Ladění výkonu PostgreSQL

Prague PostgreSQL Developer Day 2015 / 11.2.2015

Tomáš Vondra

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

Agenda

- shared_buffers
- (maintenance_)work_mem
- effective_cache_size
- effective_io_concurrency
- max_connections
- unix vs. TCP/IP socket
- temp_file_limit
- bgwriter (delay / ...)
- wal_level
- wal_log_hints
- synchronous_commit
- commit_delay / commit_siblings
- checkpoint_segments (timeout / completion_target)
- default_statistics_target
- autovacuum options

- bottom-up přístup (stavba systému od nejnižších vrstev)
- vzhledem k času jen velmi stručný přehled problematiky

Zdroje

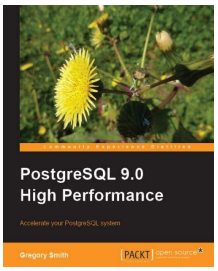
PostgreSQL 9.0 High Performance (Gregory Smith)
- vyčerpávající přehled problematiky
- víceméně základ tohoto workshopu

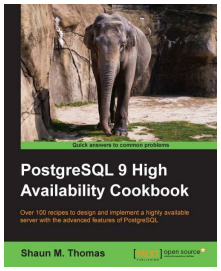
PostgreSQL 9 High Availability (Shaun M. Tomas)
- ne přímo o tuningu, ale HA je “příbuzné téma”
- hardware planning, performance triage, …

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)

#1

Každý systém má bottleneck.

- 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).

#2

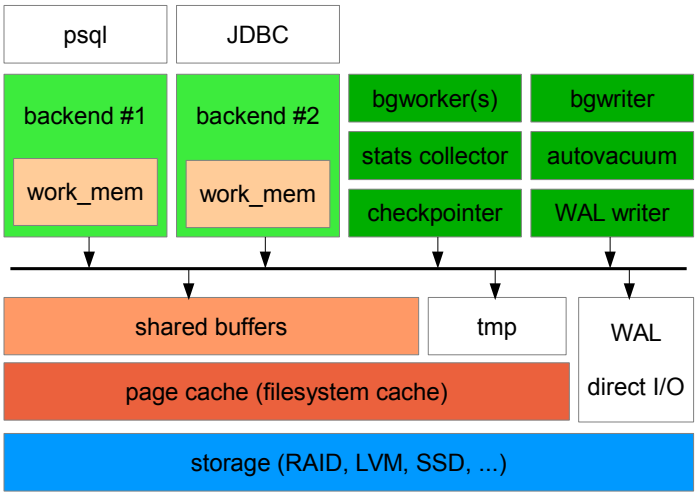
Někdy je nejlepší tuning upgrade.

- 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.

#3

- 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.

PostgreSQL architektura



PostgreSQL architektura

```
$ ps ax | grep postg
 4963 pts/2    S      0:00 /home/tomas/pg/bin/postgres -D /var/pgsql/data
 4965 ?        Ss     0:00 postgres: checkpointer process
 4966 ?        Ss     0:00 postgres: writer process
 4967 ?        Ss     0:00 postgres: wal writer process
 4968 ?        Ss     0:00 postgres: autovacuum launcher process
 4969 ?        Ss     0:00 postgres: stats collector process
```

pgbench

inicializace

```
$ createdb bench
$ pgbench -i -s 1000 bench
```

exekuce (read-write)

```
$ pgbench -c 16 -T 600 bench
```

exekuce (read-only)

```
$ pgbench -S -c 16 -j 4 -T 600 bench
```

exekuce (vlastní skript)

```
$ pgbench -c 16 -j 4 -f skript.sql -T 600 bench
```

- pgbench je stress test, nikoliv aplikační benchmark
  - říká že při daném počtu klientů je propustnost X
  - neříká (přímo) kolik uživatelů systém zvládne obsloužit apod.
- simuluje jednu konkrétní aplikaci (TPC-B), nemusí nutně vystihovat vaši aplikaci
- existují alternativní benchmarky pro jiné typy systémů (TPC-H, TPC-DS, ...)
- “standardní benchmark”
- scale (-s) – velikost datasetu, pro velikost databáze v MB stačí násobit 15
- clients (-c) – kolik databázových spojení / klientů se bude používat pro dotazy
- time (-T) – určuje délku testu ve vteřinách (alternativně počet transakcí “-t”, ale čas je praktičtější)
- read-only (-S) – použije jenom SELECT část transakce
- threads (-j) – určuje počet vláken generujících dotazy (důležité při krátkých dotazech)
- skript (-f) – umožňuje přizpůsobení transakcí vaší aplikaci (lze zadat i několik skriptů)

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 (query, autovacuum, backup ...)
- 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.

