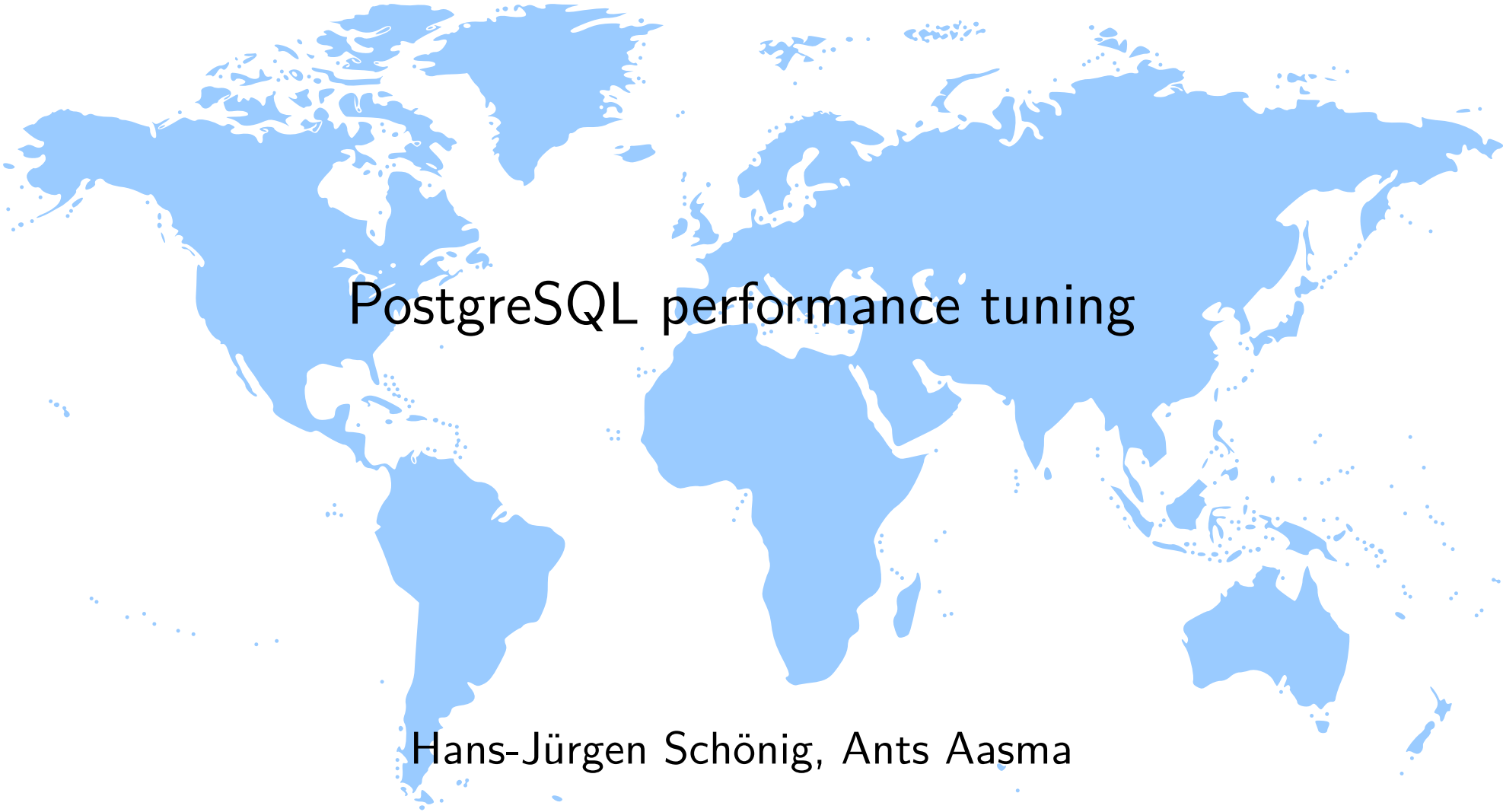

PostgreSQL performance tuning

Hans-Jürgen Schönig, Ants Aasma

www.cybertec.at

Topics

What we are talking about

- Finding bottlenecks
- Speeding up imports
- Managing missing indexes
- Improving disk performance
- Dealing with planner issues
- Optimizing functions

Finding bottlenecks

Detecting bottlenecks

- It is hard to tune without knowing what is going on
- pg_stat_statements can be used to find CPU and I/O intensive queries
- How to do it:
  - Add pg_stat_statements to shared_preload_libraries in postgresql.conf
  - Restart the database server
  - Run 'CREATE EXTENSION pg_stat_statements' in your databases

pg_stat_statements unleashed

- pg_stat_statements provides information about
  - CPU consumption
  - Memory
  - I/O consumption
  - Runtime
  - Number of calls
  - etc.

