Optimalizace SQL

Co je to cost based optimalizace a jak se využijí statistiky o databázových objektech při cost-based optimalizaci?

- Metoda výběru nejefektivnějšího prováděcího plánu na základě odhadovaných nákladů (cost)
  - Cost typicky vyjadřuje počet zpracovaných IO bloků (typicky 8kB) (CPU se většinou dá zanedbat)
  - Každá operace plánu má cost podle toho, co se v ní děje
  - Součet cen jednotlivých operací je cena prováděcího plánu
  - Vybere se ten plán, který má nejnižší cenu
- Prováděcí plán je stromové znázornění provedení dotazu
  - Začíná list (data)
  - Pokračuje operacemi (join, selekce, projekce, řazení)
  - Končí výsledkem (kořen stromu)
- Statistiky slouží k odhadu cen jednotlivých operací v prováděcím plánu
  - Zahrnují např. počet řádků v tabulce, počet různých hodnot ve sloupcích, hloubku indexu...
  - Musí se udržovat aktuální, jinak mohou vést ke špatnému výběru prováděcího plánu a snížení výkonu
    - Aktualizace neprobíhá online, zajišťuje ji typicky proces na pozadí

Jak vypadá zpracování SQL dotazu (fáze zpracování dotazu, kde a jak se při nich dá optimalizovat)?

Fáze zpracování

1. Parse
   1. Syntaktická a sémantická analýza,
   2. Kontrola práv,
   3. Výběr nejlepšího plánu pomocí cost-based optimalizace
2. Bind - přiřazení konkrétních hodnot parametrům
3. Execute - vykonání zvoleného prováděcího plánu
4. Fetch - získání výsledku a předání aplikaci

Optimalizace jednotlivých fází

Parse fáze:

- Indexy: Zajistěte, že na sloupcích používaných v podmínkách `WHERE`, `JOIN` nebo `GROUP BY` jsou vytvořeny správné indexy.
- Statistiky: Udržujte aktuální statistiky databázového systému, které optimalizátor využívá pro odhad objemu dat (ceny).
- Hinty: Některé databáze umožňují přidat "hinty", které optimalizátoru říkají, jaké strategie použít (např. jaký JOIN algoritmus preferovat).

Execute fáze:

- Partitioning: Rozdělení tabulek na části (partitions), což může zrychlit přístup k datům.
- Caching: Pokud je dotaz často spouštěn, výsledky lze uložit do mezipaměti.
- Paralelismus: Využijte paralelní exekuci, pokud ji databáze podporuje.

Jak optimalizovat

- Základem je porozumění prováděcímu plánu nebo sledování systémových statistik
  - porozumění prováděcímu plánu
    - v drtivé většině úprava/přidání indexů
    - úprava uložení dat - clustery, IOT
    - "hinty" pro optimalizátor
    - techniky pro optimalizaci (neplatí obecně, např. Oracle)
  - sledování systémových statistik
    - úpravy konfigurace serveru (buffer cache, redo log cache, …)
    - fyzická reorganizace (typicky paměť, disky, řadiče, …)

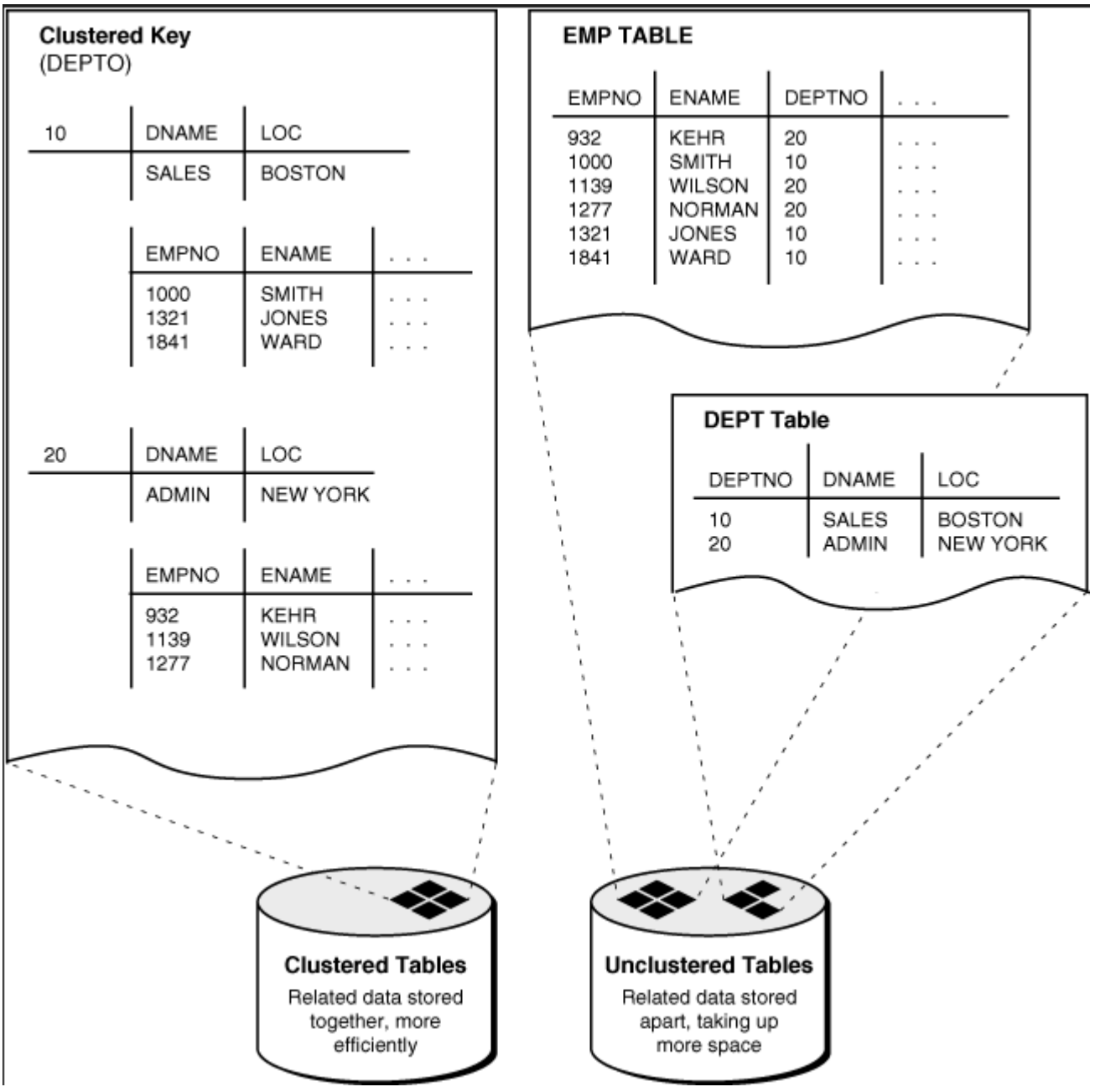
Vysvětlete rozdíl mezi heap table a index-organized table.

