PostgreSQL Tuning Basics

Warsaw PostgreSQL Meetup / 23.11.2019

Tomáš Vondra

tomas.vondra@2ndquadrant.com / tomas@pgaddict.com

Agenda

1) basic configuration
   - shared_buffers
   - (maintenance_)work_mem
   - max_connections
   - effective_cache_size

2) checkpoint tuning
   - checkpoint_segments (timeout / completion_target)
   - max_wal_size
   - bgwriter (delay / …)

3) autovacuum tuning
   - scale factor, limit, ...

4) other config options
   - wal_level
   - synchronous_commit
   - default_statistics_target
   - effective_io_concurrency

5) a little bit about hardware / OS
   - … the whole time

Sources

PostgreSQL 9.0 High Performance (Gregory Smith)

- exhaustive analysis of the topic
- more or less basis for this workshop

PostgreSQL 9 High Availability (Shaun M. Tomas)

- not really about tuning, but HA is “related topic”
- hardware planning, performance triage, …

What Every Programmer Should Know About Memory

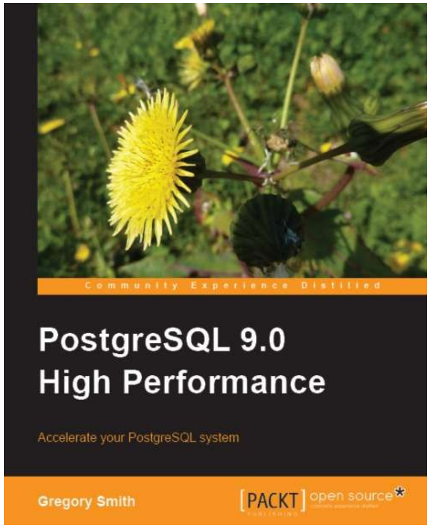
- Ulrich Drepper, Red Hat
- http://www.akkadia.org/drepper/cpumemory.pdf
- low-level features of CPU and RAM

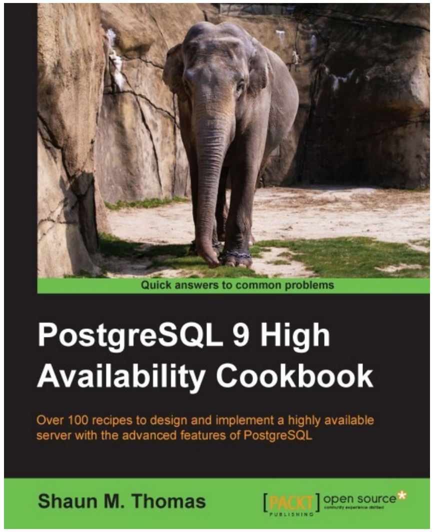
Righting Your Writes (Greg Smith)

