**Moderní databázové systémy – zkouška**

**Úvod**

**Tři vrstvy databázového modelování**

* Konceptuální – ER, UML
  + - Nejvyšší level abstrakce, modelování reálného světa a jeho vztahů
  + Logické – objekty, relace, XML, graf
    - Strojově interpretovatelné datové struktury pro ukládání modelovaných dat
  + Fyzické – datové soubory, indexový struktury
    - Jak jsou logické struktury databáze implementovány v konkrétním technickém prostředí

**Databázové modely**

* První generace – navigační: Hierarchický model (stromy), Síťový model (grafy)
* Druhá generace – relační
* Třetí generace – post-relační
  + Rozšíření relačního modelu – objektově-relační – objekty, třídy, dědičnost
  + Nové modely navazující na populární technologie – objekty, XML, NoSql (key/value, column, document, graph, …) – Big Data
  + Multi-model systém, NewSQL – zpátky k relacím

**Relační model**

* Ukládání objektů a jejich vzájemných asociací do tabulek (relací)
* Řádek v tabulce (člen relace) = objekt/asociace, sloupec (atribut) = atribut objektu/asociace
* Schéma tabulky (relace) = název schématu + seznam atributů a jejich typů
* Základní omezení integrity – unikátní identifikace řádku, simple type atributy, hodnoty NULL (žádné "díry")
* Optimální pro některé aplikace, ale: nepodporuje složité datové typy
  + Normalizace dat do tabulkové podoby ovlivňuje výkon při vyhledávání rozsáhlých, složitých a hierarchicky strukturovaných dat
  + -> objevily se objektově orientované programovací jazyky – koncept uživatelsky definovaných tříd

**Hierarchický model**

* Data jsou uspořádána do záznamů, které se rekurzivně skládají z jiných záznamů
* IBM’s IMS (Information Management System) - Jeden z prvních komerčně dostupných DBMS.
* Les stromů – vztahy typu 1:n
  + První nezávislý = redundance – záznam nemůže být uložen ve 2 různých stromech, aniž by se duplikoval
* Zpracování dat: hierarchické, počínaje kořenem, hloubkové, procházení zleva doprava
  + První ukládání na pásky – lineární přístup
  + Později (příchod disků) přímý přístup díky technikám hashování a B-stromů

**Síťový model**

* Datové záznamy propojené binárními vztahy, blízko ER modelu
  + Zpracování dat: navigační primitiva, podle kterých se k záznamům přistupuje, aktualizují se jeden po druhém -> Relační dotazovací jazyky: množinová orientace
* Uzly = typy záznamů – reprezentují entity reálného světa, mají atomická nebo složená pole, záznam = datová jednotka (má identifikátor)
* Hrany = typy množin – typ vztahu 1:n, seznam záznamů = hlavní záznam + členové množiny

**Objektový model a objektová databáze**

* Rozšíření objektů o perzistenci dat, tj. databáze
* Objekty = základ pro modelování v databázové aplikaci -> instance tříd
* Data uložená jako graf objektů (z hlediska OOP)
* Vhodné pro individuální navigační přístup k entitám, Nevhodné pro "batch operace" (datově náročné aplikace)
* Cíl: programátor se nemusí starat o perzistenci hierarchie objektů
  + Data aplikace se načítají/ukládají z/do databáze podle potřeby

**Objektově-relační databáze**

* Rozšíření databází o objekty, střední cesta mezi relačními databázemi a objektově orientovanými databázemi
* Cíl: překlenout propast mezi relačními databázemi a technikami objektového modelování používanými v programovacích jazycích
* Relační model plus: objekty, třídy, dědičnost, složité typy atributů, vlastní datové typy, metody/funkce

**XML model a databáze**

* XML – W3C markup language -> DTD, XML Schema, XPath, XQuery, XSLT, …
* XML databáze – podpora datového typu XML + související technologie
* SQL/XML (!= SQLXML !!!) - datový typ XML
  + Rozšíření jazyka SQL – publikace dat (XMLELEMENT, XMLATTRIBUTES, …), dotazování (XMLFOREST, …)

*V rámci* ***batch processingu*** *jsou úlohy nebo transakce seskupeny do souboru, který se nazývá batch soubor, a jsou zpracovávány postupně, bez interakce s uživatelem. Úlohy jsou zpracovávány ve skupinách, často v určitých časových intervalech nebo na základě specifikovaných podmínek.* *Použití: zpracování velkého objemu dat, provedení opakovaných úloh, které nevyžadují okamžitou interakci s uživatelem, např. fakturace, generování reportů, zpracování transakcí v bankovnictví*

**Úvod do Big Data** (bez standardní definice)

* „velkoobjemová, vysokorychlostní, a/nebo různorodá informační aktiva, která vyžadují nové formy zpracování umožňující lepší rozhodování, zjišťování informací a optimalizaci procesů“
* Ze sociálních medií (každý generuje data), vědecké projekty (z družic…), mobilní zařízení
* 1. VOLUME (scale) – roste exponenciálně
* 2. VARIETY (complexity) – různorodost, různé formáty, struktury (od semi-structurované až nestrukturované)
* 3. VELOCITY (speed) – rychlost nárůstu dat, data jsou generována rychle a potřebují být rychle zprocesována
* 4. VERACITY (uncertainty) – nejistá věrohodnost kvůli nekonzistenci, neúplnosti, zpoždění, nejednoznačnosti…
* 5. VALUE – vysoká hodnota pro firmu, která je vlastní
* 6. VALIDITY – limitovaná doba platnosti pro jejich využití
* 7. VOLATILITY – přechodná doba jejich nutného ukládání

**Zpracování Big data**

* OLTP: Online Transaction Processing (DBMSs)
  + Databázové aplikace, ukládání, dotazování, přístup více uživatelů, jede nonstop
* OLAP: Online Analytical Processing (Data Warehousing)
  + Odpovědi na vícerozměrné analytické dotazy
  + Finanční/marketingové výkaznictví, rozpočty, prognózy, ...
* RTAP: Real-Time Analytic Processing (Big Data Architecture & Technology)
  + Data shromažďovaná a zpracovávaná v reálném čase -> Stream způsobem
  + Data v reálném čase dotazovaná a prezentovaná online
  + Kombinace dat v reálném čase a dat z historie a jejich interaktivní vytěžování

**Klíčové technologie související s Big data**

* Distribuované file systémy (e.g., HDFS)
* Distribuované databáze – primárně NoSQL, a mnoho dalších typů
* Grid computing, cloud computing
* Analýza dat – batch, real-time, stream
* MapReduce a další nová paradigmata
* Strojové učení ve velkém měřítku

*Distribuovaný souborový systém se zaměřuje na správu souborů, zatímco distribuovaná databáze se zaměřuje na správu strukturovaných dat a poskytuje pokročilejší funkce pro práci s těmito daty.*

**Relational Database Management Systems (RDMBS)** = Systémy správy relačních databází

* Převládající technologie pro ukládání strukturovaných dat – Webové a obchodní aplikace
* Relační kalkul, SQL
* Braná jako jediná alternativa pro ukládání dat – perzistence, kontrola souběžnosti, integrační mechanismus, ...
* Alternativy: Objektové databáze nebo úložiště XML -> nikdy nezískaly stejné přijetí a podíl na trhu

**Moderní databázové systémy specifické pro Big Data**

* NoSQL databáze – key/value, column, document, graf
  + Motivace pro NoSQL: obrovské množství dat zpracováváno v reálném čase, Data i případy použití jsou stále dynamičtější, Sociální sítě (grafová data) získaly velkou dynamiku – grafové databáze, S full-textem se v systémech RDBMS vždy zacházelo nešetrně
* NewSQL databáze, Multi-model databáze, Array databáze, …

**NoSQL Databáze**

**5 výhod:**

1. Elastic scaling (Flexibilní škálovatelnost)
   * klasické databázové systémy využívají vertikální škálovatelnost (scale up) -> výkonnější hardware
   * NoSQL databáze škálují horizontálně (scale-down) -> Zpracování každé úlohy tyto systémy mohou distribuovat v rámci clusteru, který je možné dle potřeby rozšiřovat přidáváním dalších uzlů
2. Big data
   * Objem ukládaných dat se masivně zvýšil, otevírá nové dimenze, které nelze zpracovat pomocí RDBMs
3. Sbohem DBAs (Database Administrator) - správa a údržba databázových systémů
   * Automatické opravy, distribuce, ladění, ... oproti drahým, vysoce kvalifikovaným DBA RDBMS.
4. Ekonomika – levné commodity servery (horizontální škálování) -> nižší náklady na transakci/sekundu
5. Flexibilní datový model
   * Není třeba určit databázé schéma / schéma dat je volné, dodržování schématu ponecháno na aplikaci a jeho změna neznamená pro databázi významnou zátěž

**5 výzev**

1. Zralost – stále v pre-production fázi, klíčové funkce ještě nebyly implementovány
2. Uživatelská podpora – většinou open-source, překotný vývoj, omezené zdroje a důvěryhodnost
3. Administrace – vyžaduje hodně dovedností pro instalaci a hodně snahy pro údržbu
4. Standardizace přístupu k datům – v relačních databází standardizovaný přístup pomocí SQL, v NoSQL má každá databáze vlastní dotazovací jazyk a programátorské rozhraní (API) -> netriviální programátorské znalosti
5. Expertíza – málo odborníků na NoSQL dostupných na trhu

**Předpoklady pro data**

|  |  |
| --- | --- |
| RDBMs | NoSQL |
| integrita je klíčová | v pořádku, pokud je většina dat správná |
| datový formát konzistentní, dobře definovaný | formát dat neznámý nebo nekonzistentní |
| dlouhodobé uložení dat | ukládáme jen data za určité časové okno (brzo nahrazena) |
| aktualizace dat jsou časté | write-once/read-many – vložená data většinou nejsou měněna |
| předvídatelný, lineární růst | nepředvídatelný růst velikosti dat (exponenciální) |
| ne-programátoři píšící dotazy | pouze programátoři píšící dotazy |
| pravidelné zálohování | replikace dat |
| přístup přes hlavní server | data umístěna na více serverech, přístup ke clusteru uzlů |

**NoSQL data model**

* Datový model = model, podle kterého databáze organizuje data
* Každé řešení NoSQL má jiný model
  + klíč-hodnota, dokument, rodina sloupců, graf
  + První tři orientace na agregáty
* Agregát = datová jednotka se složitou strukturou (agregace – diamant v UML)
  + "Agregát je kolekce příbuzných objektů, se kterými chceme zacházet jako s jednotkou.“
* Agregačně-ignorantní
  + Neexistuje univerzální strategie, jak kreslit hranice agregátů -> záleží na manipulaci s daty
  + RDBMS a grafové databáze jsou agregačně-ignorantní
    - Umožňuje snadno nahlížet na data různými způsoby
    - Lepší volba, když nemáme primární strukturu pro manipulaci s daty
* Agregačně-orientované
  + Agregáty poskytují databázi informace o tom, s kterými částmi dat se bude manipulovat společně
    - Které by se měly nacházet na stejném uzlu
  + Výrazně pomáhá při provozu na clusteru -> při sběru dat chceme min počet uzlů pro dotazování
* Důsledek pro transakce – databáze NoSQL podporují atomickou manipulaci s jedním agregátem najednou

**Materialized view**

* Nevýhoda: agregovaná struktura je daná, jiné typy agregací nelze snadno provádět
  + RDBMS chybí agregační struktura -> podpora přístupu k datům různými způsoby (pomocí views)
* Řešení: materialized view -> předpočítané dotazy a dotazy v mezipaměti
* Uloží výsledek určitého dotazu do separátní tabulky a vytváří kopii určité části dat ve zcela odlišné struktuře

**Bez schématu**

* Když chceme ukládat data v RDBMS, musíme definovat schéma
* Svoboda a flexibilita
  + Umožňuje snadno měnit ukládání dat podle toho, jak se dozvídáme více o projektu
  + Snadnější práce s nejednotnými daty
* Fakt: obvykle je přítomno implicitní schéma -> program pracující s daty musí znát jejich strukturu

**Distribuovaný file systém**

**Apache Hadoop**

* Open-source software framework
* Běh aplikací na velkých clusterech komoditního hardwaru
* Obsahuje distribuovaný file systém – Hadoop Distributed File System (HDFS)
* Hadoop YARN, Hadoop MapReduce, spousta souvisejících projektů – databáze Cassanda, Hbase

**Hadoop Distributed File System (HDFS)**

* Free, open source, crossplatform – čistá Java, ale vazby i na jiné programovací jazyky
* Odolný proti chybám, vysoce škálovatelný
  + Idea: "selhání je spíše normou než výjimkou"
  + Instance HDFS se může skládat z tisíců strojů -> Každý z nich ukládá část dat fiel systému a každá část má netriviální pravděpodobnost selhání
  + Předpoklad: vždy je nějaká komponenta nefunkční -> detekce poruch, rychlá, automatická oprava
* **Charakteristika dat**
  + Předpokládá: streamingový přístup k datům, spíše batch processing než interaktivní přístup uživatele
  + Velké datové soubory
  + Zápis pouze jednou / čtení vícekrát
  + Optimální aplikace pro tento model: MapReduce, webcrawlery, ...
* HDFS Architecture 
  Metadata ops 
  lien 
  Re d Datanodes 
  Rack 1 
  Namenode 
  Metadata (Name, replicas, 
  /home/foo/data, 3, 
  Bloc ops 
  Replication 
  Write 
  lien 
  Datanodes 
  Bloc s 
  Rack 2 **Architektura master/slave**
  + HDFS odkrývá namespace file systému
  + Soubor – interně rozdělen na 1/víc bloků
    - Typická vel. bloku 64(128) MB
  + NameNode = hlavní server, který spravuje namespace file systému + reguluje přístup klientů k souborům
    - Otevírání/zavírání/přejmenovávání souborů a adresářů, mapování bloků na DataNodes
  + DataNode = obsluhuje požadavky na čtení/zápis od klientů + provádí vytváření/mazání bloků a replikaci podle pokynů NameNodu
    - Obvykle jeden na uzel v clusteru, spravuje úložiště připojené k uzlu, na kterém běží
* **Namespace**
  + Hierarchický file systém – adresáře a soubory
  + Vytváření, odstraňování, přesouvání, přejmenování, ...
  + NameNode spravuje file systém -> zaznamenává všechny změny metadat v file systému
  + Aplikace může určit počet potřebných replik souboru – faktor replikace souboru (uloženo v NameNodu)
* **Replikace dat**
  + Systém HDFS je navržen pro ukládání velmi velkých souborů na různých strojích ve velkém clusteru
    - Soubor = posloupnost bloků, všechny bloky v souboru -> stejná velikost (kromě posledního), velikost bloku je konfigurovatelná pro každý soubor
  + Bloky jsou replikovány kvůli odolnosti proti chybám – počet replik je konfigurovatelný pro každý soubor
  + NameNode dostává HeartBeat a BlockReport od každého DataNode
    - BlockReport obsahuje seznam všech bloků v DataNodu
* **Umístění replik (replica placement)**
  + Umístění replik je rozhodující pro spolehlivost a výkon
  + Rack-aware replica placement = zohlednění fyzického umístění uzlu při plánování úloh a alokaci úložiště
  + Idea: Uzly jsou rozděleny do racků -> komunikace mezi racky prostřednictvím switchů -> šířka pásma sítě mezi stroji na stejném racku je daleko vyšší než mezi stroji v různých rackcích -> NameNode určuje id racku pro každý DataNode
  + První idea: repliky by měly být umístěny na různých rackcích
    - Prevence ztráty dat, pokud jeden rack selže, využití šířky pásma při čtení dat z více racků
    - Zápisy jsou drahé (přenos do různých racků) - > potřebujeme zapisovat do všech replik
  + Běžný případ: replikační faktor je 3 -> Repliky umístěny: 1 na uzlu v local racku, 1 na jiném uzlu v local racku, 1 na uzlu v jiném racku -> Snižuje přenos zápisu mezi racky
* **Jak pracuje NameNode?**
  + Ukládá namespace HDFS
  + Používá transaction log **EditLog** (uložen v místním file systému NameNodu) – zaznamenává každou změnu v metadatech file systému, např. vytvoření souboru, změna replikačního faktoru
  + **FsImage** = obraz namespace file systému + mapování bloků na soubory + vlastnosti file systému
    - Uloženo v souboru v místním file systému uzlu NameNode
    - Navrženo tak, aby bylo kompaktní – načítá se v paměti uzlu NameNode, stačí 4 GB paměti RAM
  + Při spuštění file systému:
    - 1. Přečte z disku FsImage a EditLog
    - 2. Aplikuje všechny transakce ze záznamu EditLog na reprezentaci FsImage v paměti
    - 3. Vytvoří novou verzi do nového FsImage na disku = checkpoint
    - 4. Ořízne EditLog
  + Kontrolní body se pak vytvářejí pravidelně, obnova = poslední kontrolní bod
* **Jak pracuje DataNode?**
  + Ukládá data do souborů ve svém místním file systému -> nemá žádné znalosti o file systému HD
  + Ukládá každý blok dat HDFS do samostatného souboru
  + Nevytváří všechny soubory ve stejném adresáři
  + Při spuštění file systému:
    - 1. Vygeneruje seznam všech bloků HDFS = BlockReport
    - 2. Odešle ho do NameNode
* **Selhání**
  + Primární cíl: spolehlivě ukládat data v případě selhání.
  + Tři běžné poruchy: Selhání NameNode, Selhání DataNode, Network partition (rozdělení sítě)
  + Rozdělení sítě může způsobit, že podmnožina DataNodů ztratí spojení s NameNodem
    - NameNode zjistí tento stav podle absence zprávy Heartbeat
    - NameNode označí DataNody bez HeartBeat a neodesílá jim požadavky na IN/OUT operace
    - Data registrovaná k selhanému DataNode nejsou pro systém HDFS dostupná
  + Smrt DataNode může způsobit, že replikační faktor některých bloků klesne pod jejich stanovenou hodnotu → re-replikace (také při poškození repliky, selhání hard disku, zvýšení replikačního faktoru, ... )
* **API**
  + Java API pro použití aplikací – lze použít přístup v Pythonu, pro Java API je k dispozici wrapper jazyka C
  + K procházení souborů instance HDFS lze použít prohlížeč HTTP
  + Rozhraní příkazového řádku zvané FS shell – umožňuje uživateli pracovat s daty v HDFS
    - Syntaxe příkazů je podobná bash, např. pro vytvoření adresáře /foodir
      * /bin/hadoop fs -mkdir /foodir
  + Pro zobrazení namespacu je k dispozici rozhraní prohlížeče

