PostgreSQL
návrh a implementace
náročných databázových aplikací

Pavel Stěhule

Výkon libovolné aplikace spravující větší množství dat je determinován primárně návrhem aplikace a strukturou spravovaných dat. Obé musí být ve souladu s vybranou databázovou technologií.

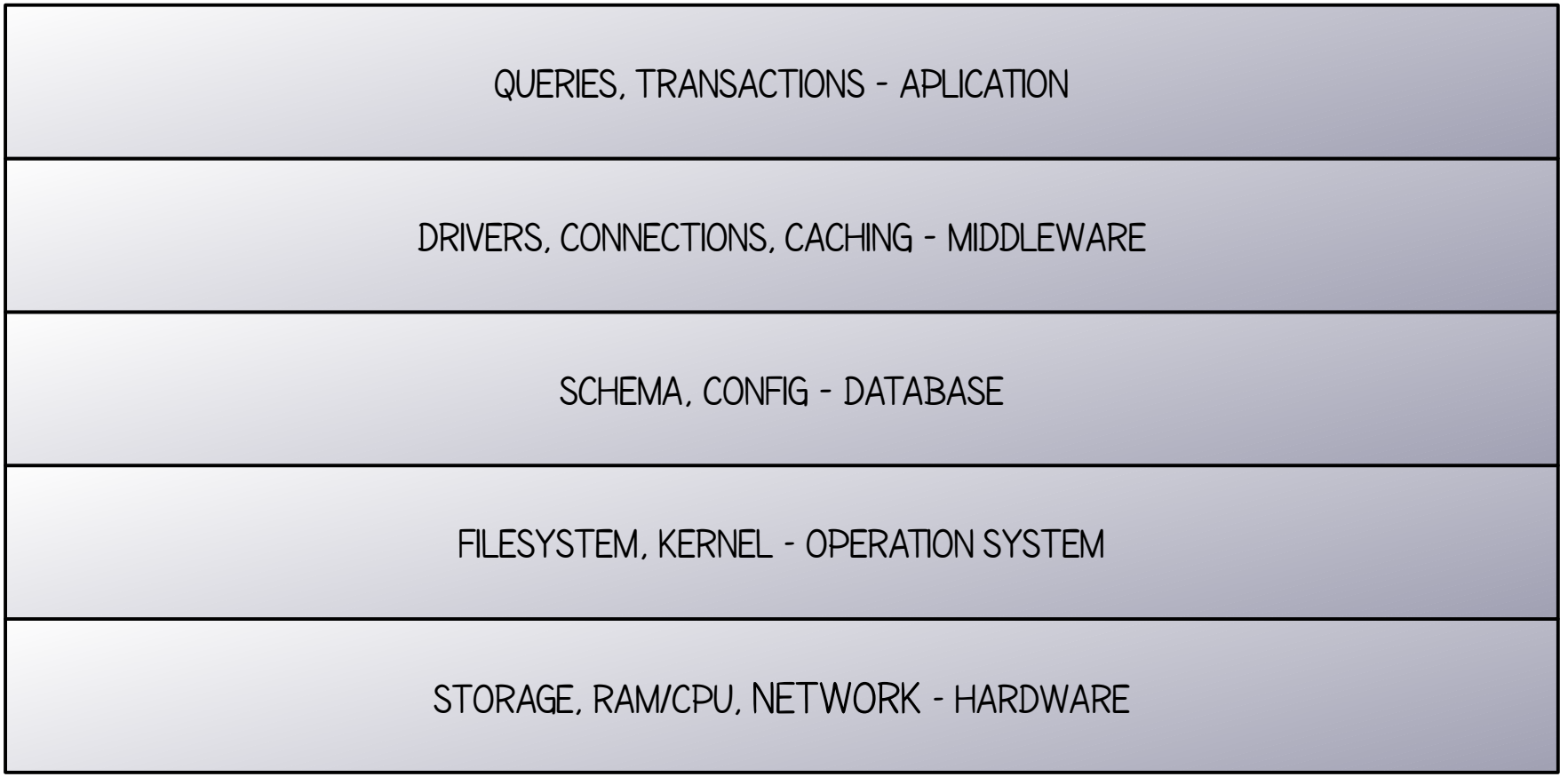
Pavel Stěhule

- Major contributor PostgreSQL
- Zakládající člen CSPUGu
- DBA v GoodData – PostgreSQL + BI + cloud
- Popularizace SQL a PostgreSQL v ČR
- Konzultace a školení PostgreSQL a SQL
- Výkonnostní audit nasazení PostgreSQL
- Poradenství při problémech s výkonem PostgreSQL

