

# Materiály ke zkoušce z Moderních databázových systémů

---

## Stránky předmětu

### Osnova

- Vrstvy databázových modelů
  - Big Data
    - Definice
    - Zdroje Big Data
    - Hlavní charakteristiky Big Data
    - Zpracování Big Data
  - Výhody NoSQL databází
  - MapReduce
  - Apache Spark
    - RDD operace
      - Transformace
      - Akce
  - Distribuční modely
  - Cloud computing
- 

## Vrstvy databázových modelů

---

### Konceptuální vrstva

- Modelování domény (reálné věci ze světa).
- Vysoká míra abstrakce.
- Používají se modely jako ER, UML.

### Logická vrstva

- Datové struktury pro uložení dat.
- Typy: relační, objektová, XML, grafová apod.

### Fyzická vrstva

- Skutečná implementace datových struktur z logické vrstvy.
  - Obsahuje indexaci, datové soubory a jejich dělení.
- 

## Big Data

---

**Definice (podle Gartneru - společnost pro IT analýzy a poradenství):**

- **Volume** – velké objemy dat.
  - **Variety** – různorodost dat.
  - **Velocity** – rychlost generování a zpracování dat.
- 

## Zdroje Big Data

- Sociální sítě.
  - Vědecké nástroje.
  - Mobilní zařízení.
  - Senzory.
- 

## Hlavní charakteristiky Big Data

### 1. Volume (objem)

- Velikost dat neustále exponenciálně roste.

### 2. Variety (různorodost)

- Podpora různých datových formátů, typů a struktur.
- Data mohou být strukturovaná, částečně strukturovaná, nebo zcela bez struktury.

### 3. Velocity (rychlost)

- Data jsou generována a musí být zpracována v reálném čase.

### 4. Veracity (věrohodnost)

- Řešení problémů jako inkonzistence, latence a neucelenost dat.
- 

## Zpracování Big Data

### OLTP: Online Transaction Processing

- Používá se v DBMS a databázových aplikacích.
- Slouží k ukládání, dotazování a přístupu k datům.

### OLAP: Online Analytical Processing

- Multi-dimenzionální analytické dotazy.
- Patří do sekce BI (Business Intelligence).

### RTAP: Real-Time Analytic Processing

- Data jsou zpracovávána jako stream v reálném čase.
  - Klíčové pro moderní Big Data architektury.
-

# Výhody NoSQL databází

---

## 1. Škálování

- Horizontální škálování ("scaling out") oproti tradičnímu vertikálnímu škálování ("scaling up").

## 2. Big Data podpora

- RDBMS nezvládají nové velké objemy dat.

## 3. Administrace

- Automatické opravy, distribuovanost, a self-tuning.

## 4. Náklady

- Nižší cena za transakci díky horizontálnímu škálování a využití [komoditního hardwaru](#).

## 5. Flexibilita

- Absence pevného schématu umožňuje snadné strukturální změny bez vysokých nákladů.
- 

# Výzvy NoSQL databází

- Nižší vyspělost technologií.
  - Omezená podpora analytických nástrojů a BI.
  - Nedostatek odborníků.
- 

# Datové předpoklady

RDBMS	NoSQL
integrity is mission-critical	OK as long as most data is correct
data format consistent, well-defined	data format unknown or inconsistent
data is of long-term value	data are expected to be replaced
data updates are frequent	write-once, read multiple (no updates, or at least not often)
predictable, linear growth	unpredictable growth (exponential)
non-programmers writing queries	only programmers writing queries
regular backup	replication
access through master server	sharding across multiple nodes

---

# HDFS

---

- Hadoop Distributed File System
- Škálovatelný a opensource

## Apache Hadoop

- opensource framework s nástroji pro zpracování Big Data
- vyšel z Google MapReducs a GFS (Google File System)
- nástroje (komponenty) Hadoopu:
  - HDFS - distribuovaný file systémů
  - Hadoop YARN - scheduling a resource management clusterů
  - Hadoop MapReduce - paralelní zpracování dat

## Fault tolerance v HDFS

- *"failure is the norm rather than exception"*
- očekáváme, že vždy nějaká část nefunguje
- detekce chyb a auto recovery

## Typ dat v HDFS

- data proudí ve streamu
- automatický batch Processing
- write-once / read-many

## Nodes v HDFS

- Master / Slave architektura
- HDFS využívá celý FS namespace
- Soubory jsou děleny na bloky (64MB, 128MB apod)

## NameNode

- master server
- řídí namespace
- regulace přístupu klientů
- řeší operace se soubory a adresáři
- určuje mapování bloků na DataNodes
- přijímá **HeartBeat** a **BlockReport** od DataNodes

## Jak funguje NameNode

- Používá transakční log - **EditLog**
  - Zaznamenává všechny změny v meta datech FS (tvorba souboru, změna repliačního faktoru)
  - je uložen ve FS NameNodu
- **FsImage** - celý FS namespace s mapováním bloků na soubory
  - opět uložen v lokálním FS NameNodu

- Je načten do paměti NameNodu (4GB RAM stačí)

### Proces zapnutí systému z pohledu NameNode:

1. NameNode přečte FsImage a EditLog z disku
2. Aplikuje všechny operace v EditLogu na FsImage reprezentaci
3. Udělá **CheckPoint** - Flush out této verze systému do nového FsImage
4. Zkrátí EditLog

### DataNode

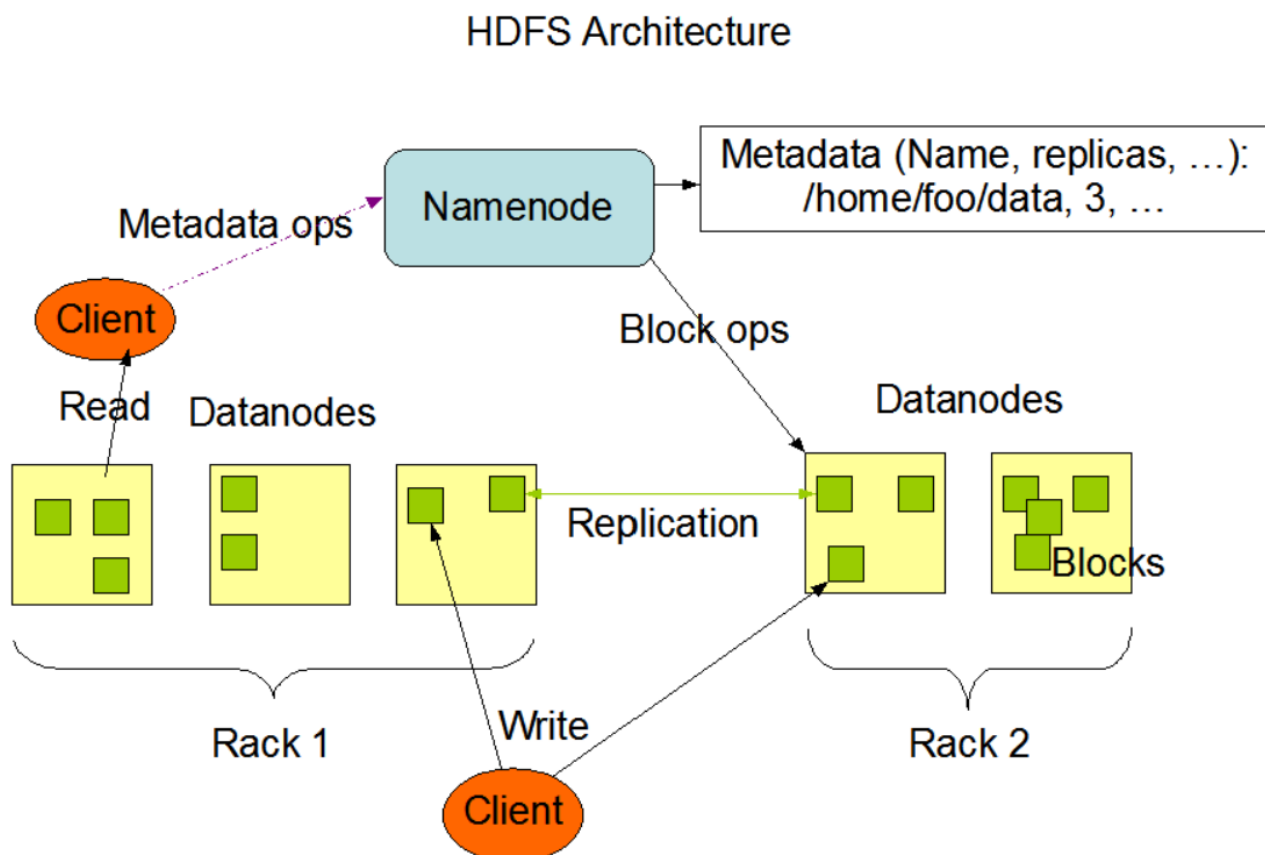
- provádí r/w requesty
- práce s bloky - tvorba, mazání a replikace na příkaz NameNode

### Jak funguje DataNode

- Ukládá data do souborů ve vlastním FS
- Každý blok HDFS je samostatný soubor
- Netvoří všechny soubory ve stejném adresáři (využívá nějakou heuristiku)

### Proces zapnutí systému z pohledu DataNode:

1. Generace seznamu všech svých HDFS bloků = **BlockReport**
2. Odešle **BlockReport** NameNodu



### Namespace

- Hierarchický FS
- CRUD operace
- NameNode je správcem FS - zaznamenává změny
- Aplikace si specifikuje replikační faktor a to je uloženo v NameNode

## Replikace

- soubor je rozdělen na posloupnost bloků
- bloky mají stejnou velikost až na poslední
- velikost bloku lze nastavit
- replikace je nastavitelná
- zajišťuje fault toleranci

## Umístění replik

### Rack-aware

- bereme v potaz fyzickou lokaci uzly
- uzly jsou děleny do racků
- racky mezi sebou komunikují skrze switche
- NameNode určí rack id pro každou DataNode
- jednoduchý přístup: umístíme uzly do jiných racků => drahé zápisy

### Standardní přístup

- replikační faktor je 3
- Repliky jsou umístěny následovně:
  - Jedna v uzlu na lokálním racku
  - Jedna na jiném uzlu v lokálním racku
  - Jedna na uzlu v jiném racku

## Chyby

### Network Failure

- ztráta připojení DataNodes k NameNode
- DataNodes přestanou posílat heartbeat a jsou NameNode označeny
- DataNode k nim neposílá I/O požadavky

### DataNode Failure

- může klesnout počet replik pod replikační faktor => nutnost re-replikace

## MapReduce

---

- Využívá paradigma **Rozděl a panuj**

## Faze Map

- Input: key/value páry
- Output: množina dočasných key/value párů - typicky jiná doména než input
- formálně:  $(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$

## Faze Reduce

- Input: Dočasný klíč a množina všech hodnot pro ten klíč
- Output: Menší množina hodnot ve stejné doméně
- formálně:  $(k_2, \text{list}(v_2)) \rightarrow \text{list}(k_2, \text{possibly smaller list}(v_2))$

## Příklady MapReduce:

### Word Frequency

```
map(String key, String value):  
    // key: document name  
    // value: document contents  
    for each word w in value:  
        EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):  
    // key: a word  
    // values: a list of counts  
    int result = 0;  
    for each v in values:  
        result += ParseInt(v);  
    Emit(key, AsString(result));
```

## Části MapReduce

### Input reader

- dělí vstup na části => každá část nalezí jedné map funkci
- generuje key / value páry

### Map funkce

- uživatelem specifikované zpracování key / value páru

### Partition funkce

- dostane klíče z map funkce a počet reducerů
- vrátí index reduceru, který se má použít
- typicky zahesujeme klíč a moduluje počtem reducerů

### Compare funkce

- setridi vstup do reduce funkce

## Reduce funkce

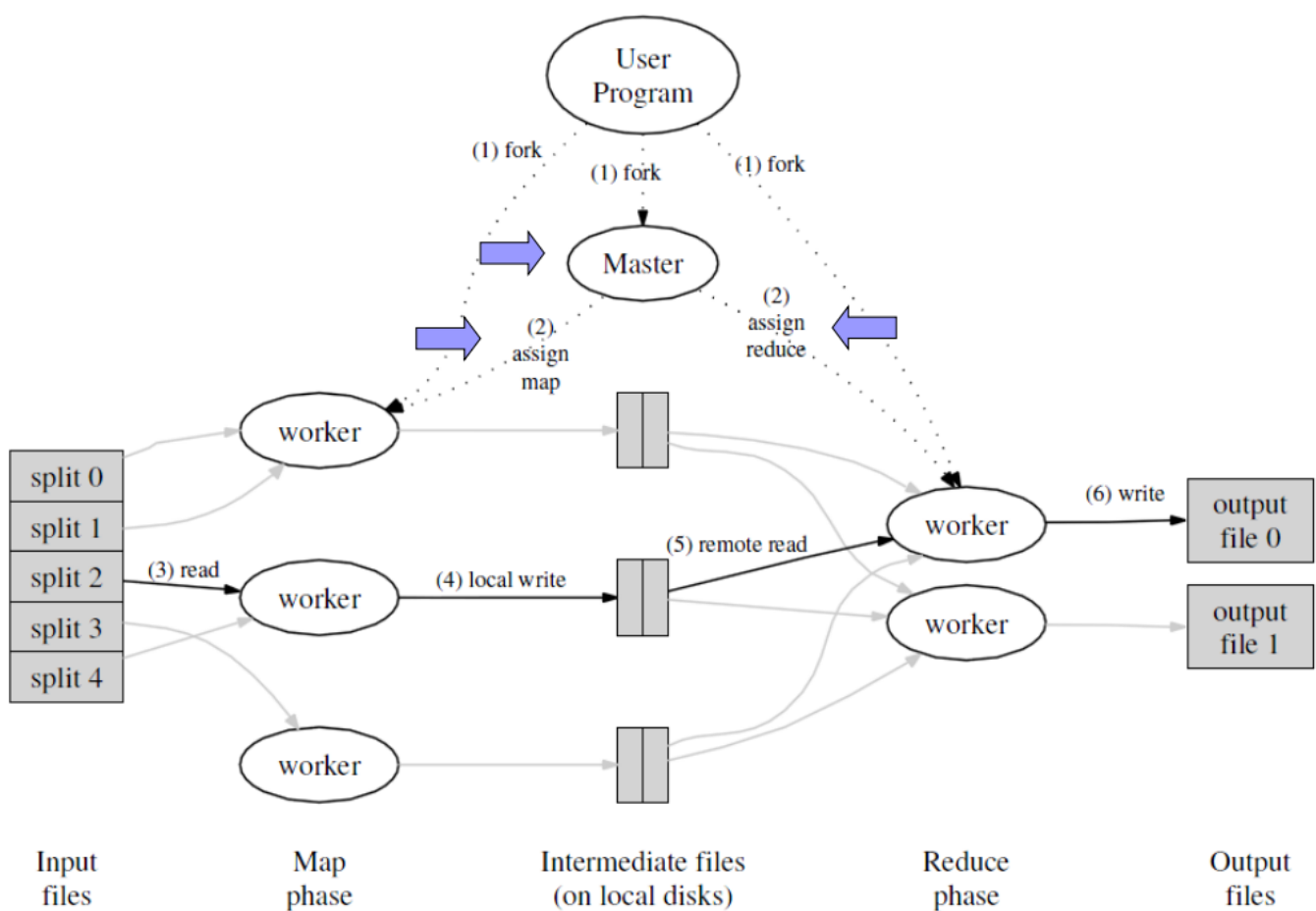
- uzivatelem specifikovane zpracovani key / values

