# Administrace, Monitoring

Pavel Stěhule

P2D2, Praha 2016

# Úkoly SQL databáze

- **Optimalizace a vyhodnocení dotazů - programátor nemusí řešit způsob přístupu k datům: index scan, seq scan, nested loop, merge join, hash join, ...**
- Zajištění izolace transakcí a konzistence dat - nikdo nevidí nepotrvzené změny ostatních uživatelů, všechna data jsou konzistentní vůči určitému času
- Nástroje a ochrana proti race-conditions (zámky, izolace transakcí)
- Efektivní práce s cache - intenzivní používání read/write cache v multiuživatelském režimu s ochranou proti havárii (transakční log).

# Faktory ovlivňující výkon aplikace

- **Architektura aplikace**
  - umožnit sharding a efektivní použití cache
- **Design aplikace**
  - možnost psát rychlé dotazy
- **Fyzické parametry serveru RAM, IO**
- *Tvar SQL příkazů a podmínek* (pokud nejsou extrémně špatné, tak nehraje příliš velkou roli)
  - optimizer friendly queries
  - index friendly predicates
- *Konfigurace serveru* (pokud není extrémně špatná, tak nehraje příliš velkou roli), **neexistuje TURBO konfigurace**.
  - konfigurace work_mem

# Základní pravidla

- Manipulace s daty (zápis a čtení) je přirozeně úzké hrdlo aplikace a **nelze jej odstranit**. Často lze odstranit zbytečné a neefektivní operace.
- Relační databáze není magická černá skříňka, která opraví “zrychlí” neefektivně napsané aplikace. Každá optimalizace má své limity, optimalizace jedné vlastnosti může vést k degradaci jiné. Cílem je dobrý kompromis.
- To, že databáze není magická černá skříňka, která umí vyřešit všechny problému, ještě neznamená, že se databáze nedají (nemají) používat. S většími daty a větší zátěží je ale dobré o databázích něco vědět.

# Stávající situace

- chybějící monitoring
- chybějící znalost limitů aplikace
- chybějící znalost chování aplikace
  - Kde jsou úzká hrdla?
  - Co je jejich příčinnou?
- chybí předcházení problémům s výkonem
  - čekáme na problém

# Konfliktní situace

- S aplikací pracují dva* nezávislé týmy
  - Vývojáři - za problémy mohou administrátoři, kteří nejsou schopni správně nakonfigurovat a provozovat databázi.
  - Administrátoři - za problémy mohou vývojáři, kteří píší špatné aplikace.
- **Administrátoři už toho moc nezachrání**. Ale mají a musí poskytovat kvalitní zpětnou vazbu. Vývojář často nemá a nesmí mít přístup k produkci.

* + další konfliktní vazby (management, obchodníci)

# Pokus o řešení problémů

- Hledání zázračné konfigurace
  - turbo = on ??
- Přidávání indexů na poslední chvíli
  - ale bez schopnosti verifikace v prováděcím plánu
- Snaha o řešení problémů brutální silou
  (load balancing, multimaster cluster)

# Ideální situace

- Znalost limitů aplikace, hw
- Funkční monitoring
  - schopnost včas identifikovat regresi a rozpoznat její důvod
- Základní znalost databázových technologií
  - schopnost interpretovat stávající chování databáze
    - čtení prováděcích plánů, detekce zámků

# Rozumná výchozí konfigurace

- work_mem, maintenance_work_mem
- shared_buffers
- effective_cache_size
- max_connection

**WM*MC*2 + SB + FS + OS < RAM**

# Databáze není lineární systém

- se zvyšující zátěží výkon databáze (schopnost zpracovávat dotazy) může klesat výrazně dramatičtěji než lineárně
  - nedostatek RAM zvyšuje zátěž IO, nižší efektivita cache
  - při větším počtu IO operací klesá rychlost čtení/zápisu
  - při pomalejších operacích může docházet k častějšímu souběhu operací (čekání na zámky)
  - některé operace mohou mít n^2 náročnost