Základní faktory ovlivňující výkon databáze

- HW (CPU, RAM, IO, NET)
- Architektura databáze (OLTP, OLAP)
- Architektura aplikace (OLTP, OLAP)
- Funkční požadavky (ACID, CAP, ...)
- Design databáze (existence indexů, použití EAV, ..)
- Aktuální stav databáze (stav indexů, uspořádání tabulek)
- Schopnost optimalizátoru generovat optimální prováděcí plány
- Design UI aplikace (skrolování, vyhledávání)
- Použití cache (aplikační, mater. pohledy, memcache)
- Konfigurace databáze
- Konfigurace OS (konfigurace virtualizace)

Aplikační vrstvy



Rady

- U většiny problémů s výkonem, které se svalují na databáze se ukáže, že se nejedná o problém databáze.
- Méně než 10% dotazů způsobuje 90% zátěže databáze.
- V libovolném čase je možné správně identifikovat a odstranit pouze jeden výkonnostní problém, jedno úzké hrdlo, jeden nejpomalejší SQL příkaz.

Etapy zpracování dotazu

- Příprava prováděcího plánu
  - parsování
  - plánování – na tvrdo specifikuje metodu JOINů (*hashjoin*, *merge join*, *nested loop*), pořadí
- Exekuce prováděcího plánu
  - většina parametrů a metod je zafixována předem (na základě odhadů) při plánování
  - *quick sort* / *external sort* dynamicky
- Mezi optimalizací a exekucí může být výrazná časová prodleva během které může dojít ke změně obsahu tabulek

CPU

- OLTP – max connection ~ 10 * CPU
- OLAP – max connection ~ 1 * CPU
- PostgreSQL je primárně OLTP databáze
  - bez dalšího sw používá pouze jedno CPU jádro pro zpracování SQL příkazů (předpoklad – hrdlem je IO)
  - komerční fork Greenplum – vestavěné MPP
  - PL/Proxy – podpora shardingu/horizontal partit.
  - GridSQL (dnes Stado) – massively parallel processing (MPP) architecture aplikace používající db backend PostgreSQL
- Plánování složitých dotazů – použití GEQE
  - zpracování velkého množství jednoduchých příkazů (ORM)
  - Pozor na partitioning – partition ~ tabulka

RAM

- Zpracování paměťově náročných úloh
  - sort, hash join (*work_mem*) ~ 10MB
  - create index (*maintenance_work_mem*) ~ 200MB
  - external sort, merge join ~ 10x pomalejší
- Cache
  - cache datových stránek (*shadow_buffers*) ~ 2 .. 20GB
  - cache celkem hint (*effective_cache_size*) ~ 2/3RAM
- Použití jako pracovní paměti při optimalizaci dotazu (zřetelné pouze při masivním použití partitioningu (používat max. 100 partitions))
- Databázový server nesmí aktivně používat SWAP
- **Výchozí konfigurace v PostgreSQL je zbytečně skromná. Pozor na windows!**

I/O

- Rychlost čtení/zápisu – RAID, SSD
  - cena/IOPS nebo cena/kapacita
  - rychlost zápisu/čtení – sekvenční čtení/náhodný přístup
  - *random_page_cost* a *seq_page_cost*
- Čtení a zápis dat stránek (bgwriter, checkpoint)
  - *fsync* transakčního logu při commitu
- Zápis do transakčního logu (commit)
  - *synchronous_commit*
- Čtení a zápis dočasných souborů (external sort)
- Zálohování
  - *throttling* např. zapnutím komprimace
  - kontinuální zálohování, zálohování slavea
- Logování

