Ladění výkonu PostgreSQL

Prague PostgreSQL Developer Day 2024 / 4.6.2024

Tomáš Vondra

tomas.vondra@enterprisedb.com / tv@fuzzy.cz

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

- vzhledem k času jen velmi stručný přehled problematiky
- Pokud by systém neměl bottleneck, měl by nekonečný výkon.
- Cílem není odstranit všechny bottlenecky, ale vědět o nich, odstranit ty zbytečné, a obecně stavět vyvážené systémy (k čemu je mi 100 jader když to vázne na I/O).
- Hromada věcí je neřešitelná na softwarové úrovni, zejména v rámci general-purpose operačního systému a general-purpose databáze.
- Spoustu věcí lze na softwarové úrovni vyřešit, ale je to neekonomické – koupit další RAM nebo pořádný RAID řadič je prostě lepší.
- Další spousta věcí se daleko jednodušeji řeší na aplikační úrovni – pokud máte bláznivou aplikaci která spouští absurdní SQL dotazy, těžko to vyřešíte změnami konfigurace databáze.
- Bohužel opravit aplikaci je často složité kvůli oddělení rolí “vývojář” a “DBA”, protože výkon produkční aplikace je často považován za starost DBA.
- Naučte se rozpoznávat o kterou situaci se jedná. Fail fast.
- Spousta z pravidel zmiňovaných dále výrazně závisí na aplikaci / workloadu. Pokusím se vysvětlit jak hodnoty závisí, jaké jsou možné přístupy při ladění.
- Pokud by existovala pravidla jak odvodit “optimální konfiguraci” tak by nejspíše bylo přímo v databázi zadrátované.
- Existují výjimky, např. pokud se používaly absurdně nízké hodnoty kvůli limitům v kernelu apod.

Zdroje

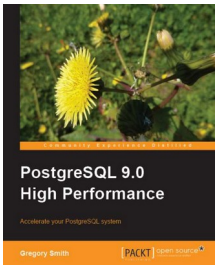
PostgreSQL 9.0 High Performance (Gregory Smith)
- vyčerpávající přehled problematiky
- víceméně základ tohoto workshopu

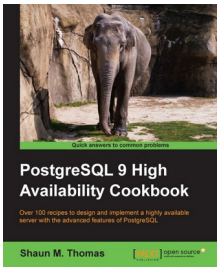
PostgreSQL 9 High Availability (Shaun M. Tomas)
- ne přímo o tuningu, ale HA je “příbuzné téma”
- hardware planning, performance triage, …

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



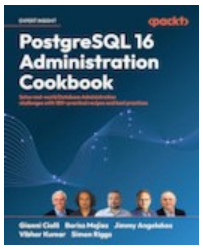


- existuje spousta kvalitních zdrojů o ladění Linuxu
  - zastaralých a nekvalitních je 10x tolik
- dávejte si pozor na doporučení na základě syntetických benchmarků
- někdy mění durabilitu za výkon (nevědomky nebo to přinejmenším nezmiňuje)

Zdroje

PostgreSQL 16 Administration Cookbook (Ciolli, Mejías, Angelakos, Kumar, Riggs)

- výborná knížka s praktickými recepty



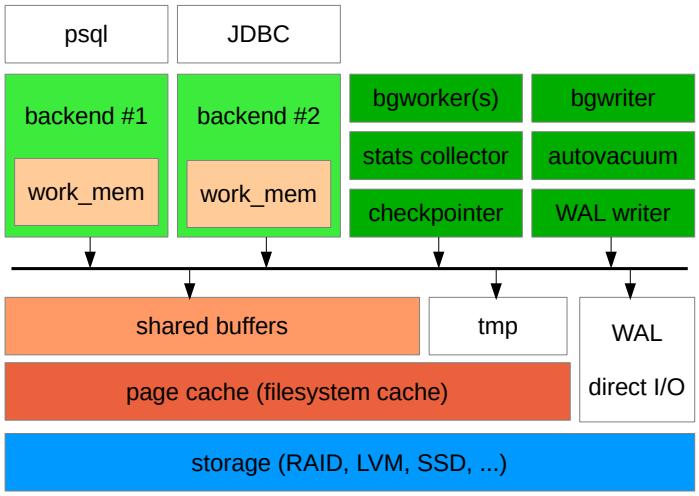
PostgreSQL 14 Internals (Egor Rogov)