## Output writer

- zapise vysledek reduce funkce do stabilniho uloziste

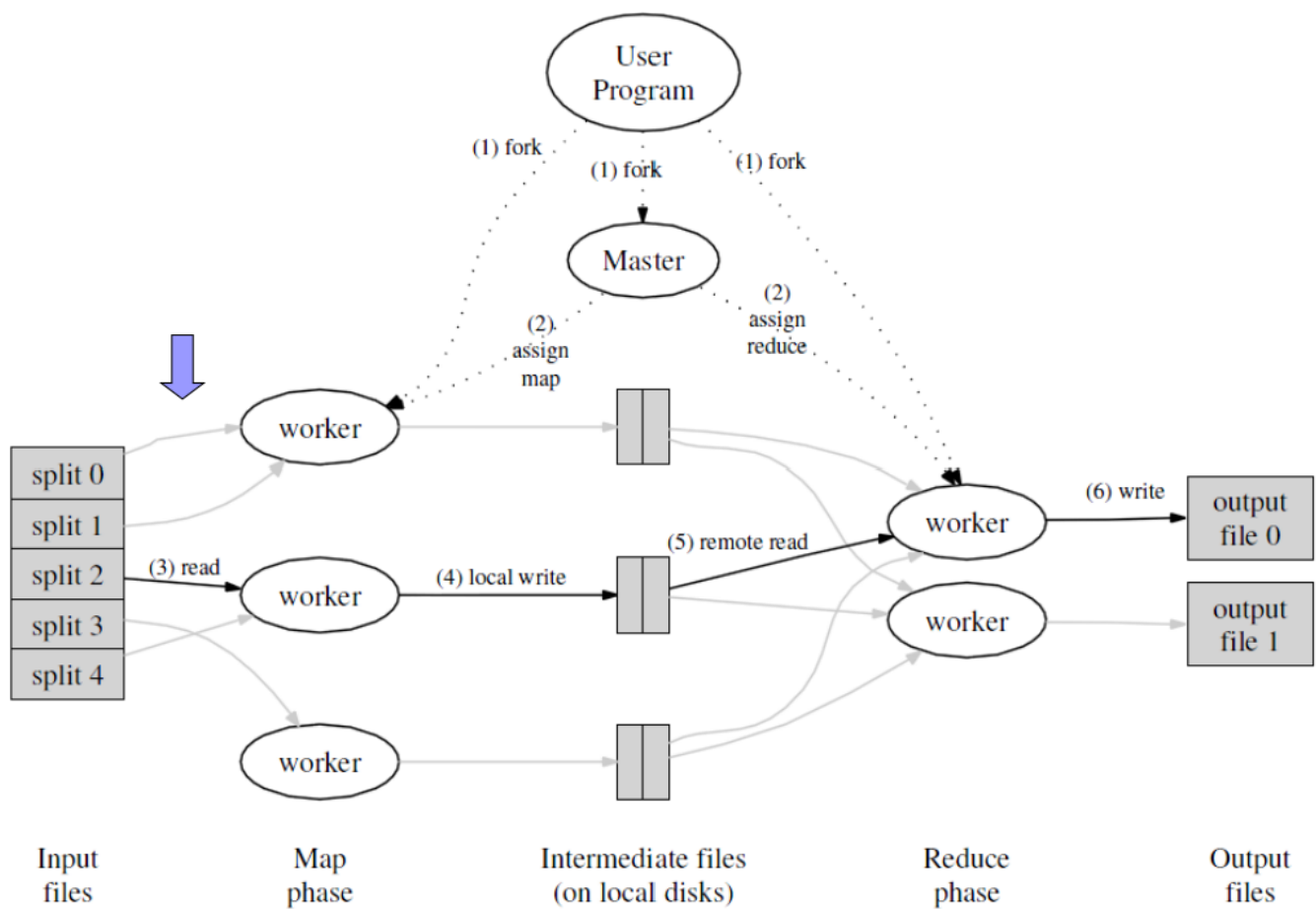
## Prubeh mapreduce

1. MapReduce knihovna rozdeli vstupni soubory do M casti (16MB - 64MB na cast)
2. Start kopii mapreduce na clusteru
3. Mame M map tasku a R reduce tasku
4. Master vybere IDLE workera a priradi mu jednu map nebo reduce tasku



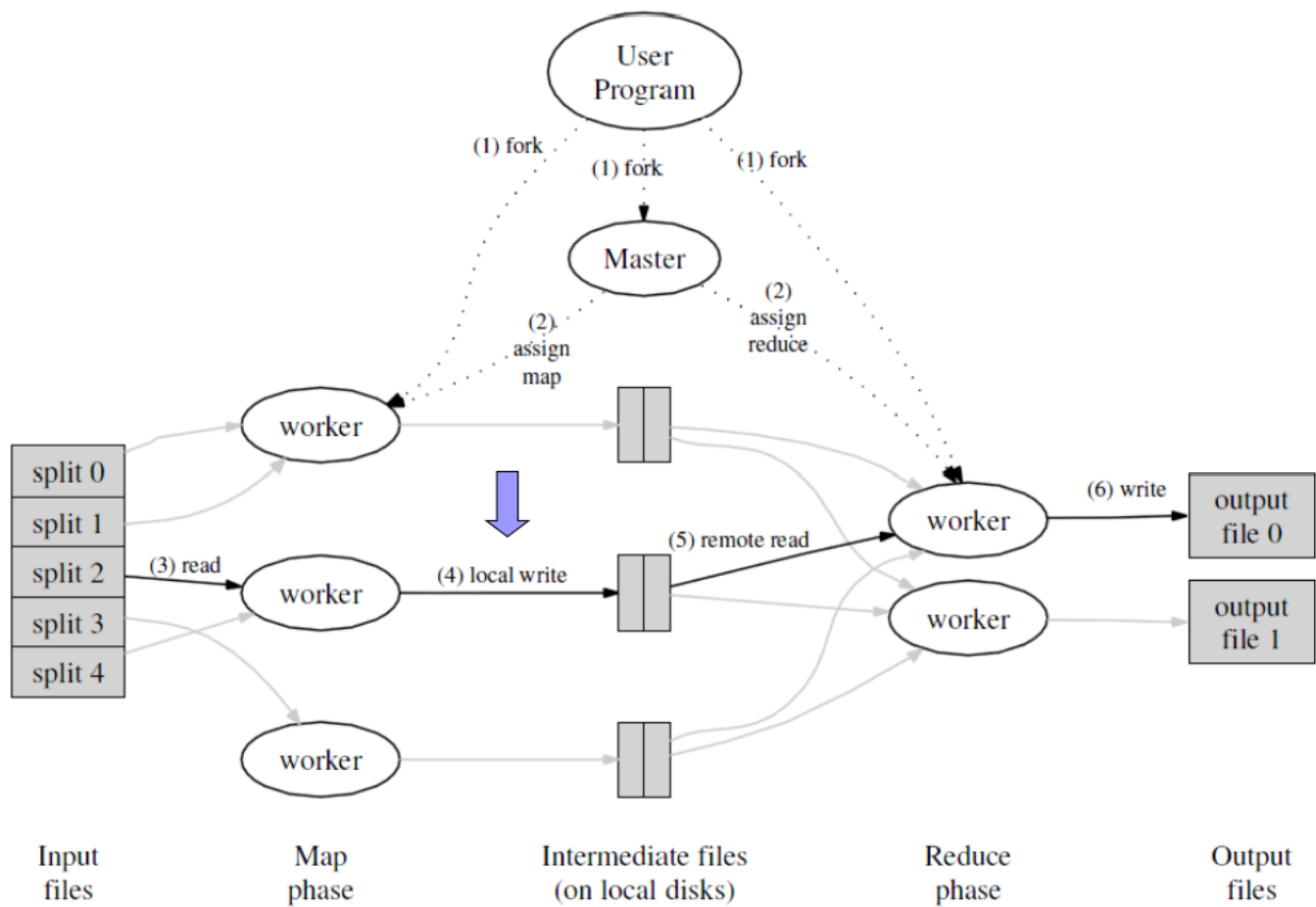
5. Worker s Map taskem precete obsah splitu na inputu
6. Naparsuje key/value pary ze vstupu
7. Preda pary do uzivatelem specifikovane Map funkce
8. Jsou vytvoreny docasne key/value pary a ulozeny do pameti