NETWORK

- Navazování spojení (ještě výrazně pomalejší je navazování SSL)
  - šetrnější poolování spojení (keep-alive)
- Přenos SQL příkazů
  - minimalizují uložené procedury
  - pozor na ORM – pro klasickou SQL db jsou výhodnější hromadné operace
- Přenos dat – latence TCP/IP, latence klienta
  - spočítaný záznam se okamžitě posílá na klienta (libpq ukládá result na klientské straně)
    - řešením problémů se zahlcením klienta může být použití explicitního kurzoru (příkaz *FETCH*) – případně opět uložené procedury (v pg inprocess)
  - pokud možno vše spočítat na serveru – klient dostává pouze výsledek
- Synchronizace klient/server

Testování hardware
výkon, spolehlivost

- IO *Bonnie++*, *Hdparm* čtení/zápis, random IO
- MEM *Cachebench*, *Memtest86* čtení/zápis
- NET *Netperf*, *ping*, *Ttcp*
- Aplikační TPC-B (OLTP) *pgbench*

Počáteční konfigurace PostgreSQL

- work_mem 10..100MB
- maintenance_work_mem 100..500MB
- shadow_buffers 2..20GB
- max_connection 10 * CPU
- effective_cache_size
- random_page_cost ↓ ~ dostatek RAM
- **SB + WorkMem * 2 * MaxCn + FileSys + OSys ≤ RAM**

Identifikace hrdel

- IO/CPU ~ *top*, *iotop*
  - Analýza logu pomalých dotazů
    - Nízká hodnota *work_mem*
      - Nepoužívá se quck sort, používá se external sort
      - Nepoužívá se hashjoin, hashagg, použije se external sort
    - Nízká hodnota *shared_buffers*
      - Data se neudrží dostatečně dlouho v cache - *pg_buffercache*
    - Chybějící indexy
    - Nevhodně napsané nebo chybné dotazy
- Zámky
  - *log_lock_waits*

Cache

- Cílem je neopakovat 2x tentýž výpočet
- Nasazení cache v závislosti na počtu uživatelů
  - vnitropodniková aplikace – možná zbytečné
  - silně navštěvovaná www aplikace – **nutnost**
- Cache
  - průběžně udržovaná (náročnější)
  - periodicky udržovaná (jednoduchá na implementaci)
- PostgreSQL „nerado“ častý update jednoho a téhož záznamu (řešením je cache a do Pg se ukládají „finální“ data) z důvodu implemetace *MVCC*.
- Některá data jsou z povahy netransakční – a proto nemají, co dělat v databázi. ACID představuje zbytečnou režii. Řešením je použití např. *Memcached* případně *UNLOGGED* tables v Postgresu (úspora zápisu do logu, stále režie *MVCC*) pokud není zbytí.

Materializované pohledy

- Představují cache na databázové úrovni
  - vestavěná podpora
  - aplikační (vlastní implementace)
  - průběžně udržované / periodicky udržované
- Při agregacích velkých dat **PostgreSQL nevyužívá umělou inteligenci** a pokaždé opakuje stejný výpočet.
  - Jednoduchá optimalizace – materializace výsledku nad archivními daty plus opakované přičtení výsledku nad denním nebo týdenním přírůstku.

Indexy v PostgreSQL

- Uspořádáné dvojice klíč/adresa
- Filtrování, agregace, JOIN (*mergejoin*), řazení
  - Nečte se balast
  - Neprovádí se masivní external sort
  - **Používá se random IO**,
    - Odkazovaná data mohou být v nehodném pořadí
      - **Přeuspořádání haldy – příkaz *CLUSTER***
- Jednoduché, složené, podmíněné (částečné), funkcionální, vzestupně, sestupně řazené
- B-Tree, GiST, GIN,
- *pg_trgm* – agregace re, LIKE, ..
- Údržba indexu není zadarmo (+ REINDEX)
- *maintenance_work_mem*

Vyhledávání
index (planner) friendly predikáty

- ~~substring(d, 1, 4)::int = 2014 and substring(d, 6, 2)::int = 12~~
- ~~d::text LIKE '2014-12-%'~~
- EXTRACT (year FROM d) = 2014 and EXTRACT(month FROM d) = 12
- **d BETWEEN '2014-12-01'::date AND '2014-12-31'::date**
- t >= '2014-12-01'::timestamp AND t < '2015-01-01'::timestamp
- Vždy používat čisté predikáty tj *sloupec = konstanta*
- V pg lze použít funkcionální indexy
  - *fce(sloupec) = konstanta*
  - Chybí statistiky ~ použije se konstantní odhad 0.5%