- hodně detailů o “vnitřnostech”, proč/jak to funguje
- PDF zdarma

Další

- https://www.postgresql.org/docs/books/

PostgreSQL architektura



- V praxi je logické začít s tuningem “od spodu” tj. od konfigurace disku, souborových systémů, jádra, … a pak pokračovat s laděním částí PostgreSQL.
- Pro vysvětlování to ale není příliš vhodné, protože pro ladění ne-PostgreSQL částí potřebujeme mít alespoň základní představu co PostgreSQL dělá např. při checkpointech.
- Takže začneme od shared_buffers, probereme PostgreSQL části, a pak se vrátíme k prostředí.
- Primárním cílem je Linux, potažmo Unix-ové systémy obecně.

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”)

- obecně “databázová cache” do které se načítají bloky z datových souborů
- bloky (aka “datové stránky”) mají standardně 8kB (stránky paměti na x86 mají 4kB)
- PostgreSQL lze překompilovat s menšími / většími stránkami
  - ale ukazuje se že 8kB je vesměs dobrý kompromis
  - větší stránky – méně “slotů” a tedy horší adaptace na aktivní dataset
  - menší stránky – větší počet I/O operací, atd.
- PostgreSQL do velké míry spoléhá na “page cache” v kernelu, která
  - je adaptivní (velikost se mění podle požadavků na paměť)
  - má nižší overhead (ale neumí tolik věcí jako shared buffers)
  - integrovaná s I/O subsystémem

shared_buffers

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

- výchozí velikost shared_buffers je extrémně nízká, a to víceméně ze čtyř důvodů
  - technické limity – dlouhou dobu bylo předmětem SHMALL kernel limitů
  - historie (byly zvoleny kdysi dávno, když systémy měly daleko méně RAM)
  - konzervativní přístup ke zvyšování (velké skoky komplikují upgrady)
  - snaha aby výchozí konfigurace “nastartovala kdekoliv” (např. i na Rasperry Pi)
- je také otázka jak určit dobrou “optimální hodnotu”
  - existuje několik relevantních metrik (různé možné výsledky)
  - záleží na konkrétní aplikaci (objemu dat, typu dotazů, …)
- změna velikosti vyžadure restart databáze (čili nepříliš dynamické změny)
- negativní dopady příliš malých shared_buffers
  - časté dotazování do page cache a kopírování do shared_buffers
  - pokud data nejsou v page cache, je nutné I/O
  - častější vyhazování bloků ze shared_buffers – pokud je “vyhazovaný” blok modifikovaný, může to znamenat opakovaný zápis (pokud by zůstal v shared_buffers tak se zápisy sloučí a provede se jenom jeden za checkpoint)

shared_buffers

- doporučeno: iterativní přístup na základě monitoringu
  1. konzervativní počáteční hodnota (1GB?)
  2. měříme důležité metriky
     - cache hit ratio (viz. pg_stat_bgwriter)
     - využití shared buffers (pg_buffercache)
     - 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 (restarty kdykoliv)

- cache hit ratio je lehce zavádějící, protože zahrnuje jenom pro shared_buffers
  - dobré hodnoty cache hit ratio jsou > 95%
- relevantní je ale spíš (shared_buffers + page cache)
- kopírování mezi shared_buffers a page cache není zadarmo, ale je levnější než I/O
- shared_buffers jsou jediná část PostgreSQL která může rozumně používat huge pages (nižší overhead)
- cache hit ratio
  - SELECT blks_hit * 100.0 / (blks_hit + blks_read) FROM pg_stat_database;
- využití shared buffers
  - SELECT isdirty, usagecount, count(*) FROM pg_buffercache GROUP BY 1, 2;
- čistění bufferů ze shared_buffers
  - SELECT buffers_checkpoint, buffers_clean, buffers_backend FROM pg_stat_bgwriter;