**Programovací modely** = soubor pravidel, konceptů a paradigmat, které definují způsob, jakým programátoři mohou vytvářet a strukturovat své aplikace

* Von Neumannův model – Provádí proud instrukcí (strojový kód), komplexní
* Modely paralelního programování
  + Message passing (Předávání zpráv) - nezávislé úlohy zapouzdřující lokální data, úlohy spolu komunikují výměnou zpráv
  + Sdílená paměť – Úlohy sdílejí společný adresový prostor, úlohy spolu komunikují čtením a zápisem z/do tohoto prostoru (asynchronně)
  + Paralelizace dat – data jsou rozdělena mezi úlohy, úlohy provádějí posloupnost nezávislých operací

map (String key, String value) : 
// key: 
document name 
// value: 
document contents 
for each word w in value: 
Emit Intermediate (w, "1 ") ; **MapReduce**

**MapReduce framework**

* Programovací model + implementace
* Distribuované, paralelní výpočty na velkých datech
* **Paradigma rozděl a panuj**
  + reduce (String key, Iterator values) : 
    // key: 
    a word 
    // values: 
    a list of counts 
    int result = 
    o; 
    for each v in values: 
    result += Parse Int (v) ; 
    Emit (key, AsString (result) ) ; **Map** rozděluje problém na sub-problémy -> zpracovává vstupní data a vytváří množinu mezilehlých dvojic key/value
  + **Reduce** přijímá a kombinuje dílčí řešení k vyřešení problému -> zpracovává mezivýsledky přiřazené ke stejnému mezikliči
* Mnoho reálných úloh lze vyjádřit tímto způsobem
  + Programátor se soustředí na kód map/reduce
  + Framework se stará o rozdělení dat, plánování provádění na různých strojích, řešení výpadků strojů, řízení komunikace mezi stroji, ...
* **Fáze map** – dostanu kus dat, udělám to, co potřebuji, a na základě zpracování vracím nějaký klíč a hodnotu
  + Input: vstupní data
  + Output: množina key/value dvojic
* **Fáze reduce** – dostaneme klíč a hodnotu, udělám, co musím, a pak vracím opět klíč a hodnotu
  + Input: key a pro něj všechny možné hodnoty
  + Output: nejmenší možná množina values

**Application parts**

* **Input reader**
  + Rozdělí vstupní data do "splitů" příslušné velikosti -> každé z nich přiřadí jedné funkci Map
  + Čte data ze stabilního úložiště, např. distribuovaného souborového systému.
  + Deer Bear River 
    Car Car River 
    Deer Car Bear 
    Splitting 
    Deer Bear River 
    Car Car River 
    Deer Car Bear 
    Mapping 
    Deer. 1 
    Bear, 1 
    Riær, 1 
    car, 1 
    car, 1 
    River, 1 
    Deer, 1 
    car, 1 
    Bear, 1 
    Shuffling 
    Bear, I 
    Bear, 1 
    Car, 
    car, 1 
    car, 1 
    Deer, 1 
    Deer, 1 
    River, 1 
    River, 1 
    Reducing 
    Bear, 2 
    car, 3 
    Deer, 2 
    River, 2 
    Final 
    Bear, 2 
    car, 3 
    Deer, 2 
    River, 2 Generuje páry klíč/hodnota
* **Map funkce** – uživatelem zadané zpracování dvojic klíč/hodnota
* **Partition funkce** (rozdělení) - Výstup mapovací funkce je přidělen reduceru
  + Funkci partition je zadán klíč (výstup z Map) a počet reducerů a vrací index požadovaného reduceru
    - Výchozí nastavení je hashování klíče a použití hashovací hodnoty modulo počtu reducerů
* **Compare funkce** – Třídí vstup pro funkci Reduce
* **Reduce funkce** – Uživatelem zadané zpracování klíče/hodnoty
* **Output writer** – Zapisuje výstup funkce Reduce do stabilního úložiště, např. distribuovaný file systém

**MapReduce execution (google)**

1. krok
   * Knihovna MapReduce v uživatelském programu rozdělí vstupní soubory na M částí – ovladatelné uživatelem prostřednictvím volitelného parametru
   * Spouští kopie programu na clusteru počítačů
2. krok
   * Master = speciální kopie programu, Workers = další kopie, kterým je přidělena práce podle masteru
   * M map tasků a R redukčních tasků k přiřazení
   * Master vybere nečinné pracovníky a každému z nich přiřadí Map task (nebo Reduce task)
3. Krok – worker, kterému je přidělen map task:
   * Přečte obsah příslušného vstupního rozdělení, rozparsuje páry klíč/hodnota ze vstupních dat
   * Předá každou dvojici uživatelem definované funkci Map
   * Zprostředkující dvojice klíč/hodnota vytvořené funkcí Funkce Map se ukládá do vyrovnávací paměti
4. Krok
   * Pravidelně se nabufferované dvojice zapisují na místní disk -> rozdělen do oblastí R pomocí partition funkce
   * Lokace nabufferovaných párů na místním disku je předána hlavní paměti -> zodpovědná za předávání lokace Reduce workerů
5. Krok
   * Reduce worker je nadřízeným pracovníkem informován o lokaci dat
   * Používá volání vzdálených procedur ke čtení nabufferovaných dat z lokálních disků Map workerů
   * Po přečtení všech mezi výstupních dat je seřadí podle mezi výstupních klíčů
     + Obvykle se mnoho různých klíčů mapuje na stejný Reduce task
     + Pokud je množství mezidat příliš velké, použije se externí třídění
6. Krok
   * Reduce Worker iteruje nad setříděnými mezidaty.
   * Pro každý nalezený meziklíč: předá klíč a odpovídající sadu mezivýsledných hodnot uživatelské funkci Reduce -> výstup je připojen ke konečnému výstupnímu souboru pro tento oddíl Reduce

**Function combine**

* Po fázi mapování předá mapper po síti celý mezisoubor dat reduktoru
* Někdy je tento soubor vysoce komprimovatelný
* Uživatel může specifikovat function combine
  + Stejně jako funkce reduce, Spustí ji mapovač před předáním úlohy reduktoru (nad lokálními daty)

**Counters**

* Může být spojen s jakoukoli akcí, kterou provádí mapper nebo reduktor
* Uživatel může countery sledovat v reálném čase, aby viděl průběh tasku

**Tolerance chyb**

* Velký počet strojů zpracovává velké množství dat → nutná odolnost proti chybám
* **Selhání workera**
  + Master pravidelně pinguje každý worker
  + Pokud do určité doby neobdrží žádnou odpověď, označí master workera za selhaného
  + Všechny jeho tasky se resetují na iniciální nečinný stav -> stanou možné pro naplánování na jiných workerů
* **Selhání mastera**
  + Strategie A:
    - Master zapisuje periodicky checkpointy hlavních datových struktur
    - Pokud dojde k jeho smrti, lze spustit novou kopii z posledního checkpointu
  + Strategie B:
    - Existuje pouze jeden master -> jeho selhání je nepravděpodobné
    - Výpočet MapReduce se jednoduše přeruší, pokud selže hlavní jednotka
    - Klienti mohou tento stav zkontrolovat a v případě potřeby operaci MapReduce zopakovat

**Stragglers**

* Straggler = stroj, kterému trvá neobvykle dlouho, než dokončí jednu z mapovacích/redukčních úloh ve výpočtu
  + Příklad: stroj se špatným diskem
* Řešení: Když se operace MapReduce blíží k dokončení, hlavní počítač naplánuje záložní provedení zbývajících rozpracovaných úloh -> task označen jako dokončený, když je dokončeno buď primární, nebo záložní provedení

**Task granularita**

* M částí Map fáze a R částí Reduce fáze
  + V ideálním případě obě mnohem větší než počet pracovních strojů, jak je nastavit?
* Master dělá O(M + R) rozhodnutí o plánování
* Master uchovává v paměti O(M \* R) stavových informací
  + Pro každou úlohu Map/Reduce: stav (nečinná/pokračující/ukončená)
  + Pro každou úlohu, která není v klidu: identita pracovního stroje
  + Pro každou dokončenou úlohu Map: umístění a velikosti R oblastí mezisouborů
* R je často omezován uživateli -> Výstup každé úlohy Reduce končí v samostatném výstupním souboru
* **Praktické doporučení (Google):** Zvolte M tak, aby každá jednotlivá úloha měla zhruba 16–64 MB vstupních dat, Udělejte R malým násobkem počtu pracovních strojů, které očekáváme, že budeme používat

**Nevýhody principu MapReduce:**

1. Princip MapReduce je vhodný pouze pro specifickou množinu úloh. Ne vždy je vhodné nebo možné daný problém řešit právě tímto způsobem.
2. Přes veškeré optimalizace může být efektivita zpracování dané úlohy velmi nízká

**Hadoop MapReduce**

* MapReduce vyžaduje: distribuovaný file systém, Engine, který dokáže distribuovat, koordinovat, monitorovat a shromažďovat výsledky
* Hadoop: HDFS + JobTracker + TaskTracker
  + JobTracker (master) = plánovač, TaskTracker (slave) - je přiřazen Map nebo Reduce (nebo jiné operace)
    - Map nebo Reduce běží na uzlu → stejně tak TaskTracker, každá úloha běží na vlastním JVM
* **Jobtracker (Master)**
  + 1. Klientská aplikace je odeslána do nástroje JobTracker
  + 2. Ten "hovoří" s uzlem NameNode (HDFS master) a lokalizuje TaskTracker (klient Hadoopu) v blízkosti dat.
  + 3. Přesune práci do vybraného uzlu TaskTrackeru
* **TaskTracker (klient)**
  + Přijímá úkoly z nástroje JobTracker – Map, Reduce, Kombinovat, ... Vstupní, výstupní cesty
  + Má několik slotů pro úlohy -> Sloty pro provádění úloh dostupné na stroji (/ strojích ve stejném racku)
  + Spouští samostatný JVM pro provádění úlohy
  + Uvádí počet dostupných slotů prostřednictvím zprávy hearbeat pro JobTracker - neúspěšná úloha je znovu provedena JobTracker

**Apache Spark**

* Jednotný analytický engine pro zpracování rozsáhlých dat (datová analýza)
* Běží na clusteru uzlů

**Aplikace Spark = řídicí program**

* = framework pro masivně distribuované výpočty nad daty v paměti, který může zdrojová data přebírat mimo jiné z HDFS nebo právě ze systémů HBase a Cassandra
* Spouští hlavní funkci uživatele, provádí paralelní operace na clusteru
  + Nezávislá sada procesů, koordinováno objektem SparkContext v programu ovladače
* SparkContext se může připojit k několika typům správců clusteru
  + Ty rozdělují prostředky mezi aplikace
* Když je připojen:
  1. Spark získává vykonavatele na uzlech v clusteru
     + Procesy, které provádějí výpočty a ukládají data pro aplikaci
  2. Odešle kód aplikace vykonavatelům
     + Definováno soubory JAR nebo Python předanými do SparkContext
  3. Posílá tasky ke spuštění vykonavatelům

**Základní principy NoSQL databází**

* Hlavní cíl: implementovat distribuovaný stav
  + Různé objekty uložené na různých serverech.
  + Stejný objekt replikovaný na různých serverech
* Hlavní myšlenka: vzdát se některých vlastností ACID (transakční zpracování dotazů) -> zlepšení výkonu
* Jednoduchý interface:
  + Zápis (=Put): potřebuje zapsat všechny repliky
  + Čtení (=Get): může získat pouze jednu repliku.
* Silná konzistence → případná konzistence

1. **Škálovatelnost – jak pracovat s čím dál většími daty bez ztráty výkonu**
2. **CAP teorém**
3. **Distribuční modely – sharding, replikace, konzistence – jak distribuovaně zpracovávat data**

**Škálovatelnost**

* flexibilně reagovat na měnící se požadavky, v našem případě zejména na zvyšující se objemy dat a zátěž systému
* **Vertikální (scaling up)**
  + Tradiční databázové systémy – silná konzistence, větší a výkonnější HW
  + Vendor lock-in (proprietární uzamčení, uzamčení poskytovatelem)
    - Výkonný hardware pro servery vyrábí pouze omezené množství firem -> často jejich proprietární řešení
    - -> Zákazník nucen další výkonnější prvky nakupovat stále u stejného výrobce -> závislost
  + **Nevýhody**: vyšší náklady, omezení nárůstu dat (i nejvýkonnější server má hranice), nutnost proaktivního přístupu (při implementaci je potřeba plánovat maximální velikost dat a odpovídající HW)
* **Horizontální (scaling down)**
  + Distribuce problému na více uzlů -> odpadají všechny tři hlavní nevýhody
  + Bylo by to ideální řešení kdyby: síť byla spolehlivá, latence nulová, šířka pásma neomezená, komunikace po síti bezpečná, neměnící se topologie sítě, síť má jediného administrátora, náklady na přenos dat jsou nulové, a síť je homogenní…

**Konzistence – ACID**

* Očekáváme od databázových transakcí (= jednotka práce v DBMS, sekvence logicky souvisejících operací)
* **Atomicita –** nedělitelnost = buď transakce proběhne celá, nebo nic
* **Consistency –** existuje jedna aktuální verze dat – po updatu všechny nody obsahují stejná data
* **Izolovanost** – skrytí operací pobíhajících uvnitř transakce před ostatními běžícími transakcemi
* **Durability** (trvalost) - zajištění, že se výsledky úspěšně provedených transakcí skutečně do databáze uloží
* Pro distribuované systémy moc drahé -> **CAP teorém**
* **Silná vs. Občasná konzistence**
  + Myšlenka občasné konzistence vychází z toho, že pro mnoho aplikací je rychlost důležitější než dokonalá konzistence ( „aktuálnost“ dat). Každá forma vynucené konzistence bude zpomalovat načtení dat

**CAP teorém**

1. **Consistency** – existuje jedna aktuální verze dat – po updatu všechny nody obsahují stejná data
2. **Availability** (dostupnost) **-** Všechny požadavky na čtení, resp. zápis dat jsou vždy systémem obslouženy
3. **Partition tolerance** (odolnost vůči rozpadu sítě) - distribuovaný systém funguje, i když se rozpadne na několik izolovaných částí

* **Teorém**: v (distribuovaném) systému jsme schopni dosáhnout vždy maximálně 2 z těchto vlastností současně
* **Alternativa k ACID** je ve světě distribuovaných databází **BASE**
  + **Převážná dostupnost** (**B**asically **A**vailable) - Systém je převážně dostupný po celou dobu. Mohou se vyskytnout částečné výpadky, ale nikdy nedojde k výpadku celého systému
  + **Volný stav** (**S**oft state) - Systém je dynamický a nedeterministický, neustále dochází ke změnám
  + **Občasná konzistence** (**E**ventual consistency): Čas od času je systém uveden do konzistentního stavu, nicméně konzistenci nemáme zaručenou neustále.