Leveraging pg_stat_statements

- The track_io_timing setting turns on I/O timing
- It helps to disginguish between CPU and I/O bottlenecks
- NOTE: Try pg_test_timing first to measure overhead

Creating relevance

- pg_stat_statements is full of information.
- You have to narrow it down to find the information you really need.
- People tend to look at pointless stuff
- HINT: A nice view can help

A sample view


```
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;
```

- https://goo.gl/Er764q

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

General procedure

- Work top down
- Check for missing indexes first
- Check I/O and mean execution time
- To reset statistics use:

```
SELECT pg_stat_statements_reset();
```

- Wait for some time and repeat the process

CPU vs. disk wait

- Compare total_time to blk_read_time + blk_write_time
- If the disk component is high, CPU time is not your problem
- Check for extensive buffer consumption
- Extensive use of buffers can indicate:
  - Table bloat
  - Fragmentation
  - Consider using the 'CLUSTER' command if possible
- If the query has a high disk component, use EXPLAIN (ANALYZE, BUFFERS)
- Lock contention can also be a problem

Logging: A common bottleneck

- Never ever set log_statement to all
- Extensive logging can be a serious bottleneck
- Check out the following URL: http://www.cybertec.at/logging-the-hidden-speedbrakes/

System level tools

- top
- iostat
- /proc/vmstat
- perf

Buffer cache

- pg_buffercache
- pg_prewarm
- pgfincore

PostgreSQL specific tools

- pg_top
- pg_activity

Importing data

Importing data: Important issues

- There are a couple of ways to tweak imports
- First rule: Do not use INSERT to load great amounts of data
  - The overhead is ways too large
- Use COPY in favor of INSERT
- Consider using pgloader

Unlogged tables

- If data is only needed temporarily or if data can be loaded again easily:
  - Consider using unlogged tables
- Unlogged tables do not have to write transaction load
- Bulk loading will be a lot faster
- In case of a crash an unlogged table will be empty

Indexing data

Importance of indexing

- In our experience around 70% of all performance problems are index related.
- There is no good performance without proper indexing
- Better get it right !

Normal btrees

- PostgreSQL uses high-concurrency btrees
  - Many people can change indexes at the same time
  - BUT: Contentions can still happen (due to block splits, etc.)
- Indexes can speed up searches
- Indexes provide SORTED OUTPUT
  - Sorted input is important
  - Many operations need sorted input

Examples (1):

```
test=# CREATE TABLE t_test AS SELECT *
    FROM    generate_series(1, 1000000) AS x;
SELECT 1000000
test=# CREATE INDEX idx_x ON t_test (x);
CREATE INDEX
```

Examples (2):


```
test=# explain SELECT *
    FROM t_test
    ORDER BY x DESC
    LIMIT 10;
                                QUERY PLAN
-------------------------------------------------------------------------
 Limit  (cost=0.42..0.86 rows=10 width=4)
   ->  Index Only Scan Backward using idx_x on t_test
         (cost=0.42..43680.43 rows=1000000 width=4)
(2 rows)
```

Examples (3):

```
test=# explain SELECT *, avg(x)
     OVER (ORDER BY x ROWS BETWEEN 3 PRECEDING
        AND 3 FOLLOWING)
       FROM t_test;
                        QUERY PLAN
-----------------------------------------------------------
 WindowAgg  (cost=0.42..45408.43 rows=1000000 width=4)
   ->  Index Only Scan using idx_x on t_test
    (cost=0.42..30408.42 rows=1000000 width=4)
(2 rows)
```

Functional indexes (1)

- Indexes can be defines on functions

```
test=# CREATE INDEX idx_cos ON t_test (cos(x));
CREATE INDEX
```