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 (reldatabase = d.oid)
GROUP BY 1, 2
ORDER BY 1, 2;
```

- detailní informace o shared_buffers – pro každý 8kB blok víte jestli je použitý, která tabulka ho používá, jak často se k němu přistupuje, zda je modifikovaný apod.

work_mem

- limit paměti pro operace
  - default 4MB (velmi konzervativní, stačí pro OLTP)
  - jedna query může použít několik operací
  - ovlivňuje i plánování (oceňování dotazů, možnost plánů)
  - už respektují snad všechny operace (i 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)

- menší work_mem (s odlíváním do temp. souborů) může být rychlejší než čistě in-memory zpracování, protože zpracování v CPU cachích je řádově rychlejší

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 * work_mem

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

- “spojené nádoby” (méně klientů -> 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

- lze také monitorovat kolik temporary souborů databáze vytváří, a na základě toho usuzovat zda je vhodné zvýšit work_mem

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 web 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

- typický problém je že databáze obslujuje skupiny uživatelů s různými nároky
- OLTP uživatelé nepotřebují velké work_mem hodnoty, ale je jich hodně
- OLAP uživatelů bývá pár, ale potřebují vyšší work_mem (reporting, …)
- Elegantní možností je nastavit hodnoty per uživatel (ALTER USER …).
- Obdobně na DB instanci může být několik databází s různými nároky. Potom jde použít podobný postup s ALTER DATABASE.

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 100 – moc vysoké nebo moc nízké číslo?
  - záleží na počtu jader apod. (N jader – maximálně N x 100% CPU)
  - očekává se že část spojení je neaktivní ...
  - … ale pokud to neplatí tak si spojení překáží
  - context switche, lock contention, cache line contention (CPU caches), …, disk contention, používá se víc RAM (work_mem)
  - výsledkem je snížení výkonu / propustnosti, zvýšení latencí, OOM, ...
- orientační “tradiční” vzorec

```
((core_count * 2) + 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 (např. CREATE INDEX, CLUSTER, COPY do tabulky vytvořené ve stejné transakci)
  - nejasné zda se vyplatí, staré bugy v kódu
- `replica`
  - WAL archivace (log-file shipping replikace, warm_standby)
  - read-only standby, zálohy, ...
- `logical`
  - možnost logické replikace (interpretuje WAL log)
- reálné rozdíly jsou minimální (hlavně replica vs. logical)

- ve starších verzích PostgreSQL existují další možnosti (archive, hot_standby)
- to je vesměs to samé jako “replica”

effective_cache_size

- default 4GB, ale neovlivňuje přímo žádnou alokaci
- slouží čistě jako “nápověda” při plánování dotazů
  - Jak pravděpodobné je že blok “X” nebudu číst z disku?
  - Jakou část bloků budu muset číst z disku?
- dobrý vzorec

  (shared buffers + 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/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 (max_wal_size)
  - kdysi dávno parametr checkpoint_segments

```
max_wal_size ~ 2 nebo 3 checkpointy
```