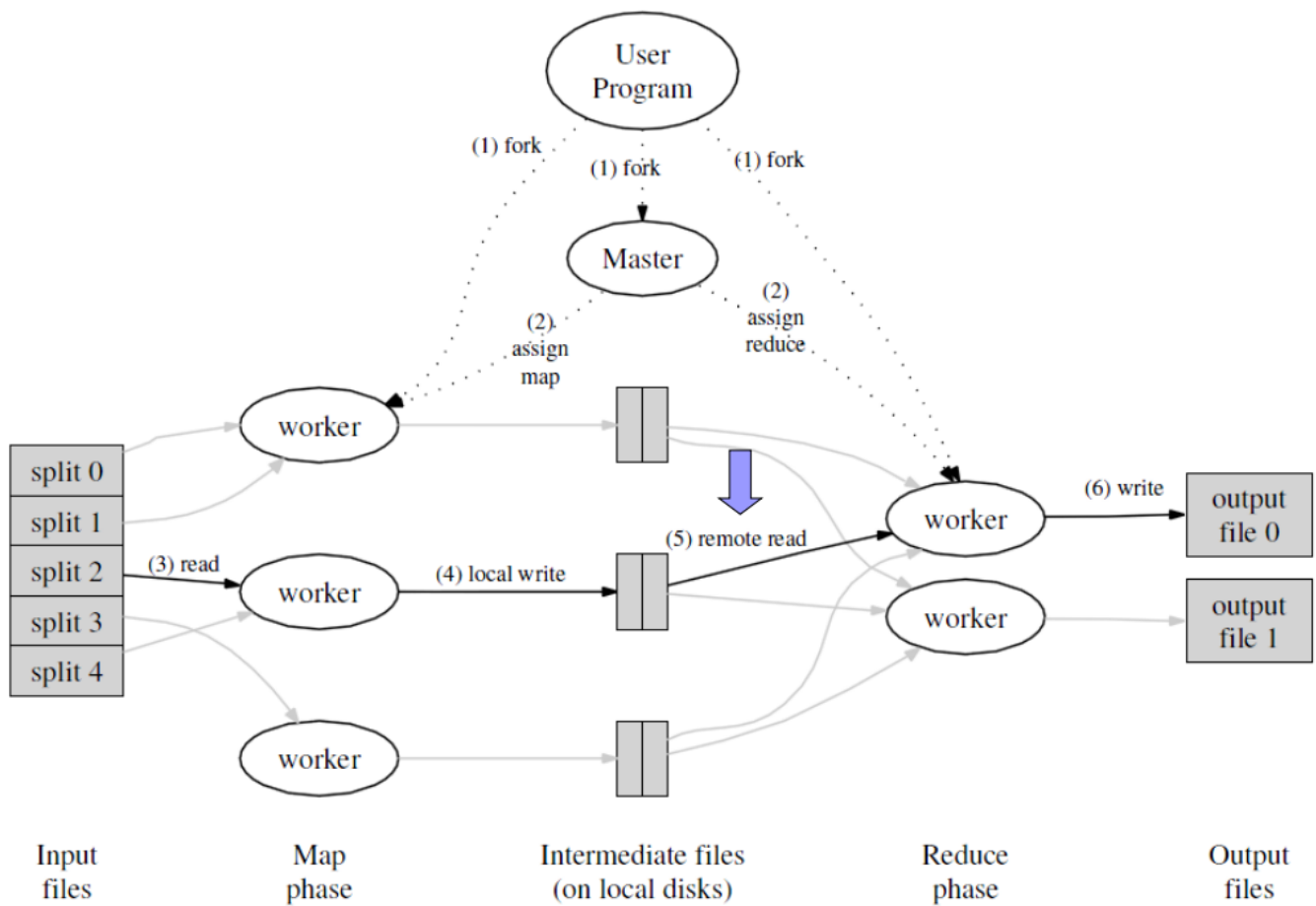


9. Key / value pary z pameti jsou periodicky zapisany na lokalni disk

10. Poloha paru na disku je forwardovana masterovi

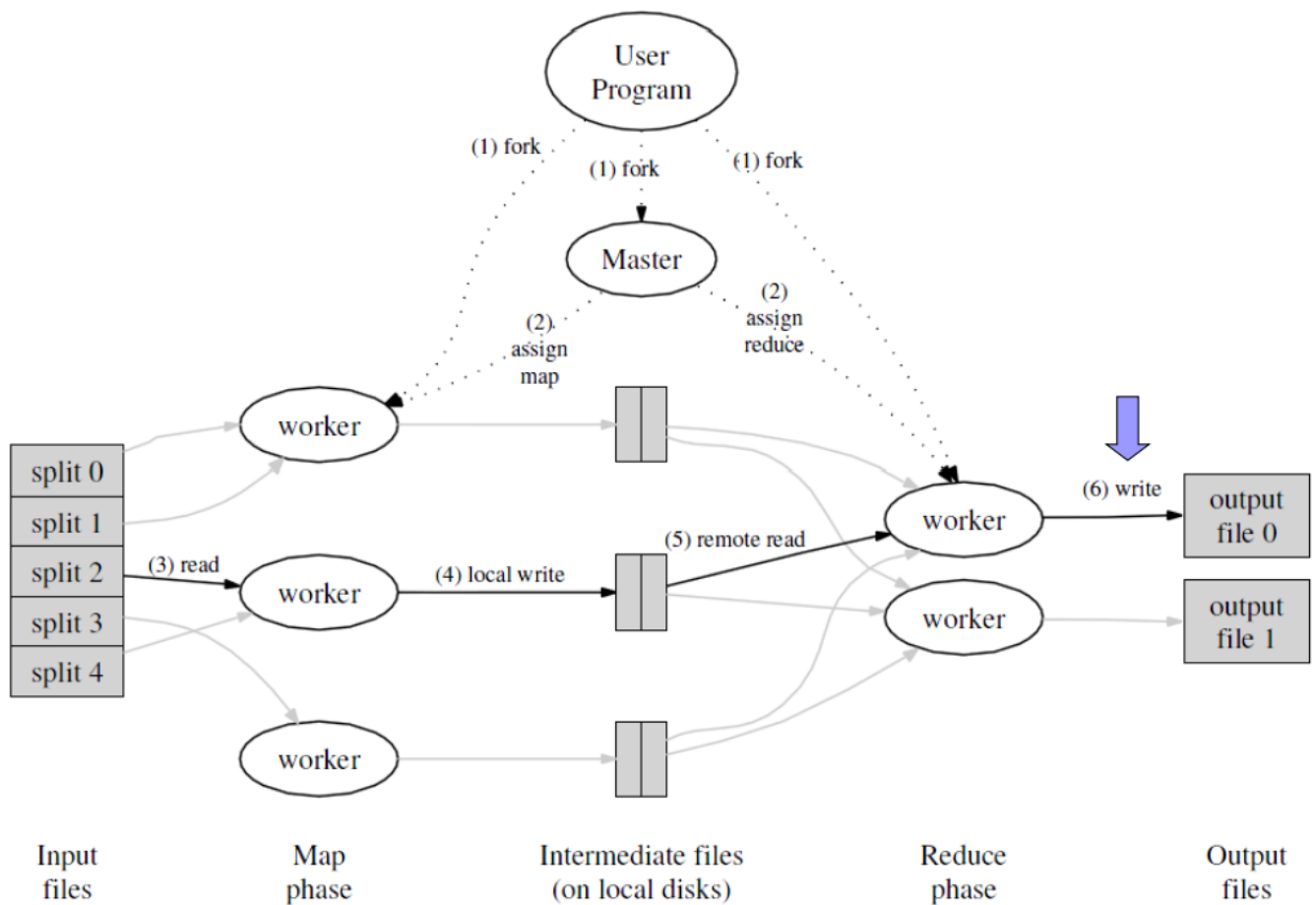


11. Master notifikuje Reduce workery o poloze dat na disku
12. Reduce worker pouzije RPC k ziskani dat z lokalniho disku map workeru
13. Az ma reduce worker vsechna docasna data, setridi je podle docasných klicu



14. Reduce worker iteruje pres setrizená dočasná data

15. Pro každý dočasný kľíč pripne jeho hodnotu do output filu pro túto reduce partition



## Combine funkce

- uživatelem specifikovaná funkce
- něco jako reduce funkce ale jeste v map fazi
- bezi pres lokalni data v mapperu
- slouzi ke kompresi posilaneho souboru

## Counters

- uživatel muze priradit pocitadlo k jakékoli akci mapperi / reduceru

## Fault Tolerance

### Worker Failure

- master pinguje workery
- pokud worker neodpovida, je oznacen za failed
- vsechny jeho tasky jsou naplanovany zpet do puvodniho idle stavu
- jeho tasky si pak rozeberou od zacatku jini workeri

### Master Failure

- 2 strategie

## Strategie A

- master si dela periodicke checkpointy master datovych struktur
- pokud master zemre, nova kopie je nastartovana z pozice posledniho checkpointu

## Strategie B

- Jeden master -> jeho selhani je malo pravdepodobne
- Pokud selze, cely prubeh MapReduce se zahodi

## Stragglers

- "struggler"
- stroj, ktery je pomaly -> nektre casti mu trvaji nezvykle dlouho

### Reseni straggleru:

- tesne pred dokoncenim MapReduce operace master naplanuje backup executions zbylych zapocatych tasku
- task je oznacena za hotovou, pokud jeji primarni nebo backup vykonavani je dokonceno

## Granularita tasku

- M casti Map faze
- R casti Reduce faze
- Master provede  $O(M + R)$  scheduling rozhodnuti
- Master uchovava  $O(M * R)$  status informaci v pameti
- R je typicky omezeno uzivatelem

# Hadoop MapReduce

---

- HDFS + JobTracker (master) + TaskTracker (slave)

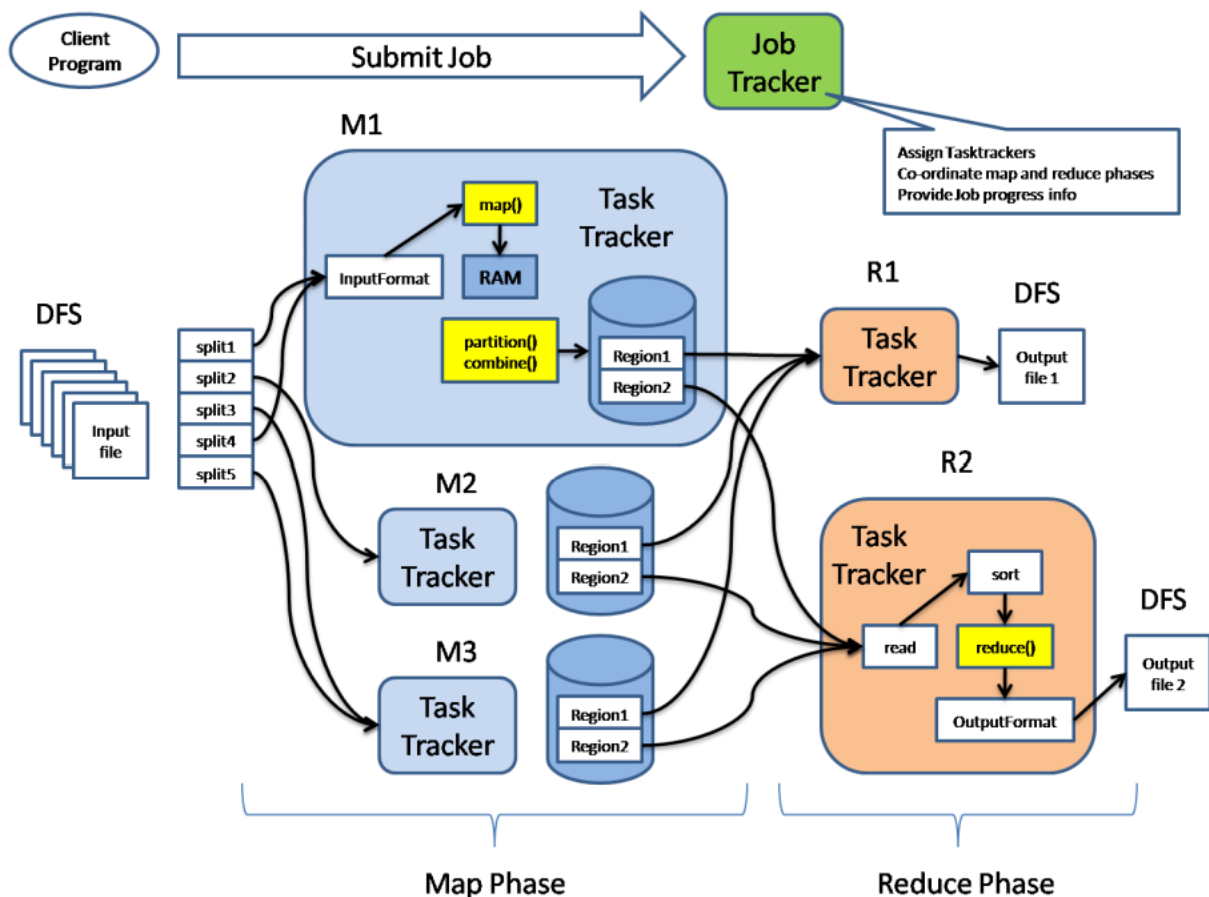
## JobTracker

- Master
- pracuje jako scheduler - deleguje praci do TaskTrackeru
- komunikuje s NameNode (HDFS master) a vyhleda TaskTracker (Hadoop client) pobliz dat
- presune skutecnou praci do TaskTracker uzlu

## TaskTracker

- client
- Prijima tasky od JobTrackeru
- Map, Reduce, Combine
- Input a output cesty
- Ma omezeny pocet task slotu
- Pro kazdy task vyrobi novou instanci JVM (Java Virtual Machine)

- Pocet volnych slotu posila pres heartbeat JobTrackeru



## Apache Spark

- data analytics engine
- narozdil od Hadoop MapReduce je rychlejsi diky praci in memory narozdil od disk I/O, který zpomaloval MapReduce
- podporuje taky Spark SQL

### Driver program

- Spark Aplikace = driver program
- Obsahuje uzivatelem specifikovany kod pro dany problem a provadi orchestraci
- koordinuje paralelni zpracovani
- nasloucha executorum (worker nodes)
- mela by bezet blizko worker nodes (idealni v jedne LAN)
- Je spravovan skrze Web UI (ukazoval nam Yaghob v Cloudu)

### SparkContext

- centralni entita v driver programu
- koordinace mezi driverem a cluste managerem
- ridi resourcy a provadeni tasku

### ClusterManager

- spark si muze vybrat Cluster Managera
- externi system, ktery alokuje resource
- Prikklady: Kubernetes, YARN (Resource Manager Hadoopu), Apache Mesos

## Resilient Distributed Dataset (RDD)

- imutabilni
- kolekce elementu rozdelenych mezi uzly v clusteru
- automaticka recovery
- lze persistovat v pameti (volani `persist` nebo `cache` funkci)
- paralelni zpracovani

### Jak vytvorit RDD

- paralelizujeme existujici kolekci v driver programu

```
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

- reference na externi dataset (treba v HDFS nebo Local file system) podporovany **Hadoopem**

```
JavaRDD<String> distFile = sc.textFile("data.txt");
```

## RDD operace

- delime na Transformace a Akce

## RDD Transformace

- vytvoreni noveho datasetu z existujiciho
- prakticky `map` funkce

### 1. `map(func)`

- **Popis:** Vytvoří nové rozdělené dataset (RDD), kde každý prvek původního datasetu je transformován funkcí `func`.

```
// Příklad: Zvětší každý prvek o 1
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));
JavaRDD<Integer> mappedRDD = rdd.map(x -> x + 1);
```

---

### 2. `union(otherDataset)`

- **Popis:** Spojí dva dataset (RDD) a vrátí nový dataset, který obsahuje všechny prvky z obou.

```
// Příklad: Spojení dvou datasetů
JavaRDD<Integer> rdd1 = sc.parallelize(Arrays.asList(1, 2, 3));
JavaRDD<Integer> rdd2 = sc.parallelize(Arrays.asList(4, 5, 6));
JavaRDD<Integer> unionRDD = rdd1.union(rdd2);
```

---

### 3. filter(func)

- **Popis:** Vrátí nový dataset, který obsahuje pouze prvky, na kterých funkce `func` vrátí `true`.

```
// Příklad: Filtrace sudých čísel
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));
JavaRDD<Integer> filteredRDD = rdd.filter(x -> x % 2 == 0);
```

---

### 4. reduceByKey(func, [numPartitions])

- **Popis:** Pokud pracujete s páry `(K, V)`, agreguje hodnoty pro každý klíč pomocí funkce `func`. Volitelně můžete nastavit počet oddílů.

```
// Příklad: Sčítání hodnot podle klíče
JavaPairRDD<String, Integer> pairs = sc.parallelizePairs(Arrays.asList(
    new Tuple2<>("a", 1),
    new Tuple2<>("b", 2),
    new Tuple2<>("a", 3)
));
JavaPairRDD<String, Integer> reducedRDD = pairs.reduceByKey((x, y) -> x + y);
```

---

### 5. sortByKey([ascending], [numPartitions])

- **Popis:** Seřadí páry `(K, V)` podle klíče. Můžete zvolit vzestupné (`true`) nebo sestupné (`false`) řazení.