- http://2ndquadrant.com/media/pdfs/talks/RightingWrites.pdf

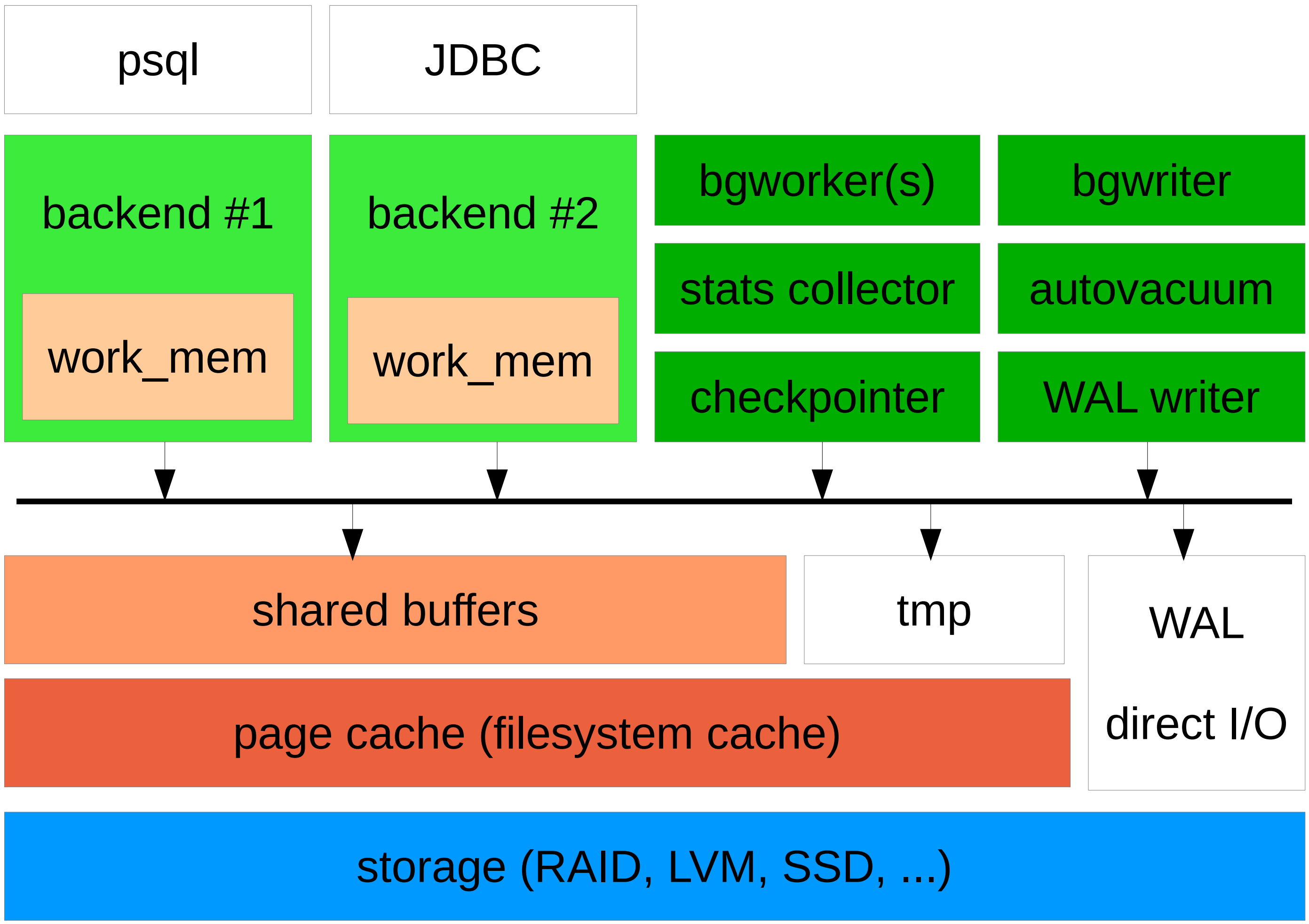
PostgreSQL Wiki

- https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server





PostgreSQL architecture



basic configuration

shared_buffers

- paměť vyhrazená pro databázi
- prostor sdílený všemi databázovými procesy
- cache “bloků” z datových souborů (8kB)
  - částečně duplikuje page cache (double buffering)
- bloky se dostávají do cache když …
  - backend potřebuje data (SQL dotaz, autovacuum, ...)
- bloky se dostávají z cache když
  - nedostatek místa v cache (LRU)
  - průběžně (background writer)
  - checkpoint
- bloky mohou být čisté nebo změněné (“dirty”)

shared_buffers

- default 128MB (used to be 32MB before 9.3)
  - goal “has to start everywhere” - low default values
  - 32MB limit is motivated by kernel limits (SHMALL)
  - 9.3 allocates shmem differently, SHMALL irrelevant
  - 128MB better, but still conservative (small systems)
  - can benefit from (explicit) huge pages
- What is optimal value?
  - high “cache hit ratio”, without wasting memory
  - larger sizes -> higher overhead, double buffering
- Why DB won’t check available RAM and pick “optimal” size?
  - depends on workload (how app uses the DB) and “active set”
  - might be sharing RAM with other stuff (appserver, ...)