- Heap table
  - Řádky nejsou uspořádané, dávají se tam, kde je místo
  - Data různých řádků spolu fyzicky nesouvisí, každý řádek má svoje ROWID určující jeho fyzickou polohu (datový soubor, datový blok, pozice řádky v datovém bloku)
  - Nad touto strukturou může být index (typicky B-Strom), který poté umožňuje přistoupit k řádce pomocí ROWID
  - Úpravy nejsou příliš nákladné
- Index-organised table (IOT)
  - Řádky jsou uspořádané podle primárního klíče
  - Uložené jsou přímo v B-Stromu primárního indexu (eliminuje dodatečný přístup, který je v heap table)
  - Tato struktura však může znamenat častější reorganizaci dat při změnách -> nákladnější úpravy

Hlavní rozdíly

	Heap table	**IOT**
Struktura uložení	Neuspořádané	Uspořádané
Výkon přístupu	2 přístupy (ROWID + data)	1 přístup (data jsou hned v indexu)
Určení	Často se měnící data (UPDATE, INSERT, DELETE)	Neměnící se data (READ heavy tabulky)

Vysvětlete rozdíl mezi heap table a cluster.

- Heap table
  - Řádky nejsou uspořádané, dávají se tam, kde je místo
  - Data různých řádků spolu fyzicky nesouvisí, každý řádek má svoje ROWID určující jeho fyzickou polohu (datový soubor, datový blok, pozice řádky v datovém bloku)
  - Nad touto strukturou může být index (typicky B-Strom), který poté umožňuje přistoupit k řádce pomocí ROWID
  - Úpravy nejsou příliš nákladné
- Cluster
  - Řádky jedné či více tabulek se ukládají do stejných datových bloků, pokud sdílejí hodnotu v cluster klíči
    - Cluster klíč je sloupec či více sloupců společných pro tabulky z clusteru
    - Zadává se při vytváření clusteru
  - Díky tomu jsou joiny tabulek ve stejném clusteru velmi rychlé
  - Je však náročnější na údržbu, tabulky se nedají snadno oddělovat
  - 

Hlavní rozdíly

	Heap table	Cluster
Struktura uložení	Neuspořádané	Seskupení podle cluster key
Určení	Často se měnící data (UPDATE, INSERT, DELETE)	Tabulky s častými joiny podle stejného sloupce

Vysvětlete rozdíl mezi B-tree a bitmap indexem, příklady vhodného použití obou typů indexů.

B-Tree Index:

- Hierarchická struktura stromu (Balanced Tree).
  - B+ strom - data/ukazatele jsou jen v listech, vnitřní nodes jsou navigační
- Hodnoty jsou uspořádané ve stromové struktuře = efektivní vyhledávání, úpravy a mazání
- Typicky 2-4 úrovně a velký faktor větvení (typicky stovky)
- Každý uzel obsahuje klíče a ukazatele na potomky, přičemž listové uzly obsahují ukazatele na fyzická data.
- Vhodné pro vysokou kardinalitu (velký počet různých hodnot ve sloupci) - např. ID, email
- Používá se pro primární a cizí klíče, unikátní hodnoty

Bitmapový Index:

- Plochá struktura
- Ukládá data ve formě bitových map, kde každý bit označuje přítomnost nebo nepřítomnost konkrétní hodnoty v určitém řádku.
- Každá jedinečná hodnota sloupce má svou vlastní bitovou mapu. T.j. pro bitmap index na sloupci se vytvoří 2D mapa.
- Rychlé, úsporné na místo
- Vhodné pro nízkou kardinalitu - např. pohlaví, kategorie
- Nevhodné pro časté zápisy

Příklady použití

B-Tree index

- Výběr osoby podle rodného čísla

Bitmap index

- Najdi všechny ženy

Jaké jsou typické statistiky pro tabulky v relační databázi a jak se udržují, když se pomocí DML mění data?

- DML = Data Manipulation Language (třeba SQL)

Statistiky (Heap table = relace R)

- n_R - počet řádků relace R
- p_R - počet stránek relace R (kolik řádků se v průměru vejde do bloku)
- $V(A, R)$ - variabilita (počet rozdílných hodnot A v relaci R)
- b_R - Block factor - průměrný počet řádek, které se vejdou do jednoho bloku/stránky
- $min(R), max(R)$ - extrémy hodnot v relaci R
- Histogramy - rozložení hodnot v relaci R

Údržba

- Neaktuální statistiky způsobují nesprávný výpočet ceny dotazů
- Statistiky se nikdy živě nemění při DML operacích (to by příliš zatěžovalo stroj)
- Automaticky se přepočítávají démonem když databáze není busy (idle time)

Jaké jsou typické statistiky pro B-tree indexy a jak se udržují, když se pomocí DML mění data?

Statistiky (index na relaci R s klíčem A)

- $I(A, R)$ - hloubka indexového stromu (typicky 2-4)
- $f(A, R)$ - faktor větvení (typicky 50-150)
- $p(A, R)$ - počet listových bloků
- Clustering faktor - počet bloků které musím přečíst, abych dostal data seřazená podle A

Údržba

- Neaktuální statistiky způsobují nesprávný výpočet ceny dotazů
- Statistiky se nikdy živě nemění při DML operacích (to by příliš zatěžovalo stroj)
- Automaticky se přepočítávají démonem když databáze není busy (idle time)

Co jsou to přístupové cesty (access paths) při vyhodnocování SQL dotazů? Uveďte příklady.