Functional indexes (2)

```
test=# explain SELECT * FROM t_test WHERE cos(x) = 10;
                              QUERY PLAN
-----------------------------------------------------------------------
 Bitmap Heap Scan on t_test
    (cost=95.17..4818.05 rows=5000 width=4)
  Recheck Cond: (cos((x)::double precision)
    = '10'::double precision)
  ->  Bitmap Index Scan on idx_cos
    (cost=0.00..93.92 rows=5000 width=0)
      Index Cond: (cos((x)::double precision)
    = '10'::double precision)
(4 rows)
```

Functional indexes (3)

- Functional indexes can only be based on IMMUTABLE functions.
- Can be used to shrink indexes:

```
CREATE INDEX idx_mail ON t_mail (hashtext(email));
```

- NOTE: In this case a functional index can be 40% smaller than a normal one

Functional indexes (4)

- Size can translate to performance
  - Less memory needed
  - Higher cache hit rates
  - More stuff in memory
- Typically used for names, etc.
  - “lower(name)”, etc.

Lower case vs. upper case

- There is a better way to handle this:

```
`CREATE EXTENSION citext;
```

- Use citext instead of text as data type
- There is no more need for “lower”
- NOTE: Lower cannot be forgotten accidentally

LIKE: A common problem

- LIKE can be a major problem
- A LIKE-query can be as expensive as 1000 properly indexed queries
- Usually LIKE is an alarm signal

How to fix LIKE

- btrees cannot be used in this case.
- A special operator class is needed.
- Create an extension first:

```
CREATE EXTENSION pg_trgm;
```

Let us give it a try (1)

- The goal is to index Austrian locations

```
test=# CREATE TABLE t_location (name text);
CREATE TABLE
test=# COPY t_location
    FROM PROGRAM 'curl www.cybertec.at/secret/orte.txt';
COPY 2354
```

Let us give it a try (2)

- Spelling names is hard
- It is a typical problem
- pg_trgm provides us with a “distance” function

Let us give it a try (2)

```
test=#  SELECT * FROM t_location
    ORDER BY name <-> 'Krammertneusiedel' LIMIT 5;
      name
-----------------
 Gramatneusiedl
 Klein-Neusiedl
 Potzneusiedl
 Kramsach
 Neusiedl am See
(5 rows)
```

Let us give it a try (3)

- This is a classical “did you mean”-type of query
- So far no indexes are involved
- PostgreSQL will use a sequential scan and a sort to answer the query

Let us give it a try (4)

- Creating the indexes works as follows:

```
test=# CREATE INDEX idx_trgm ON t_location
USING gist (name gist_trgm_ops);
CREATE INDEX
test=# explain SELECT * FROM t_location
ORDER BY name <-> 'Krammertneusiedel' LIMIT 5;
                        QUERY PLAN
-----------------------------------------------------
 Limit  (cost=0.14..0.58 rows=5 width=13)
   ->  Index Scan using idx_trgm on t_location
(cost=0.14..207.22 rows=2354 width=13)
      Order By: (name <-> 'Krammertneusiedel'::text)
```

pg_trgm can handle LIKE


```
test=# explain SELECT *
    FROM    t_location
    WHERE   name ~ '%ramat%';
                        QUERY PLAN
-----------------------------------------------------
 Index Scan using idx_trgm on t_location
    (cost=0.14..8.16 rows=1 width=13)
   Index Cond: (name ~ '%ramat%'::text)
(2 rows)
```

- Full support for regular expressions is available

Finding missing indexes (1)

- Make use of pg_stat_user_tables

```
SELECT  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
LIMIT 25;
```

Finding missing indexes (2)

- Look for expensive sequential scans
- There will always be sequential scans
- We just have to fix extensive scanning
- You might find tables, which you have already seen in pg_stat_statements

Disk and memory issues

On disk layout

- The physical layout on disk does make a difference
- Especially range scans are affected
- The optimizer has a decent estimation for the correction of values on disk

Preparing test data


```
test=# CREATE TABLE t_random AS SELECT *
    FROM    t_test
    ORDER BY random();
SELECT 1000000
test=# CREATE INDEX idx_random ON t_random (x);
CREATE INDEX
```

Comparing queries (1)


```
test=# explain (analyze true,buffers true) SELECT count(*)
    FROM t_test WHERE x BETWEEN 1 AND 50000;
-------------------------------------------------------------
Aggregate  (cost=1877.32..1877.33 rows=1 width=0)
    (actual time=12.196..12.196 rows=1 loops=1)
   Buffers: shared hit=361
   ->  Index Only Scan using idx_x on t_test
    (actual time=0.013..8.767 rows=50000 loops=1)
         Index Cond: ((x >= 1) AND (x <= 50000))
         Heap Fetches: 50000, Buffers: shared hit=361
 Execution time: 12.216 ms
```