shared_buffers

- iterative monitoring-based tuning
  1. pick conservative initial value (1GB?)
  2. measure important metric
     - cache hit ratio (viz. pg_stat_bgwriter)
     - usage of shared buffers (pg_buffercache)
     - eviction of dirty buffers (pg_stat_bgwriter, checkpoints)
     - latence of operations (queries), maintenance, data loads, …
  3. increase shared_buffers size (e.g. 2x)
  4. measure important metrics again
     - Did they improve? Repeat the shared buffer size increase (2x)
     - reduce the size again and finish

- reproducible application benchmark (not a stress test)
  - the same thing, but you can iterate much faster

shared_buffers

- pg_buffercache
  - http://www.postgresql.org/docs/devel/static/pgbuffercache.html
  - extension available with PostgreSQL (usually in -contrib package)
  - adds a new system view (list of blocks in shared buffer cache)

```
CREATE EXTENSION pg_buffercache;

SELECT
    datname,
    usagecount,
    COUNT(*) AS buffers,
    COUNT(CASE WHEN isdirty THEN 1 ELSE NULL END) AS dirty
FROM pg_buffercache JOIN pg_database d
                 ON (reldatabase = d.oid)
GROUP BY 1, 2
ORDER BY 1, 2;
```

work_mem

- memory limit for operations (sorts, hash tables, ...)
  - default 4MB (very conservative, fine for OLTP)
  - one query can do multiple operations -> multiple buffers
  - affects planning (query costing, possibility of plans)
  - some operations don’t fully respect (Hash Aggregate)
- when exceeded, a temporary file is used
  - not necessarily a slowdown (can stay in page cache)
  - may use a different algorithm (quick-sort, merge sort)
- optimální value depends on
  - RAM available (after subtracting shared buffers)
  - number of parallel queries
  - query complexity (OLTP vs. OLAP/BI)

work_mem

- example
  - system has (RAM – shared_buffers) available memory
  - we don’t want to use everything (page cache, OOM atd.)
  - expect all connections are active (consider connection pool)
  - expect each query uses 2 * work_mem

```
work_mem = 0.25 * (RAM – shared_buffers) / max_connections / 2;
```

- “spojené nádoby FIXME” (fewer queries -> higher work_mem possible)
- alternative approach
  - look at slow queries
  - would they benefit from increasing work_mem / how much?
  - chec if that risks OOM and then change config

work_mem

- work_mem not necessarily the same for everyone
- can be modified per session

```
SET work_mem = '1TB';
```

- can be modified per user

```
ALTER USER webuser SET work_mem = '8MB';
ALTER USER dwhuser SET work_mem = '128MB';
```

- can be modified per database

```
ALTER DATABASE webapp SET work_mem = '8MB';
ALTER DATABASE dwh SET work_mem = '128MB';
```

- http://www.postgresql.org/docs/devel/static/sql-alteruser.html
- http://www.postgresql.org/docs/devel/static/sql-alterdatabase.html

maintenance_work_mem

- similar to work_mem, but for “maintenance” operations
  - CREATE INDEX, REINDEX, VACUUM, REFRESH
- default 64MB – not bad, but increase can be quite beneficial
  - e.g. REINDEX of large tables etc.
- may have significant impact, but not necessarily “the more is better”

```
test=# set maintenance_work_mem = '4MB';
test=# create index test_1_idx on test(i);
CREATE INDEX
Time: 27076,920 ms

test=# set maintenance_work_mem = '64MB';
test=# create index test_1_idx on test(i);
CREATE INDEX
Time: 39468,621 ms
```

max_connections

- default value 100 is a bit too high in many cases
  - assumes many connections are inactive
  - may not be true, backends will interfere and cause slowdown
  - context switches, lock contention, disk contention, more RAM used, cache line contention (CPU caches), …
  - results in lower performance / throughput, latencies, ...
- rough “traditional” formula

```
((core_count * 2) + effective_spindle_count)
```

- better to use lower value and a connection pool (e.g. pgbouncer)

https://wiki.postgresql.org/wiki/Number_Of_Database_Connections

wal_level

- determines which information ned to be written to Write Ahead Log
- multiple levels, adding more and more information
- `minimal`
  - local recovery only (crash, immediate shutdown)
  - may skip WAL for some commands (CREATE TABLE AS, CREATE INDEX, CLUSTER, COPY into a table created in the same transaction)
- `replica (10+)`
  - WAL archiving (log-file shipping replication, warm_standby)
  - read-only standby
- `logical`
  - allows logical replication (interprets WAL log)

effective_cache_size

- default 4GB, but does not directly allocate anything
- simply a “hint” for the query planner
  - How likely is it that block “X” is in memory / no disk read needed?
  - What fraction of blocks will I need to read from the disk?
- good formula