**Distribuce**

* 2 ortogonální techniky distribuce dat, které různě kombinujeme:
  1. Rozdělení (sharding) = rozmístění různých částí dat (shards) na různé uzly v clusteru -> zvýšení kapacity systému
  2. Replikaci = vytvoření kopií dat na více uzlech v clusteru (master-slave/peer-to-peer) -> zvýšení dostupnosti a propustnosti sytému

**Distribuční modely**

* **Single server** – nemá cenu pak používat NoSQL (!!! Grafová databáze – graf je vždy kompletní)
* **Sharding**
  + pro horizontální škálování, musíme data rozdělit na vhodné (ne nutně disjunktní) podmnožiny a uložit je na různé uzly v clusteru
  + Uživatelé nepřistupují k jedinému serveru, ale každý k jinému uzlu, podle toho, jaká data potřebují
  + Kompromis mezi:
    - 1. Rovnoměrné rozmístění dat na uzlech
    - 2. Minimalizace počtu uzlů, na které musí uživatel přistoupit, aby požadovaná data získal
    - 3. Optimalizace fyzického rozmístění dat vzhledem k jejich geografické příslušnosti (např. měst)
  + Většina NoSQL databází řeší rozdělení dat automaticky (má tzv. auto-sharding)
  + Výpadek sítě -> k dané části dat ztrácíme přístup -> častá kombinace s replikací
* **Master-slave replikace**
  + Předpokládáme, že máme data replikována na určité podmnožině uzlů
  + Jeden z uzlů určen jako **primární = master**, ostatní uzly jsou **sekundární** = **slaves**
  + **Čtení** může obsloužit **kterýkoli z uzlů**, který má požadovaná data k dispozici, **zápis** řeší jen **master** a o změně pak informuje sekundární uzly
  + Vhodná pro data, jež jsou především čtena a minimálně modifikována
  + Zvyšující se požadavky na čtení -> zvýšíme počet replik / sekundárních uzlů
  + Selže-li master -> mohou slaves dále zpracovávat čtení -> mají kopie masteru -> můžou se stát masterem
  + Nového mastera určí buď správce systému, nebo si ho uzly v clusteru zvolí na základě daného algoritmu
* **Peer-to-peer replikace**
  + Master-slave -> problém s úzkým hrdlem mastera -> malá míra škálovatelnosti zápisů
  + Všechny uzly jsou si rovny -> mohou obsluhovat požadavky na čtení i na zápis
  + V případě výpadku některého z uzlů neztrácíme možnost čtení ani zápisu dat
  + Problém s udržení konzistence dat -> write-write konflikt
  + **2 způsoby řešení konfliktů**
    - při každém zápisu dat se uzly v síti koordinují (domluví), což ale výrazně zvyšuje nároky na komunikaci a zpomaluje systém
    - „rozhoduje většina“ – realizuje se prostřednictvím kvóra = stanovujeme minimální počet uzlů, na nichž musí operace proběhnout, abychom ji mohli prohlásit za úspěšnou
* **Kombinace replikace a shardingu**
  + **master-slave replikace + sharding** -> máme v clusteru více uzlů master, ale pro konkrétní data existuje master vždy pouze jeden, zatímco současně pro jiná data může být slave
  + **peer-to-peer replikace + sharding -**> např. při základním replikačním faktoru 3 každá část dat uložena na třech uzlech v clusteru, přičemž jednotlivé uzly obsahují různé části dat – column-family databáze

**Techniky a technologie** **pro zpracování Big Data**

* Co chceme dělat s Big Data? Agregace, manipulace, analýza, vizualizace

**Techniky** **pro zpracování Big Data**

* **Učení asociačních pravidel** – objevování zajímavých vztahů, tj, "asociačních pravidel" mezi proměnnými v rozsáhlých databázích -> např. analýza tržního koše
* **Klasifikace** – určení kategorií, do kterých patří nové datové body, na základě trénovací množiny obsahující datové body, které již byly kategorizovány -> Supervised learning **-** např. nákupní rozhodnutí
* **Cluster analýza** – klasifikace objektů, která rozdělí různorodou skupinu na menší skupiny podobných objektů
  + Unsupervised learning
* **Slučování a integrace dat, Zpracování signálů**
* **Crowdsourcing** – shromažďování údajů předložených velkou skupinou lidí nebo komunitou.
* **Data mining** – získávání vzorů z velkých souborů dat.
  + Zahrnuje učení asociačních pravidel, shlukovou analýzu, klasifikaci, regresi, ...
* **Analýza časových řad a prognózování -** např. hodinová hodnota burzovního indexu.
* **Analýza sentimentu** – identifikace vlastnosti/aspektu/produktu, o kterém je vyjadřován sentiment,
  + určení typu (tj. pozitivní, negativní nebo neutrální).
  + určení stupně a síly sentimentu
* **Analytické zpracování –** OLAP, reportování, přehledové zobrazení (dashboarding), prediktivní modelování
* **Vizualizace –** problém s Linked data, jak vizualizovat vzájemné propojení (grafy -> omezený interface)

**Technologie pro zpracování Big Data**

**Cloud computing**

* = způsob vytváření softwaru, který je založen na myšlence nabízet externím zákazníkům prostřednictvím internetu škálovatelné IT technologie (kombinace HW/SW) v podobě služeb
* Tři modely:
  + **Software as a Service (SaaS**) - služba, kterou využívají přímo koncoví uživatelé aplikace
  + **Platform as a Service (PaaS)** - služba, která je určena vývojářům, pro něž nabízí sadu nástrojů pro vývoj aplikací, nebo obecně pro provozování vlastního softwaru
  + **Infrastructure as a Service (IaaS)** - služba, která slouží pro poskytování infrastruktury, tj. obvykle robustního či jinak ne snadno dostupného hardwaru, typicky formou virtualizace
* Uživatel platí za jejich používání („pronájem“)
* Z hlediska cílové množiny uživatelů rozdělujeme cloudy na:
  + **veřejné** – dostupné komukoli, v nichž dochází ke sdílení technologií
  + **privátní** – provozované určitou organizací pro vnitřní účely nebo výhradně pro vybrané zákazníky
  + **komunitní** – určené pouze pro určitou vybranou skupinu uživatelů (komunitu)
* **Výhody:**
  + Uživateli **nemusí spravovat** (pořizovat, instalovat, upgradovat atd.) příslušné technologie, jen je využívá
  + Díky internetu může uživatel služby cloudu využívat **odkudkoli**
  + Poskytovatel služby je schopen nabízet zákazníkům různě robustní řešení odpovídající konkrétním požadavkům -> automaticky (ne zdarma) zajištěná škálovatelnost pro měnící se požadavky různých apps
  + Data uložena na serveru/serverech cloudu -> zjednodušuje se problematika sdílení dat, které lze zajistit vhodným zpřístupněním dat vybraným uživatelům cloudu
* **Nevýhody:**
  + Data ve veřejném cloudu nemáme pod kontrolou -> možné zneužití dat – ale u solidních poskytovatelů lze očekávat vysokou míru zabezpečení
  + Problém proprietárního uzamčení (vendor lock-in), které nedovoluje snadný přechod na platformu jinou
  + Ceny za poskytování služeb mohou být nad možnosti menších firem či akademických institucí
* **Cloud computing a Big Data**
  + efektivní zpracování Big Data vyžaduje **cluster uzlů**
    - nákup, instalace a údržba clusteru je velmi drahá a náročná -> použijeme cloudy
  + škálovatelné řešení bez nutnosti spravovat konkrétní hardware nebo software
  + pro Big Data je pronájem infrastruktury v cloudu často levnější než její nákup a údržba
  + Můžeme se zaměřit na konkrétní funkce, např. efektivní analytické zpracování dat
  + Ale: další nevýhody (bezpečnost, vendor lock-in) zůstávají
* **Core**: Key-value, Document databáze, Column-family (column-oriented/columnar) stores, Graph databáze
* **Non-core:** Object databáze, XML databáze, …
* **Further novel extensions**: Multi-model databáze, Array databáze, NewSQL databáze

**Databáze klíč-hodnota (key-value)**

**Key-value store**

* Nejjednodušší NoSQL úložiště
* Asociativní pole (map) nebo hashovací tabulku ukládající hodnoty podle unikátního klíče
* Jako tabulka v RDBMS s dvěma sloupci -> id = key, content = value
* Základní operace:
  + vložení hodnoty pro daný klíč (operace PUT)
  + získání hodnoty pro daný klíč (operace GET)
  + smazání dvojice klíč-hodnota (operace DELETE)
* + jednoduchý – skvělý výkon, snadno škálovatelný
* - jednoduchý – není určen pro složité dotazy, potřebuje agregaci
* Např. Riak, Redis
* **Vhodné případy použití**
  + Ukládání informací o relaci
    - Každé webové relaci je přiřazena jedinečná hodnota session\_id
    - Vše o relaci ukládáme pomocí jediného požadavku PUT nebo načítáme pomocí jediného GET
    - Rychle, vše je uloženo v jediném objektu
  + Uživatelské profily, předvolby
    - Každý uživatel má jedinečné user\_id, user\_name + preference (např. jazyk, barva, časové pásmo, ke kterým produktům má uživatel přístup, ... )
    - Stejně jako v předchozím případě: Rychle, jeden objekt, jeden GET/PUT
  + Údaje o nákupním košíku -> Podobně jako v předchozích případech
* **Kdy Nepoužívat**
  + Vztahy mezi různými soubory dat
    - Některá úložiště klíč-hodnota poskytují funkce procházení odkazů -> není obvyklé
  + Multioperační transakce – ukládání více klíčů
    - Neuloží-li se některý z nich → vrátit nebo vrátit zpět zbytek operace
  + Dotazování podle dat -> Vyhledávání klíčů na základě něčeho nalezeného v hodnotové části
  + Operace podle množin -> Operace jsou omezeny vždy na jeden klíč, nelze operovat s více klíči najednou
* **Dotazování** 
  + Dotazujeme se pomocí klíče, dotazovat se pomocí nějakého atributu sloupce value je (obvykle) nemožné
    - Musíme si přečíst hodnotu, abychom zjistili, zda atribut splňuje podmínky
  + Co když klíč neznáme?
    - Některé systémy umožňují získat seznam všech klíčů -> drahé
  + Některé podporují vyhledávání uvnitř hodnoty
    - Pomocí full-text indexu -> data musí být nejprve indexována – vyhledávání v Riaku
  + Jak navrhnout klíč -> pomocí algoritmu / poskytnutý uživatelem / odvození z time-stamps (jiných dat)
  + Typičtí kandidáti na uložení: data relace (s ID relace jako klíčem), data nákupního košíku (ID uživatele), uživatelské profily (ID uživatele), ...
  + Vypršení platnosti klíčů -> po uplynutí daného čas. intervalu, hodí se pro objekty relace/nákupního košíku

**Riak**

* Open-source, distribuovaná databáze, vestavěná podpora MapReduce
* Ukládá klíče do bucketů = namespace pro klíče
  + jako tabulky v RDBMS, adresáře v file systému, sada společných vlastností pro jeho obsah (počet replik…)
* **Použití**
  + HTTP -> default interface, používáme curl umožňující HTTP komunikaci s jiným serverem
  + GET (retrieve), PUT (update), POST (create), DELETE (delete)
  + Klíče a buckety -> klíče jsou uloženy v bucketech (s vlastnostmi n\_val = replikační faktor, …), pokud je klíč uložen do neexistujícího bucketu, bucket se vytvoří, klíče jsou user-specified / generované Riakem
  + Paths: /riak/<bucket> (daný bucket), /riak/<bucket>/<key> (klíč v bucketu)

Vrátí všechny buckety: *curl http://localhost:10011/riak?buckets=true*

Vrátí vlastnosti bucketu foo: *curl http://localhost:10011/riak/foo/*

Vrátí všechny klíče v bucketu foo: *curl http://localhost:10011/riak/foo?keys=true*

Změní vlastnosti bucketu foo: *curl -X PUT http://localhost:10011/riak/foo -H "Content-Type: application/json" -d '{"props" : { "n\_val" : 2 } }'*

Uloží plain text do bucketu foo pomocí vygenerovaného klíče: *curl -i -H "Content-Type: plain/text" -d "My text"* [*http://localhost:10011/riak/foo/*](http://localhost:10011/riak/foo/)

Uloží JSON do bucket artists s klíčem Bruce: *curl -i -H "Content-Type: application/json" -d '{"name":"Bruce"}'* [*http://localhost:10011/riak/artists/Bruce*](http://localhost:10011/riak/artists/Bruce)

Update objektu: *curl -i -X PUT -H "Content-Type: application/json" - d '{"name":"Bruce", "nickname":"The Boss"}' http://localhost:10011/riak/artists/Bruce*

Delete objektu: *curl -i -X DELETE* [*http://localhost:10011/riak/artists/Bruce*](http://localhost:10011/riak/artists/Bruce)

* **Links** – vytváření vztahů mezi obejkty
  + **Přidání k album link na interpreta**: *curl -H "Content-Type: text/plain" -H 'Link: ; riaktag="performer"' -d "The River"* [*http://localhost:10011/riak/albums/TheRiver*](http://localhost:10011/riak/albums/TheRiver)
  + Nalezení umělce, který je interpretem alba The River: *curl -i http://localhost:10011/riak/albums/T heRiver/artists,performer,1*
  + \_ - wild card
  + 1 – zahrnout tento krok do výsledku / 0 – nezahrnout krok do výsledku
  + Kteří umělci spolupracovali s umělcem, který vystoupil s písní The River: *curl -i http://localhost:10011/ riak/albums/TheRiver/ar tists,\_,0/artists,colla borator,1*
* **Riak Search**
  + Distribuovaný, full-text search engine
  + Poskytuje nejpokročilejší možnosti dotazování vedle MapReduce
  + Data musí být indexovaná: Přečteme dokument -> rozdělíme dokument na jeden nebo více fieldů -> rozdělíme fieldy na jeden nebo více termů -> normalizujeme každý term ve fieldu -> zapíšeme {field, term, DocumentId} jak index
  + Indexing: index <INDEX> <PATH>, Searching: search <INDEX> <QUERY>
  + Dotazy: bus\*, bus? -> wildcards, range: {red TO rum} mezi red a rum, [red TO rum] mezi red a rum včetně, AND, OR, NOT
* **Transakce v Riaku:** ne ACID, ale BASE (Basically Available, Soft state, Eventually consistent)
  + Využívá peer-to-peer replikaci spolu s **kvórem**
  + Umožňuje systému zapisovat, resp. číst data i pokud je N − W resp. N − R uzlů nefunkčních

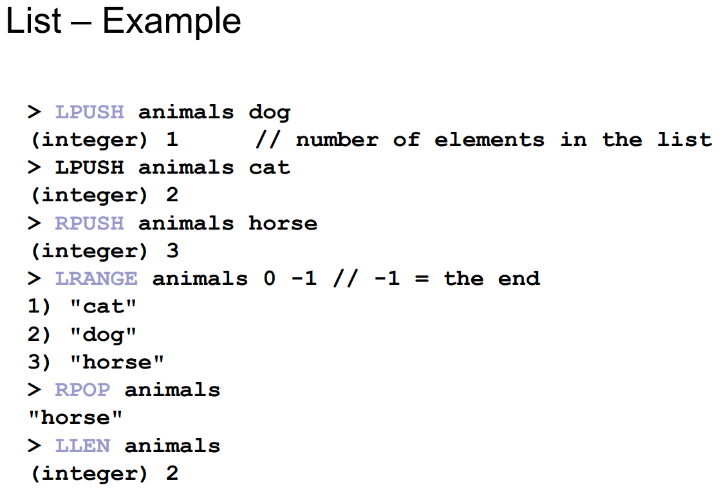
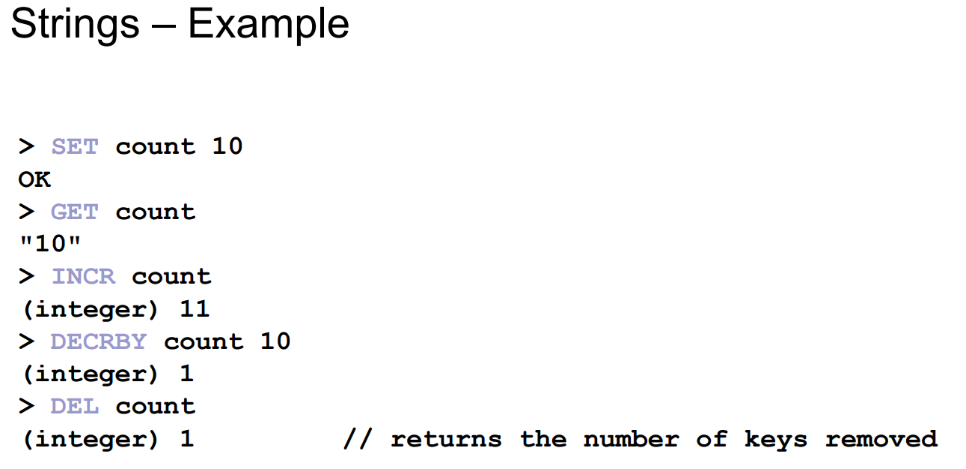
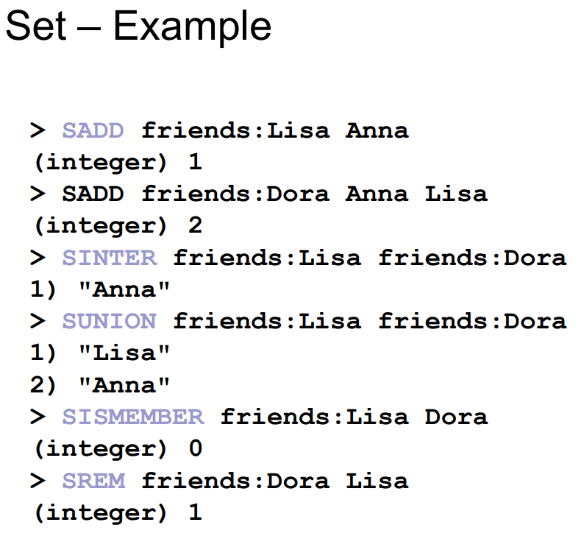
Obsah obrázku text, diagram, kruh, řada/pruh