- Způsoby, jakými DB fyzicky získává data
- Při vybírání přístupové cesty záleží i na statistikách

Způsoby

Full Table Scan

- Sekvenční čtení všech bloků tabulky
- Vhodné když
  - Potřebujeme velkou část dat z tabulky
  - Tabulka je malá
  - Neexistují vhodné indexy

Index scan

- Vstupem je index, výsledkem množina (adres) řádků (adresy se vrací pro Heap table, musím potom přečíst ještě řádek - jedna operace navíc)
- Index Unique scan - nalezení jedné hodnoty
- Index Range scan - nalezení rozsahu hodnot
- Vhodné když
  - Potřebujeme jednu hodnotu
  - Máme velkou kardinalitu hodnot sloupce

Přístup přes ROWID

- Nedoporučuje se dělat přímo na aplikační úrovni (ROWID se může měnit)

Příklady

```
SELECT * FROM R WHERE A = 'x';
```

Full table scan

- $cost = p_R/2$ pokud je hodnota unikátní
- $cost = p_R$

Index na R s klíčem A

- $cost = I(A, R)$ pro Indexem organizovanou tabulku (IOT)
- $cost = I(A, R) + 1$ pro heap table s indexem
- $cost = I(A, R) + n(R(A =' x'))$ v případě neunikátního indexu

Jaké znáte metody vyhodnocení spojení (join) v relačních databázích? Naznačte jak probíhají.

Nested loop join

- Vnořené cykly

```
foreach page r in R:
    foreach page s in S:
        output += match tuples in [r,s]
```

- Uvažuje se pouze počet přečtených bloků
- Porovnání jednotlivých n-tic se provádí v paměti ($cost = 0$)
- Vhodné pro malé tabulky, nejlepší je, když se alespoň jedna z nich vejde do paměti
- Časová složitost $O(p_R * p_S)$
- Požadavky na paměť - $M = 3$

Merge join

- Procházení seřazených relací R a S, jako slučovací fáze Merge sortu

```
sort R according to A
sort S according to A
read sorted relatins, if R.A = S.A sestroj výsledek
```

- Málo kdy se vejde do paměti -> je potřeba více běhů
- Dobré pro velké tabulky a požadavky na rovnost/nerovnost

Hash join

- Snaha snížit počet vzájemných porovnání -> rozdělení hash funkcí

```
choose hash function (ie. mod(k))
apply hash function to both relations (join attributes)
compary only groupd from R and S with same hash value
```

- Dobré pro velké tabulky a test na rovnost
- Probíhá ve dvou fázích
  1. Vytvoření hashovací tabulky z menší relace (Build)
  2. Procházení větší relace a hledání shody v menší (Probe)

Join pomocí speciálních struktur (index lookup, společný cluster)

Co to je prováděcí plán (execution plan), jak vypadá a kdy vzniká? Vyplatí se ho cachovat? Pokud ano, za jakých okolností?

- Detailní popis toho, jak databáze zpracuje dotaz
- Vzniká při každém SQL dotazu
- Reprezentován jako strom operací (listy jsou data, kořen je výsledek. Mezi jsou operace (selekce, projekce, joiny, sort...))

Kdy cachovat

- Pokud je stejný dotaz (nebo dotaz se stejnou strukturou) prováděn opakovaně, cache může výrazně snížit režii optimalizace. Nebo například mnoho uživatelů naraz se dotazuje na to stejné. (OLTP aplikace)
- U velmi složitých dotazů s více tabulkami a složitou logikou, kde vytváření prováděcího plánu zabírá značné množství času.
- Pokud se datové distribuce v tabulkách příliš nemění, plán bude pravděpodobně stále efektivní

Kdy necachovat

- Složité jednorázové dotazy (typicky OLAP aplikace)

Jaká je základní strategie pro tvorbu prováděcího plánu? Jsou situace, kdy se vyplatí spíše full-table scan přístup namísto index-based? Případně uveďte.

- Optimalizátor analyzuje různé možné strategie pro provedení dotazu
- Vygeneruje více možných prováděcích plánů pro každý dotaz
- Pomocí následujících dat a odhadu vybere nejméně náročný plán na provedení
    - Statistiky dat
        - Počet řádek v tabulce
        - Počet jedinečných hodnot
        - Distribuce dat
        - Velikost tabulky
    - Indexy
        - Typy indexů
        - Selektivita indexů
        - Velikost
    - Podmínky dotazu
        - Filtry
        - Rozsahové podmínky
        - Typy join operací
    - Náklady operací (primárně)
        - Čtení dat z disku
        - CPU náklady
        - Paměťové nároky

Kdy se vyplatí FTS

- Mám nízkou selektivitu (velká část záznamů bude odpovídat)
- Malé tabulky (režie indexu může být vyšší)
- Heap table s dotazem WHERE A < x a vysokým číslem *clustering factor*
    - Clustering factor měří, jak dobře řazení indexu odpovídá fyzickému pořadí řádků v heap table
    - Nízký CF - řádky jsou fyzicky blízko u sebe - index nemusí být špatný ani na range scan
    - Vysoký CF - řádky jsou rozeseté - index je pro range scan neefektivní

Operace řazení, v jakých situacích se používá, jaké jsou parametry pro odhad ceny řazení.

- DISTINCT
- ORDER BY
- HAVING
- Set operace (UNION, INTERSECT, EXCEPT)
- JOIN (Merge join)
- Kontrola IO (UNIQUE)

Parametry

- Velikost dostupné paměti (M)
- Počet bloků tabulky p_R

Proces

1. Rozdělíme tabulku do kusů o velikosti dle dostupné paměti (např. $i = 6$ kusů)
2. Seřadíme tyto díly
3. Zapíšeme výsledek (merge kusů) - potřebujeme $M = i + 1$ kusů paměti)

Cena

- Základ (jeden běh) má $cost = 2 * p_R$ (jednou read + jednou write)
- Výsledná cena závisí na počtu běhů, počet běhů se odvíjí od dostupné paměti
- Dá se vylepšit použitím Priority Queue - garantuje, že průměrná délka běhu je $2M$

Postup při ladění výkonu DB serveru (jak zjistíme co vázne, jak zvolíme SQL dotazy pro ladění?)