  (shared buffers + page cache) * fraction
- page cache is an estimate
  - remaining RAM without kernel memory, work_mem, other apps ...
- often used fractions
  - 0.75 - aggressive (a lot of sharing of data between backends)
  - 1/max_connections – defensive, backends on different subsets of data
- usually not worth spending a lot of time on
  - reasonable default, increasing has small impact (compared to other options)

checkpoint tuning

https://blog.2ndquadrant.com/basics-of-tuning-checkpoints/

CHECKPOINT

- WAL
  - split into 16MB segments (by default)
  - limited number of segments, recycling
- COMMIT
  - write into a transaction log (WAL) + fsync
  - sequential writes (efficient on most hardware)
  - modify data files in shared_buffers (no immediate disk write)
- CHECKPOINT
  - after “filling” WAL or timeout (checkpoint_timeout)
  - writes out changes from shared buffers to data files
  - write to page cache + fsync at the end
  - checkpoint_flush_after helps to remove “spikes”

CHECKPOINT

- checkpoints need to be done with “proper frequency”
  - too often – prevents optimizations (merging writes, ordering writes)
  - too rarely – long recovery, have to keep more WAL segments
- two basic “triggers” for checkpoint
  - expiration of a time limit (checkpoint_timeout)
  - generating too much WAL (checkpoint_segments / max_wal_size)

```
(3 * checkpoint_segments) ~ max_wal_size
```

checkpoint_timeout

- checkpoint_timeout
  - maximum distance between checkpoints
  - default 5 minut (fairly agressive), maximum 1 day
- rough upper limit on recovery time (but not quite)
  - recovery is often faster (just writes to data files)
  - but not necessarily
    - recovery is single-threaded
    - may not have anything in memory (reboot)

checkpoint_completion_target

- up to 8.2 problem with I/O spiked during checkpoint
  - write everything at once + fsync
  - goal:
    - spread writes to page cache in time
    - finish writes with enough remaining time for kernel to do flushes in the background (final fsync fast)
  - works both with “timed” and “xlog” checkpoints
  - checkpoint_flush_after – alternative solution

checkpoint tuning

- pg_stat_bgwriter

```
SELECT checkpoints_timed, checkpoints_req
  FROM pg_stat_bgwriter;

 checkpoints_timed | checkpoints_req
-------------------+-----------------
               201 |             159
```

- vast majority of checkpoints should be “timed”
- goal is to minimize checkpoints_req value
  - can’t be 100% (shutdown, CREATE DATABASE, ...)

bgwriter

- background writer (bgwriter)
  - background process regularly walking shared buffers, evicting unused ones
  - makes sure there are enough clean (not modified) buffers for queries
- pg_stat_bgwriter
  - system catalog (global) with bgwriter statistics
  - number of blocks written for various readons (and other metrics)
  - buffers_alloc – blocks loaded into shared buffers
  - buffers_checkpoint – written out by checkpointer
  - buffers_clean – written out by bgwriter
  - buffers_backend – written out by backends (impact on queries)

```
SELECT
   now(),
   buffer_checkpoint, buffer_clean, buffer_backend, buffer_alloc
FROM pg_stat_bgwriter;
```

bgwriter (delay / ...)

