



Ladění výkonu PostgreSQL

Prague PostgreSQL Developer Day 2022 / 1.6.2022

Tomáš Vondra

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

© 2022 Tomas Vondra, under Creative Commons Attribution-ShareAlike 3.0

<http://creativecommons.org/licenses/by-sa/3.0/>

Agenda

1) základní konfigurace

- 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) další konfigurační volby

- wal_level
- synchronous_commit
- default_statistics_target
- effective_io_concurrency

5) něco málo o hardwaru / OS

- ... průběžně

Zdroje

PostgreSQL 9.0 High Performance (Gregory Smith)

- vyčerpávající přehled problematiky
- víceméně základ tohoto workshopu

PostgreSQL 9 High Availability (Shaun M. Tomas)

- ne přímo o tuningu, ale HA je “příbuzné téma”
- hardware planning, performance triage, ...

What Every Programmer Should Know About Memory

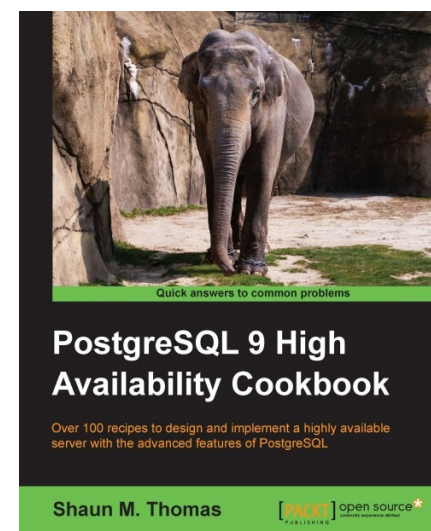
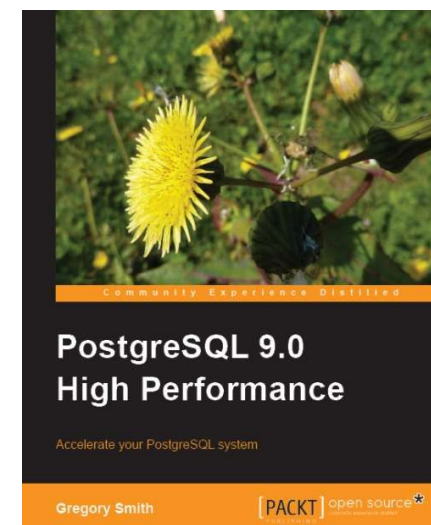
- Ulrich Drepper, Red Hat
- <http://www.akkadia.org/drepper/cpumemory.pdf>
- hutné low-level pojednání o CPU a RAM

Righting Your Writes (Greg Smith)