```
// Příklad: Seřazení párů podle klíče vzestupně
JavaPairRDD<String, Integer> pairs = sc.parallelizePairs(Arrays.asList(
    new Tuple2<>("b", 2),
    new Tuple2<>("a", 1),
    new Tuple2<>("c", 3)
));
JavaPairRDD<String, Integer> sortedRDD = pairs.sortByKey(true);
```

---

Další funkce:



- **intersection**: Vrátí dataset s prvky, které se nachází v obou vstupech.
- **distinct**: Vrátí dataset obsahující pouze unikátní prvky.

```
// Příklad: Unikátní prvky
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 2, 3, 4, 4));
JavaRDD<Integer> distinctRDD = rdd.distinct();
```

## RDD Akce

- vrátíme hodnotu do driveru po nějaký vypočtu nad datasetem
- prakticky **reduce** funkce

### 1. reduce(func)

- **Popis**: Agreguje prvky datasetu pomocí funkce **func**. Funkce musí být **komutativní** a **asociativní**, aby výpočet mohl probíhat paralelně.

```
// Příklad: Součet všech čísel v datasetu
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));
Integer sum = rdd.reduce((x, y) -> x + y);
```

---

### 2. count()

- **Popis**: Vrátí počet prvků v datasetu.

```
// Příklad: Spočítání prvků v datasetu
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));
long count = rdd.count();
```

---

### 3. first()

- **Popis**: Vrátí první prvek datasetu.

```
// Příklad: Získání prvního prvku
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));
Integer firstElement = rdd.first();
```

---

### 4. take(n)

- **Popis**: Vrátí pole s prvních **n** prvky datasetu.

```
// Příklad: Získání prvních 2 prvků
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));
List<Integer> firstTwo = rdd.take(2);
```

## 5. takeOrdered(n, [ordering])

- **Popis:** Vrátí prvních **n** prvků datasetu, seřazených buď podle jejich přirozeného pořadí, nebo podle vlastní komparace.

```
// Příklad: Získání 2 nejmenších prvků
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(4, 2, 3, 1));
List<Integer> smallestTwo = rdd.takeOrdered(2); // Výsledek: [1, 2]

// Příklad s vlastním pořadím (sestupně)
List<Integer> largestTwo = rdd.takeOrdered(2, Comparator.reverseOrder()); //
Výsledek: [4, 3]
```

## Shuffle Operace

- nektě akce spouští shuffle
- shuffle = mechanismus pro redistribuci dat přes partitions
- nutnost kopírování dat mezi executors a stroji

### Příklad Shuffle Operace: ReduceByKey

- Problém: Hodnoty stejného klíče mohou být v různých partitions nebo na různých strojích v clusteru
- Řešení (Shuffle): Spark načte hodnoty stejného klíče ze všech partitions, spojí je dohromady a spočítá finální výsledek

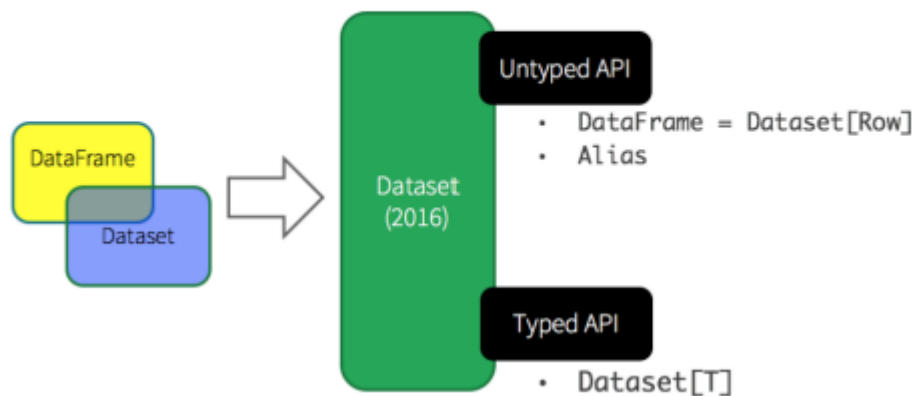
## RDD vs DataFrame vs Dataset

- RDD = primární API ve Sparku, neobsahuje optimalizace jako DataFrame nebo Dataset
- DataFrame = data jsou organizovány do pojmenovaných sloupců
  - Weak typing: `Dataset<Row>`
  - Kolekce generických objektů
  - Jednodušší práce s daty (jako tabulka v SQL)
- Dataset = distribuovaná kolekce dat
  - Strong typing: `Dataset<T>`, kde **T** je definice třídy
  - Někdy jako RDD s podporou Spark SQL zároveň

Spark 2.0 sjednotil DataFrame a Dataset do jedné struktury s dvěma API:

Nepřísně typované API (DataFrame): Pro jednoduchost a SQL-like operace. Silně typované API (Dataset): Pro typovou bezpečnost a práci s konkrétními třídami.

## Unified Apache Spark 2.0 API



## Principy MDBS

---

- vzdáme se některých ACID vlastností
- Ze silné konzistence -> slabá konzistence

## Scalability

### Vertikalni scaling (scaling up)

- v historii preferovano
- zarucovalo strong consistency (protoze stacil jen jeden stroj)
- Vendor lock-in
- drahe
- stale existuje limit pro silu a kapacitu jednoho stroje

### Horizontalni scaling (scaling out)

- system distribujeme pres vice stroju / uzlu
- staci komoditni hardware

### Klamy (fallacies) horizontalniho scalingu

- Sit je spolehliva
- Nulova letence
- Nekonecny bandwidth
- Sit je bezpecna
- Topologie se nemeni
- Mame pouze jednoho spravce
- Nulova cena transportu
- Sit je homogenni

## ACID

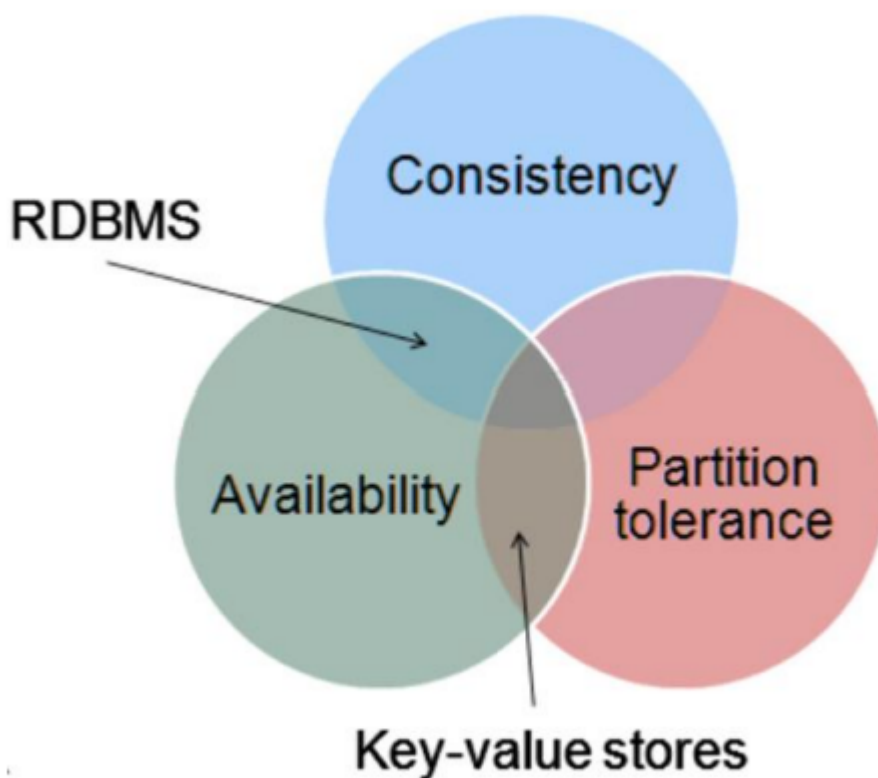
- typicke vlastnosti ocekavane u relacnich DBMS

- Databzova transakce = jednotka prace (sekvence operaci) v DBMS
- tyto vlastnosti jsou ale příliš drahé v distribuovaných systémech
- Atomicity - vše nebo nic, jedna selhala část transakce = celá transakce selhala
- Consistency - databáze se přesouvá pouze mezi konzistentními stavy
- Isolation - efekty nedokončené transakce (v průběhu, failed) nejsou viditelné zvenku
- Durability - po commitu transakce zůstane transakce committed (i přes výpadek elektriny, error)

## CAP Theorem

- CAP má 3 části
- Obecně ale nedává smysl, protože definice nejsou dostatečně přesné (např. pouze CP by naznačovalo nikdy available)

**Theorem:** Only 2 of the 3 guarantees can be given in a “shared-data” system



### Consistency

- po změně dat, všechny čtení mají vidět stejná data
- všechny uzly mají vždy obsahovat stejná data

### Availability

- všechny dotazy (čtení, zápisy) dostanou vždy odpověď
- nezávisle na výpadcích

### Partition Tolerance

- system funguje i po izolovani podcasti systemu
- problemy s pripojenim neshodi system pokud je system fault tolerantni

## BASE

- lepe skalovatelny
- sada principu jako ACID

### Basically Available

- system funguje prakticky vetsinu casu
- castecne vypadky se deji ale bez selhani celeho systemu

### Soft State

- system se neustale meni
- stav systemu je nedeterministicky (kontrast vuci consistency v ACIDu, kde vzdy mame nejaky pevny stav)

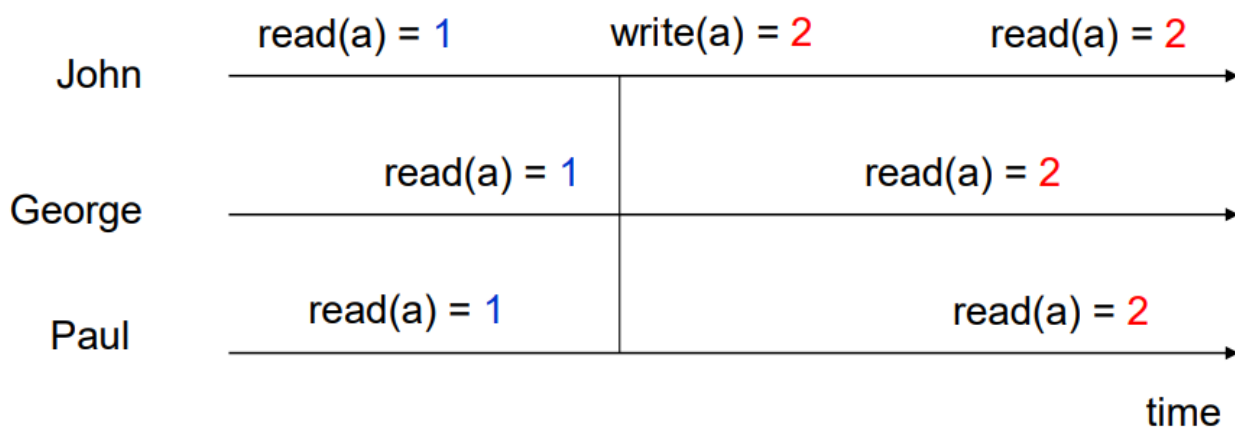
### Eventual Consistency

- nekdy v budoucnu bude system konzistentni (az treba vsechny stroje budou synced)

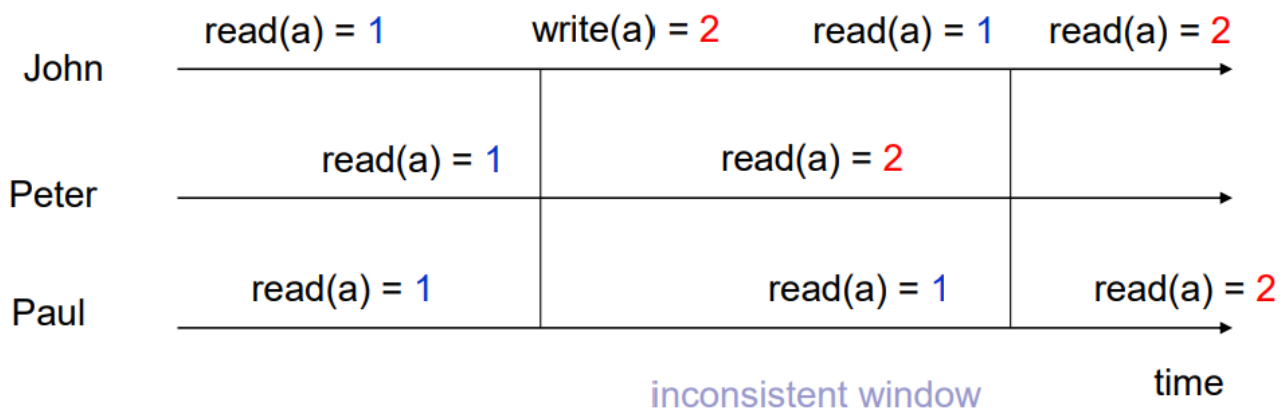
## ACID vs BASE

- ACID garantuje *Consistency* a *Availability*
  - pesimisticky pristup
  - toto dovoluje DB pouze na jednom stroji
- BASE garantuje *Availability* a *Partition tolerance*
  - optimisticke
  - distribuovane databaze
- samostatny system je **CA** system

## Silna konzistence



## Eventualni konzistence



## Distribucni modely

- Horizontalni scaling = databaze bezi na clusteru serveru
- Mame dva ortogonalni pristupy (= pristupy, které mohou být aplikovány zároveň. Jsou v jiných dimenzích / pohledech na věc)
  - **Replikace** - kopírování stejných dat přes uzly (master-slave nebo peer-to-peer)
  - **Sharding** - jiná data na jiných uzlech

### Single server

- bez jakékoliv distribuce
- DB pouze na tomto stroji
- Dobře třeba pro Grafové DB -> složitá distribuce

### Sharding

- dáváme různá data na různé uzly
- ideálně chceme pohromadě data, ke kterým přistupujeme často dohromady
- selhání uzlu -> jeho data jsou nedostupná (proto často kombinujeme s replikací)

### Rozmístění uzlu

- Jeden uživatel bere data z jednoho serveru
- Fyzická poloha
- Distribujeme rovnoměrně přes uzly

### Master-slave Replikace

- jeden uzel je primární (master), zbytek sekundární (slaves)
- master zodpovídá za zpracování a update dat
- master limituje svoji schopnosti zpracování updatu

- **Problemy:**
  - skalovatelnost zapisu (master je bottleneck)
  - nechrani pred selhanim mastera

## Volba mastera

- Manualni: user-defined
- Automaticka: cluster-elected

## Peer-to-peer replikace

- resi mnoho problemu master-slave replikace
- bez mastera
- Problem: konzistence
  - zapisem na 2 mista vzniká write-write konflikt
- Reseni:
  - pri zapisu dat repliky koordinuji pro zabraneni konfliktu
  - vsechny repliky nemusi souhlasit, staci vetsina

## Kombinace Shardingu a Replikace

### Master-slave replikace a sharding

- vice masteru, ale master je pouze pro nejaky dany datovy item
- uzel muze byt master pro nejaka data a slave pro jina

### Peer-to-peer replikace a sharding

- casta strategie pro sloupce DB
- idealne replikacni faktor 3, takze kazdy shard je na 3 uzlech

## Konzistence

### Write Consistency (Konzistence zápisu)

- **Problém:** Dva uživatelé chtějí upravit stejný záznam (write-write konflikt).
- **Důsledek:**
  - Ztracený update: Druhá transakce přepíše hodnotu z první transakce.
  - Ostatní transakce čtou nesprávnou hodnotu a vrací špatné výsledky.
- **Řešení:**
  - **Pesimistické:** Zabraňuje konfliktům (např. write lock).
  - **Optimistické:** Konflikty se nechají proběhnout, ale následně se detekují a řeší (např. podmíněný update nebo uložení obou hodnot jako konflikt).

### Read Consistency (Konzistence čtení)

- **Problém:** Jeden uživatel čte, zatímco druhý zapisuje (read-write konflikt).
- **Důsledek:**
  - Nekonzistentní čtení: Hodnota objektu se mezi dvěma čteními změní.

- Transakce, které čtou starou hodnotu, mohou vracet chybné výsledky.
- **Databáze:**
  - **Relační databáze:** Podporují ACID transakce (zajišťují konzistenci).
  - **NoSQL databáze:** Často podporují atomické operace jen v rámci jedné "agregace".
    - Pokud je update napříč více agregacemi, může dojít k **oknu nekonzistence**.
- **Další problém:** Konzistence replikace
  - Zajistit, aby všechny repliky měly stejnou hodnotu při čtení.

### Quorums (Kvóra)

- **Otázka:** Kolik uzlů je potřeba zapojit pro zajištění silné konzistence?
- **Write quorum:** Počet uzlů potvrzujících zápis musí být:  $W > \frac{N}{2}$ 
  - ( N ) = počet uzlů v replikaci (replikační faktor).
  - ( W ) = počet uzlů zapojených do zápisu.
- **Read quorum:** Počet uzlů nutných pro čtení:  $R + W > N$ 
  - ( R ) = počet uzlů kontaktovaných pro čtení.
- **Princip:**
  - Zápisy s konflikty: Pouze jeden může získat většinu.
  - Pro zajištění aktuální hodnoty musíme kontaktovat dostatečný počet uzlů.

## Zpracovani Big Data

---

### Priklady ukolu pro Big Data

- analyza
- visualizace
- agregace
- manipulace a uprava dat

### Cloud computing

- Pronajem hw/sw (servery, data, software...) poptavce
- [Virtualizace a cloud computing předmět](#)

### Modely cloudových služeb:

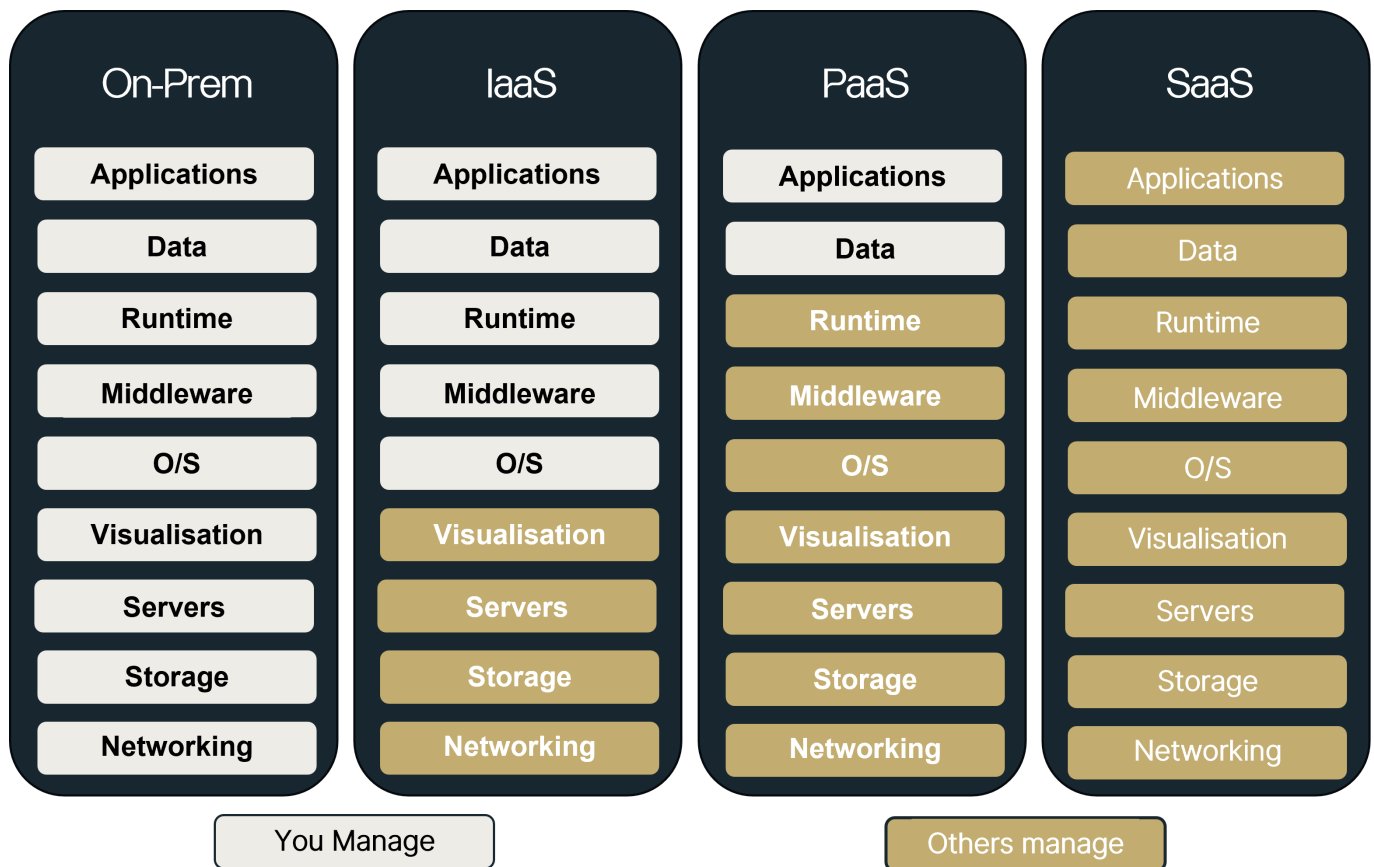
- **Software as a Service (SaaS):**
  - Primo hotovy sw produkt
  - zoom, shopify, slack
- **Platform as a Service (PaaS):**
  - Prostředí pro devs pro nasazení a vývoj včetně HW



- Nastavena DB, security, data security, hosting
- Microsoft Azure, AWS Lambda

- **Infrastructure as a Service (IaaS):**

- primo hw a infrastruktura (nejnizsi model)



## Spojeni Cloud computingu a Big Data

- nemusíme resit drahy HW, instalaci a udrzbu
- jednoduchá skalovatelnost
- nevýhoda je vendor lock-in

## Key-value databaze

- prakticky hash table
- hodnota je BLOB (nespecifikovaný typ a struktura - může být cokoliv)

## Příklady

- Riak
- Redis
- MemcachedDB

## Vhodná využití

- Session info

- klicem je `session_id`
- k ulozeni session staci `put` a pro dotaz jednoduchy `get`
- Nakupni kosiky
  - podobne jako Session info
- User preference

## Nevhodna vyuziti

- Vztahy mezi daty
- Transakce s vice operacemi
  - Ukladame vice klicu -> jedno selhani -> zadny z klicu v transakci se neulozi (revert / roll back)
- Dotazy na obsah dat

## Dotazovani

- dotazujeme se pomoci klice
- pomoci obsahu dat neni mozne (BLOB -> data nemusi byt jakkoliv definovana)
- klice jsou generovany nejakym algoritmem (auto increment), user generated nebo treba time stamps

## Riak (Key-value)

---

- open source

## Terminologie

- `bucket` = namespace pro klice
  - lze pro bucket nastavit replikacni faktor `n_val`
  - `allow_mult` - konkurentni updaty
  - `/riak/<bucket>/<key>`
- `ring`
- `hinted handoff`
- `gossiping`

Oracle	Riak
database instance	Riak cluster
table	bucket
row	key-value
rowid	key

namespace  
for keys

## Principy

- klíče jsou ukládány do bucketů (= namespaces)
- default interface je **HTTP**

## Riak Links

- umožňují tvořit vztahy mezi objekty
- tvoří se přidáním **Link** headeru k objektu (pres HTTP)

## Riak Search

- fulltext search engine
- podpora dotazování na textová data
- použití pro hledání záznamu podle obsahu

### Dotazování v Riak Search

- **Typy dotazů:**
  - **Zástupné znaky:** **Bus\***, **Bus?**
  - **Rozsahy:**
    - **[red TO rum]:** zahrnuje "red", "rum" a všechny mezi nimi
    - **{red TO rum}:** zahrnuje pouze slova mezi "red" a "rum"
  - **Logické operátory:** **(red OR blue) AND NOT yellow**
  - **Prefixové shody:** Vyhledávání podle počátečních písmen
  - **Blízkost:**
    - **"See spot run"~20:** slova v rámci 20 slov

### Proces indexace dat v Riak Search:

1. Načtení dokumentu
2. Rozdělení na pole
3. Rozdělení polí na termíny
4. Normalizace termínů
5. Zápis {Field, Term, DocumentID} do indexu

## Indexování:

```
index <INDEX> <PATH>
```

## Vyhledávání:

```
search <INDEX> <QUERY>
```

## Příklady Riaku

### Příklady použití Riaku

#### Práce s Buckets:

##### 1. Seznam všech buckets:

```
curl http://localhost:10011/riak?buckets=true
```

##### 2. Získání vlastností bucketu **foo**:

```
curl http://localhost:10011/riak/foo/
```

##### 3. Změna vlastností bucketu **foo**:

```
curl -X PUT http://localhost:10011/riak/foo -H "Content-Type: application/json" -d '{"props" : { "n_val" : 2 } }'
```

---

#### Práce s daty:

##### 1. Uložení prostého textu do bucketu **foo**:

```
curl -i -H "Content-Type: plain/text" -d "My text" http://localhost:10011/riak/foo/
```

##### 2. Uložení JSON souboru do bucketu **artists** s klíčem **Bruce**:

```
curl -i -H "Content-Type: application/json" -d '{"name":"Bruce"}' http://localhost:10011/riak/artists/Bruce
```

### 3. Získání objektu:

```
curl http://localhost:10011/riak/artists/Bruce
```

### 4. Aktualizace objektu:

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

### 5. Smazání objektu:

```
curl -i -X DELETE http://localhost:10011/riak/artists/Bruce
```

---

## Práce s Riak Links:

### 1. Přidání alba a propojení s performerem:

```
curl -H "Content-Type: text/plain" -H 'Link: </riak/artists/Bruce>;  
riaktag="performer"' -d "The River"  
http://localhost:10011/riak/albums/TheRiver
```

### 2. Najít umělce, který provedl album **The River**:

```
curl -i http://localhost:10011/riak/albums/TheRiver/artists,performer,1
```

### 3. Najít umělce, kteří spolupracovali s tím, kdo provedl **The River**:

```
curl -i  
http://localhost:10011/riak/albums/TheRiver/artists,_,0/artists,collaborator  
,1
```

## Interní mechanismy Riaku

- **BASE** principy
- používá **quora**

- $N$  = replikační faktor (default = 3)
- Zápis: data musí být zapsána aspoň na  $W$  uzlech
- Čtení: data musí být nalezena aspoň na  $R$  uzlech
- $W$  a  $R$  můžeme nastavit pro každou operaci
- Platí tyto nerovnosti:  $W > \frac{N}{2}$   $R + W > N$
- **Příklad:**
  - Cluster Riaku má:
    - $N = 5$  (počet replik)
    - $W = 3$  (minimální počet uzlů pro potvrzení zápisu)
  - **Zápis je úspěšný, pokud:**
    - Data jsou úspěšně zapsána na více než 3 uzlech
  - **Tolerované výpadky při zápisu:**
    - Cluster zvládne výpadek až  $N - W = 2$  uzlů a stále může provádět zápisy

## Clustering v Riaku

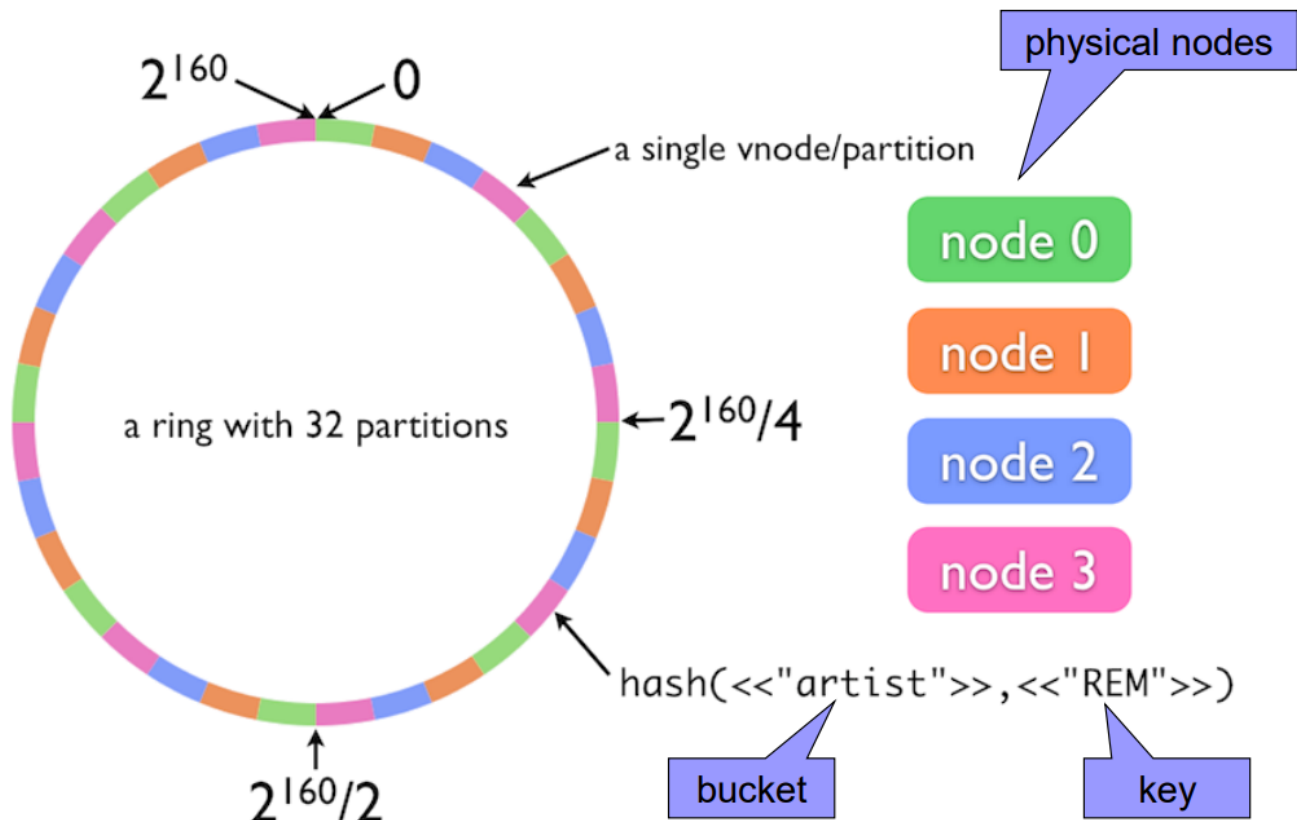
- bez mastera -> každý uzel může obsloužit jakýkoliv dotaz
- Konzistentní hashování
  - hashovací funkce mapuje klíče do kruhu
  - každý uzel zodpovědný za interval hashu (= slot) na kruhu
  - průměrně remapujeme jen  $k / n$  klíčů, kde  $k$  = počet klíčů a  $n$  = počet slotů

## Riak Ring

- střed každého clusteru
- 160-bitový prostor celých čísel rozdělený na rovnoměrné intervaly
- Každý fyzický uzel má virtuální uzly (= vnodes)
  - virtuální uzel je zodpovědný za část klíčů
  - každý fyzický uzel má na starost  $1 / (\text{počet fyzických uzlů})$  ringu
  - **Počet vnodes na každém uzlu:**

$$\text{vnodes\_na\_1\_uzlu} = \frac{|\text{partitions}|}{|\text{fyzicke\_uzly}|}$$

- Příklad:
  - Ring s 32 partitions
  - 4 fyzické uzly
  - 8 vnodes na fyzický uzel



## Replikace v Riaku

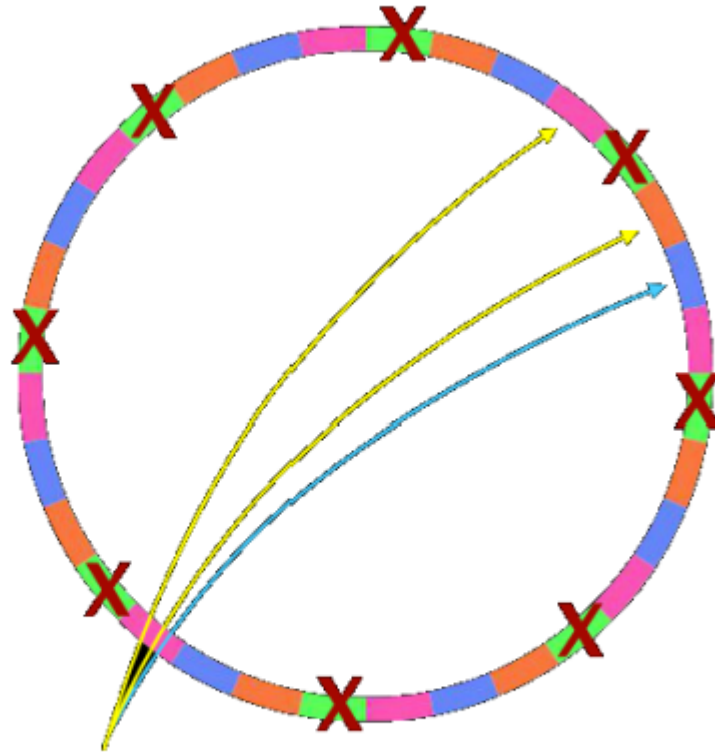
- nastavujeme **N value** (default = 3)
- objekty dedí **N value** z jejich bucketu

## Hinted kandoff

- resi selhani uzlu
- funguje díky replikaci
- Zajistuje high availability Riaku

1. **Selhání uzlu:** Pokud uzel v klastru selže, sousední uzly dočasně převzmou jeho úlohu.
2. **Dočasné převzetí:** Sousední uzly zpracovávají čtení a zápisy, aby zajistily dostupnost systému.
3. **Obnova:** Po návratu selhaného uzlu sousední uzly předají všechny mezitímní změny zpět.

**Výhoda:** Systém zůstává dostupný a data nejsou ztracena.



```
put(<<"artist">>, <<"REM">>)
```

## Gossip protokol

- robustní šíření informací
- **Gossiping** = posílání informací náhodnému uzlu
  - aktualizuje info a clusteru
- každý uzel gossipuje
  - periodicky
  - při změně na ringu

## Vector clocks

- každý uzel může zpracovávat dotaz -> jaká verze hodnoty je ale aktuální?
- řešení: vector clocks
- každá uložená hodnota je tagged vector clockem
- uloženo v headeru objektu
- při každém updatu je hodnota vector clocku aktualizována

## Riak siblings

- **siblings** = vícero objektů pod jedním klíčem
- aktivováno `allow_mult = true` příznakem
- mohou vzniknout při konkurentním zápisu, starých vector clocks, neexistujících vector clocks



## Koordinující uzel (vnode) v Riaku

1. Najde **vnode** pro klíč pomocí hashovací funkce.
2. Určí další **N-1 vnodes** pro repliky.
3. Odešle požadavek na všechny vybrané **vnodes**.
4. Čeká, dokud dostatečný počet odpovědí nesplní **kvórum** (pro čtení/zápis).
5. Vráťi výsledek klientovi.

## Redis (Key-value + multi-model)

---

- spíše dokumentová multi-model databáze s podporou key-value

### Terminologie

### Principy

- klíče jsou **binary safe** -> jakákoliv binární posloupnost může být klíčem (tedy není omezení na text nebo čitelný obsah)
- hodnota může být jakýkoliv objekt (string, hash, list, set...)
- podpora pro množinové operace (range, diff, union, intersection)

### In-Memory Data Set

- data jsou primárně uložena v paměti
- persistence je řešena dumpem datasetu na disk / přidáním příkazu do logu

### Publish/subscribe

### Cache-like chování

- klíče mohou mít nastavený **TTL**
- pak jsou automaticky vymazány -> cache charakteristika

## Datové typy Redisu

### String

- **Binary safe:** Klíč může obsahovat libovolnou binární sekvenci.
- **Maximální velikost:** 512 MB.
- **Operace:**
  - Nastavení a načtení: **SET**, **GET**.
  - Modifikace: **APPEND**, **STRLEN**, **SETRANGE**.
  - Operace s čísly: **INCR**, **DECR**, **INCRBY**, **DECRBY**.
  - Bitové operace: **GETBIT**, **SETBIT**, **BITCOUNT**.

### Příklad:

```
> SET count 10
OK
> INCR count
(integer) 11
> GET count
"11"
> DEL count
(integer) 1
```

---

## List

- **Seřazený seznam řetězců:** Prvky jsou uspořádány podle pořadí vložení.
- **Maximální délka:** Více než 4 miliardy prvků.
- **Operace:**
  - Přidání: **LPUSH** (hlava), **RPUSh** (konec), **LINSERT**.
  - Odebrání: **LPOP**, **RPOP**, **LRM**.
  - Přístup k prvkům: **LRANGE**, **LINDEX**.
  - Délka seznamu: **LLen**.

### Příklad:

```
> LPUSH animals cat
(integer) 1
> RPUSh animals dog
(integer) 2
> LRANGE animals 0 -1
1) "cat"
2) "dog"
```

---

## Set

- **Neuspořádaná kolekce unikátních řetězců.**
- **Maximální velikost:** Více než 4 miliardy prvků.
- **Operace:**
  - Přidání/Odebrání: **SADD**, **SREM**.
  - Test členství: **SISMEMBER**.
  - Množinové operace: **SUNION**, **SINTER**, **SDIFF**.

### Příklad:

```
> SADD colors red green blue
(integer) 3
> SINTER colors:1 colors:2
1) "green"
```

---

## Sorted Set

- Seřazená kolekce s hodnotami přiřazenými skóre.
- Operace:
  - Přidání/Odebrání: **ZADD**, **ZREM**.
  - Počítání: **ZCARD**, **ZCOUNT**.
  - Získání prvků podle skóre: **ZRANGEBYSCORE**.

### Příklad:

```
> ZADD scores 10 Anna 20 John
(integer) 2
> ZRANGE scores 0 -1
1) "Anna"
2) "John"
```

---

## Hash

- Mapa mezi poli a hodnotami řetězců.
- Operace:
  - Nastavení/Načtení: **HSET**, **HGET**, **HMSET**.
  - Všechny hodnoty/pole: **HGETALL**, **HKEYS**, **HVALS**.
  - Smazání: **HDEL**.

### Příklad:

```
> HSET user:id name Sara age 25
(integer) 1
> HGET user:id name
"Sara"
> HGETALL user:id
1) "name"
2) "Sara"
3) "age"
4) "25"
```

## Transakce v Redisu

- každý příkaz je atomický
- podporuje transakce při použití více příkazů (zachová pořadí) -> vše v jedné atomické operaci
- bez roll backu