- alternative approach to sizing shared buffers
  - smaller shared buffers + more aggressive background eviction
  - often you can’t have sufficiently large shared buffers
- `bgwriter`
  - monitor number of buffers needed by backends per interval
  - evict a multiple of the number (in the background)
- `bgwriter_delay = 200ms`
  - delay between runs of bgwriter process
- `bgwriter_lru_multiplier = 2.0`
  - multiple of pages needed in previous round
- `bgwriter_lru_maxpages = 100`
  - max number of pages written out in each round

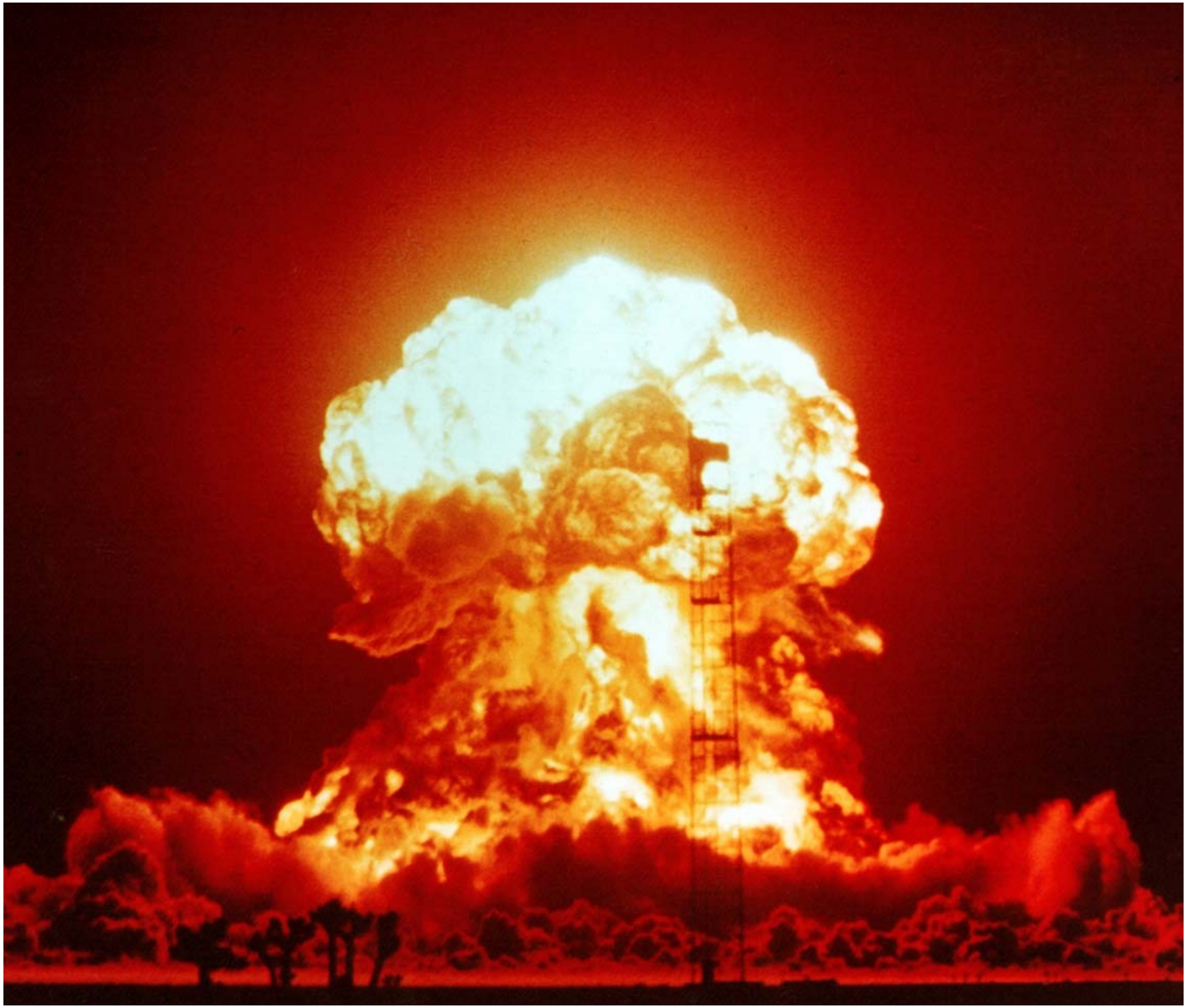
autovacuum tuning

https://blog.2ndquadrant.com/autovacuum-tuning-basics/

autovacuum options

- autovacuum = on
- autovacuum_work_mem = -1 (maintenance_work_mem)
- autovacuum_max_workers = 3
- autovacuum_naptime = 1min
- **autovacuum_vacuum_threshold = 50**
- **autovacuum_analyze_threshold = 50**
- **autovacuum_vacuum_scale_factor = 0.2**
- **autovacuum_analyze_scale_factor = 0.1**
- ...
- autovacuum_vacuum_cost_delay = 20ms
- **autovacuum_vacuum_cost_limit = -1** (vacuum_cost_limit = 200)
- vacuum_cost_page_hit = 1
- vacuum_cost_page_miss = 10
- vacuum_cost_page_dirty = 20