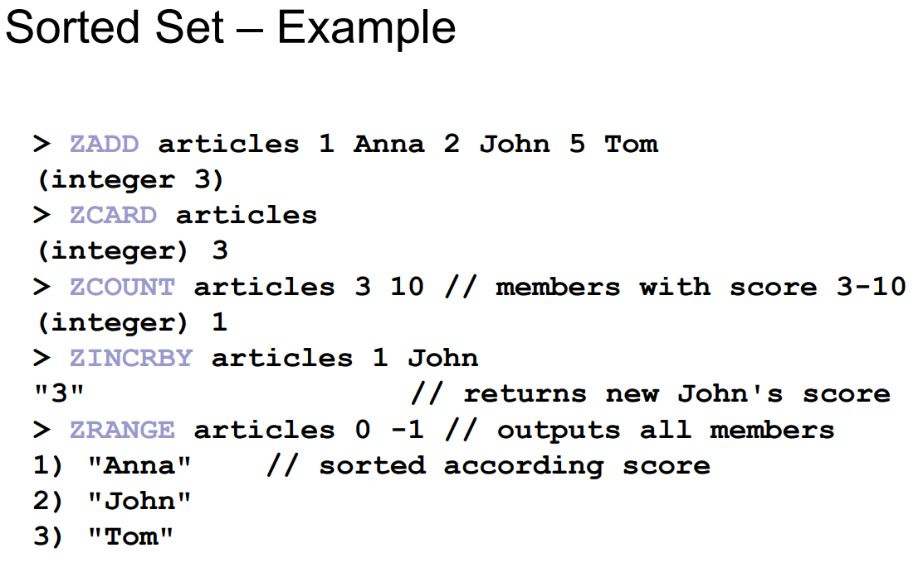
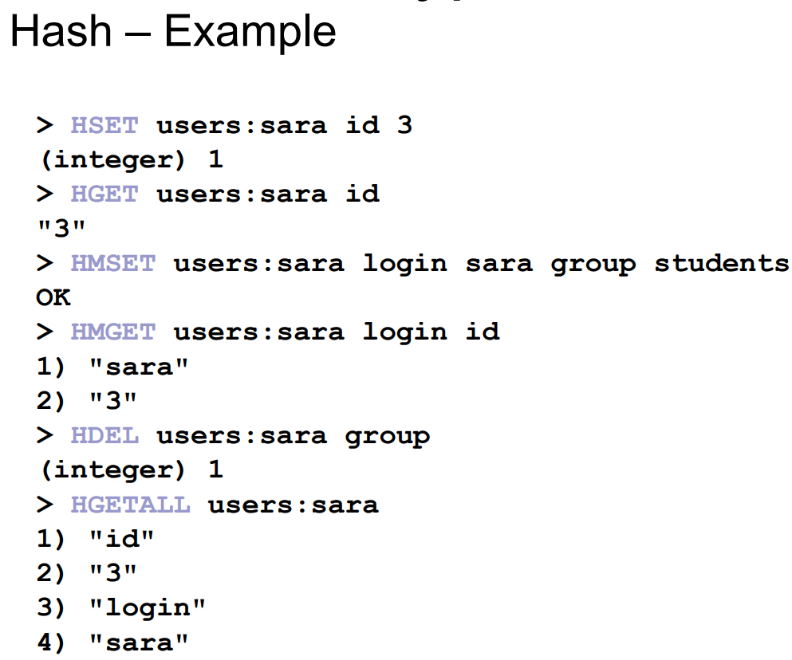
Popis byl vytvořen automaticky**Clustering v Riaku**

* Žádný master – každý uzel schopen obsloužit požadavek klienta
* **Princip konzistentního hashování (consistent hashing)**
  + Každý uzel zodpovědný za souvislý interval hashovacích klíčů
  + Každému uzlu je přiřazen hashovací klíč ze stejné domény, kterou má hashovací funkce na klíčích
  + Každý uzel pak spravuje všechny klíče v intervalu mezi klíčem předcházejícího uzlu a svým klíčem
  + Pro rozšíření infa o změně v množině uzlů se používají tzv. **gossip protokoly** -> každý uzel v pravidelných časových intervalech vybere náhodně jeden ze svých známých uzlů, který kontaktuje a předá mu „novinky“
  + Slabina -> nerovnoměrné rozložení dat mezi spolupracující uzly – hashovací funkce nedokáže zaručit, že intervaly mezi uzly budou stejně dlouhé
* Obsah obrázku text, kruh, snímek obrazovky, diagram

  Popis byl vytvořen automaticky**Konceptu virtuálních uzlů**
  + doména hašovací funkce je rozdělena na fixní počet stejně velkých intervalů neboli virtuálních uzlů
  + Tyto intervaly jsou postupně přiřazovány participujícím fyzickým uzlům (serverům), které jsou v obrázku vyznačeny různým podbarvením. Jak je vidět i z obrázku, při použití této techniky mohou být data na fyzické servery rozdělena rovnoměrně

**Redis**

* Nejedná se o standardní key-value features (spíše o druh dokumentové databáze):
  + Klíče jsou binárně bezpečné = klíčem může být jakákoli binární posloupnost
  + Uloženou hodnotou může být libovolný objekt: řetězce, hashe, seznamy, množiny a setříděné množiny
  + Lze provádět operace typu range, diff, union, intersection, ...
* Hybridně persistentní databáze: výchozím úložištěm je **operační paměť (RAM)** -> masivní propustnost operací (throughput) -> pravidelně ukládána na disk pro zvýšení odolnosti (resiliency)
* **Transakce v Redisu** -> každý příkaz je atomický – podpora transakcí při použití více příkazů
  + Příkazy prováděné ve frontě, I když příkaz selže, všechny ostatní příkazy ve frontě jsou zpracovány
  + Dva druhy chyb: Příkaz může být nezařazen do fronty / Příkaz může selhat po volání EXEC
* **Master-slave replikace**
  + Master může mít více podřízených zařízení, Slave může sloužit jako master pro jiné slave
  + Slave se mohou automaticky znovu připojit, když spojení master-slave z nějakého důvodu vypadne
  + **Replikace na master straně neblokující** -> master pokračuje v obsluze dotazů, když se slave synchronizuje
  + **Replikace na slave straně neblokující -**> Zatímco slave provádí synchronizaci, může odpovídat na dotazy pomocí staré verze dat, volitelně může v případě potřeby blokovat
    - Nastává okamžik, kdy je třeba smazat starou datovou sadu a načíst novou = blokování



**Sloupcové databáze (column-family)**

**Column-family stores -** HBase, Cassandra, SimpleDB, …

* = řádky mají mnoho sloupců asociovaných s klíčem řádku (row key)
* Každý sloupec v daném řádku má column name, column value a také time stamp, kdy byla hodnota uložena
* Column family = kolekce podobných řádků – řádky nemusí mít stejné sloupce
* Druhý význam – ukládání tabulek do sloupců místo řádků
* **Vhodné případy použití:**
  + **Event logging – můžeme** ukládat libovolné datové struktury – ideální pro ukládání informací o eventech
  + **Systémy pro správu obsahu, blogovací platformy**
    - V různých sloupcích ukládáme blogové příspěvky s tagy, kategoriemi, odkazy a zpětnými odkazy
    - Komentáře lze ukládat: do stejného řádku / přesunout je do jiného keyspacu
    - Uživatele blogů a vlastní blogy lze umístit do různých skupin sloupců
* **Kde NEpoužívat:**
  + Systémy vyžadující transakce ACID -> column-family stores nejsou jen speciálním druhem RDBMS s proměnlivou množinou sloupců
  + Agregace dat pomocí dotazů (např. SUM nebo AVG) - musí být (obvykle) provedena na straně klienta
  + Pro rané prototypy -> nevíme, jak se budou měnit vzory dotazů -> musíme měnit column-family design

Obsah obrázku text, snímek obrazovky, účtenka, Písmo

Popis byl vytvořen automaticky**Cassandra**

* Sloupcová databáze vytvořená původně interně ve společnosti Facebook, Java
* Column = základní jednotka, skládá se z dvojice název-hodnota – název slouží jako klíč
  + Ukládá se s časovým razítkem
* Row = kolekce sloupců spojených s klíčem
* Column family = kolekce podobných řádků – řádky nemusí mít stejné sloupce
* **Column-family vs. Relace**
  + Nemusíme modelovat všechny sloupce předem – obvykle předpokládáme podobné sady sloupců
  + Žádné formální cizí klíče: Dotaz -> předem vypočítat / použít sekundární index

**Sloupce**

* Sloupec je nejmenší přírůstek dat = Název + hodnota (může být prázdná) + časové razítko
* Lze indexovat na základě jejich názvu -> použití sekundárního indexu
  + Primární index = klíč řádku – zajišťuje jedinečnost, urychluje přístup, může ovlivnit pořadí ukládání
* **Typy**:
  + **Expirující** – s volitelným datem expirace zvaným TTL, Lze se dotazovat
  + **Counter** – pro uložení čísla, které inkrementálně počítá výskyty určité události nebo procesu
    - Např. pro počítání počtu zobrazení stránky, Interně zajišťuje konzistenci ve všech replikách
  + **Super** – přidání další úrovně vnoření = Seskupení více sloupců na základě společné hodnoty vyhledávání
    - Má název a hodnotu zahrnující mapu sloupců

**Column families**

* Může definovat metadata o sloupcích
  + Skutečné sloupce řádku určuje klientská aplikace, každý řádek může mít jinou sadu sloupců
* **Statické** – podobné tabulce relační databáze
  + Řádky mají stejnou sadu sloupců, není nutné mít definovány všechny sloupce
* **Dynamické** – využívají možnosti Cassandry používat libovolné názvy sloupců zadané aplikací

*// rodina sloupců "users"*

*{*

*// řádek s klíčem "honza"*

*"honza": {*

*"firstName": {"142926822455": "Jan"},*

*"lastName": {"142926822455": "Novák"},*

*"email": {"142926822643": "honza@seznam.cz”}*

*},*

*"janicka": {*

*"firstName": {"142926845345": "Jana"},*

*"lastName": {"142926845345": "Novotná"},*

*"web": {"142926845345": "http://jnovotna.cz/"}*

*}*

*}*

* + Předem vypočtené sady výsledků, uloženo v jednom řádku pro efektivní vyhledávání dat
  + Řádek = snímek dat, která vyhovují danému dotazu
* Musí zadat klíč, lze zadat datové typy sloupců a specifikovat možnosti

*CREATE COLUMNFAMILY Fish (key blob PRIMARY KEY);*

*CREATE COLUMNFAMILY FastFoodEatings (user text PRIMARY KEY)*

*WITH comparator=timestamp AND default\_validation=int;*

*CREATE COLUMNFAMILY MonkeyTypes (*

*KEY uuid PRIMARY KEY,*

*species text,*

*alias text,*

*population varint*

*) WITH comment='Important biological records'*

*AND read\_repair\_chance = 1.0;*

* **Komparátor** = datový typ pro název sloupce
  + V řádku sloupce uloženy seřazené podle názvu sloupce
  + Statické column family: Typicky řetězce, na pořadí nezáleží
  + Dynamické column family -> pořadí je obvykle důležité (např. časové značky)
* **Validátor** = datový typ pro hodnotu sloupce / klíče řádku
  + Definujte výchozí validátor klíče řádku pomocí vlastnosti **key\_validation\_class**
  + Statické rodiny sloupců -> definujte každý sloupec a jeho přidružený typ
  + Dynamické rodiny sloupců -> názvy sloupců nejsou známy dopředu, specifikujte **default\_validation\_class**
* Datové typy nemusí být definovány
  + Výchozí hodnota: ByteType, tj. libovolné hexadecimální bajty
* Základní operace: GET, SET, DEL
* Z nových verzí Cassandry a CQL: nová strategie
  + Možnosti však zůstávají stejné, tj. stále můžeme vytvářet tabulky s libovolnými sloupci

**CQL – nový přístup**

* Cassandra query jazyk, SQL-like příkazy (CREATE, ALTER, UPDATE, DROP, DELETE, …), jednodušší než SQL
* Jiný přístup než column families (od CQL 3 říkáme table)
* **CREATE KEYSPACE Excelsior WITH replication = {'class': 'SimpleStrategy', 'replication\_factor' : 3};** 
  + Vytvoření keyspace se zadanou strategií replikace a parametry
  + USE Excelsior; -> použítí keyspacu
* **CREATE TABLE** timeline (userid uuid, posted\_month int, posted\_time uuid, body text, posted\_by text, **PRIMARY** **KEY** (userid, posted\_month, posted\_time)) **WITH** compaction = {'class': 'LeveledCompactionStrategy'};
  + Vytvoření tabulky s názvem, sloupci a dalšími možnostmi
  + Primární klíč je povinný
    - Partition key = první sloupec (nebo sada sloupců, pokud je v závorce)
      * Záznamy jsou uloženy na stejném uzlu
    - Clustering columns = Určení clusteringu na oddíl, tj. pořadí pro fyzické uložení řádků v oddílu!!!
* **Kolekce - set<text>, list<text>, map<text,text>**
  + **List** – […] - uspořádaný seznam prvků, stejná hodnota vícekrát, vrací seřazené podle indexu
  + **Set** – { ….} - soubor jedinečných hodnot, při dotazu se vrací v abecedním pořadí
  + **Map** – {… : …, … : … } - dvojice název + hodnota
  + Každý prvek je interně uložen jako jeden sloupec Cassandra
* **Práce s table**
  + DROP TABLE timeline; //smaže tabulku včetně všech dat,
  + TRUNCATE timeline; //smaže všechny data v tabulce
  + CREATE INDEX userIndex ON timeline (posted\_by); //vytvoří index, DROP INDEX userIndex;
* **Dotazování –** Žádné joiny, jen jednoduché podmínky
  + SELECT \* FROM users WHERE firstname = 'jane' and lastname='smith' ALLOW FILTERING;
  + SELECT \* FROM emp WHERE empID IN (130,104) ORDER BY deptID DESC;
  + SELECT select\_expression FROM keyspace\_name.table\_name WHERE relation AND relation ... GROUP BY columns ORDER BY ( clustering\_key ( ASC | DESC )...) LIMIT n ALLOW FILTERING
  + Agregační funkce – min, count, max, sum, avg

**Distribuce a replikace**

* **distribuovaná architektura, peer-to-peer replikace**
* rozdělují data mezi jednotlivé uzly podle dělicího klíče (partition key)
* Primárním přístupem k distribuci dat není hašovací funkce, ale intervalové dělení
* různé názvy pro vzniklé části tabulky: tablets (Bigtable), regions (HBase) nebo partitions (Cassandra)
* Pokud není určeno jinak, partition key tabulky tvoří první název sloupce z primárního klíče tabulky
* Primární klíč složen z více sloupců -> názvy ostatních sloupců za partition key jako tzv. clustering columns -> v rámci každého uzlu záznamy fyzicky uloženy seřazeně podle hodnot těch sloupců -> zefektivnění práce s dotazy

**Lokální organizace dat**

* Úkol lokálního úložiště: persistence dat, trvalost provedených změn, co nejvyšší propustnost W/R operací
  + využívá kombinace technik zápisového logu (write-ahead log) a paměťových tabulek (memtable)
* Na disku jsou data často ukládána ve struktuře zvané SSTable (sorted string table)
  + obsah každé SSTable je totiž vždy neměnný v čase (immutable)
  + Všechny operace modifikující data nejprve zaznamenány do zápisového logu, do kterého se zapisuje pouze na konec, a proto bývá velmi rychlý
  + Změny jsou dále zaznamenány do paměťové struktury memtable, která se uloží na disk do SSTable ve chvíli, kdy je plná
  + Operace čtení berou v úvahu data v paměti i na disku
* Čas od času je v systému spuštěna operace **konsolidace (Compaction Process)**
  + struktury SSTable se nahrají do paměti a data z nich se přeskupí tak, aby bylo čtení co nejefektivnější
  + odstraněny starší hodnoty, které byly nahrazeny novějšími
  + fyzicky odstraněny záznamy, které byly smazány operacemi promítnutými zatím pouze do paměti
* **Tombstone**: Tombstone (hrobka) je záznam v Cassandře, který označuje smazání záznamu. Při provádění operace smazání (Delete) je do memtable a SSTable vložen tombstone místo skutečného záznamu. Tombstone informuje Cassandra, že záznam byl smazán a nebyl přístupný. Tombstone se postupně replikuje mezi všemi uzly v clusteru a po dosažení konzistence je záznam považován za definitivně smazaný.