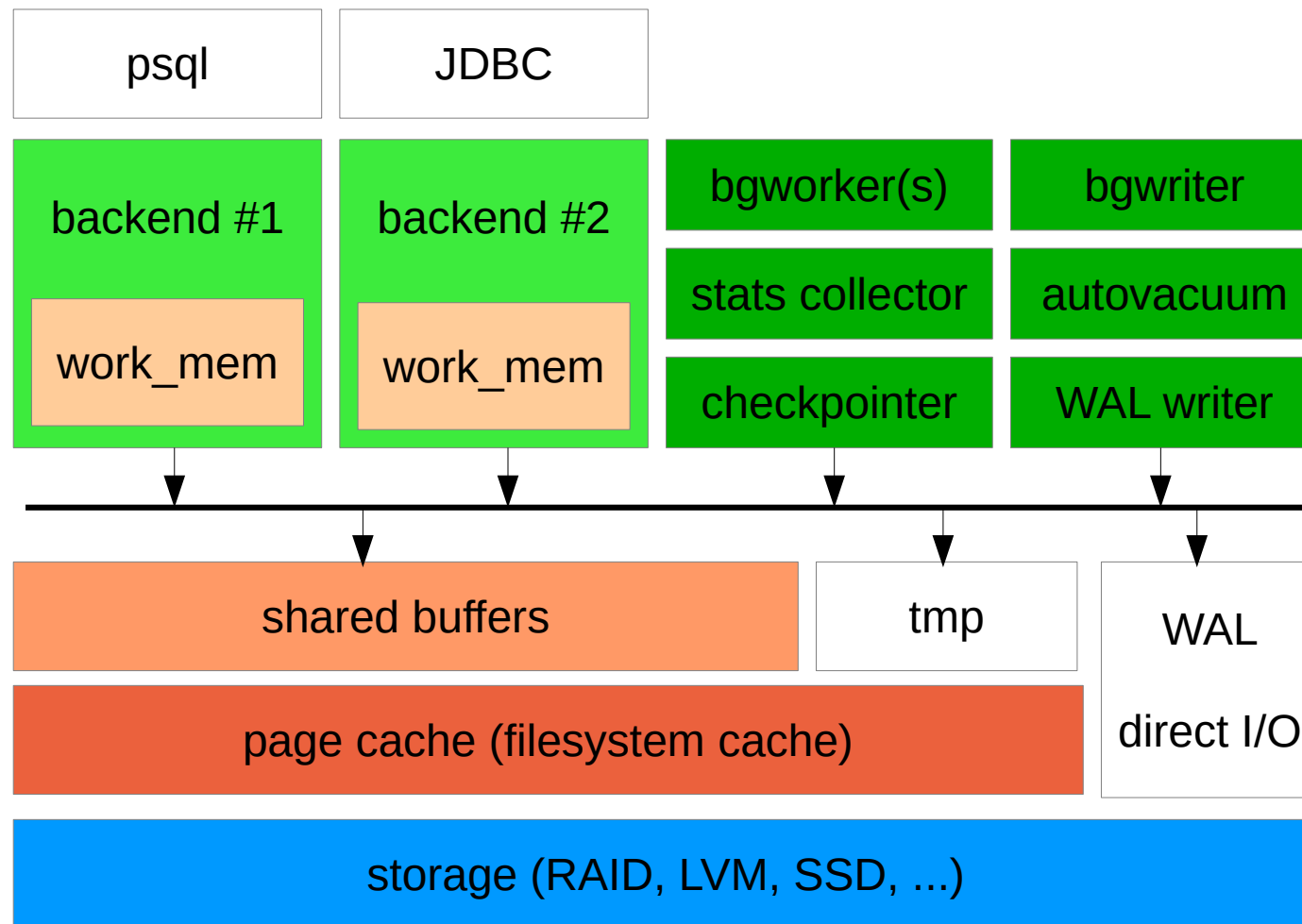
- <http://2ndquadrant.com/media/pdfs/talks/RightingWrites.pdf>

PostgreSQL Wiki

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



PostgreSQL architektura



základní konfigurace

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 32MB, resp. 128MB (od 9.3)
 - cílem je “musí nastartovat všude” - nízké výchozí hodnoty
 - 32MB limit je motivovaný limity kernelu (SHMALL)
 - 9.3 alokuje sdílenou paměť jinak, nepodléhá SHMALL
 - 128MB lepší, ale stále konzervativní (malé systémy)
 - lze využít huge pages (explicitní)
- co je optimální hodnota ...
 - vysoká “cache hit ratio” hodnota, bez plýtvání pamětí
 - větší cache -> větší overhead, double buffering
- Proč si DB nezjistí RAM a nenastaví “optimální” hodnotu?
 - závisí na workloadu (jak aplikace používá DB)
 - závisí objemu dat (aktivní části)

shared_buffers

- iterativní přístup na základě monitoringu
 1. konzervativní počáteční hodnota (1GB?)
 2. změříme důležité metriky
 - cache hit ratio (viz. pg_stat_bgwriter)
 - využití shared buffers (pg_buffers)
 - vyhazování špinavých bufferů (pg_stat_bgwriter, checkpointy)
 - latence operací (queries), maintenance, data loads, ...
 3. zvýšíme objem shared_buffers 2x
 4. znovu změříme důležité metriky
 - zlepšily se – opakujeme zvýšení shared buffers (2x)
 - jinak se vrátíme o krok zpět a konec
- reprodukovatelný aplikační benchmark (ne stress test)
 - to samé ale iterace lze provádět daleko rychleji

shared_buffers

- pg_buffercache
 - <http://www.postgresql.org/docs/devel/static/pgbuffercache.html>
 - extenze dodávaná s PostgreSQL (často v extra -contrib balíku)
 - přidá další systémový pohled (seznam bloků v 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 (relidatabase = d.oid)
```