- Ladění musí probíhat na zahřáté databázi (po nějaké době běžného provozu)
- Začneme nalezením typické zátěže
- Pokračujeme optimalizací vybraných SQL příkazů
  - Postupujeme po jednom
  - Stanovíme měřitelné metriky
  - Základem je porozumění prováděcímu plánu a sledování statistik
    - Porozumění plánu
      - Většinou vede na úpravy indexů
      - Úprava struktury uložení dat (cluster / IOT)
      - Hinty pro optimalizátor
      - DB specifické techniky pro optimalizaci
    - Sledování statistik
      - Úprava konfigurace serveru (velikosti cache)
      - Fyzická reorganizace
    - Další možnosti
      - Materializování často využívaných pohledů (fyzické uložení dat výsledku)

OLTP vs OLAP, ORDBMS

Vysvětlete rozdíly mezi OLTP a OLAP databází.

OLTP

- Online Transaction Processing
  - Určeny pro zpracování běžných transakcí
- Mnoho malých, rychlých transakcí
- Rychle se měnící data
- Reálný čas (potřeba nízké latence)
- Mnoho konkurentních uživatelů
- Důraz na ACID vlastnosti
- Typické použití

    - Eshopy
    - Blogy

OLAP

- Online Analytical Processing
    - Určeny pro historickou analýzu dat
- Velké a složité dotazy
- Menší počet
- Málo konkurentních uživatelů
- Typické použití
    - DSS (Decision Support System)

Rozdíly TLDR

Zatímco OLTP systémy jsou optimalizované pro rychlé zpracování mnoha jednoduchých operací, OLAP systémy jsou navržené pro komplexní analýzu velkých objemů dat. Mohou se lišit také ve způsobu uložení dat (denormalizace v OLAP pro zlepšení výkonu)

Vysvětlete, případně uveďte na příkladech hlavní přínos objektově relačních databázových systémů oproti čistě relačním.

- ORDBMS je rozšíření klasického RDBMS
    - Možnost vytvářet vlastní datové typy
    - Dědičnost
    - Lepší práce se složitými strukturami (kolekce, hierarchická data, vnořené objekty)
    - Přiřazení metod k vlastním typům
    - Lepší mapování na objektový kód

Vysvětlete co je reference na objekt (typ REF) v objektově-relačních databázích. Jaký je rozdíl mezi referencí na objekt a cizím klíčem?

- REF je reference (pointer) na jiný objekt v databázi
- Narozdíl od cizího klíče neodkazuje na klíč, ale přímo na objekt
    - Při rušení odkazovaného objektu se REF nullují
- Umožňuje jednoduchou dereferencí pomocí DEREF (pohodlnější, než psát join)

Vysvětlete rozdíl mezi relační tabulkou obsahující uživatelem definovaný datový typ a objektovou tabulkou.

Relační tabulka

- Klasická relační tabulka
- Jeden nebo více sloupců má uživatelem definovaný typ
- Řádek není objekt, jen obsahuje složitější hodnotu
- Databáze nepřiděluje řádku OID, nelze na něj odkazovat pomocí REF

Objektová tabulka

- Tabulka obsahující pouze objekty (přiřazuje OID, lze na ně odkazovat pomocí REF)
- Lze volat metody objektu

Hlavní rozdíl tedy spočívá v tom, že relační tabulka pouze využívá složené typy jako své sloupce, zatímco objektová tabulka je sama o sobě kolekcí objektů s vlastní identitou, metodami a možností dědičnosti.

V jakém jsou vztahu objektově-relační databázový stroj a ORM (object-relational mapping) technologie? (co to řeší, kdy je co vhodné)

- Jedná se o rozdílné přístupy řešení stejného problému - překlenutí rozdílu mezi OOP a relačním ukládáním dat
- Řešení na rozdílné úrovni

ORDBMS (na úrovni databáze)

- Rozšíření relačního modelu o objektové prvky
- Přirozené ukládání a manipulace s objekty přímo v databázi

ORM (na úrovni aplikace)

- Mapování aplikačních objektů na klasické relační tabulky
- Software převede objekt na tabulková data, která následně uloží v relační databázi
- Při čtení software z databáze získá tabulková data a na jejich základě instanciuje objekty
- V DB nejsou objekty, ale obyčejné tabulky
- Umožňuje mapovat třídy a jejich atributy na tabulky a sloupce, takže vývojář může pracovat s databází prostřednictvím objektů, aniž by musel psát přímo SQL dotazy.

Kdy je co vhodné?

Použití ORDBMS:

- Složité datové modely: Potřebujete pracovat s komplexními typy dat (např. multimédia, geografická data, hierarchické struktury).
- Potřebujeme objektové vlastnosti pro všechny aplikace
- Lepší výkon: Potřebujete zpracovat logiku přímo na úrovni databáze (např. metody, funkce nebo složité indexy na objektové typy).
  Nevýhody:
- Vyšší složitost správy databáze.
- Tzv. Impedance mismatch - nesoulad mezi objektovým světem aplikace a relačním světem databáze

Použití ORM:

- Rychlý vývoj aplikací: Pokud je potřeba rychle vytvořit aplikaci a pracovat s daty bez psaní SQL.

- Jednodušší přenositelnost mezi DB
- Tradiční relační databáze: Když vaše aplikace používá klasický RDBMS a databáze není připravena na práci s objekty.
  Nevýhody:
- Výkonnostní režie, zvláště u složitých dotazů.

CAP, BASE, ACID, BigData

Uveďte a vysvětlete CAP theorem.

- Předpoklady
  - Distribuovaný systém se shardingem a replikací
  - Operace čtení a zápisu pouze na jednom agregátu
- Teorém tvrdí, že nemohou být poskytnuty všechny 3 záruky zároveň
  - Konzistence (Consistency): Operace pro čtení a zápis musí být spouštěny atomicky. Po zapisu musi vsichni readers videt stejna data. Každé čtení vrátí nejaktuálnější zápis.
  - Dostupnost (Availability): Pokud node funguje, musí odpovídat na požadavky
  - Odolnost k přerušení (Partition tolerance): systém musí fungovat, i když se vypadne část uzlů
- Vybírají se tedy ty dvojice, které nejvíce požadujeme
  - U distribuovaných systému se reálně vybírá mezi AP a CP
  - Tradiční relační databáze jsou z tohoto pohledu zpravidla CA

Vysvětlete rozdíly mezi koncepcí ACID a BASE.