**Dokumentové databáze (document databases)** – MongoDB, CouchDB

* hlavní koncept jsou dokumenty – ukládání a vyhledávání – XML, JSON
* Dokument = datová struktura, která má samopopisný charakter, tedy obsahuje kromě samotných dat i metadata popisující význam jednotlivých částí datové struktury
* Stromové datové struktury obsahující maps, kolekce a základní datové typy
* Dokumentové databáze používají dokumenty nejen pro ukládání dat, ale i pro komunikaci s klienty a aplikacemi
* Očekává se, že dokumenty v kolekci budou podobné – jejich schéma se může lišit
* Dokumentové databáze ukládají dokumenty jako value části key-value store
* **Vhodné případy použití**
  + **Event Logging -**> mnoho různých aplikací chce zaznamenávat události
    - Typ zaznamenávaných dat se neustále mění, eventy lze dělit podle názvu aplikace / typu
  + **Systémy pro správu obsahu, platformy pro blogování**
    - Správa uživatelských komentářů, uživatelských registrací, profilů, dokumentů na webu, ...
  + **Webová analýza / real-time analýza -**> části dokumentu lze aktualizovat
    - Nové metriky lze snadno přidávat beze změn schématu
  + **E-commerce aplikace** – flexibilní schéma pro produkty a objednávky
* **Kdy NEpoužívat**
  + **Complex Transactions Spanning Different Operations** (Složité transakce zahrnující různé operace)
    - Atomické operace napříč dokumenty, některé dokumentové databáze podporují (např. RavenDB)
  + **Queries against Varying Aggregate Structure** (Dotazy proti proměnlivé agregované struktuře)
    - Návrh agregátu se neustále mění -> potřebujeme ukládat agregáty na nejnižší úrovni granularity

Obsah obrázku text, snímek obrazovky, Písmo, číslo

Popis byl vytvořen automaticky**MongoDB**

* Dokumenty JSON -> dynamická schémata
* Features: Vysoký výkon – indexy, Vysoká dostupnost – replikace + případná konzistence + automatický failover, Automatické škálování – automatický sharding napříč clusterem, Podpora MapReduce
* Každá instance mongoDB má více databází
* Každá databáze může mít více kolekcí
* Když ukládáme dokument, musíme vybrat databázi a kolekci

**Dokumenty**

* Používají JSON, ukládají se v BSON (binární repr. JSON)
* Mají maximální velikost: 16 MB (v BSON)
* Omezení názvů polí:
  + \_id je vyhrazeno pro použití jako primární klíč
    - Unikátní v kolekci, neměnné, jakýkoli jiný typ než pole
  + Názvy polí nesmí začínat znakem $ -> vyhrazeno pro operátory
  + Názvy polí nemohou obsahovat znak . -> vyhrazeno pro přístup k polím

**Datový model**

* Dokumenty mají flexibilní schéma -> kolekce nevynucují strukturu dat, v praxi jsou si dokumenty podobné
* Výzva: balancing – potřeby aplikace, výkonnostní charakteristiky databázového stroje, vzory vyhledávání dat
* Klíčové rozhodnutí: references vs. embedded documents – struktura dat, vztahy mezi daty
* **Reference** 
  + Včetně odkazů z jednoho dokumentu na druhý, normalizované datové modely
  + Odkazy poskytují větší flexibilitu než vložení
  + Používání normalizovaných datových modelů:
    - Pokud by vložení vedlo ke zdvojení dat, které není vyváženo výkonem čtení,
    - K reprezentaci složitějších vztahů m:n, K modelování velkých hierarchických datových sad
  + Nevýhody: Mohou vyžadovat více cest na server (následné dotazy)
* **Embedded data (Vnořená data)**
  + Související data v jedné struktuře dokumentu
    - Dokumenty mohou mít subdokumenty (v poli pole) -> aplikace mohou potřebovat méně dotazů
  + Denormalizované datové modely
  + Umožňují aplikacím získávat a manipulovat se souvisejícími daty v rámci jedné databázové operace
  + Použití embedded datových modelů:
    - Pokud máme mezi entitami vztahy "obsahuje" -> Vztahy 1:1
    - Při vztazích 1:m, kdy se podřízené dokumenty objevují vždy u jednoho nadřazeného dokumentu
  + Lepší výkon při čtení, možnost načítat/aktualizovat související data v rámci jedné databázové operace
  + Nevýhody: Dokumenty se mohou po vytvoření výrazně rozrůst
    - Ovlivňuje výkon zápisu – Dokument musí být přemístěn na disk, pokud jeho velikost přesahuje přidělené místo, Může vést k fragmentaci dat

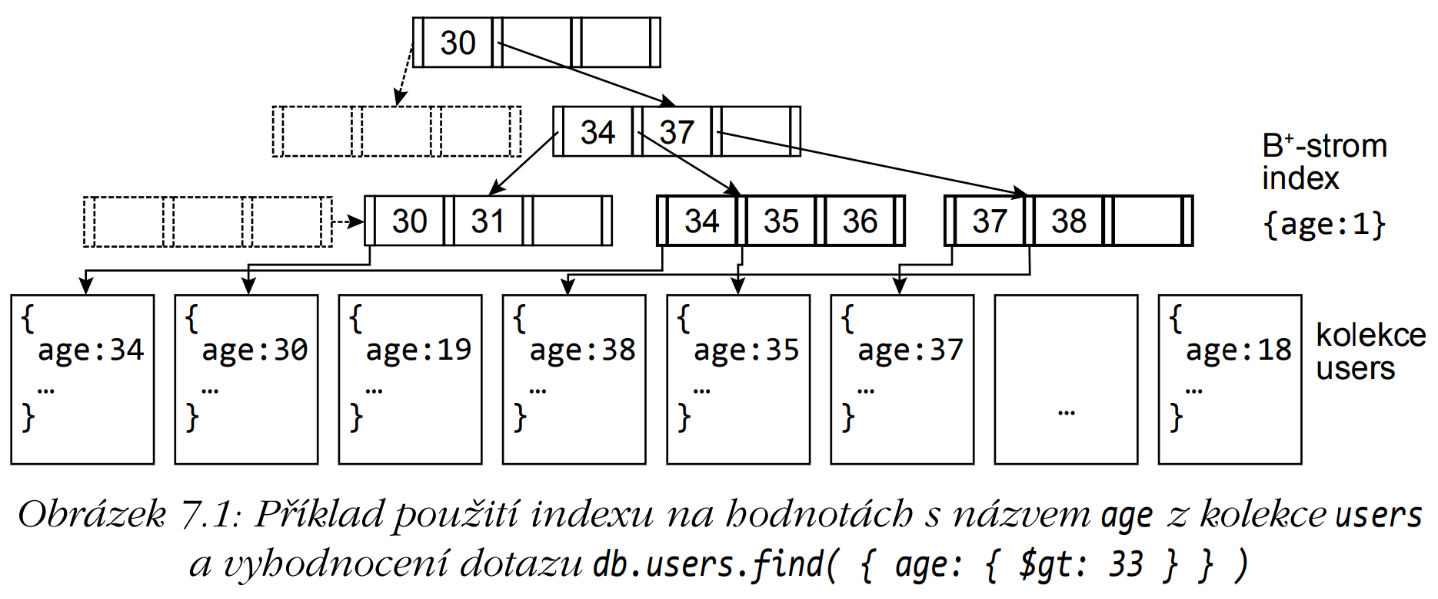
**Modifikace dat**

* Operace: create, update, delete -> Úprava dat jedné kolekce dokumentů
* Pro update / delete: kritéria pro výběr dokumentů, které se mají updatovat / vymazat
* **Vkládání a odstraňování dat**
  + db.inventory.save( { type: "book", item: "notebook", qty: 40 } ) - Vytvoří nový dokument v inventory sbírky, pokud není zadáno \_id nebo ve sbírce neexistuje.
  + db.inventory.remove( { type : "food" } ) - Odstraní z kolekce inventory všechny dokumenty s typem food
  + db.inventory.remove( { type : "food" }, 1 ) - Odstraní z kolekce inventory 1 dokument typu food
* **Data update**
  + db.inventory.update( { type : "book" }, { $inc : { qty : -1 } }, { multi: true } ) - Vyhledá všechny dokumenty s typem book a změní jejich pole qty o -1
  + db.inventory.save( { \_id: 10, type: "misc", item: "placard" } ) - Nahradí dokument s \_id = 10

**Dotazování**

* Zaměřuje se na určitou kolekci dokumentů
* Určuje kritéria, která identifikují vrácené dokumenty
* Může obsahovat projekci, která určuje pole z odpovídajících dokumentů, která se mají vrátit.
* Může stanovit omezení, pořadí řazení, ...
* db.inventory.find( {} ), db.inventory.find() - všechny dokumenty v kolekci
* db.inventory.find( { type: "snacks" } ) - všechny dokumenty, kde pole type má hodnotu snacks
* db.inventory.find( { type: { $in: [ 'food', 'snacks' ] } } ) - všechny dokumenty, kde hodnota pole type je buď food, nebo snacks
* db.inventory.find( { type: 'food', price: { $lt: 9.95 } } ) - všechny dokumenty, kde pole type má hodnotu food a hodnota pole price je nižší než ($lt) 9,95
* db.inventory.find( { type: 'food', $or: [ { qty: { $gt: 100 } }, { price: { $lt: 9.95 } } ] } ) - Všechny doklady, kde hodnota pole type je food a hodnota qty je větší než ($gt) 100 nebo hodnota pole price je menší než 9,95
* **Subdokumenty**
  + db.inventory.find( { producer: { company: 'ABC123', address: '123 Street' } } ) - Všechny dokumenty, jejichž hodnotou pole producer je subdokument, který obsahuje pouze pole company s hodnotou ABC123 a pole address s hodnotou 123 Street, a to v přesném pořadí
  + db.inventory.find( { 'producer.company': 'ABC123' } ) - Všechny dokumenty, kde hodnota pole producer je subdokument, který obsahuje pole společnost s hodnotou ABC123 a může obsahovat další pole
* **Pole subdokumentů:** db.inventory.find( { 'memos.0.by': 'shipping' } ) - Všechny dokumenty, kde pole memos obsahuje pole, jehož prvním prvkem je subdokument s polem by s hodnotou shipping

**Indexy**

* Bez indexů: prohledávání každého dokumentu v kolekci, aby vybrala dokumenty, které odpovídají dotazu
* Indexy uchovávají část datové sady kolekce ve formě, kterou lze snadno procházet
  + Ukládá hodnotu konkrétního pole nebo množiny polí seřazených podle hodnoty pole
  + Struktury podobné B-stromu
* Definovány na úrovni kolekce
* Účel: Zrychlení běžných dotazů, Optimalizace výkonu jiných operací ve specifických situacích
* db.people.ensureIndex( { "phone-number": 1 } ) Vytvoří single-field index na phone-number v kolekci people
* db.products.ensureIndex( { item: 1, category: 1, price: 1 } ) - Vytvoří složený index na item, category a price

Obsah obrázku text, skica, diagram, kresba

Popis byl vytvořen automaticky

**Replikace dat a dostupnost systému**

* Princip master-slave
* **Množina replik** (replica set) = několik instancí MongoDB, které kooperují a obsahují stejná data
  + Jedna z těchto instancí je určena jako master a pouze tato instance může provádět operace zápisu
* Tento princip:
  + zvyšuje propustnost operací čtení (počet obsloužených operací), protože čtení může být realizováno ze všech uzlů v množině replik
  + zvyšuje dostupnost systému a dat při krátkodobém výpadku některého z uzlů, protože klienti mohou dále číst a případně i zapisovat data přes ostatní uzly
  + zvyšuje celkovou odolnost systému vůči ztrátě dat – i při fatálním výpadku některého z uzlů o data nepřijdeme
* Klientské aplikace nekomunikují přímo s jedním konkrétním uzlem, ale s „celou množinou replik“
* Aplikace se nejprve připojí k jednomu uzlu, jehož adresu zná, ale driver může požadovanou operaci přesměrovat (např. operaci zápisu vždy přesměruje na uzel master)
* **CAP teorém**
  + Když master zjistí, že s ostatními nemůže navázat spojení a je „sám“, tak automaticky přestane být masterem a bude dále obsluhovat pouze čtení
  + Naopak zbylé uzly (např. 2) spolu dále komunikují, a protože jich je více než polovina původního počtu, tak mezi sebou zvolí nový master
  + -> systém replikace v MongoDB **preferuje konzistenci dat před dostupností systému**
* Vhodné, aby počet replik byl lichý, pokud je sudý MongoDB nabízí možnost přidat **arbitra**
  + **=** uzel, který nenese žádná data, ale účastní se volby nového uzlu typu master v případě výpadku
* Vlastní replikace dat je realizována pomocí přeposílání **operačního logu (oplog**) z mastera na ostatní
* Můžeme slave nastavit jako **zpožděnou kopii** -> změny z logu se pak na tento uzel aplikují až po uplynutí daného časového intervalu, hodí se např. pro obnovu dat poškozených chybou v aplikaci

**Sharding (rozdělení dat)**

* Můžeme data rozdělit podle hodnot \_id nebo jakéhokoli jiného klíče (sharding key), , který je v rámci kolekce unikátní, neměnný a je na něm postavený index
* Rozdělení dat z kolekce na jednotlivé uzly pomocí rozdělení domény tohoto klíče na intervaly / hašovací funkce
* Umožněno komunikovat s distribuovaným systémem pomocí **směrovače (procesu s názvem mongos**), který dostává informace z konfiguračního serveru o tom, kde jsou jaká data, a který směruje dotazy na příslušné uzly
* **Balancing** = vyrovnávání velikosti dat uložených na jednotlivých uzlech
  + provádí dělení a migraci kusů dat tak, aby objemy dat na jednotlivých uzlech byly co nejvyrovnanější
* Operace CRUD (create, read, update, delete) jsou směrovačem přeposlány na příslušnou část kolekce, případně na několik částí

**ACID pro jednotlivé operace a transakce**

* musíme řešit, jak jsou tyto vlastnosti zajištěny pro jednotlivé operace na replikovaných datech
* pouze jeden master -> zabráněno vzniku write-write konfliktu
* read-write konflikt (čtení zastaralých dat) - Při zápisu si může aplikace zvolit, jestli má systém nahlásit úspěšné ukončení operace hned po zápisu na master, nebo až po rozšíření změny na slaves
  + operace zápisu je pak blokující (můžeme nastavit maximální čas blokování)
* Při čtení můžeme naopak nastavit, jestli se má operace realizovat pouze na uzlu master, nebo jestli se může číst i z ostatních uzlů, které mohou teoreticky obsahovat neaktuální data
* Při zápisu můžeme (globálně / pro jednotlivé operace) nastavit parametr **writeConcern** takto:
  + w=0: operace ukončena hned, bez jakékoli kontroly, jestli změnu systém opravdu proved
  + w=1: operace ukončena po úspěšném provedení změny do paměťového prostoru databáz. systému
  + w=1, j=true: operace ukončena po úspěšném zápisu změny do diskového operačního logu (j = journal)
* Operace zápisu jsou vždy atomické na úrovni jednotlivých dokumentů
* Pokud operace provádí hromadnou změnu několika dokumentů, tak jsou operace atomické pro jednotlivé dokumenty, ale ne jako celek -> můžeme změnit pomocí parametru **$isolated** = zaručí, že dokud celá operace hromadné změny nebude dokončena, tak se na modifikovaných dokumentech žádná jiná operace neprovede
* **Transakčního zpracování** – možnost celou sekvenci operací vrátit (**rollback**), pokud je jedna neúspěšná
  + pomocí algoritmu dvoufázového COMMIT protokolu -> velké zpomalení celého zpracování