Stránkování index friendly

- ***Relace není pole***, nelze jednoduše získat Ntý řádek
  - Záznamy v heapu jsou nejsou uspořádané (pořadí se neustále mění)
- Nepoužívat samotný `OFFSET n+1 LIMIT m`
- `WHERE pk > last_pk LIMIT m`
  - Stále vyžaduje dotaz na počet záznamu ve výběru
    - Limitovaný count ~ přesný do 10K řádků/více než 10K
    - Fake count
- Kontinuální zobrazení (facebook, linkedin, ..)
  - Ušetří 1x COUNT
  - Generuje index friendly dotazy
  - Odpadá mapování číslo stránky => pk

Životní cyklus databázové aplikace

- Vývoj [+ migrace (čištění) stávajících dat]
  - Testovací data/testovací provoz
  - Testovací databázi dimenzovaná jakoby po 5-7 let provozu
- Startup (první 3 – 21 dní provozu)
  - Kritické pro úspěch aplikace (i dodavatele)
  - Důležitá dostatečná rezerva, diagnostika a monitoring
- Provoz
  - Průběžná údržba – VACUUM, REINDEX
  - Archivace dat
  - Opakované doladění vycházející ze skutečné zátěže
    - Možné finální doladění 80/20 díky skutečným datům a skutečné zátěži
    - Změny v zátěži, změny v databázi – nárůst velikosti dat

Použití polí pro časové řady

- Každá verze záznamu v PostgreSQL obsahuje 27 bajtů v neviditelných (příp. nedostupných sloupcích: *xmin*, *xmax*, *cmax*, *ctid*, ..)
- Dlouhé úzké tabulky (typické pro časové řady v rel. db) jsou neekonomické (27/12)
  - Někdy nutné – vazební tabulky *m*:*n* – zde se nedoporučuje použití polí z důvodu optimalizace planneru.
- Řešením je zabalení hodnot do bloků (60, 1440, 3600, ..) s použitím polí.
  - Zanedbatelná režie
  - Transparentní komprimace

E-A-V tables ? hstore

- Struktura uložení dat v relační databázi
  - **Normalizovaná data**
  - Denormalizovaná data – široké tabulky
  - Entity - Class ~ Relation mapping
    - Komplikovaný přístup k entitám
      - Větší množství JOINů – dematerializace (náročnější údržba dat, duplicitní data)
  - Strukturovaná data
    - XML, JSON, *hstore*
  - Entity - Attribute - Value (Open Schema)
    - **Nepoužívat pro větší data a často využívané atributy**

hstore

- Polymorfní, stále kompaktní
- Podpora GiST, GIN indexů
- Podpora statistik
- Extenze z *contrib* balíku

```
'a=>1,b=>foo'::hstore -> 'a'
'a=>1,b=>foo'::hstore -> ARRAY['a','b']
'a=>1, b=>foo'::hstore ? 'a'
'a=>1, b=>foo'::hstore @> 'a=1'
%# 'a=>1, b=>foo'::hstore = {{a,1},{b,foo}}
hstore_to_json('a=>1, b=>foo')
CREATE INDEX ON table USING GIST (h)
```

Zamykání

- *MVCC*
  - *UPDATE neblokuje SELECT*
  - *UPDATE blokuje UPDATE*
- Skryté zamykání – implementace referenční integrity pomocí triggerů
  - `SELECT * FROM ft WHERE key=X FOR SHARE`
  - `SELECT * FROM ft WHERE key=X FOR KEY SHARE` (9.3+)
- *log_lock_waits*