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, záleží na prefetchingu)
- checkpoint_completion_target
  - kdysi dávno checkpoint zapsal vše + fsync => dopad na latenci
  - 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;
```

- Background Writer se zabývá jenom “špivanými” stránkami – pokud se stránka nijak nezměnila, není problém ji prostě vyhodit z cache.
- Obecně chcete aby buffer_backend hodnota byla co nejnižší, tak aby backendy nemusely samy zapisovat špinavé bloky.
- Celkem dobré je také minimalizovat buffer_alloc (zvětšením shared buffers).

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

- Hodnoty jsou dost konzervativní, ale zase opatrně se zvyšováním, jinak se dostanete do situace že buffery zapisujete stále dokola (kvůli opakovaným změnám).
- konzervativní je zejména hodnota bgwriter_lru_maxpages – při shared_buffers=1GB to odpovídá cca 0.1% za vteřinu (4 MB/s)
- z pg_stat_bgwriter se dá spočítat kolik zhruba potřebujete

  $$\Delta \text{(buffer\_clean + buffer\_backend)} / \Delta \text{čas}$$

  a pak použít např. 1.5x tolik (je to jenom maximum, aby byl prostor pro multiplier)
- bgwriter_lru_multiplier asi měnit nepotřebujete, ale pokud zvýšíte maxpages tak může být lepší ho o trochu snížit

autovacuum tuning

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_vacuum_cost_limit (integer)**

  Limit ceny po jejímž dosažení vacuum proces usne (na dobu určenou “delay” parametrem), Výchozí hodnota je stejná jako vacuum_cost_delay (200).

- **autovacuum_vacuum_cost_delay** (integer)

  Doba (v milisekundách) kolik bude autovacuum proces spát po dosažení cost limitu. Nastavení na 0 cost-based autovacuum vypne, tj. autovacuum nebude spát a pojede na plný plyn (což není úplně dobrý nápad).

  Vhodné hodnoty jsou většinou dost malé (10 nebo 20 milisekund apod.) aby se systém choval “interaktivně” bez větších záseků. Lepší je modifikovat cost_limit nebo následující ceny jednotlivých operací.

- **vacuum_cost_page_hit** (1)

  Cena při zpracování bufferu nalezeného v shared bufferech. Víceméně jenom získání zámku na bufferu poolu, lookup v hash tabulce a průchod obsahem stránky.

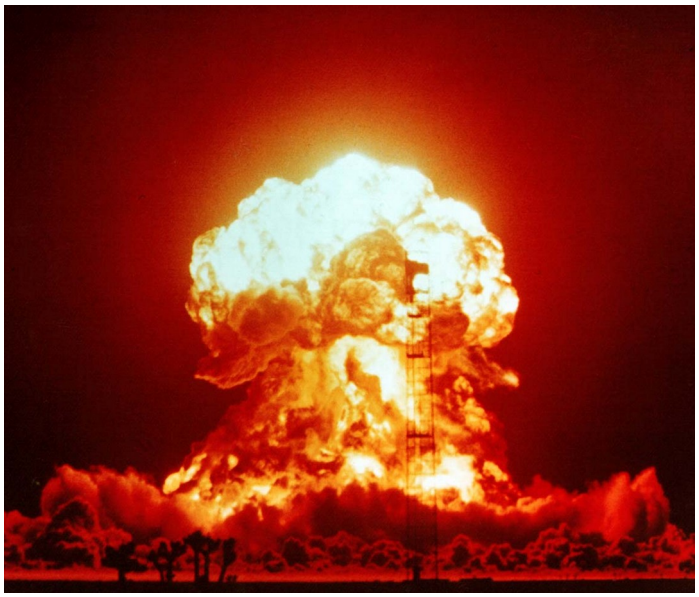
- **vacuum_cost_page_miss** (10)

  Cena při načtení bufferu z disku (resp. z page cache). Tj. přibližně to samé jako “hit” ale navíc ještě získání bloku z disku.

- **vacuum_cost_page_dirty** (20)

  To samé jako page_miss, ale s tím že vacuum musel blok upravit (tj. našel tam staré nepotřebné řádky) a musel je zapsat zpět na disk.

autovacuum = off



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

- Často se stává že admin dostane ticket o pomalém systému, podívá se na systém (iotop, iostat, ...), zjistí že většinu I/O generuje autovacuum proces a vypne ho (nebo alespoň notně omezí).
- Chvíli to funguje, ale o to horší jsou pak důsledky – v systému se hromadí bloat (mrtvé řádky), narůstá místo zabrané na disku apod. Případně se ještě narazí na wraparound.
- Důsledkem je přinejmenším velmi vysoké vytížení systému kvůli nutné údržbě, nebo I úplný výpadek (např. v případě wraparoundu).
- Řešení je paradoxně opačné – agresivnější autovacuum ;-)
- Může být dobré udělat review aplikace, jestli skutečně musí generovat tolik změn (opakovaný UPDATE jednoho řádku stále dokola v jedné transakci, apod.)

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 (integer)

  Specifies the minimum number of **updated or deleted** tuples needed to trigger a VACUUM in any one table. The default is 50 tuples. This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by changing storage parameters.

- autovacuum_analyze_threshold (integer)

  Specifies the minimum number of **inserted, updated or deleted** tuples needed to trigger an ANALYZE in any one table. The default is 50 tuples. This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by changing storage parameters.

- autovacuum_vacuum_scale_factor (floating point)

  Specifies a fraction of the table size to add to autovacuum_vacuum_threshold when deciding whether to trigger a VACUUM. The default is 0.2 (20% of table size). This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by changing storage parameters.

- autovacuum_analyze_scale_factor (floating point)
- Specifies a fraction of the table size to add to autovacuum_analyze_threshold when deciding whether to trigger an ANALYZE. The default is 0.1 (10% of table size). This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by changing storage parameters.

autovacuum thresholds

- `autovacuum_vacuum_threshold = 50`
- `autovacuum_vacuum_insert_threshold = 1000`
- `autovacuum_analyze_threshold = 50`

- `autovacuum_vacuum_scale_factor = 0.2`
- `autovacuum_vacuum_insert_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