```
GROUP BY 1, 2
```

```
ORDER BY 1, 2;
```

work_mem

- limit paměti pro operace
 - default 4MB (velmi konzervativní)
 - jedna query může použít několik operací
 - ovlivňuje plánování (oceňování dotazů, možnost plánů)
 - některé operace nerespektují (Hash Aggregate)
- při překročení se použije temporary file
 - nemusí nutně znamenat zpomalení (může být v page cache)
 - může se použít jiný algoritmus (quick-sort, merge sort)
- optimální hodnota závisí na
 - množství dostupné paměti
 - počtu paralelních dotazů
 - složitosti dotazů (pgbench-like dotazy vs. analytické dotazy)

work_mem

- příklad
 - systému zbývá (RAM – shared_buffers) paměti
 - nechceme použít všechno (vytlačilo by to page cache, OOM atd.)
 - očekáváme že aktivní budou všechna spojení (connection pool)
 - očekáváme že každý dotaz použije $2 * \text{work_mem}$

$\text{work_mem} = 0.25 * (\text{RAM} - \text{shared_buffers}) / \text{max_connections} / 2;$

- “spojené nádoby” (méně dotazů -> víc work_mem)
- alternativní postup
 - podívej se na pomalé dotazy,
 - zjisti jestli mají problém s work_mem / kolik by potřebovaly
 - zkontroluj jestli nehrozí OOM a případně změň konfiguraci

work_mem

- work_mem nemusí být nastaveno pro všechny stejně
- lze měnit per session

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

- lze nastavit per uživatele

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

- lze nastavit per databázi

```
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

- obdobný význam jako work_mem, ale pro “maintenance” operace
 - CREATE INDEX, REINDEX, VACUUM, REFRESH
- default 64MB – není špatné, ale může se hodit zvýšit
 - např. REINDEX velkých tabulek apod.
- může mít výrazný vliv na operace, ale ne nutně “více je lépe”

```
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 hodnota 100 je často příliš vysoká
 - očekává se že část spojení je neaktivní
 - to často neplatí, backendy si “šlapou po prstech”
 - context switche, lock contention, disk contention, používá se víc RAM, cache line contention (CPU caches), ...
 - výsledkem je snížení výkonu / propustnosti, zvýšení latencí, ...
- orientační “tradiční” vzorec
$$((\text{core_count} * 2) + \text{effective_spindle_count})$$
- radši použijte nižší hodnotu a connection pool (např. pgbouncer)

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

wal_level

- které informace se musí zapisovat do Write Ahead Logu
- několik úrovní, postupně přidávajících detaily
- `minimal`
 - lokální recovery (crash, immediate shutdown)
 - může přeskočit WAL pro některé příkazy (CREATE TABLE AS, CREATE INDEX, CLUSTER, COPY do tabulky vytvořené ve stejné transakci)
- `replica`
 - WAL archivace (log-file shipping replikace, `warm_standby`)
 - read-only standby
- `logical`
 - možnost logické replikace (interpretuje WAL log)

effective_cache_size

- default 4GB, ale neovlivňuje přímo žádnou alokaci
- slouží čistě jako “nápopěda” při plánování dotazů
 - Jak pravděpodobné je že blok “X” nebudu číst z disku?
 - Jakou část bloků budu muset číst z disku?
- dobrý vzorec
$$(\text{shared buffers} + \text{page cache}) * X$$
- page cache je odhad
 - zbývající RAM bez paměti pro kernel, work_mem, ...
- často se používá
 - $X = 0.75$ - agresivní hodnota (hodně sdílení mezi backendy)
 - $X = 1/\text{max_connections}$ – defensivní hodnota
- většinou nemá cenu to příliš detailně ladit
 - rozumný default, zvyšování má malý vliv (ve srovnání s dalšími parametry)

checkpoint tuning

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

CHECKPOINT

- WAL
 - rozdělený na 16MB segmenty
 - omezený počet segmentů, recyklace
- COMMIT
 - zápis do transakčního logu (WAL) + fsync
 - sekvenční povaha zápisů
 - úprava dat v shared_buffers (bez zápisu na disk)
- CHECKPOINT
 - při “zaplnění” WAL nebo timeoutu (checkpoint_timeout)
 - zápis změn ze shared buffers do datových souborů
 - zapisuje se do page cache + fsync na konci
 - checkpoint_flush_after pomáhá odstranit “špičky”

CHECKPOINT

- checkpointy chceme dělat “tak akorát často”
 - příliš často – brání optimalizacím (služování zápisů, řazení)
 - příliš zřídka – dlouhá recovery, akumulace WAL segmentů
- dva základní důvody pro checkpoint
 - vypršení časového limitu (checkpoint_timeout)
 - vygenerování množství WAL (checkpoint_segments/max_wal_size)

$$(3 * \text{checkpoint_segments}) \sim \text{max_wal_size}$$

checkpoint_timeout apod.

- `checkpoint_timeout`
 - maximální vzdálenost mezi checkpointy
 - default 5 minut (dost agresivní), maximum 1 den
 - jakýsi orientační horní limit na recovery time, ale ...
 - recovery je většinou rychlejší (jenom samotné zápisy)
 - ale ne nutně (recovery je single-threaded, nemá data v paměti)
- `checkpoint_completion_target`
 - až do 8.2 problém s I/O při checkpointu (write všeho + fsync)
 - `completion_target` rozkládá zápisy v čase
 - cíl: dokončit zápisy do page cache s předstihem, nechat kernelu čas na zápis dat na disk (rychlý fsync na závěr)
 - funguje s “timed” i “xlog” checkpointy
 - `checkpoint_flush_after` – alternativní řešení

checkpoint tuning

- pg_stat_bgwriter

