



Konfigurace PostgreSQL a základy "tuningu"

Prague PostgreSQL Developer Day 2017 / 15.2.2017

Tomáš Vondra, 2ndQuadrant

tomas.vondra@2ndquadrant.com

© 2017 Tomas Vondra, under Creative Commons Attribution-ShareAlike 3.0

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

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
- wal_min_size / wal_max_size
- checkpoint_timeout /
checkpoint_completion_target
- default_statistics_target
- autovacuum options

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

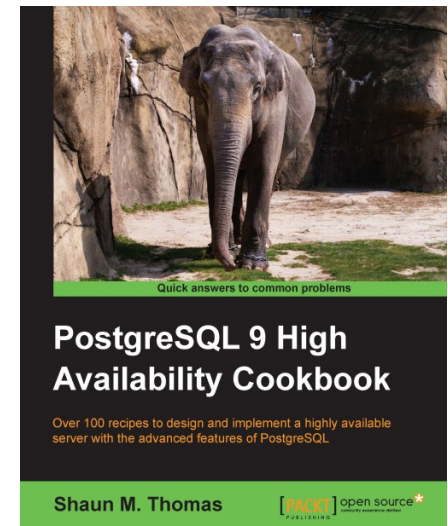
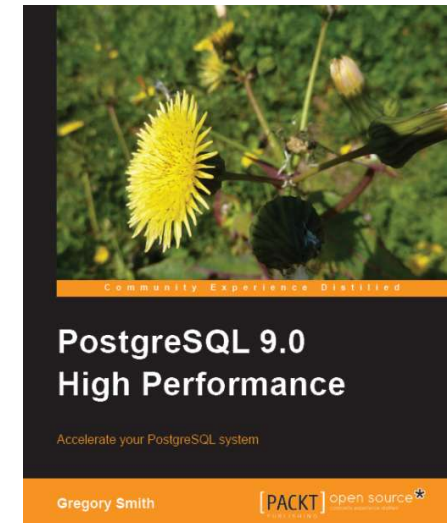
- Ulrich Drepper, Red Hat
- <http://www.akkadia.org/drepper/cpumemory.pdf>
- hutné low-level pojednání o CPU a RAM

Righting Your Writes (Greg Smith)