**Grafové databáze** – Neo4j, OrientDB

* Ukládání entit a vztahů mezi těmito entitami
  + Uzel je instance objektu, Uzly mají vlastnosti (např. jméno)
  + Hrany mají směrový význam, Hrany mají typy (např. likes, friend, ...)
* Uzly jsou uspořádány podle vztahů -> Umožňují najít zajímavé vzory
* **RDBMS vs. Grafová databáze**
  + Pokud v RDBMS ukládáme strukturu podobnou grafu, je to pro jeden typ vztahu ("Kdo je můj manažer")
    - Přidání dalšího vztahu obvykle znamená změnu schématu, přesun dat atd.
    - V grafových databázích lze vztahy dynamicky vytvářet / odstraňovat – neomezený počet a druhy
  + V RDBMS modelujeme graf předem na základě požadovaného procházení (Traversal)
    - Pokud se Traversal změní, budou se muset změnit i data
    - Obvykle potřebujeme mnoho operací join
  + V grafových databázích se vztahy nepočítají v době dotazu, ale zachovávají se
    - Přesunutí většiny práce při procházení grafu na vkládání, takže dotazy jsou co nejrychlejší

**Charakteristika**

* Uzly mohou mít mezi sebou různé typy vztahů – reprezentovat vztahy mezi entitami domény
  + Mají sekundární vztahy – kategorie, cesty, časové stromy, …
* Počet a druh vztahů, které může uzel mít, není omezen
* Vztahy mají typ, počáteční uzel, koncový uzel, vlastní vlastnosti, např. od kdy se stali přáteli

**Neo4j**

* Ukládá data do uzlů propojených orientovanými, typizovanými vztahy – s vlastnostmi na obou = property graf
* **Hlavní vlastnosti** (podle autorů)
  + intuitivní – grafový model pro reprezentaci dat, spolehlivá – s plnými transakcemi ACID
  + trvalá a rychlá – nativní úložný engine založený na disku
  + masivně škálovatelná – až několik miliard uzlů / vztahů / vlastností
  + vysoce dostupná – při distribuci na více strojů, rychlá – výkonný rámec pro procházení
  + expresivní – výkonný, lidsky čitelný jazyk pro grafové dotazy, embeddovatelná
  + jednoduchý – přístupný pomocí rozhraní REST / objektově orientovaného Java API
* **vs. RDBMS** – pro agregovatelná data, Neo4j optimalizovaná pro vysoce propojená data
* **vs. Key-value (Column-family)** 
  + Model klíč-hodnota je určen pro vyhledávání jednoduchých hodnot nebo seznamů
    - Column-family úložiště lze považovat za krok ve vývoji úložišť typu klíč/hodnota
  + Neo4j umožňuje rozpracovat jednoduché datové struktury do složitějších dat – vzájemně propojené
* **Vs. Document store** pojme data, která lze snadno reprezentovat jako strom (Bez schématu)
  + Odkazy na jiné dokumenty v rámci stromu = výraznější reprezentace

**Datový model**

* Základní jednotky: uzly + hrany
* Hrana má vždy typ určený svým názvem a spojuje právě dva uzly -> možné vytvářet multigrafy
* Oba mohou mít atributy = dvojice (klíč, hodnota), přičemž klíč je typu řetězec, hodnota může být boolean, byte, float, char, String, popř. pole hodnot některého z těchto datových typů
* Null není validní hodnota – pokud chceme vyjádřit, že daný atribut nemá, prostě ho nevytvoříme
* **Přístup** pomocí **Java API,** Rest API
  + firstNode = graphDb.createNode();
  + firstNode.setProperty( "message", "Hello, " );
  + secondNode = graphDb.createNode();
  + secondNode.setProperty( "message", "World!" );
  + relationship = firstNode.createRelationshipTo(secondNode, RelTypes.KNOWS);
  + relationship.setProperty("message", "brave Neo4j ");
  + Samotný průchod grafem reprezentován objektem **Traversal** = zahrnuje informace o tom, jakou část grafu a jakým způsobem chceme projít. Konkrétně se skládá z těchto charakteristik:
    - Ve kterých uzlech průchod začíná
    - Směr hran a případně typy vztahů, které chceme do průchodu zahrnout
    - Pořadí uzlů při průchodu (tj. chceme-li procházet do šířky nebo do hloubky)
    - Povolení nebo zakázání opakovaného navštěvování uzlů a/nebo hran
    - Jakou část grafu chceme projít (např. celý graf, pouze do určité hloubky apod.)
    - Jakou část průchodu grafem chceme zahrnout do výsledku
* **Speciální jazyky: Gremlin a Cypher**

**Gremlin**

* = jazyk pro procházení grafů pro procházení grafů vlastností (property graph)
* Skripty se spouštějí v databázi serveru
* Výsledky jsou vráceny jako reprezentace uzlů a vztahů Neo4j
* gremlin> g = new Neo4jGraph('I:\\tmp\\myDB.graphdb')   
  ==> neo4jgraph[EmbeddedGraphDatabase[I:\tmp\myDB.graphdb]]
* gremlin> v = g.v(1)   
  ==>v[1]
* gremlin> v = g.v(1)   
  ==>v[1]
* gremlin> v.name   
  ==>marko

**Cypher**

* **Neo4j** grafový dotazovací jazyk, Jednoduchý a uživatelsky příjemný
* Deklarativní = nepopisujeme, jak chceme grafem projít, ale co chceme průchodem získat
* **Klauzule:**
  + START: Určení počátečních uzlů grafu.
  + MATCH: Vzor navázaný na počáteční uzly, kterému musí požadovaný graf odpovídat.
  + WHERE: Filtrovací kritéria.
  + RETURN: Návratové hodnoty.
  + CREATE: Vytváření uzlů a hran. DELETE: Mazání uzlů, hran a atributů.
  + SET: Nastavení/změna hodnot atributů uzlů/hran.
  + FOREACH: Provedení změn nad všemi prvky daného seznamu.
  + ORDER BY: Seřazení výsledku podle zvoleného kritéria.
* CREATE (a {name : 'Jaroslav'}) RETURN a -> a Node[2]{name:"Jaroslav"} 1 row Nodes created: 1 Properties set: 1
* CREATE n = {name : 'Jaroslav', title : 'professor'} -> (empty result) Nodes created: 1 Properties set: 2
* START a = node(1), b = node(2) CREATE a-[r:FRIEND]->b
* START a = node(1), b = node(2) CREATE a-[r:FRIEND {name : a.name + '<->' + b.name }]->b -> atributy hrany
* CREATE p = (jarda {name:'Jaroslav'})-[:WORKS\_AT]->mff<-[:WORKS\_AT]-(irena {name:'Irena'}) RETURN p
* START n = node(2) SET n.name = null RETURN n -> mazání pomocí null
* START user=node(1,2,4) MATCH user-[:knows]->friend WHERE friend.age > 30 AND friend.name =~ 'D.\*' RETURN user, friend.name
* **Další features:** agregační funkce (sum, min, max, avg,…), LIMIT n, SKIP n, ALL, ANY, LENGTH, …

**Transakce**

* Podpora ACID vlastností
* Veškeré operace editace grafu (zápis) musí probíhat v transakci (jinak chyba např. NotInTransactionException)
* Pokud dojde ke zrušení některé vnitřní transakce, je zrušena celá transakční hierarchie. Operace tedy probíhají v následujících krocích:
  + 1. Začátek transakce.
  + 2. Operace nad grafem zahrnující zápisy.
  + 3. Označení transakce jako úspěšné, nebo neúspěšné.
  + 4. Ukončení transakce, kdy dojde k uvolnění paměti i zámků.
* Veškeré operace zápisu probíhají v paměti -> velké editace grafu se dělí do menších celků
* Jakákoli **editační operace pak implicitně využívá zámky**. Konkrétně:
  + Přidání/změna/smazání atributu uzlu/hrany aplikuje zámek pro zápis na příslušném uzlu/hraně.
  + Vytvoření/smazání uzlu aplikuje zámek na příslušný uzel.
  + Vytvoření/smazání hrany aplikuje zámek na hranu a oba její uzly.
* Důsledkem využití zámků může být samozřejmě v určitých případech **deadlock** -> výjimka
* **Operace smazání** má navíc speciální sémantiku:
  + Smazání uzlu/hrany znamená smazání všech příslušných atributů.
  + Není možné smazat uzel, do/z něhož vedou hrany. Dojde k vyvolání chyby (výjimky)
  + Pokus o editaci uzlu/hrany smazané v aktuální transakci, která ještě nebyla potvrzena, způsobí chybu. Ale odkaz na takový uzel/hranu lze získat a s tímto odkazem je možné pracovat až do operace COMMIT.
* Operace čtení vždy přečte poslední hodnotu, pro kterou byl úspěšně proveden COMMIT
  + -> problém neopakovatelného čtení, kdy dvě po sobě jdoucí operace čtení, které se vyskytnou v jedné transakci, vracejí různou hodnotu

**Indexy**

* speciální datová struktura sloužící pro efektivnější vyhledávání uzlů, respektive hran grafu
* Každý index má unikátní, uživatelem zadané jméno a umožňuje asociovat libovolné množství dvojic (klíč, hodnota) s libovolným množstvím indexovaných entit, tj. uzlů nebo hran
* **Automatická indexace**
  + Implicitní indexy – konkrétně jeden index pro uzly a jeden pro hrany, přičemž oba automaticky indexují vybrané atributy uzlů/hran
  + Nutno v databázi povolit, pak se s indexem pracuje jako s explicitním

**Neo4j HA (Neo4j High Availability)**

* speciální část systému, která je založena na klasické master-slave architektuře – ta umožňuje:
  + nakonfigurovat několik uzlů typu slave tak, aby byly přesnými replikami uzlu master, a tudíž zajišťovaly odolnost vůči výpadkům
  + rozložit zátěž na více uzlů, tedy horizontálně škálovatelné čtení dat
* -> Neo4j může pracovat na: **jediném serveru / clusteru uzlů**
* Cluster uzlů -> **transakce stále atomické, konzistentní a trvalé**, ale do slaves se propagují jen občasně
* Každý uzel v clusteru stejná logika, která mu umožňuje koordinaci s jinými uzly. Po startu se instance databáze nejprve pokusí připojit ke clusteru, který má uveden v konfiguraci: **povede se = slave / nepovede se = master**
* Pokud **v běžícím clusteru dojde k výpadku**, nastane jedna z následujících možností:
  + Výpadek na slave – ostatní uzly to zjistí díky ztrátě komunikace a master s ním data dále nesynchronizuje
  + Výpadek na master – uzly typu slave zvolí nový uzel master
* Opětovné navázání komunikace, tak se původní master chová jako slave
* Každý uzel uloženou kopii celé grafové databáze, existují **omezení na velikost grafu,** který neo4j může uložit

**Multimodel databáze** – ArangoDB, OrientDB

* Polyglotní perzistence -> nutnost vzniku různých typů databázových systémů pro různorodé účely
  + Pokud máte strukturovaná data s určitými rozdíly -> document store
  + Pokud máte vztahy mezi entitami a chcete se na ně efektivně dotazovat -> grafová databáze
  + Pokud si strukturu dat spravujete sami a nepotřebujete složité dotazy -> key/value store
* -> Problémy: vývojář musí znát různé databáze, integrace různých databází, dotazy a transakce napříč modely?
* -> Multimodel databáze = Jedna jednotná databáze pro data z více modelů
* Navržena tak, aby podporovala více datových modelů na jednom integrovaném backendu
* **1. Jedna velikost nemůže vyhovovat všem**
  + Analytika SQL, real-time decision support a datové sklady nemohou být podporovány jedním enginem
* **2. Jedna velikost může vyhovovat všem**
  + OctopusDB navrhuje jednotnou, univerzální architekturu zpracování dat pro OLTP, OLAP, streamovací systémy a databázové systémy orientované na skenování
  + Všechna data se shromažďují v centrálním protokolu – Insert a update-operations = log-entries
  + Na základě tohoto protokolu definuje několik typů volitelných pohledů na úložiště
  + Optimalizace dotazů, údržba zobrazení, výběr indexů i problémy s výběrem úložiště se náhle stávají jediným problémem: výběrem zobrazení úložiště
* **3. Jedna velikost je vhodná pro všechny**
  + Semistrukturovaný data model – Bez schématu i s full-schématem, Textová, časová, prostorová, ... data
  + Dotazovací jazyk podobný SQL
* **Plusy**: Zpracování dat z více modelů, Jeden systém implementuje odolnost proti chybám, Konzistence dat
  + Jednotný dotazovací jazyk pro multimodelová data
* **Zápory**: Komplexní systém, Nezralý a vyvíjející se, Mnoho výzev a otevřených problémů

Obsah obrázku text, snímek obrazovky, Písmo, číslo

Popis byl vytvořen automaticky**ArangoDB**

* ArangoDB je open-source databáze s více modely a flexibilními datovými modely - Document, graf, key/value
* Ukládá všechna data jako dokumenty
* Vrcholy a hrany grafů jsou dokumenty -> umožňuje kombinovat všechny tři datové modely

**OrientDB**

* Podpora modelů grafů, dokumentů, klíč/hodnota a objektů
* Vztahy jsou řízeny jako v grafových databázích s přímými vazbami mezi záznamy.
* Podporuje režimy bez schématu, s plným schématem a se smíšeným schématem.
* Dotazy: SQL rozšířený o procházení grafů
* SELECT expand( out("Knows").Orders.orderlines. Product\_no ) FROM Customers WHERE CreditLimit > 3000

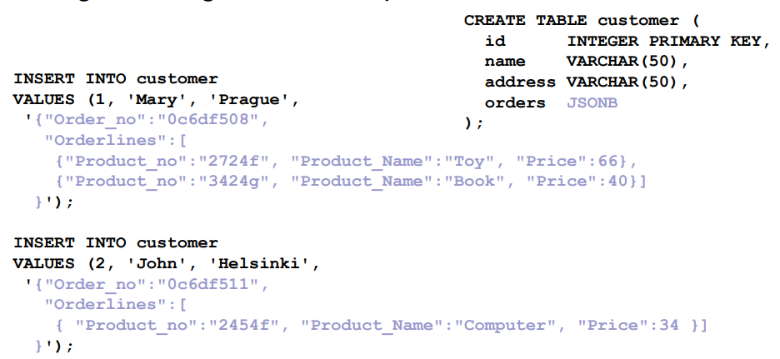
**Rozšíření o multimodely**

1. Přijetí zcela nové strategie ukládání vhodné pro nový datový model, např. databáze s podporou XML.

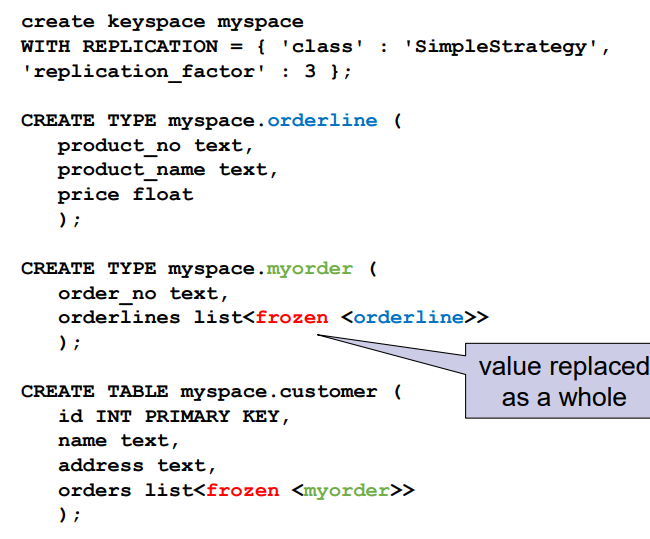
2. Rozšíření původní strategie ukládání pro účely nového datového modelu, např. ArangoDB – speciální kolekce hran nesou informace o hranách v grafu

3. Vytvoření nového rozhraní pro původní strategii ukládání dat, např. MarkLogic – ukládá data ve formátu JSON stejným způsobem jako data ve formátu XML

4. Žádná změna původní strategie ukládání – Ukládání a zpracování datových formátů jednodušší než původní