# Silná negativní zpětná vazba

- Několik pomalých dotazů může výkon silně degradovat
  - generují více požadavků na IO, které je pak pomalejší
  - snižují efektivitu cache
- Odstraněním několika pomalých dotazů můžeme znatelného zlepšení výkonu (pravidlo 80/20)

# Spojování tabulek

- Nested loop
  - rychlé pro menší počet řádků
  - vyžaduje index - množství random IO operací
- Merge join
  - efektivní pro větší počet řádků
  - vyžaduje seřazená data (index - random IO
    nebo external sort - zápisy čtení IO)
- Hash join
  - významná část menší tabulky se musí vejít do RAM (*work_mem*)
  - nevyžaduje index, pouze sekvenční čtení

# Chování

- přidání řádku tabulky může změnit prováděcí plán a tudíž i charakter využití zdrojů (IO, RAM)
- prováděcí plán je statický, generovaný na základě statistik před zpracováním dotazu - pracuje se s stochastickým modelem - navíc statistiky nemusí být aktuální
- změna velikosti dat musí vyvolat změny prováděcích plánů (pro netriviální dotazy neexistuje věčný ideální plán)
- vyjma vyjímek to překvapivě funguje

# NESTED LOOP

```
postgres=# EXPLAIN SELECT * FROM obce o JOIN okresy k ON o.okres_id = k.id
postgres-# WHERE k.nazev = 'Benešov';
                                    QUERY PLAN
═══════════════════════════════════════════════════════════════════════════════
 Nested Loop  (cost=0.28..12.56 rows=81 width=58)
   ->  Seq Scan on okresy k  (cost=0.00..1.96 rows=1 width=17)
         Filter: (nazev = 'Benešov'::text)
   ->  Index Scan using obce_okres_id_idx on obce o  (cost=0.28..9.79 rows=81 width=41)
         Index Cond: ((okres_id)::text = k.id)
(5 rows)

Time: 2.109 ms
```

# Penalizace nested loop(u)

```
postgres=# SET enable_nestloop TO off;
SET
Time: 0.775 ms
postgres=# EXPLAIN  SELECT * FROM obce o JOIN okresy k ON o.okres_id = k.id
WHERE k.nazev = 'Benešov';
                                QUERY PLAN
═══════════════════════════════════════════════════════════════════════════
 Hash Join  (cost=1.97..147.72 rows=81 width=58)
   Hash Cond: ((o.okres_id)::text = k.id)
   ->  Seq Scan on obce o  (cost=0.00..121.50 rows=6250 width=41)
   ->  Hash  (cost=1.96..1.96 rows=1 width=17)
         ->  Seq Scan on okresy k  (cost=0.00..1.96 rows=1 width=17)
               Filter: (nazev = 'Benešov'::text)
(6 rows)
```

# HASH JOIN

```
postgres=# EXPLAIN SELECT * FROM obce o JOIN okresy k ON o.okres_id = k.id;
                                QUERY PLAN
═══════════════════════════════════════════════════════════════════════════
 Hash Join  (cost=2.73..210.17 rows=6250 width=58)
   Hash Cond: ((o.okres_id)::text = k.id)
   ->  Seq Scan on obce o  (cost=0.00..121.50 rows=6250 width=41)
   ->  Hash  (cost=1.77..1.77 rows=77 width=17)
         ->  Seq Scan on okresy k  (cost=0.00..1.77 rows=77 width=17)
(5 rows)
```

# Pozor

- **To, že něco funguje efektivně může být náhoda!**
  - při testování, při vývoji - může svádět k falešným doměnkám
- CTE může být efektivnější (blokuje optimalizaci)
- CTE může být horší (blokuje optimalizaci)
- Záleží jak věrohodné jsou odhady - liší se pro jednotlivé hledané hodnoty
- Záleží jak rychle dokážeme získat data - v cache (10GB/s), načtení z málo zatíženého IO (600MB/s), načtení ze zatíženého IO (60MB/s)