```
SELECT checkpoints_timed, checkpoints_req  
FROM pg_stat_bgwriter;
```

checkpoints_timed		checkpoints_req
-----+-----		
201		159

- obecně většina checkpointů by měla být “timed”
- cílem je minimalizovat checkpoints_req
 - nelze 100% (shutdown, CREATE DATABASE, ...)

bgwriter

- background writer (bgwriter)
 - proces pravidelně procházející buffery, aplikuje clock-sweep
 - jakmile “usage count” dosáhne 0, zapíše ho (bez vyhození z cache)
- pg_stat_bgwriter
 - systémový pohled (globální) se statistikami bgwriteru
 - (mimo jiné) počty bloků zapsaných z různých důvodů
 - buffers_alloc – počet bloků načtených do shared buffers
 - buffers_checkpoint – zapsané při checkpointu
 - buffers_clean – zapsané “standardně” bgwriterem
 - buffers_backend – zapsané “backendem” (chceme minimalizovat)

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

bgwriter (delay / ...)

- alternativní přístup k velikosti shared buffers
 - menší shared buffery + agresivnější zápisy
 - ne vždy lze libovolně zvětšovat shared buffers
- `bgwriter_delay = 200ms`
 - prodleva mezi běhy bgwriter procesu
- `bgwriter_lru_maxpages = 100`
 - maximální počet stránek zapsaných při každém běhu
- `bgwriter_lru_multiplier = 2.0`
 - násobek počtu stránek potřebných během předchozích běhů
 - adaptivní přístup, ale podléhá `bgwriter_lru_maxpages`
- problém
 - statické hodnoty, nenavázané na velikost shared buffers

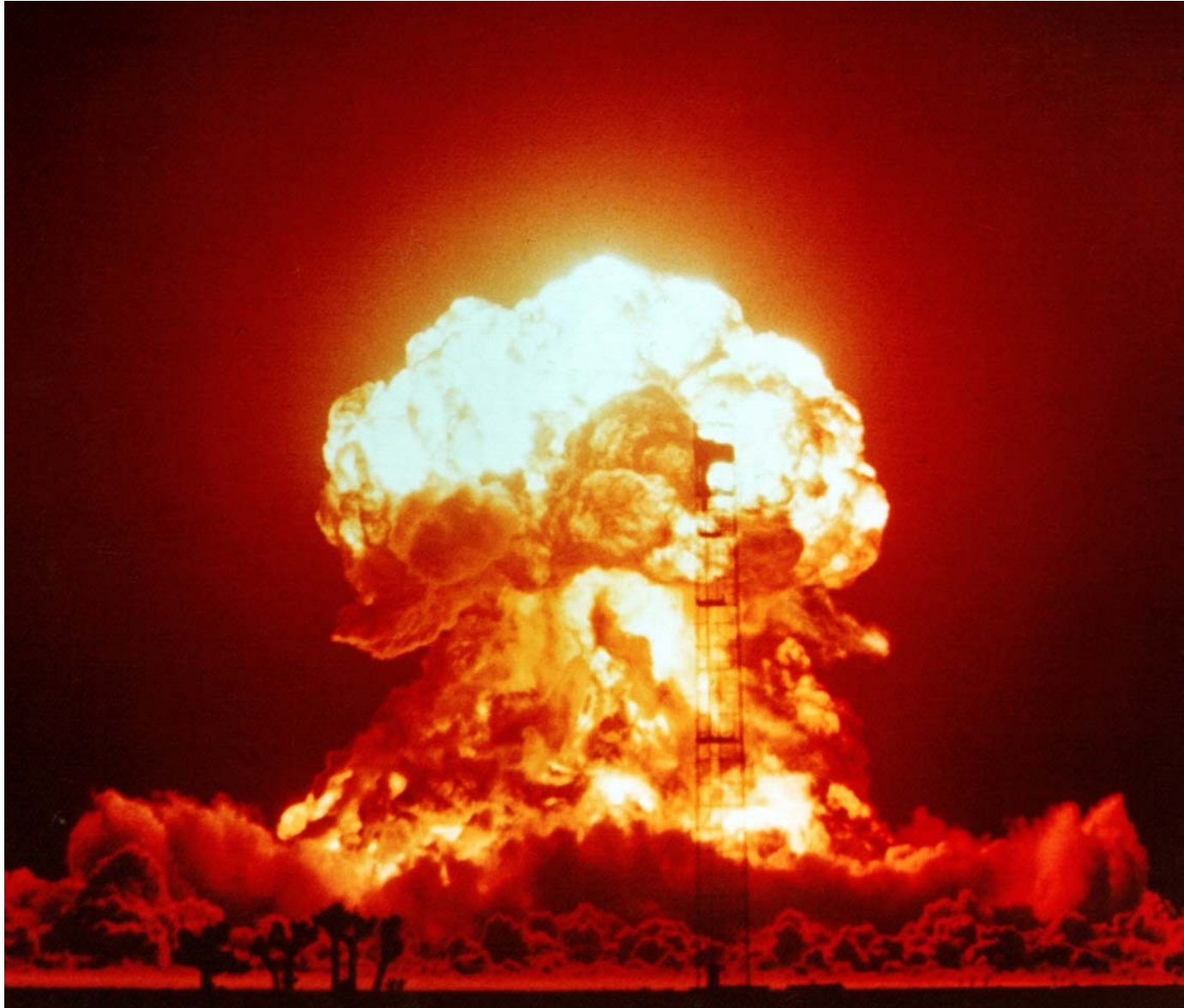
autovacuum tuning

[**https://blog.2ndquadrant.com/autovacuum-tuning-basics/**](https://blog.2ndquadrant.com/autovacuum-tuning-basics/)

autovacuum options

- `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_freeze_max_age = 200000000`
- `autovacuum_multixact_freeze_max_age = 400000000`
- `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

autovacuum thresholds

- `autovacuum_naptime = 1min`
 - prodleva mezi běhy “autovacuum launcher” procesu
 - v rámci běhu se spustí autovacuum na všech DB
 - interval mezi autovacuum procesy (1 minuta / počet DB)
 - jinak – interval mezi běhy autovacuum na konkrétní DB
- `autovacuum_vacuum_threshold = 50`
- `autovacuum_analyze_threshold = 50`
- `autovacuum_vacuum_scale_factor = 0.2`
- `autovacuum_analyze_scale_factor = 0.1`
 - parametry určující které tabulky se mají čistit / analyzovat
 - informace se berou ze systémového katalogu `pg_stat_all_tables`

$$\text{změněných_řádek} > (\text{threshold} + \text{celkem_řádek} * \text{scale_factor})$$

autovacuum thresholds

- `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`
- parametry určující agresivitu autovacuum operace
 - maximálně 200 stránek za kolo / 10000 za vteřinu (jen buffer hits)
 - pokud musí načíst do shared buffers tak 20 / 1000
 - pokud skutečně vyčistí (tj. označí jako dirty) tak 10 / 500
- problém pokud není dostatečně agresivní
 - bloat (nečistí se smazané řádky) – pomalejší dotazy, diskový prostor
 - wraparound (32-bit transaction IDs)

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 fails

- přístup “méně práce ale častěji” může v některých situacích uškodit
- pokud existuje dlouhá transakce (zapomenutá session, dlouhý dotaz, ...)
 - autovacuum nic vyčistit nemůže
 - bude se pouštět stále dokola (vytěžovat CPU ...)
- pokud autovacuum pustí “VACUUM ANALYZE”
 - obě fáze podléhají throttlingu
 - VACUUM uvolňuje zámek na tabulce po každé 8kB stránce
 - ANALYZE nikoliv – může blokovat DDL

logování a monitoring

logging & monitoring

- důležité volby
 - `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)
- zajímavé nástroje
 - <http://pgfouine.projects.pgfoundry.org/>
 - <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 ...

<http://www.postgresql.org/docs/devel/static/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

další konfigurace

Durability tuning

bezpečné

- synchronous_commit = off
- checkpoint_segments = vysoké číslo
- unlogged tables (při pádu DB zmizí, nereplikují se)

nebezpečné

- fsync = off
- full_page_writes = off
- unlogged tables (při pádu DB zmizí, nereplikují se)

<http://www.postgresql.org/docs/9.4/static/non-durability.html>

synchronous_commit

- má se čekat na dokončení commitu?