```
změněných_řádek > (threshold + celkem_řádek * scale_factor)
```

- autovacuum_vacuum_threshold (integer)

  Specifies the minimum number of **updated or deleted** tuples needed to trigger a VACUUM in any one table. The default is 50 tuples. This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by changing storage parameters.

- autovacuum_analyze_threshold (integer)

  Specifies the minimum number of **inserted, updated or deleted** tuples needed to trigger an ANALYZE in any one table. The default is 50 tuples. This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by changing storage parameters.

- autovacuum_vacuum_scale_factor (floating point)

  Specifies a fraction of the table size to add to autovacuum_vacuum_threshold when deciding whether to trigger a VACUUM. The default is 0.2 (20% of table size). This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by changing storage parameters.

- autovacuum_analyze_scale_factor (floating point)
- Specifies a fraction of the table size to add to autovacuum_analyze_threshold when deciding whether to trigger an ANALYZE. The default is 0.1 (10% of table size). This parameter can only be set in the postgresql.conf file or on the server command line. This setting can be overridden for individual tables by changing storage parameters.

autovacuum thresholds

- `vacuum_cost_page_hit = 1`
- `vacuum_cost_page_miss = 2 (dříve 10)`
- `vacuum_cost_page_dirty = 20`
- `autovacuum_vacuum_cost_delay = 2ms (dříve 20ms)`
- `autovacuum_vacuum_cost_limit = -1 (vacuum_cost_limit=200)`
- parametry určující agresivitu autovacuum operace (PG14+)
  - maximálně 200 stránek za kolo / 100000 za vteřinu (jen buffer hits)
  - pokud musí načíst do shared buffers tak 100 / 50000
  - pokud skutečně vyčistí (tj. označí jako dirty) tak 10 / 5000
  - 800 MB/s, 400 MB/s, 40 MB/s
- dříve (11-12) 80 MB/s, 40 MB/s, 4 MB/s
- a ještě dříve 80 MB/s, 8 MB/s, 4 MB/s (hrůza)

autovacuum limit

- pokud autovacuum není dostatečně agresivní
  - bloat (nečistí se smazané řádky) – pomalejší dotazy, diskový prostor
  - wraparound (32-bit transaction IDs)
- `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”
  - doporučovaný, ale 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/

- log_line_prefix (string)

  printf-like formát hodnoty zalogované na začátku každé řádky. Rozhodně je dobré doplnit alespoň %t (timestamp), %u (uživatel), %d (databáze), %p (PID), případně další (viz. postgresql.conf).

- log_min_duration_statement (integer)

  Zaloguje všechny příkazy delší než daný limit (milisekundy). -1 vypíná (default), 0 loguje všechno.

  Je dobré nastavit na nějakou konzervativní hodnotu (tak aby většina dotazů byla drtivě delší), a poté pravidelně kontrolovat zalogované dotazy. Lze kombinovat s auto_explain extenzí.

  Trik: Parametr lze nastavit per databáze ALTER DATABASE … SET ...

- log_checkpoints (boolean)

  Checkpointy jsou jednou z nejčastějších příčin I/O spiků, a rozhodně se hodí mít je v logu tak aby se daly korelovat s pomalými dotazy apod. Kromě důvodu checkpointu (xlog, timed, immediate) a začátku/konce, zaloguje i statistiky jako počet bufferů, čas strávený zápisy a fsync operací.

  Alternativa je sledování pg_stat_bgwriter v rámci monitoringu – grafy apod. To je fajn, lepší než nic, ale často tam chybí statistiky a granularita nemusí být dostatečná (např. nevíte kdy přesně k checkpointu došlo, což ztěžuje korelaci s dotazy).

- log_temp_files (integer)

  Umožňuje logování temporary souborů – jmen a velikostí. Týká se třídění, hashů, dočasných výsledků dotazů apod. -1 neloguje, 0 – všechny soubory, jinak jen soubory větší než hodnota (kB).

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