Čtení prováděcích plánů

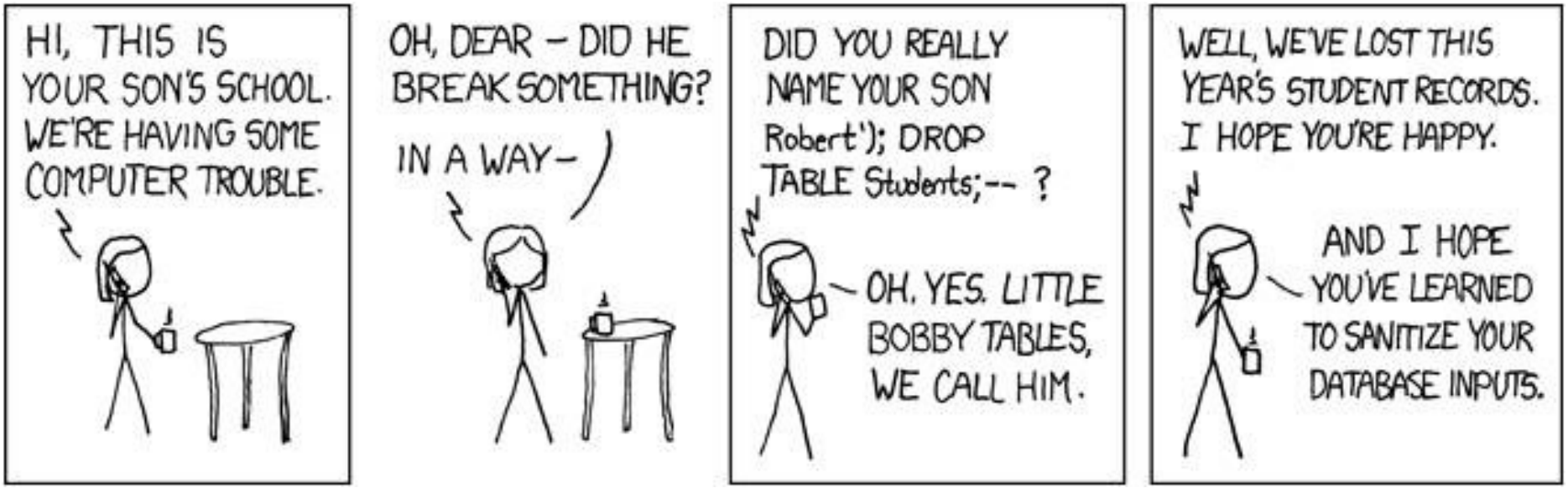
- EXPLAIN
- EXPLAIN (ANALYZE, TIMING OFF)

```
postgres=# EXPLAIN ANALYZE SELECT * FROM foo
                                        WHERE a BETWEEN 10 AND 10 LIMIT 10;
                                    QUERY PLAN
-------------------------------------------------------------------------------------------
 Limit  (cost=0.00..9.70 rows=10 width=8)
        (actual time=0.024..0.150 rows=10 loops=1)
   ->  Seq Scan on foo  (cost=0.00..194248.00 rows=200333 width=8)
                        (actual time=0.023..0.146 rows=10 loops=1)
         Filter: ((a >= 10) AND (a <= 10))
 Total runtime: 0.173 ms
(4 rows)
```

Možné problémy s předpřipravenými dotazy

- Prepared statements (*server side*)
  - Ochrana před *SQL injection*
  - Čitelnější zápis SQL parametrického SQL příkazu
  - Znovupoužití prováděcího plánu
    - Starší CPU
    - Intenzivně opakované příkazy – INSERT
    - Opakované komplexní dotazy (velký počet JOINů)
    - Uložené v paměti session
  - Optimalizované bez znalosti parametrů (slepá optimalizace)
    - Optimalizace na nejčastěji se vyskytující hodnotu
    - Optimalizace na první použitou hodnotu – použito v 9.3 (5 pokusů pro rozhodnutí o volbě optimalizace)

SQL injection!



Fast upload - COPY

- Etapy SQL příkazu
  - *PREPARE*, { *BIND vector (**row**)*, *EXEC* }, *COMMIT*
  - Pozor na *autocommit*!
  - *synchronous_commit off*– asynchronní fsync
- Etapy příkazu COPY
  - *PARSE*, *EXEC **stream***, *COMMIT*
  - COPY tablename FROM stdin CSV

Cost based optimizer

- Minimalizuje IO operace
  - Základem je odhad účinnosti filtrů a převod na cenu
- Volba metody přístupu k datům
  - *Seqscan*, *indexscan*, *seqscan + qsort*,
- Voba metody agregace
  - *Ordered*, *hashagg*
- Volba metody slučování
  - *Nestedloop*, *mergejoin*, *hashjoin*, ..