- ACID
  - Atomicity - transakce se provede celá, nebo vůbec
  - Consistency - databáze jde z validního stavu do validního stavu (zápis pouze jsou-li splněny IO)
  - Isolation - operace uvnitř transakce neovlivní ostatní transakce
  - Durability - provedené změny jsou bezpečně uloženy
- BASE
  - Basically Available - Systém by měl prioritizovat dostupnost
  - Soft-state - různé uzly nemusí vracet stejná data
  - Eventually consistent - systém garantuje, že bude za nějakou dobu konzistentní

Rozdíly obecně

- ACID nabízí silné garance konzistence a determinističnosti, kdežto BASE lepší škálovatelnost
- BASE je typický pro distribuované databáze s velkou potřebou škálovatelnosti
- ACID je typický v klasických RDBMS
- BASE zase v NoSQL databázích

Co je to horizontální a co vertikální škálování databáze a jak souvisí s CAP?

Vertikální

- Zvyšování výkonu stroje (jednoho uzlu)
  - Upgrade CPU, RAM ...
- Typické pro klasické RDBMS
- Výhody
  - Stále jeden node (odpadá potřeba řešit distribuovaný systém)
  - Silná konzistence
- Nevýhody
  - Dražší upgrade
  - Potenciální vendor lock-in
  - Nelze škálovat donekonečna

Horizontální

- Přidání dalších uzlů do clusteru
- Data jsou distribuována mezi uzly
- Výhody
  - Teoreticky neomezené škálování
  - Levné uzly (nemusí být tolik výkonné)
  - Teoreticky nezávislé na výrobci
- Nevýhody
  - Nutnost řešit synchronizaci uzlů
  - Rozhodnutí mezi konzistencí a dostupností

Vztah k CAP teorému:

- Horizontální škálování nutně vyžaduje partition tolerance (P z CAP)
- Pak je nutné volit mezi:
  - Konzistencí (CP systémy jako MongoDB)
  - Dostupností (AP systémy jako Cassandra)
- Vertikální škálování se CAP teorému vyhýbá, protože nemá distribuovaná data

Jak lze použít CAP theorem ke klasifikaci databázových strojů? Uveďte příklady databázových strojů, které znáte, a pokuste se je klasifikovat na základě CAP theoremu.

- CA (Consistency-Availability)
  - Zachovávají ACID vlastnosti
  - Mohou nastat výpadky
  - Tradiční RDBMS v single-node setup (MySQL, PostgreSQL, Oracle)
- CP (Consistency-Partition Tolerance)
  - Distribuované systémy upřednostňující konzistenci
  - Upřednostňují konzistenci nad dostupností => distributed locking
  - Typicky maximalizují i dostupnost
  - MongoDB, MemcacheDB, Redis (v cluster modu)

- AP (Availability-Partition Tolerance)
  - Upřednostňují dostupnost nad konzistencí = BASE vlastnosti
  - Data jsou typicky konzistentní v řádu milisekund → eventuálně konzistentní DB
  - Cassandra, DynamoDB, RiakKV, DNS
- Některé NoSQL databáze umožňují do jakési míry ovlivnit vlastnosti stroje, např. Cassandra je v základu AP, ale lze nakonfigurovat jako CP systém, nebo RiakKV taky, pomocí Write Quorum.-

Jaký je rozdíl mezi replikací a technikou sharding? Jsou to techniky, které se vzájemně vylučují nebo se mohou doplňovat?

Sharding

- Rozděluje data na části (shardy)
- Tyto části distribuuje mezi uzly
- Výzvy
  - Uniformní rozložení dat - volba dobrého klíče (aby nedošlo k tzv. Hotspottingu)
  - Balancovaná zátěž
  - Respektování fyzických lokalit

Replikace

- Kopie stejných dat na více uzlech
- Výhody replikace
  - Zvýšení dostupnosti
  - Zrychlení čtení (může se použít uzel fyzicky blíže k nám)
  - Geografická distribuce

Vztah mezi replikací a shardingem

- Doplňují se vzájemně
- Každý shard je replikován do několika uzlů, což zvyšuje dostupnost
- Sharding jako takový zvyšuje škálovatelnost díky rozložení zátěže

Co je to silná a slabá konzistence v NoSQL databázích? Jak souvisí s CAP?

- Konzistence plně souvisí s jednou garancí CAP (Consistency, Availability, Partition tolerance)
- Typicky se volí (v NoSQL) mezi CP a AP systémy
- Některé systémy umožňují konzistenci nastavit (Cassandra, RiakKV)

Slabá konzistence

- Systém preferuje dostupnost na úkor konzistence
- Může nastat okamžik, kdy různé uzly vidí různá data
- Systém garantuje, že se po nějaké době dostane do konzistentního stavu (řádově ms)

Silná konzistence

- Systém preferuje konzistenci na úkor dostupnosti
- Každý uzel vidí data stejně
- Po zápisu jsou aktualizovaná data okamžitě viditelná pro všechna následující čtení, bez ohledu na to, který uzel provádí čtení.

Vysvětlete co je "quorum" a jak se používá k zajištění silné či slabé konzistence?

- Quorum je minimální počet nodů, které musí potvrdit požadavek na čtení (Read Quorum) nebo zápis (Write Quorum), aby byl úspěšný
- Používají se v NoSQL databázích
- Lze je nastavit a ovlivnit tak nastavení konzistence na úkor dostupnosti
- Pokud nenastane Quorum, požadavek nemůže být proveden

Zajištění silné konzistence (N je počet uzlů)

- Write Quorum: $W > \frac{N}{2}$
- Read Quorum: $R > N - W$
- Obviously, pro zajištění slabé, nastavit tyto čísla třeba velmi nízko (1 a 1)

Jak jsou charakterizována BigData (3V+)!

- BigData
  - Buzzword
  - Charakterizují data, která přicházejí velmi rychle, ve velkých objemech a / nebo s velkou roznorodostí
- 3V
  - Volume - velké objemy dat
  - Velocity - vysoká rychlost generování nových dat
  - Variety - různé formáty, strukturovaná i nestrukturovaná data
- Tyto data dali potřebu vzniku nových technologií k jejich zpracování (v zásadě proto vidíme boom NoSQL databází)

Porovnání DB modelů

Uveďte podstatné rozdíly (výhody a nevýhody) relační a dokumentové databáze.