 - “durability tuning” dlouho před NoSQL hype
 - stále plně transakční / ACID
- až do 9.0 jenom on / off
- 9.1 přidala synchronní replikaci – daleko víc možností
 - on (default) – čekej na commit
 - remote_write – čekej i na zápis do WAL na replice (9.1)
 - local – nečekej na repliku, stačí lokální WAL (9.1)
 - off – nečekej ani na lokální WAL
- jde nastavovat “per transakce”
 - důležité “on”, méně důležité “local”

<http://www.postgresql.org/docs/9.4/static/runtime-config-wal.html>

wal_log_hints

- MVCC
 - zpřístupnění více verzí řádek paralelně běžícím transakcím
 - moderní alternativa k zamykání (řádek, tabulek, ...)
 - u každého řádku jsou ID dvou transakcí – INSERT / DELETE
 - nutná kontrola zda daná transakce skončila (commit / rollback)
 - náročné na CPU, takže se výsledek cachuje pomocí “hint bitu”
 - původně se nelogovalo do WAL (kontrola se zopakuje)
 - problém po recovery / na replikách (hint bity nenastaveny, všechno se musí zkontrolovat znovu proti commit logu)
- 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

- při plánování se používá zjednodušený model “ceny” plánu
- pět “cost” proměnných
 - seq_page_cost = 1
 - random_page_cost = 4
 - cpu_tuple_cost = 0.01
 - cpu_index_tuple_cost = 0.005
 - cpu_operator_cost = 0.0025
- změna hodnot se velmi obtížně se ověřuje
 - jednomu dotazu to pomůže, druhému ublíží :-(
- asi jediné co se obecně vyplatí tunit je random_page_cost
 - na SSD, velkých RAID polích zkuste snížit na 2, možná 1.5

statement_timeout

- optimalizovat rychlost dotazů je fajn, ale ...
- čas od času se objeví “nenažraný dotaz”
 - např. kartézský skoučín produkující 100 trilionů řádek
 - žere spoustu CPU času nebo I/O výkonu (případně obojí)
 - ovlivňuje ostatní aktivitu na systému
- je dobré takové dotazy průběžně zabíjet / fixovat
- statement_timeout
 - limit na maximální délku dotazu (milisekundy)
 - oblivňuje “všechno” (loady, ...)
 - stejně jako work_mem apod. jde nastavit per user / db
- alternativa
 - cron skript (umožňuje např. regulární výrazy na dotaz, ...)

temp_file_limit

- alternativní způsob omezení “nenažraných” dotazů
- pokud dotaz generuje moc temporary souborů
- většinou to znamená že běží dlouho
 - nakonec ho zabije statement_timeout
 - do té doby ale bude přetěžovat I/O subsystém
 - z page cache vytlačí všechna zajímavá data
 - nežádoucí interakce s write cache kernelu (dirty_bytes, background_dirty_bytes)

hardware a OS

Disk layout

- části data directory mají rozdílné charakteristiky
 - nároky na trvanlivost zapsaných dat
 - způsob přístupu (sekvenční / náhodný, čtení / zápis)
- WAL
 - sekvenční zápisy/čtení, čte se výjimečně (recovery/replikace)
 - kritická část databáze (ztráta nepřijatelná)
- data files
 - mix zápisů a čtení, náhodně i sekvenčně
 - kritická část databáze (ztráta nepřijatelná)
- temporary soubory (pgsql_tmp)
 - mix zápisů a čtení, v podstatě jenom sekvenčně
 - nekritická část (žádný fsync, irelevantní po pádech/restarted)

io scheduler

- říká se že ...
 - **cfq** – se snaží o férové rozdělení I/O kapacity v rámci systému
 - **deadline** - lehký jednoduchý scheduler snažící se o rovnoměrné latence
 - **noop** - dobrá volba tam kde si to zařízení stejně chytře přeorganizuje (nerotační média aka SSDs, RAID pole s řadičem a BBWC)
 - ~~**anticipatory** – koncepčně podobný deadline, se složitější heuristikou která často zlepšuje výkon (a někdy naopak zhoršuje)~~
 - multiqueue deadline, multiqueue Kyber, multiqueue BFQ, ...
- ale **cfq** je “default” scheduler, a např. “ionice” funguje jenom v něm
- ve většině případů minimální rozdíly

souborové systémy

- ext3
 - ne, zejména kvůli problémům při fsync (všechno)
- ext4, XFS
 - cca stejný výkon, berte to co je “default” vaší distribuce
 - mount options: noatime, barrier=(0|1), discard (SSD, ext4 i xfs)
 - XFS tuning: allocsize, agcount/agsize (mkfs)
- ZFS / BTRFS
 - primárním motivem není vyšší výkon, ale jiné vlastnosti (odolnost na commodity hw, snapshoty, komprese, ...)
 - špatný výkon na random workloadech (zejména r-w)
 - dobrý výkon na sekvenčních (lepší než XFS / EXT4, ...)
 - <http://www.citusdata.com/blog/64-zfs-compression>

readahead