Comparing queries (2)

```
test=# explain (analyze true,buffers true) SELECT count(*)
    FROM t_random WHERE x BETWEEN 1 AND 50000;
-------------------------------------------------------------
Aggregate ... (actual time=19.315..19.315 rows=1 loops=1)
 Buffers: shared hit=4564
 ->  Bitmap Heap Scan on t_random
    (actual time=5.205..16.095 rows=50000 loops=1)
     Recheck Cond: ((x >= 1) AND (x <= 50000))
     Heap Blocks: exact=4425 Buffers: shared hit=4564
    ->  Bitmap Index Scan on idx_random
          Index Cond: ((x >= 1) AND (x <= 50000))
          Buffers: shared hit=139
 Execution time: 19.365 ms
```

Low correlation

- Low correlation can lead to excessive buffer usage
- This in turn leads to low performance
- Too many blocks have to be touched
- CLUSTER can help

Clustering a table

```
test=# \h CLUSTER
Command:     CLUSTER
Description: cluster a table according to an index
Syntax:
CLUSTER [VERBOSE] table_name [ USING index_name ]
CLUSTER [VERBOSE]
```

- NOTE: A clustered table will not maintain its state forever. Reclustering might be useful. The table is locked while clustering, which can be an issue.

The Linux I/O scheduler

- Linux offers more than just one I/O scheduler
- The scheduler can have an impact on speed under high load.
- Checking the I/O scheduler:

```
[hs@paula sda]$ cat /sys/block/sda/queue/scheduler
noop anticipatory deadline [cfq]
```

General rules

- We want to be fast, not fair
- “deadline” is usually better than cfq
- “noop” can make sense if you happen to use extremely fast SSDs
- Add “elevator=” to your boot line to set the default I/O scheduler permanently

Adjusting work_mem

- work_mem is an essential runtime parameter
- It is influences sorts, aggregates, etc.
- It is a “per operation” parameter
- NOTE: Adjustments might change your execution plan

work_mem in action (1)

- Preparing data:

```
CREATE TABLE t_test (id serial, name text);
CREATE TABLE
INSERT INTO t_test (name) SELECT 'hans'
    FROM generate_series(1, 100000);
INSERT 0 100000
INSERT INTO t_test (name) SELECT 'paul'
    FROM generate_series(1, 100000);
INSERT 0 100000
```

work_mem in action (2)

```
test=# explain analyze SELECT count(*)
        FROM t_test GROUP BY name;
                          QUERY PLAN
-----------------------------------------------------------
HashAggregate  (cost=4082.00..4082.02 rows=2 width=5)
    (actual time=49.914..49.915 rows=2 loops=1)
  Group Key: name
  ->  Seq Scan on t_test
      (actual time=0.005..12.526 rows=200000)
 Execution time: 49.941 ms
```

work_mem in action (3)


```
test=# explain analyze SELECT count(*)
        FROM t_test GROUP BY id;
                          QUERY PLAN
-----------------------------------------------------------
GroupAggregate
    (actual time=98.272..160.843 rows=200000 loops=1)
  Group Key: id
  ->  Sort (actual time=98.267..117.808 rows=200000 )
        Sort Key: id
        Sort Method: external sort  Disk: 2736kB
    ->  Seq Scan on t_test
        (actual time=0.005..20.681 rows=200000)
 Execution time: 183.451 ms
```

work_mem in action (4)

- Adjusting work_mem will fix the slow plan
- Changes can be made globally in postgresql.conf or per session:

```
test=# SET work_mem TO '1 GB';
SET
```

work_mem in action (5)

```
test=# explain analyze SELECT count(*)
    FROM    t_test  GROUP BY id;
                    QUERY PLAN
-----------------------------------------------------
 HashAggregate
    (actual time=78.921..129.767 rows=200000)
   Group Key: id
   ->  Seq Scan on t_test
    (actual time=0.005..14.811 rows=200000)
 Planning time: 0.037 ms
 Execution time: 141.156 ms
(5 rows)
```

work_mem in action (6)

- Note that this is not about input data
- It is about the number of groups (= output data)
- work_mem will also have an impact on the sort algorithm used by PostgreSQL
  - external sort disk
  - quick sort memory
  - top-N heapsort

Adjusting maintenance_work_mem

- maintenance_work_mem is used for administrative tasks
- CREATE INDEX, VACUUM, etc. will benefit
- NOTE: Do not increase this value blindly because too excessive settings can harm CREATE INDEX.