- Někdy se stává že exekuční plán dotazu se neočekávaně změní – load velkého objemu dat, nestačí se to zanalyzovat apod. Dotaz se pustí se špatným plánem (a trvá dlouho), následně se zanalyzuje a plán se spraví. Vy k tomu ráno přijdete a netušíte proč protože plán je OK.
- Tohle umožňuje problém odhalit.
- Pozor – instrumentace není zadarmo a musí se instrumentovat všechno (předem se neví co potrvá dlouho). Takže má dopad na všechno.
- Hlavní problém je logování “skutečného času”, ale to většinou není ta nejzajímavější informace – nejzajímavější je (a) podoba exekučního plánu, (b) odhadované a skutečné počty řádek a (c) celkový čas. Takže log_timing lze nastavit na “off”.
- Tím se overhead výrazně sníží (maximálně nízké jednotky).
- Případně lze nastavit per user či per databáze (ALTER USER / ALTER DATABASE) a zapínat dle potřeby když honíte “ducha”.

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`

- Alternativní přístup k analýze SQL dotazů – namísto extrakce z logů (nekompletní) se statistiky sbírají během exekuce.
- Tradiční aplikace má relativně malý počet SQL dotazů (šablon).
- Výhoda – kompletní pohled (ne jenom příliš dlouhé dotazy).
- Nevýhoda – netriviální overhead (cca 10% podle workloadu) oproti “čistému” PostgreSQL, ale při srovnání s případem kdy se logují všechny dotazy to klidně může být lepší varianta.
- Overhead je nepřímo úměrný délce dotazů – pro kratičké dotazy může být velký, čím delší dotazy tím lépe se overhead amortizuje.

další konfigurace

Durability tuning

bezpečné

- synchronous_commit = off
- max_wal_size = 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

- seq_page_cost neměňte, považujte za konstantu a ostatní hodnoty jsou relativní

statement_timeout

- optimalizovat rychlost dotazů je fajn, ale ...
- čas od času se objeví “nenažraný dotaz”
  - např. kartézský skoučin 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
  - nekriricá 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

- není dokonalý souborový systém
- každý souborový systém má čas od času bug (delalloc, 2009)
- např. Lance Armstrong Bug (ext4)
  - … the code never fails a test, but evidence shows it’s not behaving as it should
  - http://www.pointsoftware.ch/en/4-ext4-vs-ext3-filesystem-and-why-delayed-allocation-is-bad/
- čas od času se změní default mount options (mezi verzemi kernelu)
  - moc fajn na produkci ...
- dokumentace v kernelu
  - https://www.kernel.org/doc/Documentation/filesystems/ext4.txt
  - https://www.kernel.org/doc/Documentation/filesystems/xfs.txt
- vývojáři “ZFS on Linux” tvrdí že 0.6.3 je stable / production ready
  - … mimochodem mezi 0.6.2 a 0.6.3 a jsme úplně překopali X a Y
  - prostě změny jsou stále dost masivní
  - navíc vrstvy ZPL (ZFS POSIX Layer) a SPL (Solaris Portability Layes)
  - licenční problémy (nemožnost implementace některých vlastností – např. crypto)
  - BTRFS na tom technicky není o moc líp (brzy default na distribucích, ale …)
  - ARC – lepší na některých workloadech, horší na dalších, memory hog, alien v Linuxu
  - https://www.linuxdays.cz/video/Pavel_Snajdr-ZFS_na_Linuxu.pdf

readahead

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

- default 256 sektorů (256 * 512B = 128kB)
- 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

- Některé RAID řadiče mají implementovaný vlastní readahead (různé varianty).
- Většinou funguje hůře než ten kernelový, např. proto že nevidí “do filesystému” (fungují na úrovni RAW device). Navíc kernelový readahead byl do jisté míry “ušit na míru” PostgreSQL.

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 50000000 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 <= 50000000))
   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 <= 50000000))
         Buffers: shared hit=3 read=136611
 Total runtime: 46251.375 ms
```