- Datové modely
  - Tabulky s řádky
  - Kolekce s dokumenty

Relační databáze

- Výhody
  - Silná konzistence dat (ACID)
  - Komplexní dotazy pomocí SQL

    - Transakční zpracování
    - Definované schéma (schema-aware)
    - Zaručená integrita dat
- Nevýhody
    - Pevně dané schéma vede na horší flexibilitu
    - Horší horizontální škálovatelnost
    - Složitější práce s vnořenými strukturami
- Příklady: Postgres, Oracle, MSSQL...

Dokumentová databáze:

- Využívají dokumenty
    - Hierarchická, samopopisná stromová struktury (JSON, BSON)
    - Identifikuje je unikátním klíčem, organizuje je do kolekcí
- Výhody
    - Flexibilní schéma (schema-free)
    - Přirozená práce s vnořenými daty
    - Lepší horizontální škálování
    - Snadnější práce s různorodými daty
    - Lepší mapování na objekty v kódu
- Nevýhody
    - Slabší podpora pro komplexní dotazy
    - Obvykle slabší konzistence (eventual consistency)
    - Složitější zajištění integrity dat
    - Méně efektivní pro data s mnoha vztahy
    - Možná duplicita dat
    - Není zde joinovani tabulek - struktura se bud vnoří “dovnitř” dokumentu nebo se tam vytvoří reference na dokument v jiné tabulce (je to jen hodnota, není to foreign key!)
- Příklady: MongoDB, Couchbase, DynamoDB

Uveďte podstatné rozdíly (výhody a nevýhody) relační a XML-nativní databáze.

- Datové modely
    - Tabulky s řádky
    - XML dokumenty

Relační

- Výhody
    - Silná konzistence dat (ACID)
    - Komplexní dotazy pomocí SQL
    - Transakční zpracování
    - Definované schéma (schema-aware)
    - Zaručená integrita dat

- Nevýhody
  - Pevně dané schéma vede na horší flexibilitu
  - Horší horizontální škálovatelnost
  - Složitější práce s vnořenými strukturami
- Příklady: Postgres, Oracle, MSSQL...

XML-nativní databáze:

- Využívají XML
- Dotazovací jazyk je XQuery (nadmnožina XPath)
- Výhody
  - Přirozená práce s hierarchickými daty
  - Podpora XML standardů (XPath a XQuery)
  - Flexibilní schéma
  - Dobrá pro polo-strukturovaná data
  - Dobrá integrace s XML systémy
- Nevýhody
  - Vyšší paměťová náročnost
  - Pomalejší
  - Nevhodné pro propojená data (neexistence foreign key)
- Příklady: BaseX

Uveďte podstatné rozdíly (výhody a nevýhody) relační a key-value databáze.

- Datové modely
  - Tabulky s řádky
  - Kolekce klíč-hodnota (cokoliv)

Vyhody KV:

- Rychlost (čtení i zápis)
- Jednoduchý model
- Dobrá škálovatelnost
- Vhodné pro cachování, user sessions apod

Nevýhody KV

- Není zde obecně možnost vyhledat podle hodnoty
- Chybí provázanost dat
- Omezené dotazovací možnosti

Uveďte podstatné rozdíly (výhody a nevýhody) relační a grafové databáze.

- Datové modely

    - Tabulky s řádky
    - Property Graph
        - Orientovaný labeled (uzly i hrany mají labely) multigraf (mezi dvěma uzly může být více hran)
        - Uzel
            - Má identifikátor, labely a vlastnosti
        - Hrana
            - Má identifikátor, label a vlastnosti

Výhody grafových DB:

- Přirozená reprezentace propojených dat
- Vysoce efektivní pro průchod vztahů
- Snadné modelování komplexních vztahů
- Výkonné pro rekurzivní dotazy
- Flexibilní schéma pro uzly a vztahy
- Přidávání nových typů vztahů je jednoduché

Nevýhody:

- Složitější škálování (vztahy mezi uzly)
- Méně efektivní pro tabulková data
- Menší podpora nástrojů

Uveďte podstatné rozdíly (výhody a nevýhody) relační a sloupcové (wide-column) databáze.

- Datový model sloupcové databáze je Column family -> Row (řádek) -> Column (sloupec)
    - Column family se dá připodobnit k tabulce, obsahuje podobné řádky
    - Row je skupina sloupců (mohou být různé) s identifikátorem RowKey
    - Sloupec je skupina názvu a hodnoty, může obsahovat kromě skalárních hodnot i množiny, seznamy nebo mapy
- Typicky používá jazyk podobný SQL (např. Cassandra CQL)

Výhody wide-column DB

- Dobré pro analytické dotazy
- Efektivní pro velké objemy dat
- Dobrá škálovatelnost
- Vysoká rychlost zápisu

Nevýhody wide-column DB

- Nepodporují join operace
- Místo join se musí data denormalizovat
- Neefektivní pro OLTP aplikace

Uveďte výhody a nevýhody přístupů schema-free a schema aware databází.

schema-aware

- Výhody
  - Garantovaná struktura dat
  - Definice uložených dat - víme, co v DB může být
  - Pokročilé indexování, optimalizace → možnost vytváření efektivních dotazů
  - Kontrola validity (integrity) dat
  - Konzistentní datový model
- Nevýhody
  - Nutné vytvořit schéma a držet se ho (a aplikace, které s DB pracují) → menší flexibilita
  - Složitější změny schématu
  - Složitější práce s nestandardními typy

schema-free

- Výhody
  - Flexibilita
  - Snadné změny struktury
- Nevýhody
  - Nevíme, co je v DB (struktura dat není určena) → zmatenost
  - DB nekontroluje integritu dat
  - Složitější optimalizace dotazů
  - Složitější údržba

V praxi se přístupy mixují - například MongoDB umožňuje validaci i když je schema-free

Probírané DB stroje

Vysvětlete koncepci databázového stroje MongoDB. Uveďte jeho silné stránky a uveďte příklady, kdy je jeho použití vhodné a kdy je naopak nevhodné.