```
blockdev --getra /dev/sda
```

- default 256 sektorů ($256 * 512\text{B} = 128\text{kB}$)
- zvýšení většinou zlepší výkon sekvenčního čtení
 - nemá negativní dopad na náhodný přístup (adaptivní)
- nemá cenu to dlouho tunit
 - zejména ne na základě syntetického testu (dd, fio, bonnie++, ...)
 - výkon se většinou zpočátku rychle zlepší, pak už roste pomalu
- dobré hodnoty
 - 2048 (1MB) – standardní disky
 - 16384 (8MB) – střední RAID pole
 - 32768 (16MB) – větší RAID pole

effective_io_concurrency

- Kolik paralelních I/O requestů dokáží disky odbavit?
 - samostatné disky -> 1 nebo víc (díky TCQ, NCQ optimalizacím)
 - RAID pole -> řádově počet disků (spindlů)
 - SSD disky -> hodně paralelních requestů
- více I/O requestů zaslaných “předem” umožňuje
 - optimalizace v disku (pořadí, slučování, ...)
 - využití paralelních součástí disků (RAID spindles, SSD kanály)
- překládá na počet stránek které se mají načíst dopředu
- používá se jenom pro “Bitmap Heap Scan”
 - přeskakuje některé stránky podle bitmapy
 - sequential scan apod. spoléhají na “kernel readahead”
- není součástí plánování (používá se až při exekuci)

effective_io_concurrency

pgbench scale 3000 (45GB database), Intel S3500 SSD

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT * FROM pgbench_accounts
WHERE aid BETWEEN 1000 AND 500000000 AND abalance != 0;
```