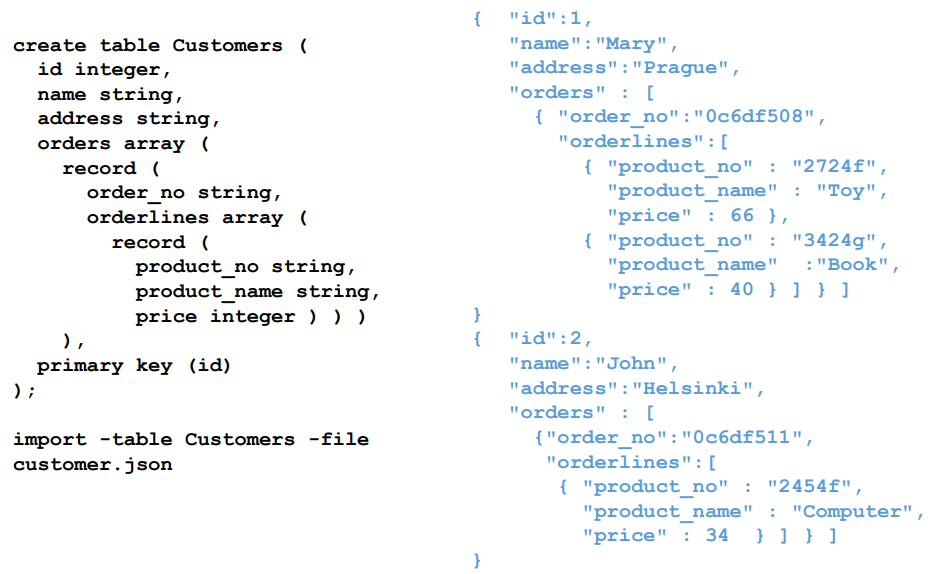
* ****Typy přechodů mezi modely
  + Reference mezi modely (Inter-model references)
  + Vložení modelu (model embedding)
  + Redundance mezi modely (cross-model redundancy)

**Relační multi-model DBMSs – např. PostgreSQL**

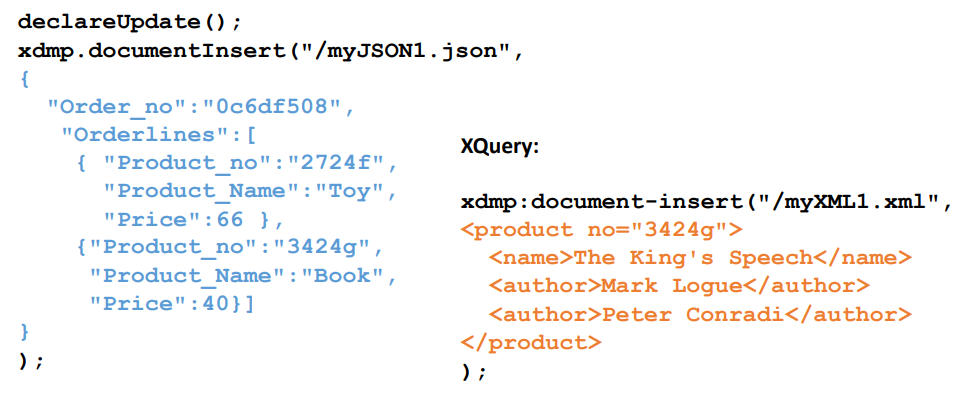
* Největší množství multimodel databází – Nejoblíbenější typ databází,
* SQL byl rozšířen na další datové formáty (např. SQL/XML).
* Jednoduchost a univerzálnost relačního modelu

**Column multimodel DBMSs**

* Column-oriented (columnar) ukládá data do tabulek jako sloupce, nikoliv jako řádky – Ne nutně NoSQL
* Column-family (wide-column) = databáze NoSQL, která podporuje tabulky s různým počtem a typem sloupců – Základní strategie ukládání je libovolná

**Key/value multi model DBMSs**

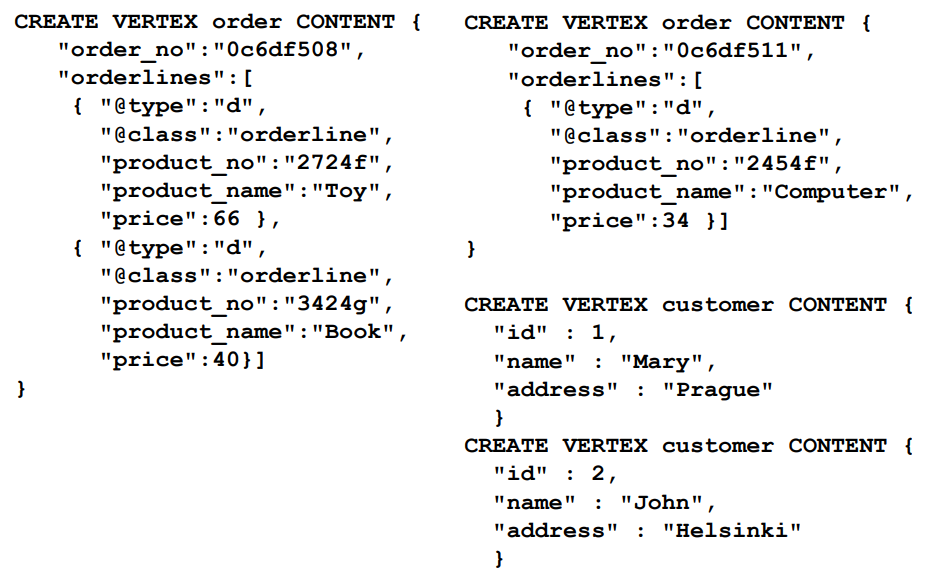
* Nejjednodušší typ databáze NoSQL
  + Get / put / delete + klíč
  + Často rozšířená o pokročilejší funkce
* Multi-model extenze: Složitější indexy nad hodnotovou částí + nové API (např. JSON, SQL, ...)



**Document multimodel DBMSs**

Odlišné strategie:

* ArangoDB: speciální kolekce hran
* MarkLogic: ukládá JSON jako XML



**Graph multimodel DBMSs**

* Založeno na objektové databázi = nativní podpora více modelů
* -> Element úložiště = záznam = dokument / BLOB / vrchol / hrana
* Třídy – definují záznamy
* Třídy mohou mít vztahy: Referenced – ukládají se podobně jako ukazatele mezi dvěma objekty v paměti, Vložené (Embedded) – uložené v rámci záznamu, který vkládají

**Jazyky pro dotazování se nad multimodely**

1. Jednoduché API
   * Ukládání, načítání, mazání dat – Typicky klíč/hodnota, ale i jiné případy použití
   * DynamoDB – jednoduchý přístup k datům + dotazování nad indexy pomocí srovnávacích operátorů
2. Rozšíření SQL a jazyky podobné SQL – Nejběžnější
   * Ve většině typů systémů (relační, sloupcové, dokumentové, ...)
3. Rozšíření dotazů SPARQL
   * IBM DB2 – SPARQL 1.0 + podmnožina funkcí z SPARQL 1.1
     + SELECT, GROUP BY, HAVING, SUM, MAX, ...
     + Pravděpodobně žádné rozšíření pro relační data – Ale: RDF trojice jsou uloženy v tabulce  SQL dotazy lze použít i nad nimi
4. Rozšíření dotazů XML
   * MarkLogic – k JSON lze přistupovat pomocí XPath
     + Stromová reprezentace jako u XML, lze volat z XQuery a JavaScriptu
5. Fulltextové vyhledávání - Obecně docela běžné
   * Riak – index Solr + operace -> Wildcards, proximitní search, range search, booleovské operátory, grouping…

**Multimodel query processing**

* Závisí do značné míry na způsobu, jakým byl systém rozšířen.
  + Žádná změna
  + Nové rozhraní, např. MarkLogic
  + Rozšíření původní strategie ukládání, např. ArangoDB
  + Zcela nová strategie ukládání, např. nativní podpora XML ze strany Oracle
* Obecné tendence:
  + Využití stávajících strategií ukládání v co největší míře
  + Využívat ověřené přístupy k optimalizaci dotazů

**MarkLogic multiple modely**

* Indexuje data XML i JSON stejným způsobem.
* Data bez schématu
* Univerzální index – optimalizovaný tak, aby umožňoval kombinovat vyhledávání textu, struktury a hodnoty do jednoho – Slovní indexování, Indexování frází, Indexování vztahů, Indexování hodnot
* Další indexy definované uživatelem: Indexování rozsahu, Lexikony slov, Zpětné indexování, Trojitý index

**NewSQL databáze**

* = systémy, které nabízejí škálovatelné úložiště i veškerou funkcionalitu jako u klasických relačních databází typu klient-server – jak jazyk SQL, tak relační model dat a vlastnosti ACID
* 2 přístupy se stejným cílem:
* 1. **distribuované systémy** s výhodami relačního modelu a ACID vlastnosti, např. Clustrix, ScaleArc, VoltDB…
* 2. **původně relační databáze** rozšířeny o techniky pro horizontální škálovatelnost, např. TokuDB16, JustOne DB
* **Cloud**: **NewSQL as a Service** - NewSQL databáze jako cloudová služba, tedy horizontálně škálovatelný relační databázový systém - Amazon Relational Database Service, Microsoft Azure Database – relační model + SQL
* **Proč je potřeba?**
  + 1. Mnoho aplikací pracujících s relačním modelem -> potřebují řešit náhlý nárůst objemu dat
    - Přechod na model v NoSQL databázích -> časově a finančně velmi náročné modifikace
  + 2. Kdyby přechod na jiný datový model nebyl takovým problémem, mnoho aplikací si nemůže dovolit vzdát se požadavků na silnou konzistenci dat, kterou NoSQL databáze běžně nenabízejí
* -> potřebujeme zachovat vlastnosti relačních databází a současně horizontálně škálovat
* Není hrozba pro NoSQL systémy

**VoltDB**

* vybudován na základě experimentálního akademického systému H-Systém
* škálovatelný systém, který garantuje ACID vlastnosti a je založen na relačním modelu dat
* jazyk SQL + relační model -> snadná práce s daty a významně se neliší od klasických databázových systémů
  + CREATE / ALTER / DROP TABLE, INSERT INTO, CHECK, SELECT (including GROUP BY), set operations, nested queries, stored procedures, database views
* Big data – automactická distribuce (ale uživatel může systému poradit podle jakého sloupce distribuci dělat)
* Distribuovaná architektura – shared-nothing = uzly v clusteru mezi sebou nesdílí ani paměť, ani disk
  + -> autonomní jednotky, které mezi sebou komunikují prostřednictvím zpráv
* Data jsou primárně zpracovávána v paměti (in-memory database), čímž je zajištěna maximální efektivita
* **Pro opravdovou efektivitu**: distribuce tabulek vhodná pro konkrétní aplikaci -> odpovídá příslušným distribuovaným uloženým procedurám -> umožňují jednotlivé části dat zpracovat lokálně
  + Každá uložená procedura = 1 transakce
  + VoltDB transakce v rámci jednoho uzlu serializuje -> bez zámků, logů, …
  + Díky distribuci dat umožňuje paralelní zpracování více transakcí najednou
  + Pokud procedura potřebuje data z více uzlů, jeden z uzlů = koordinátor, spustí lokální požadavky, sloučí jejich výsledky a proceduru dokončí -> pomalejší než nezávislé distribuované procedury

**Array databáze**

* Databázové systémy určené speciálně pro data, která jsou reprezentována jako jedno nebo vícerozměrná pole
  + -> když potřebujeme ukládané hodnoty reprezentovat v prostoru a/nebo v čase
  + Astronomická měření, klimatické změny, satelitní snímky Země, oceánografická data, lidský genom, …
* Big Data s velmi specifickými a náročnými požadavky na reprezentaci i analýzu
* Tato data nejsou vhodná pro ukládání do plochých relací klasických relačních databázových systémů
  + Uložit by je sice uměly, ale efektivita operací by byla značně limitována.
* např. SciDB, Rasdaman, Oracle Spatial and Graph

**SpiDB**

* jedna z nejpopulárnějších reprezentací
* **Datový model: vícerozměrné uspořádané poli**, předpokládá, že **data nejsou přepisována**
  + Pokaždé je vytvořena nová verze dat – umožňuje analyzovat změny a opravy v čase
  + umí efektivně ukládat jak hustá, tak řídká pole libovolně velkých dimenzí
* Pro práci s daty nabízí SciDB dva nástroje:
  + **deklarativní jazyk AQL (Array Query Language)** – inspirované SQL, místo tabulek pracujeme s poli
    - DDL (Data Definition Language) pro vytváření polí a jejich naplnění daty
    - DML (Data Manipuation Language) pro dotazování a operace s daty
  + **funkcionální jazyk AFL (Array Functional Language)**
* Každé pole – minimálně 1 atribut, hodnotu určitého datového typu ukládanou do jednotlivých buněk pole
  + Každé pole aspoň jedna dimenze -> každá dimenze definované souřadnice, velikost datového úseku (chunk) a případný překryv úseků (overlapping)
* **CREATE ARRAY A <x: double, err: double> [i=0:99,10,0, j=0:99,10,0];LOAD A FROM '../examples/A.scidb';**
  + 2 atributy x, err, 2 dimenze i, j, souřadnice (0:99), velikost datového úseku (10), a nulový překryv
* Distribuuje data na datové úseky (chunks) - ideálně 10-20 MB (ne moc malé/velké úseky)
* Data v polích distribuuje a zpracovává právě po specifikovaných úsecích
* Překryvy úseků využívat nemusíme, ale hodí pro efektivní dotazování na nejbližší sousedy
* // vypsání hodnot souřadnice i z pole: A SELECT i FROM A;   
  [(0),(1),(2),(3),(4),(5),(6),(7),(8),(9)]
* **SELECT** pro spojování vice polí -> základní operace = vnitřní spojení -> spojí hodnoty v buňkách dvou polí
  + Spojovaná pole musí být kompatibilní – stejné počáteční souřadnice, velikosti datových úseků a překryvy
  + Množství a typy atributů se mohou lišit – ty jsou prostě sloučeny dle zadané operace
  + // spojení hodnot polí A a B a uložení výsledku do pole C SELECT \* INTO C FROM A, B; [(1,101),(2,102),(3,103),(4,104),(5,105),(6,106),(7,107),(8,108),(9,109),(10,110)]
* MERGE (slévání hodnot 2 polí se stejnými charakteristikami do výsledného pole, při němž jsou vybírány nenulové hodnoty primárně z prvního, případně druhého pole)
* CROSS (kartézský součin hodnot 2 polí, tedy vytvoření pole všech kombinací hodnot spojovaných polí)
* CROSS\_JOIN (kartézský součin hodnot 2 polí, navíc s podmínkou rovnosti na souřadnice odpovídajících dimenzí)
* JOIN ON (spojení přes zadanou podmínku)

**Search engine**

* Někdy se označují jako systémy pro správu dat vyhledávačů – search engine data management systems
* Rozdíly oproti relačním DBMS
  + Žádné pevné strukturální požadavky
    - Data mohou být strukturovaná, polostrukturovaná, nestrukturovaná, ...
    - Žádné relace, žádná omezení, žádné spojování, žádné transakční chování, ...
  + Případy použití: vyhledávání na základě relevance, fulltextové vyhledávání, vyhledávání synonym, analýza logů, ... -> není typické pro databáze
  + Data mohou být velká -> Distribuované výpočty
* Rozdíly oproti NoSQL DBMS
  + Primárně určeny k vyhledávání, nikoliv k úpravám
  + Specializované funkce: fulltextové vyhledávání, stemming, složité vyhledávací výrazy, řazení a seskupování výsledků vyhledávání, geoprostorové vyhledávání -> Big Data analýza

**Elasticsearch**

* Distribuovaný full-text search engine -> škálovatelné vyhledávácí řešení
* Elastic Stack = Elasticsearch +
  + Logstash – shromažďuje, zpracovává a přeposílá události a zprávy z logů.
  + Kibana – analytická a vizualizační platform
* Lze použít pro všechny druhy dokumentů
* **Vyhledávání téměř v reálném čase** -> Mírná latence (cca 1 sekunda) od okamžiku indexování (nebo aktualizace či odstranění) dokumentu do doby, než se dokument stane prohledávatelným
* **Index** = kolekce dokumentů s podobnými vlastnostmi, např. údaje o zákaznících, katalog výrobků, ...
  + Má název, V clusteru může být libovolný počet indexů
* Indexy můžeme rozdělit na části – každý shard může mít repliky, vyvažování a routing se provádí automaticky
* Každý uzel může fungovat jako koordinátor, který deleguje operace na příslušné oddíly

**Indexy**

* Při vytváření indexu definujte počet oddílů a počet replik - index nemusí být definován předem.
  + Každý shard je sám o sobě plně funkční a nezávislý index
  + Části umožňují: Horizontální škálování velkých objemů dat, Paralelizaci operací
  + Repliky umožňují: vysokou dostupnost (částečné výpadky), Paralelizace operací
* Výchozí nastavení: 5 primárních oddílů a 1 replika, tj. 10 oddílů na index
* GET /\_cat/indices? -> vrátí všechny indexy
* PUT /customer?pretty -> vytvoří index “customer” (a pěkně vytisknout výsledek, pokud existuje)
* PUT /customer/\_doc/1?pretty { "name": "John Doe" } – index daného dokumentu s ID = 1
* GET /customer/\_doc/1?pretty – vrátí document s ID = 1
* DELETE /customer/\_doc/1?pretty – smaže document s ID = 1
* DELETE /customer – smaže index “customer”