- Dokumentová NoSQL databáze
- Hierarchie úložiště je: Instance -> databáze -> kolekce -> dokument
- 1 dokument = 1 JSON objekt uložený v BSON (binární JSON) (ekvivalent řádku)
- Každý dokument má identifikátor a je v nějaké kolekci (ekvivalent tabulky)
- schema-free databáze
  - Umožňuje však nastavit validace pomocí JSON Schema
- Příbuzná data se buď embedují, nebo se na ně dá odkazovat pomocí ID (je ale vhodnější embedovat)
- Built in podpora pro replikaci a sharding
  - Řešeno pomocí Replica Setu (master slave architektura)

        - Primární a sekundární uzly
        - Zápisy jdou pouze na primární
        - Pokud primární spadne, volí se nový

Silné stránky:

1. Flexibilní schéma
2. Snadná horizontální škálovatelnost
3. Vysoký výkon pro read-heavy operace
4. Dobrá práce s velkými objemy dat
5. Open source, komunita

Použití

- Vhodné pro
    - Katalogy produktů
    - Nestrukturovaná data
    - Logging
- Nevhodné pro
    - Propojená data

Vysvětlete koncepci databázového stroje Cassandra. Uveďte jeho silné stránky a uveďte příklady, kde je jeho použití vhodné a kdy je naopak nevhodné.

- Wide-column schema-aware NoSQL distribuovaná databáze
- Model úložiště: Instance -> Keyspaces -> Tables -> Row -> Column
    - Table (Column family) je kolekce podobných řádků
    - Row (řádek) je kolekce sloupců identifikovaná pomocí primárního klíče
    - Column (sloupec) se skládá z názvu a hodnoty. Může obsahovat kromě skalárních hodnot i množiny, listy nebo mapy
- Decentralizovaná (peer-to-peer) architektura
- Primárně AP systém, dá se ale nakonfigurovat konzistence
- Podpora MapReduce a uživatelských datových typů
- Vysoce škálovatelná
- Nepodporuje Joiny

Silné stránky

1. Extrémně vysoká škálovatelnost
2. Výborný výkon pro zápis
3. Žádný single point of failure
4. Volnost v konfiguraci shardingu a replikace

Použití

- Vhodné pro
  - Masivní objemy dat
  - Hodně zápisů
  - Logy a auditní záznamy
  - Real-time data ze senzorů (např. CERN to používá pro ATLAS)
- Nevhodné pro
  - ACID transakce
  - Propojená data vyžadující JOIN

Vysvětlete koncepci databázového stroje Neo4j. Uveďte jeho silné stránky a uveďte příklady, kdy je jeho použití vhodné a kdy je naopak nevhodné.

- Grafová databáze pracující s Orientovaným property graphem
  - Uzly mají identifikátor, labely a atributy (klíč, hodnota)
  - Hrany mají identifikátor, label a atributy
- Má vlastní dotazovací jazyk Cypher

Silné stránky

- Přirozená reprezentace propojených dat
- Dobré pro analýzu závislostí
- ACID compliant - garantuje transakční zpracování

Použití

- Vhodné pro
  - Sociální sítě
  - Doporučovací systémy
  - Knowledge grafy
- Nevhodné pro
  - Jednoduchá tabulková data
  - Nepropojená data
  - OLAP / BI systémy
  - Systémy vyžadující horizontální škálování a distribuovanou architekturu
- Zkrátka hodí se tam, kde jsou vztahy mezi daty důležité tak, jako data samotná

Uveďte koncepci databázového stroje RiakKV. Uveďte jeho silné stránky a uveďte příklady, kdy je jeho použití vhodné a kdy je naopak nevhodné.

- Distribuovaná Key-Value AP databáze
- Datový model: Instance -> (bucket types ->) buckety -> objekty
  - Bucket types jsou nepovinné, shlukují podobné buckety
  - Bucket je logická kolekce key-value objektů (dá se přiřovnat k tabulce například)

    - Objekt - jeden key-value pár
- Umožňuje vyhledávat v hodnotách, pomocí Search 2.0
- Řešení konfliktů pomocí vector clocks a CRDTs
- Sharding a replikace (peer to peer)
    - Řešeno pomocí Riak Ring
        - Má konzistentní hashovací funkci, která hashuje jméno bucketu a object key to 160bitového celého čísla => Riak Ring
    - Celý ring je rozdělen do disjunktních částí, tzv partitions, a každý fyzický uzel potom spravuje několik partitionů (tzv. virtuální uzly)
    - Každý soused dané partition je na jiném fyzickém node -> replikace probíhá tak, že primární replika se uloží tam, kam to pošle hash funkce a repliky se vytvoří v partitions sousedících s primární podle hodinových ručiček

Silné stránky:

1. Extrémně vysoká dostupnost
2. Fault tolerance
3. Lineární škálovatelnost
4. Jednoduchý model přístupu k datům
5. Přístup přes HTTP

Použití

- Vhodné pro
    - Session storage
    - Logování
    - Cache systémy
    - IoT/senzor data
    - Systémy vyžadující vysokou dostupnost
- Nevhodné použití:
    - Komplexní dotazy
    - Relační data
    - Transakční systémy
    - Když potřebujeme dotazy přes hodnoty
    - Agregační dotazy
    - Systémy vyžadující silnou konzistenci
    - Aplikace potřebující range queries

Probírané jazyky

Krátce popište, případně vysvětlete na vhodných příkladech dotazovací jazyk Cypher.

- Deklarativní dotazovací jazyk pro Neo4j grafovou databázi
    - Uzly jsou reprezentovány v závorkách ()

    - Hrany jsou reprezentovány pomocí [ ], smer --> nebo <-- nebo --
    - Labely začínají dvojtečkou :Person
    - Vlastnosti jsou v složených závorkách {}
- Klauzule se dají téměř libovolně řetězit
- Je založen na matchování podgrafů

Příklady

Vyhledávání

```
MATCH (p:Person {name: 'John'})-[:FOLLOWS]->(friend) RETURN friend
```

```
MATCH (m:MOVIE)-[:PLAY]->(a:ACTOR)
WHERE m.title = "Medvídek"
RETURN a.name, a.year
ORDER BY a.year
```

Tvorba uzlu

```
CREATE (p:Person {name: 'John', age: 30})
```

Agregace

```
MATCH (p:Person)<-[:FOLLOWS]-(follower)
RETURN p.name, COUNT(follower) as followers_count
```

Krátce popište, případně vysvětlete na vhodných příkladech dotazovací jazyk XQuery.