- Bitmap Heap Scan “přeskakuje” některé stránky v tabulce, podle toho jestli odpovídají bitmapě (vygenerované z indexu). Proto na něj nefunguje readahead.

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.
```

- dá se testovat I synteticky, např. pomocí “fio” benchmarku
- http://freecode.com/projects/fio
- http://github.com/axboe/fio
- http://github.com/axboe/fio/blob/master/HOWTO

```
[random-read]
rw=randread
ioengine=libaio
filesize=1GB
iodepth=16
runtime=15
blocksize=8192
direct=1
time_based
```

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)

- vysoký limit – vyšší propustnost, ale potenciální “záseky” (latence)
- nízký limit – nižší propustnost, ale hladší průběh (žádné zásadní záseky)

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

- Na novějších moderních systémech s více sockety je NUMA nejspíše automaticky zapnutá (detekce při bootu).

  `cat /proc/sys/vm/zone_reclaim_mode`
- Nejčastější problém který to v praxi působí je že page cache (filesystémová cache udržovaná kernelem) se nikdy pořádně nezaplní, I když databáze je dostatečně aktivní a objemově je několikanásobná oproti RAM. Důvodem je že systém se snaží udržet část "local" paměti k dispozici.
- Řešením je zone_reclaim_mode úplně vypnout, což se udělá např. přidáním

  `vm.zone_reclaim_mode = 0`

  do `/etc/sysctl.conf` (a pak `sysctl -p` pro aplikaci změněných hodnot).
- After disabling zone_reclaim_mode, the filesystem cache fills up and performance improves.
- podrobnější diskuse viz. diskuse na mailing listu (jsou I další)

  http://archives.postgresql.org/pgsql-performance/2012-07/msg00215.php
- PostgreSQL není jediný projekt který má s NUMA/zone_reclaim_mode problém:

  http://blog.jcole.us/2010/09/28/mysql-swap-insanity-and-the-numa-architecture/

  http://lwn.net/Articles/254445/

  http://marc.info/?l=linux-mm&m=128528098927584&w=2

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 řadiči)
  - 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, ...