QUERY PLAN

```
-----
Bitmap Heap Scan on pgbench_accounts
    (cost=1059541.66..6929604.57 rows=1 width=97)
    (actual time=5040.128..23089.651 rows=1420738 loops=1)
    Recheck Cond: ((aid >= 1000) AND (aid <= 500000000))
    Rows Removed by Index Recheck: 3394823
    Filter: (abalance <> 0)
    Rows Removed by Filter: 48578263
    Buffers: shared hit=3 read=1023980
    -> Bitmap Index Scan on pgbench_accounts_pkey
        (cost=0.00..1059541.66 rows=50532109 width=0)
        (actual time=5038.707..5038.707 rows=49999001 loops=1)
        Index Cond: ((aid >= 1000) AND (aid <= 500000000))
        Buffers: shared hit=3 read=136611
Total runtime: 46251.375 ms
```


effective_io_concurrency

<http://www.postgresql.org/message-id/CAHyXU0yiVvfQAnR9cyH=HWh1WbLRsioe=mzRJTHwtr=2azsTdQ@mail.gmail.com>

effective_io_concurrency 1:	46.3 sec, ~ 170 mb/sec
effective_io_concurrency 2:	49.3 sec, ~ 158 mb/sec
effective_io_concurrency 4:	29.1 sec, ~ 291 mb/sec
effective_io_concurrency 8:	23.2 sec, ~ 385 mb/sec
effective_io_concurrency 16:	22.1 sec, ~ 409 mb/sec
effective_io_concurrency 32:	20.7 sec, ~ 447 mb/sec
effective_io_concurrency 64:	20.0 sec, ~ 468 mb/sec
effective_io_concurrency 128:	19.3 sec, ~ 488 mb/sec
effective_io_concurrency 256:	19.2 sec, ~ 494 mb/sec

Did not see consistent measurable gains > 256
effective_io_concurrency. Interesting that at setting of '2'
(the lowest possible setting with the feature actually working)
is pessimal.

vm / dirty memory

- data se nezapisují přímo na disk, ale do “write cache”
 - kernel to dle svého uvážení zapíše na disk
 - zápis lze vynutit pomocí fsync()
 - konfliktní požadavky na velikost write cache
- chceme velkou write cache
 - kernel má větší volnost v reorganizaci requestů
 - spíše sekvenční zápisy, rewrite bloků → jediný zápis
- chceme malou write cache
 - rychlejší shutdown (nutnost zapsat všechno)
 - chceme také read cache (read-write ratio 90%)
 - požadavek na hodně paměti (složité dotazy -> nutnost zapsat)

vm / dirty memory

- kernel používá dva prahy / thresholdy

```
dirty_background_bytes <= dirty_bytes  
dirty_background_ratio <= dirty_ratio
```