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

PostgreSQL Wiki

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





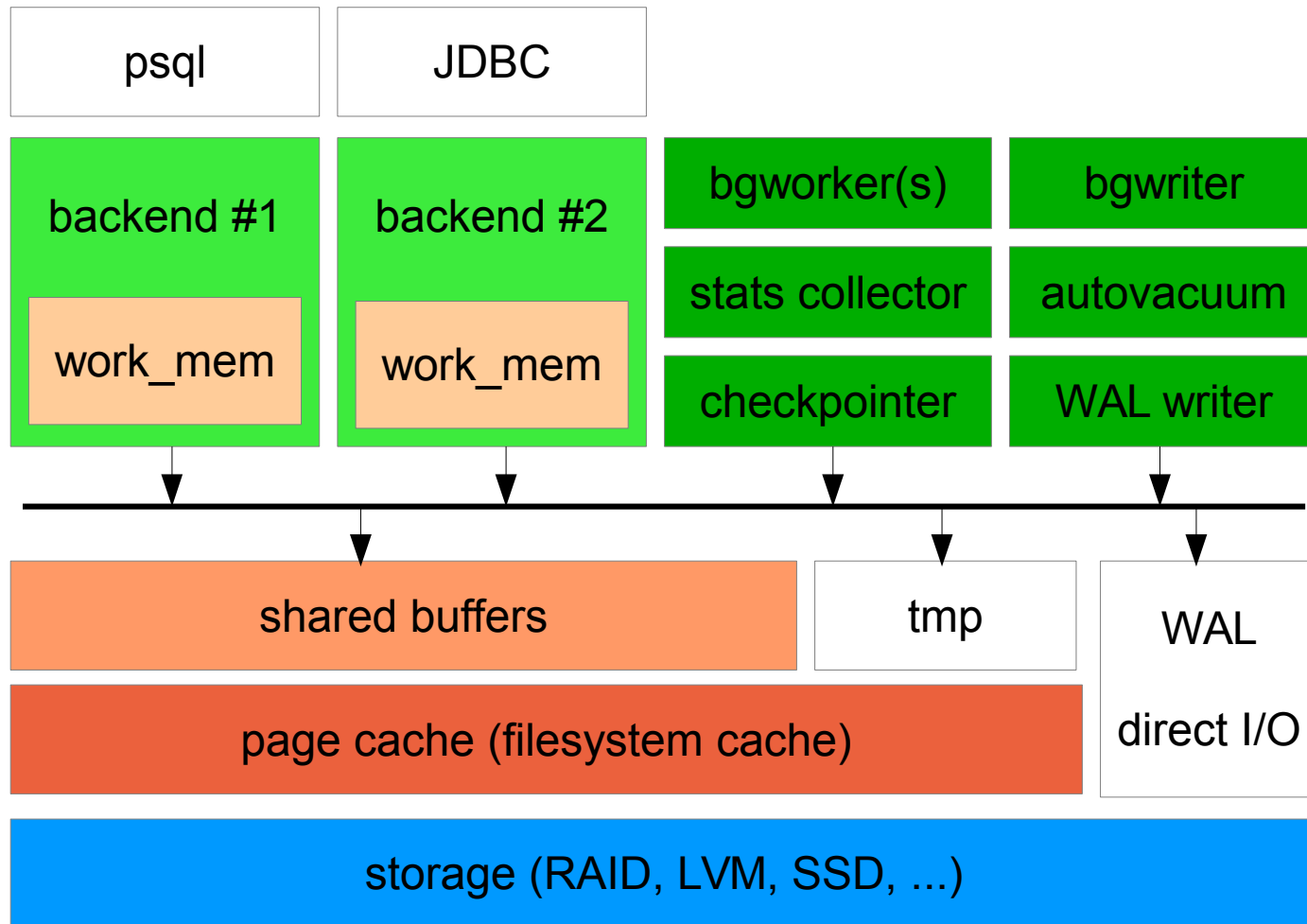
#1

Každý systém má bottleneck.

#2

Někdy je nejlepší tuning upgrade.

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 (init)

```
$ createdb test
$ time pgbench -h localhost -i -s 1000 test
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
NOTICE: table "pgbench_accounts" does not exist, skipping
NOTICE: table "pgbench_branches" does not exist, skipping
creating tables...
100000 of 100000000 tuples (0%) done (elapsed 0.02 s, remaining 16.21 s).
200000 of 100000000 tuples (0%) done (elapsed 0.03 s, remaining 16.35 s).
...
99900000 of 100000000 tuples (99%) done (elapsed 99.26 s, remaining 0.10 s).
10000000 of 100000000 tuples (100%) done (elapsed 99.41 s, remaining 0.00 s).
vacuum...
set primary keys...
done.

real      6m43.962s
user       0m18.254s
sys        0m0.767s
```



pgbench (test)

```
$ pgbench -j 4 -c 4 -T 60 test
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 1000
query mode: simple
number of clients: 4
number of threads: 4
duration: 600 s
number of transactions actually processed: 702704
tps = 1171.160579 (including connections establishing)
tps = 1171.163316 (excluding connections establishing)
```



pgbench (read-only test)

```
$ pgbench -s -j 4 -c 4 -T 600 test
starting vacuum...end.
transaction type: SELECT only
scaling factor: 1000
query mode: simple
number of clients: 4
number of threads: 4
duration: 600 s
number of transactions actually processed: 2136869
tps = 3561.435957 (including connections establishing)
tps = 3561.445152 (excluding connections establishing)
```

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”)