Chyby v odhadech

- Chybějící statistiky
  - Chybí podpora pro použitý datový typ
  - Odtržení od statistik
    - Použití výrazu v predikátu
    - Filtrování výsledku další operace s daty
- Zastaralé statistiky
- Nedostatečné statistiky ~ hrubý histogram
- Korelace v datech
- Výsledkem chyby může být nižší nebo vyšší odhad, u složitějších dotazů se chyby mohou násobit (ale také neutralizovat)

Řešení chyb v odhadu

- Zvýšení přesnosti histogramu
  - *default_statistic_target*
  - Vylepšuje odhady, zpomaluje plánování
- Zavření očí
  - Pokud je suboptimální plán dostatečně rychlý
- Rozbití dotazu + dočasné tabulky + ANALYZE
- Použití CTE
  - V PostgreSQL implementováno jako optimizer fence
    - Použíjí se fixní 0.5% odhady

Neoptimalizovatelné konstrukce

- Záměrně neoptimalizované CTE
- Nepodporované optimalizace – *push group by*
- Minimální rozdíly v semantice umožňují výraznou optimalizace – `NOT IN` / `NOT EXISTS`
- Pozor na pohledy
  - JOIN removal odstraní pouze nadbytečné slučované relace
  - UNION, korelované poddotazy zůstávají
- Optimalizátor se s každou verzí mění

Co monitorovat?

- Pomalé dotazy (50, 200, 5000ms)
  - analyzátory logu *pgBadger*, *pgFouine*
  - *log_min_duration_statement*
- CPU load, IO waits (*munin*, *top*)
- IO utilization (*munin*, *iotop*)
- Buffer hit
- Transaction per sec
- Jednorázově četnost všech typů dotazů
- Počet otevřených spojení
  - Pozor na aplikace neuzavírající spojení (connection leaks)
  - Pozor na *<IDLE> in transaction* (v PostgreSQL)
- Dostatek místa na disku!

Co monitorovat?

```
SELECT last_vacuum, last_autovacuum,
       last_analyze, last_autoanalyze
  FROM pg_stat_user_tables;

SELECT blks_read, blks_hit,
       xact_commit, xact_rollback,
       deadlocks, temp_bytes
  FROM pg_stat_database;

SELECT * FROM pg_stat_activity;

SELECT * FROM pg_prepared_xacts;
```

Co monitorovat?

- Stav indexů a tabulek
  - extenze *pg_stat_tuple* ~ je pracné najít hledanou informaci (příliš úrovní, příliš prvků v seznamu)
- Bobtnání tabulek a indexů
  - nízká hustota dat ~ čte se balast
- viz http://wiki.postgresql.org/wiki/Show_database_bloat

Použití specializovaných problémově orientovaných db

- PostgreSQL / MySQL / Firebird / SQLite
- OLAP / OLTP
- ACID / CAP
- noSQL / SQL
- Relační / Grafové (síťové) / key-value
- Relační / Stream / Array
- Row oriented (OLTP) / Column oriented (OLTP)
- Klasické / Memory / Only memory
- Relační / Map-Reduce

Architektura - doporučení

- Při návrhu myslet na možnost shardingu – horizontálního partitioningu
  - PL/Proxy nebo vlastní řešení
- Při návrhu myslet na obnovu a migraci
  - Oddělit kritické a nekritické části databáze
    - logy, archivní data
    - kritická data pro kritické části aplikace
- Při návrhu myslet na cache – do PG neukládat krátkodobá data, ale i zbytečně nepřistupovat do databáze.
- Při návrhu myslet na limitované IO – nekritické pomalé dotazy počítat mimo pracovní dobu nebo na dedikovaných serverech. Snažit se o index friendly dotazy.

Doporučení

- **Nepoužívejte ORM u datově náročných aplikací!**
  - Ladění ORM může zabrat výrazně víc času než ušetří
  - Ladění ORM je větší magie než ladění databáze
  - Napsat rychlou aplikaci není zas až tak obtížné, pokud znáte základy a základní charakteristiky databází
  - Je zásadní znát parametry aplikace, parametry databáze, u existujících aplikací znát úzká hrdla
- **Výkon testujte průběžně! Neodkládejte testování na dokončení produktu**.

Dotazy?

- uvidíme se na P2D2 únor 2014

konfigurace terminálu

```
export PAGER=less
export LESS="-iMSx4 -FX"
```