- `dirty_background_bytes`
 - kernel začne zapisovat na pozadí (pdflush writeback)
 - procesy stále zapisují do write cache
- `dirty_bytes`
 - kernel stále zapisuje ...
 - ... procesy nemohou zapisovat do write cache (vlastní writeback)

<https://www.kernel.org/doc/Documentation/sysctl/vm.txt>

<http://lwn.net/Articles/28345/>

<http://www.westnet.com/~gsmith/content/linux-pdflush.htm>

vm / dirty memory

- dobré hodnoty (vyladěné pro I/O subsystém)
 - pokud máte řadič s write cache


```
vm.dirty_background_bytes = ... write cache ...  
vm.dirty_bytes = (8 x dirty_background_bytes)
```
 - pokud RAID řadič (nebo SSD) nemáte, tak radši nižší hodnoty
- /proc/meminfo
 - Dirty — waiting to get written back to the disk (kilobytes)
 - Writeback — actively being written back to the disk (kilobytes)
- nepoužívejte “_ratio” varianty
 - s aktuálními objemy RAM (stovky GB) příliš velké hodnoty
 - 1% z 128GB => víc než 1GB
 - problémy při přidání RAM (najednou jiné thresholdy)

vm / dirty memory

- nízké hodnoty dirty_background_bytes ale mohou být problém
 - dotazy s velkými temp soubory (třídění, ...) začnou dělat I/O
- PostgreSQL 9.6 toto řeší pravidelným “flush”
 - bgwriter_flush_after = 512kB
 - backend_flush_after = 0
 - wal_writer_flush_after = 1MB
 - checkpoint_flush_after = 256kB
- od 9.6 můžete nechávat dirty_background_bytes vyšší

zone_reclaim_mode

- NUMA (Non-Uniform Memory Access)
 - What Every Programmer Should Know About Memory (PDF)
 - architektura na strojích s mnoha CPU / velkými objemy RAM
 - RAM rozdělená na části, každá připojená ke konkrétnímu CPU
 - přístupy konkrétního jádra k částem RAM jsou dražší / levnější
 - `numactl --hardware`
- zone_reclaim_mode
 - snaha uvolnit paměť v aktuální zóně (snaha o data locality)
 - <https://www.kernel.org/doc/Documentation/sysctl/vm.txt>

For file servers or workloads that benefit from having their data cached, zone_reclaim_mode should be left disabled as the caching effect is likely to be more important than data locality.
 - <http://frosty-postgres.blogspot.cz/2012/08/postgresql-numa-and-zone-reclaim-mode.html>

Transparent Huge Pages

- umožňují kernelu spravovat paměť v 2MB blocích (namísto 4kB)
 - možné i jiné velikosti (1GB apod.) ale 2MB nejčastější (x86)
 - transparentní pro userspace, kernel řeší čistě interně
 - menší overhead (jediná “page fault” pro 2MB, méně položek v TLB)
 - vyžaduje “defragmentaci” paměti (souvislé 2MB bloky)
- rozhodně se doporučuje vypnout pro PostgreSQL
 - PostgreSQL není THP-aware, alokace se všelijak prolínají atd.
 - v důsledku to znamená že spotřeba paměti může být vyšší
 - defragmentace paměti ovlivňuje rychlost “malloc” a může zásadně ovlivňovat latenci dotazů (hlavně v OLTP aplikacích)

```
echo never > /sys/kernel/mm/redhat_transparent_hugepage/enabled  
echo never > /sys/kernel/mm/redhat_transparent_hugepage/defrag
```

Neplést s huge pages pro shared_buffers! (huge_pages = try)

metodika

- mějte jasnou představu o výkonu systému
 - ideálně výsledky sady měření která můžete rychle zopakovat
 - ověření že HW je v pořádku apod. (umřel disk, baterka na radiči)
 - slouží jako baseline pro ostatní (např. na mailinglistu)
- mějte monitoring
 - spousta věcí se dá zjistit pohledem na grafy (náhlé změny apod.)
 - může vás to rovnou navést na zdroj problému nebo symptomy
 - může vám to říct kdy k problému došlo, jestli rostl pomalu nebo se to stalo náhle, apod.
- zkuste rychle vyřešit “seshora” (fix SQL dotazu, ...)
 - může se jednat o “klasický problém” (chybějící index, ...)
- pokud nejde, postupujte systematicky odspodu (neskákejte)
 - hardware, OS, databáze, aplikace, ...