**Modifikace dat**

* ID dokumentu
  + Pokud je použit existující dokument: dokument je nahrazen (a znovu zaindexován)
  + Pokud je použito jiné: je uložen nový dokument -> Stejný dvakrát
  + Pokud není zadáno žádné: je vygenerováno náhodné ID
* Aktualizace dokumentu -> Žádné aktualizace na místě, dokument je smazán a je vytvořen a indexován nový
* POST /customer/\_doc/1/\_update?pretty { "doc": { "name": "Jane Doe" } } změní “name” dokumentu s ID = 1

**Apache Lucene**

* Invertovaný index, High-performance, text search engine knihovna
* Dokument = jednotka vyhledávání a indexu -> nemusí to být skutečné dokumenty, ale i databázové tabulky
* Dokument se skládá z jednoho nebo více polí -> Dvojice jméno-hodnota
* Vyhledávání vyžaduje, aby byl již vytvořen index
* Pro vyhledávání používá vlastní jazyk: Shoda: vyhledávání podle klíčových slov, zástupných znaků, blízkosti, rozsahu, ..., Logické operátory, Posílení termínů/výrazů

**Problém z rozmanitostí (variety) Big Data – dvě řešení:**

1. **Multimodel databáze** – Použití jediného integrovaného backendu
2. **Polystores** – Společné použití více technologií ukládání dat, které se vybírají na základě způsobu, jakým jsou data využívána jednotlivými aplikacemi

**Polystores**

* Použijte správný nástroj pro (každou část) práce...
  + Pokud máte strukturovaná data s určitými rozdíly – Použijte úložiště dokumentů
  + Pokud máte vztahy mezi entitami a chcete se na ně efektivně dotazovat – Použijte databázi grafů
  + Pokud si strukturu dat spravujete sami a nepotřebujete složité dotazy – Použijte úložiště klíč-hodnota
* ...a slepte vše dohromady
* **Plusy:** Zpracování dat z více modelů, Pomozte svým aplikacím dobře škálovat, Bohaté zkušenosti s úložišti s jedním modelem
* **Mínusy:** Vyžaduje, aby společnost najala lidi na integraci různých databází, Vývojáři se musí naučit různé databáze, Je náročné zvládnout dotazy a transakce napříč modely
* **Tři typy polystore systémů**
* **1. Loosely-coupled (volně vázané) systémy** – Podobně jako architektura mediátor-wrapper, společná rozhraní, autonomie místních úložišť
* **2. Tightly-coupled (těsně propojené) systémy** – Využití přímo lokálních rozhraní, Výměna autonomie za výkon, Materializované pohledy, indexy
* **3. Hybridní**
* **Žádná "univerzální velikost"...**
  + Heterogenní analýza dat: rámce pro zpracování dat (Map/Reduce, Spark, Flink), NoSQL, ...
  + Polystore idea: V tomto případě se jedná o balíček více dotazovacích motorů
    - Sjednocení (federace) různých specializovaných úložišť, každé s odlišným (nativním) datovým modelem, interními možnostmi, jazykem a sémantikou
    - Svatý grál: platformově agnostická datová analytika
  + Použití správného úložiště pro každý specializovaný scénář
  + Případně spoléhat na middleware vrstvu pro integraci dat z různých zdrojů
* **Dimenze Polystore**
  + **Heterogenita** – Různé datové modely, modely dotazů, expresivita, dotazovací motory
  + **Autonomie** – Asociace s polystores, provádění (podpora nativních aplikací + federace), vývoj vlastních modelů a schémat
  + **Transparentnost** – Umístění (data mohou přesahovat i více úložných strojů), transformace / migrace dat
  + **Flexibilita** – Uživatelem definovaná schémata a rozhraní (funkce), modulární architektura
  + **Optimalita** – Sdružené plány, umístění dat
* **Tightly Integrated Polystores (TIPs)** - Úzce integrované Polystore, např. Polybase, HadoopDB, Estocada
  + Vyměňte autonomii za efektivní dotazování na různé druhy dat pro analýzu Big Data
    - K datovým úložištím lze přistupovat pouze prostřednictvím vícesložkového systému
    - Menší nejistota s rozšířenou kontrolou nad různými úložišti
    - Úložiště, ke kterým se přistupuje přímo prostřednictvím jejich lokálního jazyka
  + Efektivní / adaptivní pohyb dat mezi datovými úložišti
  + Počet datových úložišť, která lze propojit, je obvykle omezený
  + Rozšiřitelnost

Obsah obrázku text, snímek obrazovky, Písmo, číslo

Popis byl vytvořen automaticky

* **Loosely Integrated Polystores** (volně integrované) - BigIntegrator, Forward/SQL++, QoX
  + Připomíná systémy s multimodel databázemi
  + Sledujte architekturu mediator-wrapper (jeden wrapper na každé datové úložiště)
    - Jeden globální společný jazyk
  + Obecný přístup
    - Rozdělení dotazu na dílčí dotazy – pro každé datové úložiště, stále ve společném jazyce
    - Odeslat do wrapperu, Překlad, Získat výsledky, Přeložit do běžného formátu, Integrovat
* **Hybridní polystores** – např. BigDawg, SparkSQL, CloudMdsQL
  + Spoléhat se na těsné propojení u některých obchodů, u jiných na volné propojení.
  + Podle architektury mediator-wrapper -> Procesor dotazů však může k některým datovým úložištím přistupovat i přímo
* **BigDAWG**
  + Soubor datových úložišť, ke kterým se přistupuje pomocí jediného dotazovacího jazyka
  + Klíčová abstrakce: **island of information**
    - Datový model + operace + úložný(é) stroj(y), cross-island queries
  + Spoléhá na různé data islands
    - Relační, pole, NoSQL, streamování, ..., v současné době: PostgreSQL, SciDB, Accumulo.
  + Žádný společný datový model, dotazovací jazyk / procesor -> Každý ostrov má vlastní
  + **Shim** spojuje island s jedním nebo více úložnými mechanismy.
    - Mapuje dotazy z jazyka islandu do nativního dotazovacího jazyka konkrétního úložného stroje
  + **Cast** = operátory pro přesun datových sad mezi islands
    - Zpracování v úložném stroji, který nejlépe vyhovuje vlastnostem dat
  + Jádrem je middleware, který podporuje společné rozhraní API pro soubor úložných zařízení.
  + Klíčové prvky:
    - **Optimalizátor**: analyzuje vstupní dotaz a vytvoří sadu životaschopných stromů plánů dotazů s možnými motory pro každý subdotaz
    - **Monitor**: používá údaje o výkonu z předchozích dotazů k určení stromu plánu dotazu s nejlepším motorem pro každý dílčí dotaz
    - **Executor**: zjistí, jak nejlépe spojit kolekce objektů, a poté provede dotaz
    - **Migrátor**: přesouvá data z motoru do motoru, pokud plán takový pohyb dat vyžaduje

**Další klasifikace:**

* **Federativní systémy**: soubor homogenních data úložišť, obsahuje jediné standardní rozhraní pro dotazování
* **Polyglotní systémy**: soubor homogenních datových úložišť, vystavuje uživatelům více dotazovacích rozhraní
* **Multistore systémy:** data v heterogenních datových úložištích, podpora jediného dotazovacího rozhraní
* **Polystore systémy**: zpracování dotazů napříč heterogenními datovými úložišti, podpora více query interfaces

**Otevřené problémy a výzvy**

* **Mnoho výzev**: optimalizace dotazů, provádění dotazů, rozšiřitelnost, rozhraní, transakce napříč platformami, samočinné ladění, umisťování dat / migrace, benchmarking, ... -> vysoká míra nejistoty
* **Transparentnost**: nevyžadovat, aby uživatelé určovali, kde mají získat / uložit data, kde mají spouštět dotazy / poddotazy -> Vysvětlení a umožnění uživatelských nápověd
* Více než kdy jindy potřeba automatizace, přizpůsobivosti, učení se za běhu

**Pokročilé aspekty Big Data Managementu**

**CAP teorém**

* **konzistence** (consistency), **dostupnost** (availability) a **odolnosti vůči rozpadu sítě** (partition tolerance)
* Všechny tyto tři vlastnosti nejsme schopni v distribuovaném systému zajistit na 100 %
* **Consistency + Availability** -> Single-site databases, cluster databases, …
* **Consistency + Partition Tolerance** -> Distributed databases, distributed locking, majority protocols, …
* **Availability + Partition Tolerance** -> Web caching, DNS

**Spravování transakcí**

* Kritici databází NoSQL se zaměřují na nedostatečnou podporu transakcí.
* **Business transakce** - např. prohlížení katalogu produktů, výběr láhve Taliskeru za dobrou cenu, vyplnění údajů o kreditní kartě a potvrzení objednávky
* **Systémové transakce** – Na konci interakce s uživatelem, Zámky jsou drženy pouze po krátkou dobu
* **Business transakce** = série systémových transakcí
* **Offline souběžnost** (concurrency) zahrnuje manipulaci s daty pro obchodní transakci, která zahrnuje více požadavků na data -> Mít otevřenou systémovou transakci pro celou obchodní transakci obvykle není možné
  + Dlouhé systémové transakce nejsou podporovány
* Problémy:
  + **Přepisování neschválených (Overwriting uncommitted) dat** -> Více transakcí vybere stejný řádek, poté aktualizuje řádek na základě původně vybrané hodnoty, aniž by si to uvědomovaly ostatní transakce
  + **Čtení nekomitovaných (Reading uncommitted) dat** -> Transakce přistupuje ke stejnému řádku několikrát a pokaždé čte jiná data
* tj. výpočty a rozhodnutí mohou být prováděny na základě dat, která jsou změněna, např. může být aktualizován ceník, někdo může aktualizovat adresu zákazníka, změnit poštovné, ...

**Optimistický offline zámek (Optimistic Offline Lock)**

* Předpokládá, že pravděpodobnost konfliktu je nízká
* Forma podmíněného updatu -> aby změny, které mají být commitovány jednou relací, nebyly v konfliktu se změnami jiné relace
* Validace před odesláním (pre-commit validace)
  + 1. Klientská operace znovu přečte všechny informace, na které se obchodní transakce spoléhá
  + 2. Zkontroluje, zda se nezměnily od doby, kdy byly původně přečteny a zobrazeny uživateli
* Získání zámku, který indikuje, že je v pořádku pokračovat ve změnách dat záznamu

**Pesimistický offline lock (Pessimistic Offline Lock)**

* **Problémy optimistického algoritmu**: mnoho konfliktů, konflikt lze zjistit až na konci dlouhé business transakce
* **Pesimistické řešení:** umožňuje přístup k datům pouze jedné obchodní transakci v daném okamžiku.
* Nutí business transakci, aby získala zámek na každý kus dat předtím, než je začne používat
  + Jakmile business transakce začne, určitě se dokončí
* **Správce zámků (Lock manager) ->** Jednoduchý, jediný (pro všechny business transakce), centralizovaný (nebo založený na databázi v distribuovaném systému)
* **Standardní problém: deadlock**
  + Časový limit pro aplikaci -> Automatické vrácení zpět po určité době neodpovídání
  + Atribut časového razítka pro zámek -> Automatické uvolnění po určité době

**Coarse-grained Lock (hrubozrnný zámek)**

* Když jsou objekty upravovány jako skupina -> Logicky související objekty, např. zákazník a jeho sada adres
  + Chceme uzamknout libovolný z nich
* Samostatný zámek pro jednotlivé objekty = řada problémů -> všechny najít, abychom je mohli uzamknout
  + S přibývajícími skupinami zámků je to složitější, Když se skupiny zkomplikují -> Vnořené skupiny
* Myšlenka: jeden zámek, který pokrývá mnoho objektů -> Sofistikovaný správce zámků

**Implicitní zámek**

* Problém: zapomeneme napsat jediný řádek kódu, který získá zámek -> celé offline zamykací schéma je k ničemu
  + Nezískání zámku pro čtení -> jiné transakce používají zámky pro zápis -> nezískání aktuálních dat relace
  + Nepoužití počítadla verzí -> nevědomé přepsání něčích změn
  + Neuvolnění zámků -> zastaví produktivitu práce.
* **Fakt**: Pokud může být položka uzamčena někde, musí být uzamčena všude
* Myšlenka: **zámky se získávají automaticky** -> Ne explicitně vývojáři, ale implicitně aplikací

**Performance tuning (ladění výkonu)**

* MapReduce vytváří způsob škálování bez úzkých míst
* **Snížení latence**
  + Latence:
    - Neparalelní systémy: čas potřebný k provedení celého programu
    - Paralelní systémy: čas potřebný k provedení nejmenší atomické dílčí úlohy
  + Strategie:
    - Zkrácení doby provádění programu, Výběr nejoptimálnějších algoritmů pro vytvoření výstupu
    - Paralelní provádění dílčích úloh
* **Zvýšení propustnosti**
  + **Propustnost** = množství vstupů, které lze v rámci procesu zpracovat pro vytvoření výstupu
  + **Neparalelní** **systémy** – Omezené dostupnými zdroji (množství paměti RAM, počet procesorů)
  + **Paralelní systémy** – bez omezení, Paralelizace umožňuje použití libovolného množství komoditního HW
* **Lineární škálování**
  + Typický horizontálně škálovaný model založený na MapReduce: lineární škálovatelnost
    - "Jeden uzel clusteru může zpracovat x MB dat každou sekundu -> n uzlů může zpracovat x  n množství dat každou sekundu."
      * Čas potřebný ke zpracování y množství dat na jednom uzlu = t sekund
      * Čas potřebný ke zpracování y objemů dat na n uzlech = t / n sekund
  + Předpoklad: úlohy lze paralelizovat do stejně vyvážených jednotek
* **Amdahlův zákon**
  + Vzorec pro zjištění maximálního zlepšení výkonu systému při vylepšení jeho části
    - P = podíl programu, který je paralelizován
    - 1–P = podíl programu, který nelze paralelizovat
    - N = časy, které paralelizovaná část vykoná ve srovnání s neparalelizovanou částí, tj. kolikrát je rychlejší, např. počet procesorů
  + **Příklad**: proces, který běží 5 hodin (300 minut); lze paralelizovat celou část programu kromě malé části, která trvá 25 minut
    - Procento celého programu, které lze paralelizovat: 91,6 %.
    - Procento, které nelze paralelizovat: 8,4 %.
    - Maximální zvýšení rychlosti: 1 / (1 - 0,916) = ~11,9krát rychlejší -> N má tendenci k nekonečnu
* **Littleův zákon** (Původ v ekonomii a teorii front (matematika)) – L = kW
  + Analýza zatížení stabilních systémů -> Zákazník nastoupí do fronty a je obsloužen (v konečném čase).
  + "Průměrný počet zákazníků (L) ve stabilním systému je součinem průměrné rychlosti příchodu (k) a doby, kterou každý zákazník v systému stráví (W)."
    - Intuitivní, ale pozoruhodný výsledek, tj. vztah není ovlivněn rozdělením procesu příchodu, rozdělením služeb, pořadím služeb ani prakticky ničím jiným.
  + Příklad: Čerpací stanice s platbami pouze v hotovosti přes jeden pult.
    - Každou hodinu přijedou 4 zákazníci, Každý zákazník stráví na čerpací stanici přibližně 15 minut
    - -> V každém okamžiku by měl být v průměru 1 zákazník, pokud by na jednu stanici přijeli více než 4 zákazníci, vedlo by to k úzkému hrdlu
* **Message Cost Model (model nákladů na zprávy) – C = a +bN**
  + Rozděluje náklady na odeslání zprávy z jednoho konce na druhý z hlediska fixních a variabilních nákladů.
    - C = náklady na odeslání zprávy z jednoho konce na druhý.
    - a = počáteční náklady na odeslání zprávy
    - b = náklady na jeden bajt zprávy
    - N = počet bajtů zprávy
  + Příklad: gigabitový Ethernet
    - a je přibližně 300 mikrosekund = 0,3 milisekundy
    - b je 1 sekunda na 125 MB
    - Znamená přenosovou rychlost 125 MBps.
    - 100 zpráv o velikosti 10 KB => trvá 100  (0,3 + 10/125) ms = 38 ms.
    - 10 zpráv po 100 KB => trvá 10  (0,3 + 100/125) ms = 11 ms
    - Způsob, jak optimalizovat náklady na zprávy, je poslat pokaždé co největší paket.