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

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;
```


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, nenačítané na velikost shared buffers

zone_reclaim_mode

- NUMA (Non-Uniform Memory Access)
 - architektura na strojích s mnoha CPU / velkými objemy RAM
 - RAM rozdělená na části, každá připojená ke konkrétnímu CPU
 - přístupy k RAM u jiného CPU jsou dražší
 - `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>

work_mem

- limit paměti pro jednotlivé operace
 - default 4MB (velmi konzervativní)
 - jedna query může použít několik operací
 - 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)
 - Hash Aggregate je výjimka (pouze během plánování)
- optimální hodnota závisí na
 - dostupné paměti, počtu paralelních dotazů
 - složitosti dotazů (OLTP vs. OLAP)
 - menší hodnoty mohou být lepší (CPU cache)

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, dotazy vytvářející temporary files
 - 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 pro danou session

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

- lze nastavit per uživatele

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

- nebo pro 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
- 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

effective_cache_size

- default 4GB, ale neovlivňuje přímo žádnou alokaci
- slouží čistě jako “náповěda” při plánování dotazů
 - Jak pravděpodobné je že blok “X” nebude číst z disku?
 - Jakou část bloků budu muset číst z disku?
- vyšší hodnoty -> náhodný přístup levnější
- defenzivní hodnota
 - (shared buffers + page cache) / max_connections
- v praxi se data často sdílejí mezi backendy (stejná data)
- page cache je odhad
 - zbývající RAM bez paměti pro kernel, work_mem, ...
- většinou nemá cenu to nějak detailněji ladit

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
```



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.

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

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`)

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
 - 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

min / max_wal_size

- 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)

min / max_wal_size

- `max_wal_size = 1GB (default)`
 - pokrývá cca 3 checkpointy (2 + `completion_target`)
 - to znamená cca 340MB / checkpoint
 - konzervativní (minimální nároky na diskový prostor)
- praktičtější hodnoty
 - pro větší systémy se často používají desítky GB
 - 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 (orientační)
 - recovery je většinou rychlejší (jenom samotné zápisy)
 - ale ne nutně (single threaded)
- `checkpoint_completion_target`
 - až do 8.2 problém s I/O při checkpointu (write + fsync)
 - `completion_target` rozkládá zápisy v čase
 - další checkpoint skončil než začne následující (0.5 - 50%)
 - funguje s “timed” i “xlog” checkpointy

checkpoint tuning

- většina checkpointů by měla být “timed” a ne příliš často
- základní metodika
 1. zvolte si “rozumnou” vzdálenost checkpointů (např. 30 minut)
 2. odhadněte kolik potřebujete WAL na checkpoint
např. s `max_wal_size=1GB` jsou checkpointy každých 10 minut => 3GB
 3. nastavte `max_wal_size` na 3x odhadu (9GB)
 4. nastavte vhodný `completion_target` (tak aby OS zbylo dost času)

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

* `_flush_after`

- rychlá akumulace velkého objemu dirty dat v page cache
 - OS se rozhodne zapsat naráz - latency spike
- řešení v rámci DB - donutit OS zapsat data na disk
 - `wal_writer_flush_after = 1MB`
 - `checkpoint_flush_after = 0 (Linux: 256kB)`
 - `bgwriter_flush_after = 0 (Linux: 512kB)`
 - `backend_flush_after = 0`
- pokud se data vejdou do `shared_buffers` - OK
- pokud ne, může generovat random I/O

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 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)