autovacuum = off



http://en.wikipedia.org/wiki/Nuclear_explosion

launcher and workers

- `autovacuum_naptime = 1min`
  - interval between runs of “autovacuum launcher” process
  - in each interval, the launcher will try to start a worker on each db
  - interval between starting autovacuum workers (1min / num of DBs)
  - else – interval between autovacuum runs on a given DB
- `autovacuum_workers = 3`
  - number of “worker” processes doing the actual work is limited
  - if all workers are busy, new can’t be started

autovacuum thresholds

- how does the autovacuum system which tables need cleanup?
  - monitoring of number of deleted/modified rows
  - after exceeding a limit, the table is considered for cleanup

```
modified_rows > (threshold + total_rows * scale_factor)
```

- `autovacuum_vacuum_threshold = 50`
- `autovacuum_vacuum_scale_factor = 0.2`
- `autovacuum_analyze_threshold = 50`
- `autovacuum_analyze_scale_factor = 0.1`
- row counts come from system catalogs (pg_class, pg_stat_all_tables)

autovacuum limit

- `autovacuum_vacuum_cost_limit`
  - globální limit, sdílený všemi autovacuum worker procesy
  - zvýšení `autovacuum_max_workers` většinou nic neřeší (je jich víc ale pracují pomaleji)
- lze předefinovat pro jednotlivé tabulky

```
ALTER TABLE t SET (autovacuum_vacuum_cost_limit = 1000);
```

  - tabulka (resp. autovacuum worker) je vyjmuta z globálního limitu a limit je aplikován na samostatného workera
  - ale stále to nezaručuje že volný worker bude k dispozici

autovacuum throttling

- autovacuum workers should not use too much resources (I/O, CPU)
- throttling activity over time, based on basic operations
  - `vacuum_cost_page_hit = 1        # read from cache`
  - `vacuum_cost_page_miss = 10      # read from OS`
  - `vacuum_cost_page_dirty = 20     # modified`
- time is divided into small intervals, with budget per interval
  - `autovacuum_vacuum_cost_delay = 20ms`
  - `autovacuum_vacuum_cost_limit = -1 (200)`
- so there’s total “per second” budget 10000 (= 50 x 200)
  - 10000 cache hits / second
  - 1000 reads / second => 8MB/s
  - 500 writes / second => 4MB/s

autovacuum tuning

- DON’T DISABLE AUTOVACUUM!
  - Seriously. Don’t repeat our mistakes.
- increase the throttling limits (4/8 MB/s is way too low)
  - increase cost_limit, decrease cost_delay
  - depends on available hardware resources (CPU, I/O)
  - current systems should handle 10x that
- trigger autovacuum more often
  - “If it hurts, do it more often.”
  - 10% is fine on 10GB table, not so much on 1TB one
  - Don’t overdo it, a bit of bloat is natural / beneficial.
- maybe increase number of workers

autovacuum fails

- Triggering autovacuum more often can be making things worse.
- If there’s a long transaction (forgotten session, prepared transaction, …)
  - autovacuum can’t actually cleanup anything
  - it’ll be triggered over and over (CPU utilization, I/O traffic, ..)
- autovacuum can also do ANALYZE to collect stats
  - both phases are throttled
  - VACUUM cancels itself when there’s lock request on the table
  - ANALYZE can’t cancel itself – may block DDL

logging and monitoring

logging & monitoring

- important options
  - `log_line_prefix (string)`
  - `log_min_duration_statement (integer)`
  - `log_checkpoints (boolean)`
  - `log_temp_files (integer)`
  - `log_lock_waits (integer)`
  - `log_auto_vacuum_min_duration (integer)`