Adjusting effective_cache_size

- It tells the optimizer about the total amount of memory for caching on your system.
- It helps PostgreSQL to adjust I/O costs during planning.
- Rule of thumb: Set if to 70% of RAM

Adjusting shared_buffers

- shared_buffers are somewhat tricky to set.
- Rule of thumb: 25-40% of RAM but not more than 8-16 GB.
- PostgreSQL does not scale with too much memory.
- Remember: PostgreSQL does not use direct I/O but fetches stuff from the filesystem.

The ideal value

- We know that speed improves with more memory.
- However, at some point the trend seems to reverse.
- The idea value depends on data and workload.
- Some experiments might turn out to be helpful.

Setting values

- Keep in mind that most settings can be changed on a per session value.
- You can also use "ALTER DATABASE ... SET" or "ALTER USER ... SET" for a more fine grained setup.

Finding optimizer issues

What is is all about

- Usually the optimizer creates a close-to-perfect plan.
- However, sometimes it does not.
- How can that be attacked?

Common issues

- Wrong estimates due to
  - statistics
  - functions
  - correlation

How to approach issues

- Run 'EXPLAIN ANALYZE' to see, where estimates and real values start to differ.
- Check for missing indexes first.
- Missing indexes are more common than people tend to think.

Common issues: Nested loops

- Nested loops: Remember, nested loops are O(n^2)
- If a nested loop is underestimated, it can turn into a nightmare.
- Try: 'SET enable_nestloop TO off'

Common issues: Function calls

- Functions can lead to wrong estimates.
- Set cost parameters accordingly (ROWS, COST).
- Remember: PostgreSQL does not have statistics given your input parameters.

Common issues: Expensive sorting

- Check for expensive sorts.
- Make sure sorts can happen in memory.
- Consider fixing work_mem.

Common issues: Correlation

- PostgreSQL stores statistics about each column.
- PostgreSQL has (as of 9.5) no cross-column statistics.
- Problem: 20% of all people like skiing, 20% of all people live in the desert. How many people living in the desert like skiing?
- Those two columns are not statistically independent.
- Estimates might be too high.
- Hard to approach: Try to use a combined index or try to rewrite the query.
- To fix over estimated: Add dummy clauses

Common issues: UNION vs. UNION ALL

- 'UNION ALL' appends results
- 'UNION' filters duplicates
  - Ways more expensive
- Make sure that if you really want to filter duplicates or if duplicates can actually happen.
- In many cases 'UNION ALL' is enough.

Common issues: Random I/O

- Spinning disks have slow access times.
- Queries can have very unpredictable runtimes when there is too much random I/O.
- A typical queries:

```
SELECT ... FROM tab WHERE field IN (... large list ...);
SELECT ...
    FROM    tab
    WHERE   field = 'frequent_value_in_large_table';
```

Common issues: Random I/O

- Try to keep data together.
- Shrink data to increase cash hits if possible.
- Try to run 'CLUSTER'
- Try to go for index-only scans
- Keeping data together locally makes sense in most cases.

Optimizing functions

Writing functions

- Make sure the function is marked properly:
  - VOLATILE: The result is inherently unstable
  - STABLE: Same transaction, same input, same output
  - IMMUTABLE: Same input, same output (always)

What the optimizer does


```
SELECT * FROM x WHERE y = func(10);
```

- If func is VOLATILE (default), PostgreSQL cannot make use of indexes
- The function might be called millions of times
- STABLE and IMMUTABLE functions can make use of indexes

Adjusting costs

- Various parameters can be set:

```
| COST execution_cost
| ROWS result_rows
| SET configuration_parameter
    { TO value | = value | FROM CURRENT }
```

- Config parameters can be used to adjust work_mem, optimizer settings and alike

General advice

- Go for an SQL function if possible
- Keep in mind: PL/pgSQL is an interpreted language and therefore slower than compiled code
- Avoid extensive result sets: They have to be kept in memory entirely