shared_buffers

- default 32MB, resp. 128MB (od 9.3)
  - 32MB je zoufalý default, motivovaný limity kernelu (SHMALL)
  - 9.3 alokuje sdílenou paměť jinak, nemá problém s limity
  - 128MB lepší, ale stále dost konzervativní (kvůli malým systémům)
- co je optimální hodnota ...
  - Proč si DB nezjistí dostupnou RAM a nenastaví “správnou” hodnotu?
  - závisí na workloadu (jak aplikace používá DB)
  - závisí objemu dat (aktivní části)
  - větší cache -> větší počet bloků -> větší management overhead
- pravidlo (často ale označované za obsolete)
  - 20% RAM, ale ne víc než 8GB

- příklad zbytečně nízké default hodnoty (kvůli kernel limitům)

shared_buffers

- optimální hodnota #2
  - tak akorát pro “aktivní část dat”
  - minimalizace evict-load cyklů
  - minimalizace vyplýtvané paměti (radši page cache, work_mem)
  - Ale jak zjistit?
- iterativní přístup na základě monitoringu
  - konzervativní počáteční hodnota (256MB?)
  - postupně zvětšujeme (2x), sledujeme výkon aplikace
  - tj. latence operací (queries), maintenance, data loads, …
  - zlepšila se latence? pokud ne – snížit (vyžaduje restart)
- reprodukovatelný aplikační benchmark (ne stress test)
  - to samé ale iterace lze udělat 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 (reldatabase = d.oid)
    GROUP BY 1, 2
    ORDER BY 1, 2;
```

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 buffery
- `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

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

work_mem

- limit paměti pro operace
  - default 4MB (velmi konzervativní)
  - jedna query může použít několik operací
  - některé operace nerespektují (Hash Aggregate)
  - ovlivňuje plánování (oceňování dotazů, možnost plánů)
- 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)

- problém s CPU cachemi
- iterativní pipelined exekutor

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
  - očekáváme že každý dotaz použije 2 * work_mem

```
work_mem = 0.25 * (RAM – shared_buffers) / 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 USER webapp SET work_mem = '8MB';
ALTER USER 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

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

- radši použijte nižší hodnotu a connection pool (např. pgbouncer)

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

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
- page cache je odhad
  - zbývající RAM bez paměti pro kernel, work_mem, ...
- často se používá
  - 50% paměti jako konzervativní hodnota
  - 75% paměti jako agresivní hodnota
- většinou nemá cenu to detailně tunit