- interesting tools
  - http://dalibo.github.io/pgbadger/

auto_explain

- `auto_explain.log_min_duration (integer)`
- `auto_explain.log_analyze (boolean)`
- `auto_explain.log_buffers (boolean)`
- `auto_explain.log_timing (boolean)`
- `auto_explain.log_triggers (boolean)`
- `auto_explain.log_verbose (boolean)`
- `auto_explain.log_format (enum)`
- `… další volby …`

https://www.postgresql.org/docs/current/auto-explain.html

pg_stat_statements

- userid
- dbid
- queryid
- query
- calls
- total_time
- rows
- shared_blks_hit
- shared_blks_read
- shared_blks_dirtied
- shared_blks_written
- local_blks_hit
- local_blks_read
- local_blks_dirtied
- local_blks_written
- temp_blks_read
- temp_blks_written
- blk_read_time
- blk_write_time

other config options

Durability tuning

<u>safe</u>

- synchronous_commit = off
- checkpoint_segments / max_wal_size = high number
- unlogged tables (lost after DB crash, not replicated)

<u>unsafe</u>

- fsync = off
- full_page_writes = off
- unlogged tables (lost after DB crash, not replicated)

https://www.postgresql.org/docs/current/non-durability.html

synchronous_commit

- should we wait for confirmation tuning?
  - “durability tuning” long before the NoSQL hype
  - still fully transactional / ACID
- up to 9.0 only on / off options
- 9.1 added sync replication – many more options
  - on (default) – wait for commit confirmation
  - off – don’t wait for local WAL confirmation
  - local – do not wait for a replica, local WAL is enough
  - remote_write – wait for write into WAL on a replica
  - remote_apply – applied on replica (visible)
- can be set “per transaction”
  - important transactions “on”, less important “local”

https://www.postgresql.org/docs/current/runtime-config-wal.html

wal_log_hints

- MVCC
  - each row has ID of two transactions – INSERT / DELETE
  - when reading data, we need to check visibility of those xids
  - expensive (CPU), hint bits are “flags” caching the results
  - not necessarily WAL-logged (can be recalculated)
  - problem after recovery on replicas (after failover / hot standby)
  - hint bits not set, everything has to be checked from scratch
  - data checksums enable this automatically
- odkazy
  - http://en.wikipedia.org/wiki/Multiversion_concurrency_control
  - http://www.postgresql.org/docs/current/static/mvcc-intro.html
  - http://momjian.us/main/writings/pgsql/internalpics.pdf
  - http://momjian.us/main/writings/pgsql/mvcc.pdf

random_page_cost

- our optimization is based on computing “cost” of queries
  - amount of resources (CPU, I/O consumed by the execution)
  - lower cost => can execute faster => lower duration
- five basic cost parameters determining cost of basic operations
  - seq_page_cost = 1
  - random_page_cost = 4
  - cpu_tuple_cost = 0.01
  - cpu_index_tuple_cost = 0.005
  - cpu_operator_cost = 0.0025
- difficult to verify changes to those values
  - may improve one query, hurt other
- the most common thing is tweaking random_page_cost
  - on SSD, big RAID arrays maybe lower to 2, or even 1.5

statement_timeout

- from time to time you’ll get “runaway query”
  - e.g. cartesian product generating 100 trilions of rows
  - using a lot of CPU or I/O (or both)
  - affects other activity on the system (user queries, vacuuming)
- it’s better kill / fix such queries, because they’ll not finish anyway
- statement_timeout
  - limit on maximal query duration (milliseconds)
  - affects “everything” (data loads, …)
  - just like work_mem etc. can be set per user / db
- alternative
  - cron skript (allows e.g. matching queries by regexp, ...)

temp_file_limit

- another way to limit “greedy” queries
  - if query requires too much temporary files
- usually means the query will run for a long time anyway
  - so statement_timeout will kill it eventually
  - but until then it’ll put pressure on the I/O subsystem
  - may push all interesting data from page cache
  - may interact with kernel write cache config (dirty_bytes, background_dirty_bytes)