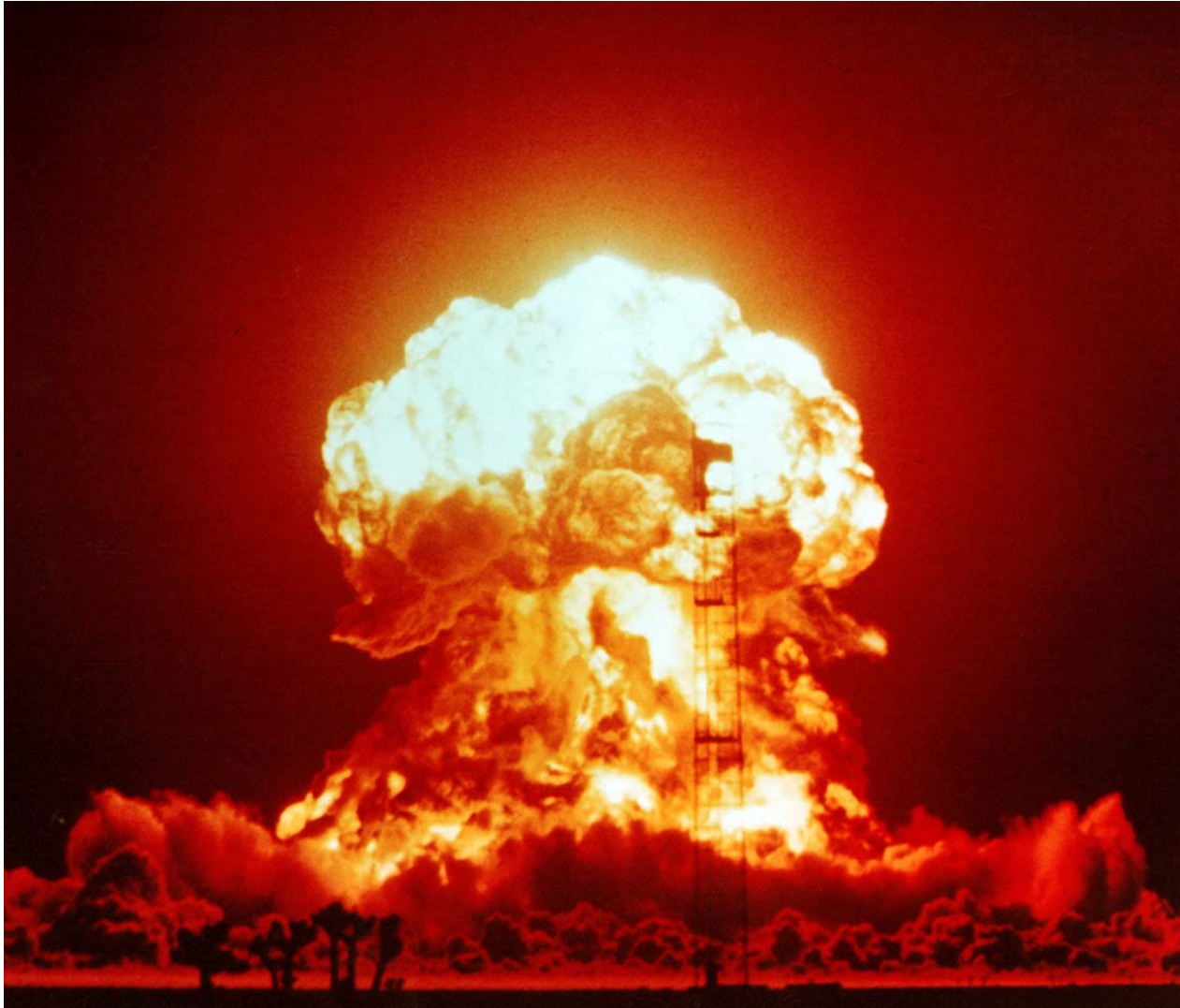
io scheduler

- “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

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

`změněných_řádek > (threshold + celkem_řádek * scale_factor)`

autovacuum thresholds

- `autovacuum_vacuum_cost_delay = 20ms`
- `autovacuum_vacuum_cost_limit = -1 (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

logging & monitoring

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

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
 - pomalá změna / náhlá změna?
- postupujte systematicky “seshora”, zbytečně neskákejte
 - většinou víte co v aplikaci je pomalejší
 - drill down, Occamova břitva

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>

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>

Load data (iniciální)

- vypněte AUTOCOMMIT (BEGIN/END)
- použijte COPY namísto INSERT
- odstraňte INDEXy a FK (cizí klíče)
- zvyšte maintenance_work_mem a checkpoint_segments
- vypněte WAL archivaci a streaming replikaci
 - wal_level=minimal, archive_mode=off,
max_wal_senders=0
- po loadu proveďte ANALYZE
- pokud používáte pg_dump, zkuste použít paralelní mód
 - pg_dump -j N ... / pg_restore -j N ...

<http://www.postgresql.org/docs/9.4/static/populate.html>

Disk layout

- WAL
 - sekvenční zápisy
 - téměř se nečte (s výjimkou recovery)
- data files
 - častý náhodný přístup (zápis i čtení)
 - ...