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 -> dobré zvládají hodně paralelních requestů
- 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
```

readahead

- blockdev --getra /dev/sda
- default 256 sektorů (256 * 512B = 128kB)
- zvýšení většinou zlepší výkon sekvenčního čtení
- dobré hodnoty
  - 2048 (1MB) – standardní disky
  - 16384 (8MB) – střední RAID pole
  - 32768 (16MB) – větší RAID pole
- nemá cenu to dlouho tunit
  - zejména ne na základě syntetického testu (dd, fio, bonnie++, ...)
  - výkon se většinou zpočátku rychle zlepší, pak už roste pomalu

- 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.

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)

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 řadič 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

- jak dlouho bude trvat zápis (odhad “záseku”)
  - co když bude velká část zápisů náhodná
  - disková pole, SSD disky – můžete si dovolit vyšší limity
- dobré hodnoty (vyladěné pro I/O subsystém)
  - pokud máte řadič s write cache

    ```
    vm.dirty_background_bytes = (cache řadiče)
    vm.dirty_bytes = (8 x dirty_background_bytes)
    ```
  - pokud řadič 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)

vm / dirty memory

- alternativně lze tunit pomocí timeoutů
  - vm.dirty_expire_centisecs – jak dlouho mohou být data v cache před zápisem na disk (default: 30 vteřin)
  - vm.dirty_writeback_centisecs – jak často se pdflush probouzí aby data zapsal na disk (default: 5 vteřin)

io scheduler

- “který scheduler je nejlepší” je tradiční bullshit téma
  - ideální pro pěkné grafy na Phoronixu, v praxi víceméně k ničemu
  - minimální rozdíly, s výjimkou zkonstruovaných “corner case” případů
- říká se že ...
  - **noop** - dobrá volba pro in-memory disky (ramdisk) a tam kde si to zařízení stejně chytře přeorganizuje (nerotační média aka SSDs, RAID pole s řadičem a BBWC)
  - **deadline** - lehký jednoduchý scheduler snažící se o rovnoměrné latence
  - **anticipatory** - koncepčně podobný deadline, se složitější heuristikou která často zlepšuje výkon (a někdy naopak zhoršuje)
  - **cfq** – se snaží o férové rozdělení I/O kapacity v rámci systému
- ale **cfq** je “default” scheduler, a např. “ionice” funguje jenom v něm

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)
- `archive`
  - WAL archivace (log-file shipping replikace, warm_standby)
- `hot_standby`
  - read-only standby
- `logical`
  - možnost logické replikace (interpretuje WAL log)

wal_level

- `minimal`
  - může ušetřit zápis WAL pro “ignorované” operace
  - vesměs málo časté maintenance operace (CREATE INDEX, CLUSTER)
  - nebo nepříliš používané operace (CREATE TABLE + COPY, CTAS)
  - irelevantní pokud potřebujete lepší recovery
  - možná pro iniciální load dat
- `archive / hot_standby / logical`
  - minimální rozdíly mezi volbami (objem, CPU overhead, ...)

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

wal_buffers

- od 9.1 víceméně obsolete
  - nastavuje se automaticky na 1/32 shared_buffers
  - maximálně 16MB (manuálně lze nastavit víc)
  - není důvod na to víc šahat
- na starších verzích používejte stejnou logiku
  - těžko vymyslíte něco lepšího

checkpoint_segments

- COMMIT
  - zápis do transakčního logu (WAL) + fsync
  - sekvenční povaha zápisů
  - úprava dat v shared bufferech (bez zápisu na disk)
- WAL
  - rozdělený na 16MB segmenty
  - omezený počet, recyklace
- CHECKPOINT
  - zápis změn ze shared buffers do datových souborů
  - náhodná povaha zápisů
  - nutný při recyklaci nebo timeoutu (checkpoint_timeout)

checkpoint_segments

- checkpoint_segments = 3 (default)
  - 48MB změn, v praxi velmi nízká hodnota
  - konzervativní (minimální nároky na diskový prostor)
- praktičtější hodnoty
  - 32 (512MB), 64 (1GB), ...
  - souvisí s velikostí write cache na řadiči
- Pozor!
  - čím vyšší hodnota, tím déle může trvat recovery
  - musí se přehrát všechny změny od posledního checkpointu

checkpoint_timeout apod.

- checkpoint_timeout
  - maximální vzdálenost mezi checkpointy
  - default 5 minut (dost agresivní), maximum 1 hodina
  - v postatě horní limit na recovery time
  - recovery je většinou rychlejší (jenom samotné zápisy)
- checkpoint_completion_target
  - až do 8.2 problém s I/O při checkpointu (write + fsync)
  - completion_target rozkládá zápisy a fsync v čase
  - další checkpoint skončil než začne následující (0.5 – 50%)
  - funguje s “timed” i “xlog” checkpointy

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

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

commit_delay / siblings

- způsob jak “obejít” IOPS limit disků
  - 7.2k disk má limit ~100 IOPS, 15k ~250 IOPS, ...
  - víceméně limit pro “fsync rate” (modulo TCQ/NCQ apod.)
- co kdybychom transakce necommitovali po jedné?
  - chvilku počkáme a “nahromaděné” commitneme najednou
- potřebujeme aby
  - běželo několik transakcí
  - skončily skoro ve stejnou chvíli
- parametry
  - commit_siblings – počet transakcí které musí běžet
  - commit_delay – jak dlouho čekat (1/IOPS)

commit_delay / siblings

- funguje jenom pro krátké transakce (řádově 1/IOPS)
- obsolete od 9.2 – implementován “group commit”
  - lepší (chytřejší, automaticky, ...)
- příklad v praxi
  - http://www.linux.cz/pipermail/linux/2011-December/270072.html
  - dSpam ukládající data do PostgreSQL
  - několik INSERT/UPDATE procesů + autocommit

	medián (s)	pct90 (s)
commit_delay=0	7.66	25.32
commit_delay=1500	2	4.86
synchronous_commit=off	0.42	0.56

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.

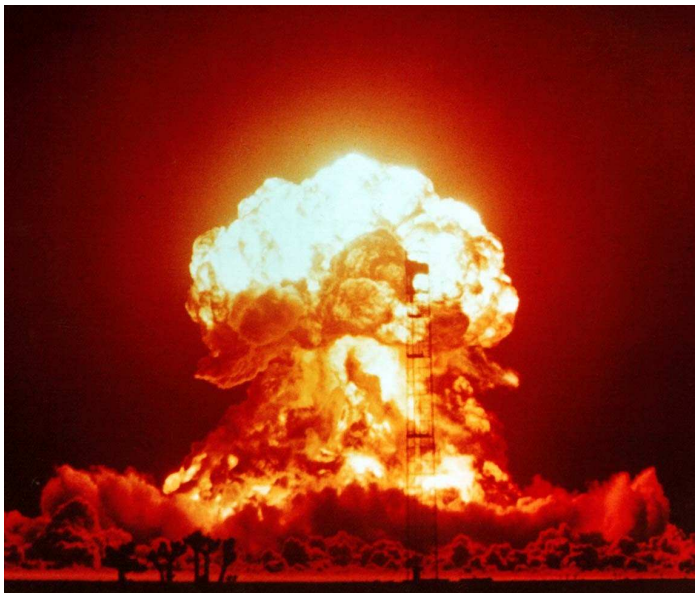
- **vacuum_cost_page_miss** (10)

  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 = 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

```
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

- `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)

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

logging & monitoring

- důležité volby
  - `log_line_prefix (string)`
  - `log_min_duration_statement (integer)`
  - `log_checkpoints (boolean)`
  - `log_temp_files (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.

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, ...

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