# Monitorování regresí IO

- Utilizace IO - nedostatek RAM se projeví ve vyšším zatížení IO (cache, tmp files)
- Utilizace IO - pomalejší zápis transakčních logů a checkpointů - pomalejší DML
- Velikost a počet dočasných souborů generovaných Postgresem - při nedostatku RAM Pg generuje tmp files (viz *pg_stat_database*)
- Důležitá 20-30% rezerva v dlouhodobém pohledu
- RAM používá se jako cache, Pg nemusí používat dočasné soubory, Indexy - lze snížit objem čtených dat z disku (potenciálně lepší efektivita cache)

# Monitorování pomalých dotazů

- Pomalé dotazy dlouhodobě čerpají zdroje
  - statement_timeout
  - SELECT pg_cancel_backend(pid)
    FROM pg_stat_activity
    WHERE
      current_timestamp - query_start > interval “10min”
- **Pozor na nechtěné pomalé dotazy** - kartézský součin (fatální), nebo dobíhající pomalé dotazy - klient již neexistuje, dotaz v db stále běží (www aplikace)
- Alert na dotazy delší než 10 min, 1h, 24h
  - většinou neočekávané - lépe včas zastavit než riskovat přetížení serveru
- Nástroje - *log_min_duration_statement*, pgFouine, pg_stat_statements, auto_explain
- Dotazy pod 50ms nemá cenu optimalizovat - větší smysl má použití aplikačních cache

# Index friendly predicates

- WHERE

  EXTRACT(month FROM inserted) = 12

  AND EXTRACT(year FROM inserted) = 2016

- **WHERE**

  **inserted >= '2016-12-01'**

  **AND inserted < '2017-01-01'**

# Monitorování spojení

- Možnost identifikace leakování spojení
- Možnost identifikace kritického problému - uváznutí spojení ve stavu “**IDLE IN TRANSACTION**”
  - blokuje poolování
  - blokuje VACUUM
- Optimum cca 10x CPU cores
- Postgres nemá timeout na “idle in transaction”
  - pgbouncer, cron

# Monitorování regresí CPU

- Při dostatku RAM se stává uzkým hrdlem CPU
- Vysoká utilizace CPU - neefektivně prováděné dotazy, OLAP dotazy
- Vysoká utilizace CPU - potenciálně problém se spinlocky - typické pro neadekvátně velký počet připojení do databáze: perf top (identifikace), pgbouner (poolování spojení). Při N*100 (N > 2) spojení do DB i idle spojení může být drahé (dramatické snížení výkonu).

# Monitorování počtu transakcí

- Regrese:
  - efekt optimalizace (odstranění zbytečných dotazů, cache)
  - Také ale nedostupnost aplikace pro zákazníka (firewall, ...)

# Monitoring autovacuum

- Při velkém počtu tabulek nemusí být 3 výchozí procesy dostačující
  - je nutné zvýšit počet procesů
- Při větší velikosti tabulek může dojít k stornování (cancel) autovacuua z důvodů požadavků na držené zámky (vacuum nikdy nedoběhne)
  - je nutné zvýšit intenzitu VACUUM, zkrátit sleep time

# Monitorování zámků

- Zámky jsou ochranou před race conditions
- “Zpomalují” provádění operací
- Pro začátek postačí log_lock_waits a lock_timeout
- Řešení:
  - zrychlit operace
  - reorganizovat schéma (tabulky, které drží důležité PK, co nejméně aktualizovat)
  - přejít na pesimistické zamykání

# Benchmarking

- Verifikace hw
- Identifikace limitů
- Pozor na falešné alarmy
  - test na slabším stroji
- Pozor na falešný pocit bezpečí
  - test na silnějším nebo nezatíženém stroji
- **Čím dříve, tím lépe**
  - Poslední víkend před startem produkce bývá již pozdě

# Dotazy?