- Jedná se o jazyk pro komplexní dotazy a transformace nad XML dokumenty
- Nadmnožina navigačního jazyka XPath
- Datovým modelem je XDM

XPath

- Osa, node test a další predikáty
- Absolutní a relativní cesty

Příklad

```
/movies/child::move/attribute::year
```

XQuery

- Cesty - XPath výrazy
- FLWOR (For, Let, Where, Order by, Return)

- Kvantifikátory
- If-else
- Konstruktory (pro sestrojení nového XML například)

Příklad

- Sestrojí nový movies element

```
<movies>
    <count>{ count(//movie) }</count>
    {
        for $m in //movie
        return <movie year="{ data($m/@year) }">{ $m/title/text() }</movie>
    }
</movies>
```

Krátce popište, případně vysvětlete na vhodných příkladech dotazovací jazyk MongoDB.

- Dotazovací jazyk podobný Javascriptu
- Příkazy se vyvolávají oproti jedné kolekci dokumentů
- Podporuje
    - Základní CRUD
    - Agregace

Příklady

- Výběr jmen filmů z roku 2006 a výše, seřazených podle názvu sestupně

```
db.movies.find(
    { year: { $gt: 2005 }}, // selekce
    { _id: false, title: true } // projekce
).sort({ title: -1 });
```

- Insert filmů

```
db.movies.insertMany( // lze taky insert one pro vložení jednoho záznamu
    [
        {
            _id: ObjectId("9"),
            title: "Želary",
            year: 2003,
            actors: [ ObjectId("4"), ObjectId("8") ]
        },
        {
            title: "Anthropoid",
            year: 2016,
            actors: [ ObjectId("8") ]
        },
```

```
    ]
);
```

Databázové benchmarky

Charakterizujte rozdíly mezi tzv. micro a complex benchmarkem v databázích.

Micro benchmark

- Testuje nějaký konkrétní aspekt databázového stroje (např. práce s cachí)
- Poskytuje detailní pohled na tento testovaný aspekt
- Příklady
  - Rychlost čtení jednoho řádku
  - Výkon konkrétního typu indexu
  - Rychlost zápisu do logu
  - Latence jednotlivých dotazů

Complex benchmark

- Testuje celou databázi nad předem daným scénářem
- Simuluje reálné zatížení a use-cases
- Většinou jsou velmi rozsáhlé
- Musí být velmi dobře definované všechny podmínky a struktury pro správné provedení
- Příklady
  - TPC-C (OLTP Eshop)
  - TPC-E (OLTP Burza)
  - TPC-H (OLAP simulace)

Co je TPC a jak souvisí s databázovými benchmarky?

- TPC je nezisková organizace zaměřující se na výkon a benchmarky databází
- Definují různé standardizované databázové benchmarky a zajišťují také záznam a ověření výsledků (certifikuje platné výsledky)

Hlavní benchmarky

TPC-C

- Benchmark pro OLTP
- Poměrně jednoduchý benchmark na dnešní stroje
- Simuluje prostředí eshopu

TPC-E

- Benchmark pro OLTP
- Dalo by se říct nástupce TPC-C benchmarku

- Simuluje prostředí online brokera akcií

TPC-H

- Benchmark pro OLAP systémy
- Simuluje velké a náročné transakce (BI)

Metriky používané v testech

- Transakce za sekundu
- Cena/transakce
- Energetická efektivita
- Doba odezvy

Vysvětlete princip benchmarku TPC-C. Co je výstupem benchmarku?

- Benchmark pro OLTP systémy
- Simuluje objednávkový systém / eshop

Požadavky na latenci

- 90% transakcí má limit 5s
- Stav skladu 20s

Provádění

- Při provádění se stále zvyšuje zátěž systému, dokud není porušen požadavek na latenci, nebo není dosaženo HW limitu
- Výsledné metriky se berou z posledního úspěšného testu

Metriky

- Dvě hlavní
  - tmpC (transactions per minute - TPC-C)
    - Počet dokončených transakcí za minutu
    - Čím vyšší, tím lepší
  - price/tpmC
    - Cena systému za jednotku výkonnosti
    - Čím nižší, tím lepší

Vysvětlete princip benchmarku TPC-E. Co je výstupem benchmarku?

- Benchmark pro OLTP systémy
- Nástupce TPC-C
  - Rozmanitější transakce
  - Více tabulek

    - Uvěřitelnější data
- Simuluje prostředí online investičního brokera
- Náročné na implementaci - mohou si to dovolit jen velké firmy

Provádění

- Při provádění se stále zvyšuje zátěž systému, dokud není porušen požadavek na latenci, nebo není dosaženo HW limitu
- Výsledné metriky se berou z posledního úspěšného testu

Metriky

- tpsE (transactions per seconde TPC-E)
    - Počet transakcí za sekundu
    - Hlavní metrika výkonu
    - Čím vyšší, tím lepší
- $/tpsE
    - Cena za jednotku výkonu
    - Čím nižší, tím lepší

Vysvětlete princip benchmarku TPC-H. Co je výstupem benchmarku?

- Benchmark pro OLAP systémy
- Má fixní velikost DB, předgenerovaná data (GB až TB)
- Komplexní SQL dotazy
    - Agregace, JOINy, GROUP BY a poddotazy
- Simuluje datový sklad

Rozdíly oproti TPC-C/E:

- Zaměření na analytické dotazy místo transakcí.
- Důraz na objem dat a komplexnost dotazů
- Měření v hodinách místo transakcí za sekundu/minutu

Metriky

- Composite Query-per-Hour (QphH@Size)
    - Hlavní metrika výkonu
    - Čím vyšší, tím lepší
- $/QphH (cena/výkon)
    - Čím nižší, tím lepší

Poznámka k měření

- Výkon se měří při sekvenčním (power test) a paralelním (throughput test) zpracování
- Hodnota výsledku potom reflektuje několik aspektů, např:

- Zvolenou velikost databáze oproti zvoleným dotazům
- Výpočetní výkon na jednom streamu
- Propustnost dotazu pro více uživatelů najednou