```
> MULTI // start definice transakce
OK
```

```
> INCR foo
QUEUED
> INCR bar
QUEUED
> EXEC // provedeni transakce
1) (integer) 1
2) (integer) 1
```

## Replikace v Redisu (master-slave)

- master-slave
  - master ma vice slavu
  - uzel muze byt master a slave zaroven
- replikace je **neblokujici** na strane **mastera**
  - pri syncu slavu master pracuje dal
- replikace je **neblokujici** na strane **slavu**
- pri syncu slavu slave pracuje dal

## Synchronizace v Redisu

1. Po připojení k masteru slave odešle příkaz **SYNC**.
2. Master spustí **background saving** a začne ukládat nové příkazy do bufferu.
3. Po dokončení uložení master přenesou celý soubor databáze na slave.
4. Slave uloží soubor na disk a načte jej do paměti.
5. Master pošle slave také všechny **bufferované příkazy**.

## Sharding v Redisu

### Redis Cluster (od verze 3.1)

- **Nepoužívá konzistentní hashování.**
- Klíče jsou přiřazeny do **16384 hash slotů** (CRC16 klíče modulo 16384).
- **Každý master uzel spravuje subset hash slotů.**

#### Příklad:

- 3 uzly:
  - **Node A:** Hash sloty 0–5500
  - **Node B:** Hash sloty 5501–11000
  - **Node C:** Hash sloty 11001–16383
- Přidání uzlu **D:** Některé sloty z A, B, C se přesunou na D.
- Odebrání uzlu **A:** Jeho sloty se přesunou na B a C.

**Bez přerušení provozu:** Přesun hash slotů probíhá bez nutnosti zastavit systém.

## Redis sentinel

- system pro managing Redis instanci
- monitorovani, notifikace, automaticky failover

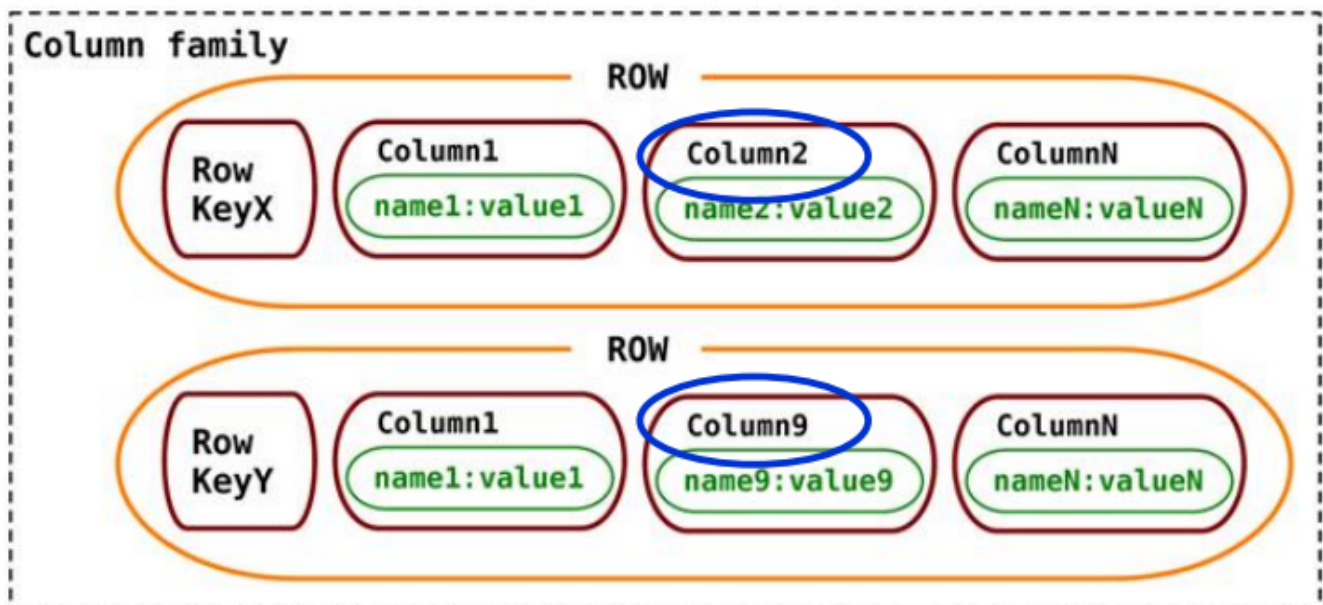
# Sloupcove databaze

- "column-oriented" je neco jineho

## Terminologie

- **column family** = neco jako **table** v relacnich = radky s mnoha sloupci asociovane s **row key**
- data uchovava jako sloupce
  - datove zaznamy jsou mapovany na rowIDs

```
10:001,12:002,11:003,22:004;  
Smith:001,Jones:002,Johnson:003,Jones:004;
```



## Vhodna vyuziti

- Event logging
- CMS, Blogy

## Nevhodna vyuziti

- Systemy vyžadující ACID vlastnosti
- Agregování dat v dotazech

# Cassandra (sloupcove)

- Vyvinuta ve FB
- Ma vlastni query jazyk **CQL**

# Terminologie

- Column = zakladni jednotka
  - Name + value + timestamp
  - name je klic
  - value muze byt prazdna
  - indexace podle jmena a primary indexu = row key
  - Typy:
  - **Expiring** - nastaveny TTL
  - **Counter** - cislo inkremtenujici pri nejake udalosti
  - **Super** - seskupeni vice sloupctu pod jednou hodnotou -> dalsi uroven hierarchie

▪ Priklad Super sloupce:

Row Key (Customer ID)	Super Column (Order ID)	Columns (Order Details)
123	Order_001	date: 2024-12-01, total: \$50
	Order_002	date: 2024-12-05, total: \$75
456	Order_001	date: 2024-11-30, total: \$30

- Row = kolekce sloupctu spojenych ke klici
- Column family = kolekce podobnych rows

RDBMS	Cassandra
database instance	cluster
database	keyspace
table	column family
row	row
column (same for all rows)	column (can be different per row)

Usually one per application

## Column families

- musime specifikovat key
- Comparator = datovy typ pro jmenou sloupce
- Validator = datovy typ pro hodnotu sloupce

## Staticke

- jako tabulka v relační db
- všechny řádky mají stejnou sadu sloupců
- povolujeme null -> každý sloupec nemusí mít hodnotu

## Dynamicke

- dynamicky generované sloupce
- uložena v jednom řádku pro efektivní získání dat
- v tomto kontextu je **row** něco jako snapshot dat / materializovaný **view** -> efektivnější

## Typy kolekce v CQL

- **set** - množina -> jedinečné hodnoty, vrací v abecedním pořadí
- **list** -> seřazené a vrací podle indexu
- **map** -> name + value páry

## CQL - Cassandra query language

### 1. Operace s Keyspace

#### Vytvoření keyspace

```
CREATE KEYSPACE Excelsior
WITH replication = {
  'class': 'SimpleStrategy',
  'replication_factor': 3
};
```

- Definuje **keyspace** s replikací typu **SimpleStrategy** a faktorem replikace **3**.

---

#### Použití keyspace

```
USE Excelsior;
```

- Nastaví **Excelsior** jako aktuálně používaný keyspace.

---

#### Úprava keyspace

```
ALTER KEYSPACE Excelsior
WITH replication = {
  'class': 'SimpleStrategy',
  'replication_factor': 4
};
```

- Změní faktor replikace u existujícího keyspace.
- 

## Odstranění keyspace

```
DROP KEYSPACE Excelsior;
```

- Smaže keyspace a všechna data v něm.
- 

## 2. Operace s tabulkami

### Vytvoření tabulky s primárním klíčem

```
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' };
```

- **Primární klíč:**
    - `userid` je **partition key** (hlavní klíč, který určuje rozdělení dat mezi uzly).
    - `posted_month` a `posted_time` jsou **clustering columns** (určují pořadí dat uvnitř partice).
  - **Strategie komprese:** Nastavena na `LeveledCompactionStrategy`.
- 

### Smazání tabulky

```
DROP TABLE timeline;
```

- Smaže tabulku a všechna její data.
- 

### Vymazání dat z tabulky

```
TRUNCATE timeline;
```

- Odstraní všechna data z tabulky, ale zachová její strukturu.
-



## Vytvoření indexu

```
CREATE INDEX userIndex ON timeline (posted_by);
```

- Vytvoří sekundární index na sloupci `posted_by` pro efektivní dotazování mimo primární klíč.

---

## Smazání indexu

```
DROP INDEX userIndex;
```

- Smaže vytvořený index.

---

## 3. Expirace dat v tabulce

### Vytvoření tabulky

```
CREATE TABLE excelsior.clicks (  
  userid uuid,  
  url text,  
  date timestamp,  
  name text,  
  PRIMARY KEY (userid, url)  
);
```

### Vložení dat s TTL (Time-To-Live)

```
INSERT INTO excelsior.clicks (userid, url, date, name)  
VALUES (  
  3715e600-2eb0-11e2-81c1-0800200c9a66,  
  'http://apache.org',  
  '2013-10-09',  
  'Mary'  
) USING TTL 86400;
```

- Data budou automaticky smazána po 86,400 sekundách (1 den).

### Zjištění zbývajících doby života dat

```
SELECT TTL(name) FROM excelsior.clicks  
WHERE url = 'http://apache.org' ALLOW FILTERING;
```

- Určuje, kolik času zbývá, než data vyprší.

---

## 4. Práce s kolekcemi

### Set (množina)

```
CREATE TABLE users (  
  user_id text PRIMARY KEY,  
  first_name text,  
  last_name text,  
  emails set<text>  
);  
  
INSERT INTO users (user_id, first_name, last_name, emails)  
VALUES ('frodo', 'Frodo', 'Baggins', {'fb@baggins.com', 'baggins@gmail.com'});  
  
UPDATE users SET emails = emails + {'fb@friendsofmordor.org'}  
WHERE user_id = 'frodo';  
  
SELECT user_id, emails FROM users WHERE user_id = 'frodo';  
  
UPDATE users SET emails = emails - {'fb@friendsofmordor.org'}  
WHERE user_id = 'frodo';  
  
UPDATE users SET emails = {} WHERE user_id = 'frodo';
```

- **Set** ukládá jedinečné hodnoty.
- Přidávání: +.
- Odebírání: -.
- Vymazání všech hodnot: nastavení na {}.

---

### List (seznam)

```
ALTER TABLE users ADD top_places list<text>;  
  
UPDATE users SET top_places = ['rivendell', 'rohan']  
WHERE user_id = 'frodo';  
  
UPDATE users SET top_places = ['the shire'] + top_places  
WHERE user_id = 'frodo';  
  
UPDATE users SET top_places = top_places + ['mordor']  
WHERE user_id = 'frodo';  
  
UPDATE users SET top_places[2] = 'riddermark'  
WHERE user_id = 'frodo';
```

```
DELETE top_places[3] FROM users WHERE user_id = 'frodo';

UPDATE users SET top_places = top_places - ['riddermark']
WHERE user_id = 'frodo';
```

- **List** je uspořádaný seznam hodnot.
- Přidávání na začátek: `['value'] + list`.
- Přidávání na konec: `list + ['value']`.
- Přepis hodnoty podle indexu: `top_places[index]`.

---

## Map (mapa)

```
ALTER TABLE users ADD todo map<timestamp, text>;

UPDATE users SET todo = {
  '2012-9-24': 'enter mordor',
  '2012-10-2 12:00': 'throw ring into mount doom'
}
WHERE user_id = 'frodo';

UPDATE users SET todo['2012-10-2 12:00'] =
'throw my precious into mount doom'
WHERE user_id = 'frodo';

DELETE todo['2012-9-24'] FROM users WHERE user_id = 'frodo';
```

- **Map** ukládá páry klíč-hodnota.
- Přidávání/aktualizace hodnot: `todo['key'] = 'value'`.
- Smazání hodnoty podle klíče: `DELETE todo['key']`.

## Dotazy v Cassandře

- Cassandra nepodporuje **joins** ani složité podmínky.
- Dotazy jsou optimalizované pro rychlé čtení jednoduchých dat.
- **Primární klíč** hraje klíčovou roli při určování výkonu dotazů.

---

### 1. Základní SELECT dotaz

```
SELECT * FROM users
WHERE firstname = 'Jane' AND lastname = 'Smith'
ALLOW FILTERING;
```

- Používá **WHERE** pro filtrování výsledků.

- **ALLOW FILTERING** umožňuje filtrovat výsledky mimo primární klíč, ale může mít negativní dopad na výkon.
- 

## 2. Filtrování (WHERE)

```
SELECT * FROM emp
WHERE empID IN (130, 104);
```

- **IN** umožňuje vybírat více hodnot.
- 

## 3. Řazení (ORDER BY)

```
SELECT * FROM emp
WHERE deptID = 10
ORDER BY empID DESC;
```

- Řazení lze použít pouze u **clustering columns**.
  - Směr řazení: **ASC** (výchozí) nebo **DESC**.
- 

## 4. Syntaxe SELECT dotazů

```
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;
```

- **select\_expression**:
    - Výběr sloupců (např. **firstname**, **lastname**).
    - **DISTINCT**: Používá se pro jedinečné hodnoty v rámci partice.
    - **COUNT**: Spočítá řádky.
    - **Aliases**: Pomocí **AS** lze přejmenovat sloupce.
    - **TTL(column\_name)**: Ukáže zbývající čas života hodnoty.
    - **WRITETIME(column\_name)**: Zobrazí čas posledního zápisu hodnoty.
- 

## 5. Podmínky (relation)

- Základní podmínky:

```
column_name ( = | < | > | <= | >= ) value
```

- Použití seznamů:

```
column_name IN (value1, value2, ...)
```

- Použití **TOKEN**:

```
TOKEN(column_name) ( = | < | > | <= | >= )
```

---

## 6. GROUP BY

```
SELECT country, COUNT(*)  
FROM users  
GROUP BY country;
```

- **Skupinování řádků** podle sloupců.
- Povolené pouze pro sloupce obsažené v **primárním klíči**.
- Použitelné agregační funkce:
  - **COUNT, MIN, MAX, SUM, AVG.**
  - Uživatel může definovat i vlastní agregační funkce.

---

## 7. ALLOW FILTERING

- Cassandra vyžaduje, aby dotazy byly **predikovatelné** a efektivní.
- **ALLOW FILTERING**:
  - Umožňuje spustit drahé dotazy, které filtrují velké množství dat.
  - Může být kombinováno s **LIMIT** pro omezení počtu vrácených řádků.

### Příklad:

```
SELECT * FROM users  
WHERE birth_year = 1981  
ALLOW FILTERING;
```

- Použití filtru na sloupec, který není součástí primárního klíče.

---

## 8. Vytvoření indexu

```
CREATE INDEX ON users(birth_year);
```

- Index umožňuje efektivní dotazování na sloupce mimo primární klíč.
- Použití s indexem:

```
SELECT firstname, lastname  
FROM users  
WHERE birth_year = 1981;
```

---

## Příklady dotazů:

### Výběr všech uživatelů:

```
SELECT * FROM users;
```

### Vyhledání uživatele podle primárního klíče:

```
SELECT * FROM users  
WHERE username = 'frodo';
```

### Filtrování s řazením:

```
SELECT firstname, lastname  
FROM users  
WHERE birth_year = 1981  
ORDER BY lastname ASC  
ALLOW FILTERING;
```

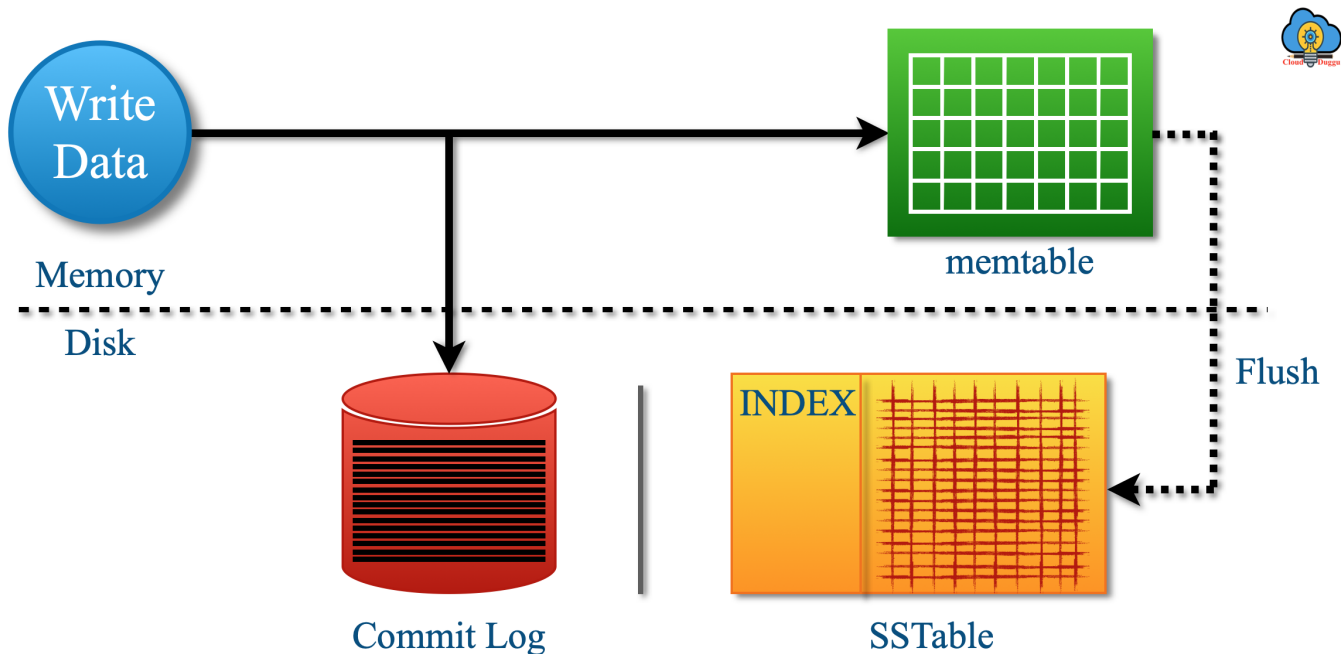
## Zapisy

- zapis je atomicky na urovni radku
- Memtable a SSTable jsou udržovány pro každou tabulku

### Průběh zápisu

1. Při zápisu jsou data uložena v paměti -> **memtable**
2. Zapis je přidán do **commit logu** na disku (durability)
3. Memtable je flushed do **SSTable** (= sorted string table) na disku

4. Data v commit logu jsou **purged** (= odstranena) po flushnutí jejich odpovídajících dat z **memtable** do **SSTable**



## SSTable

- = Sorted string table
- **SSTable** je immutable
  - radek je zapsan pres vice **SSTable** souboru
- read kombinuje fragmenty z **SSTable** a neflushnutych **Memtablu**
- kazdy SSTable si udržuje:
  - **partition index** -> lokalizace dat
  - **partition summary** -> vice rozseka

## Write Request

- request zpracovava jakykoliv uzel -> stava se z nej **coordinator**
  - komunikuje mezi klientem a ostatními uzly s replikami
  - posle write request vsem replikam, které mají radek, který se má zapsat
- **Write consistency level** = kolik replik musí uspet
  - uspech = data jsou zapsana do commit logu a memtablu

## Cteni

- typy read requestu:
  - primy read request
  - background read repair request

## Prubeh cteni v Cassandre:

### 1. Koordinátor kontaktuje repliky podle úrovně konzistence.

- Např. **ONE**, **QUORUM**, nebo **ALL**.

- Vybere nejrychleji odpovídající repliky.

## 2. Porovnání dat z replik.

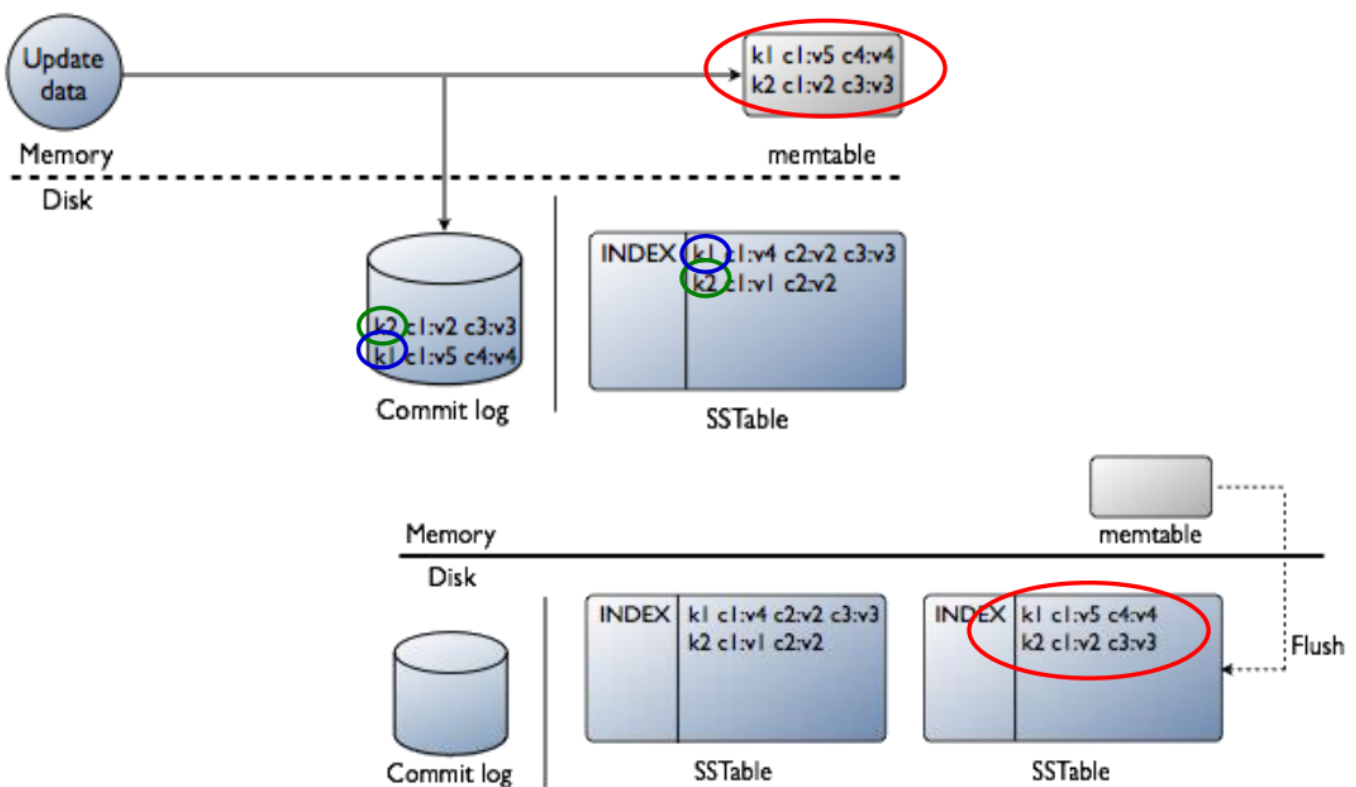
- Pokud jsou konzistentní, vrátí se klientovi.
- Pokud jsou nekonzistentní, použije se nejnovější hodnota podle **timestampu**.

## 3. Read Repair (oprava čtení):

- Na pozadí koordinátor zkontroluje zbývající repliky.
- Opraví zastaralé nebo nekonzistentní repliky.

## Updates

- insert a update jsou stejné operace
- neprepisuje readky -> seskupuje inserty/updaty v memtable
- **Upsert** = insert nebo update podle toho, jestli data existují
  - sloupce jsou prepisany pouze pokud jsou timestamps novejsi
  - jinak jsou updaty ukladany do noveho SSTablu
    - pak je to margnuto na pozadi behem **compaction processu**



## Deletes v Cassandře

- **Smazání řádku:** Odpovídá smazání všech jeho sloupců.
- **Mazání není okamžité:**

## Tombstone

- **Definice:**



- Značka, která označuje, že sloupec nebo řádek byl smazán.
- Cassandra používá tombstones k opětovnému odeslání požadavku na mazání replikám, které byly při mazání nedostupné.

- **Doba platnosti tombstones:**

- Sloupce označené tombstonem existují po **nastavitelnou dobu platnosti** (grace period).
- Po uplynutí této doby jsou při procesu **kompakce** (compaction) trvale odstraněny.
- Kompakce také slučuje více SSTables.

## **Možné problémy s mazáním:**

- Pokud je uzel nedostupný déle, než je nastavená grace period:
  - Může dojít k tomu, že smazaná data se na tomto uzlu objeví znovu, protože mazání nebylo na tento uzel aplikováno.

## **Řešení:**

- **Pravidelná oprava uzlů (node repair):**
  - Správci musí pravidelně spouštět opravy uzlů, aby se předešlo situacím, kdy by některé repliky měly stará data.

## **Compaction process**

- Cassandra **nevkládá/neupravuje/nesmaže data přímo na místě:**
  - **Vkládání/úpravy:** Vytvoří novou verzi dat s časovou značkou v nové SSTable.
  - **Mazání:** Označení dat pomocí tombstonu.
- Compaction robíhá pravidelně, aby byla data sloučena a zoptimalizována.

## **Kroky Compaction**

### **1. Sloučení dat z SSTables** podle partition key:

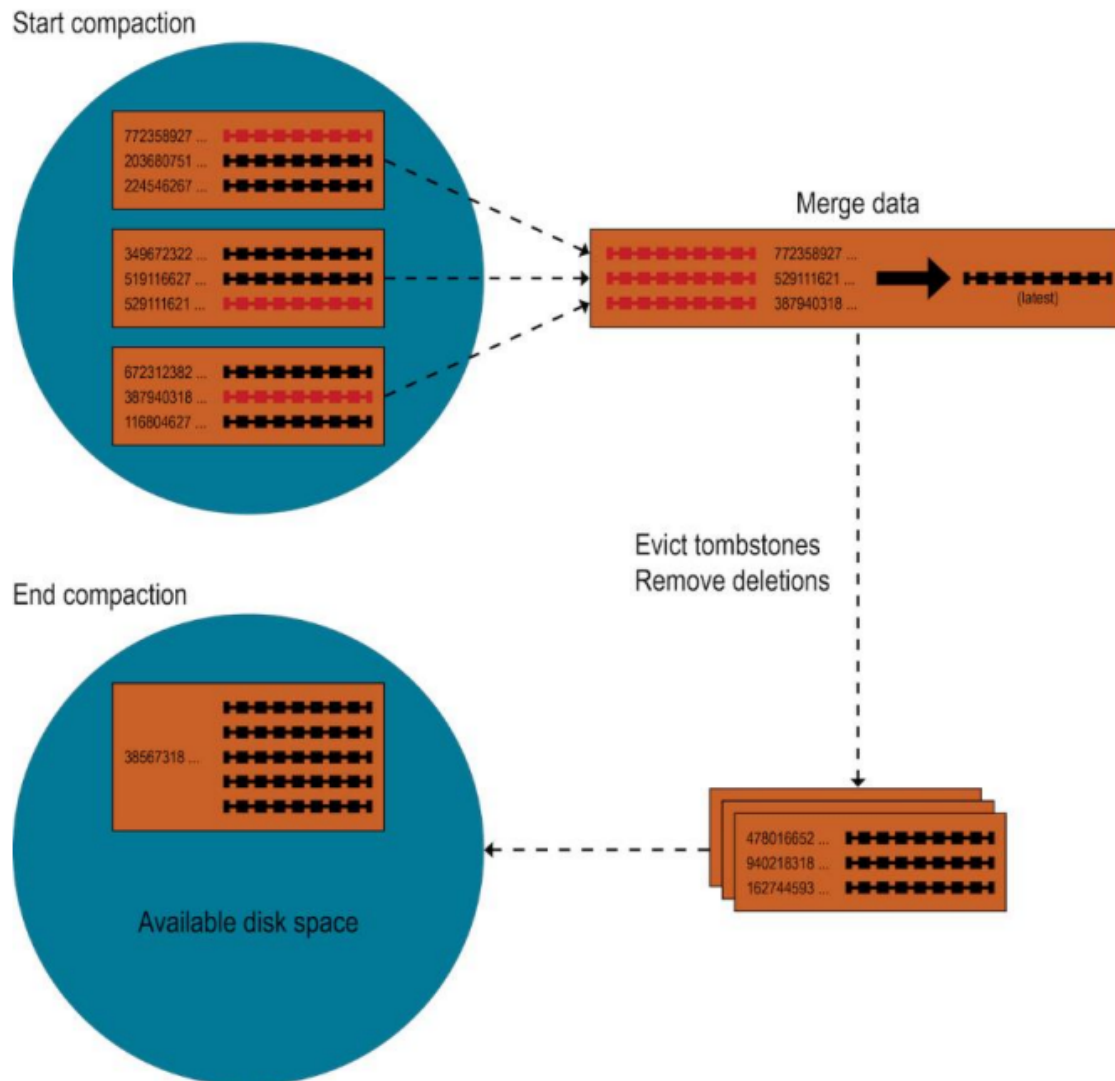
- Výběr nejaktuálnějších dat na základě timestampu.
- Synchronizace je nutná.
- SSTables jsou seřazené → není třeba náhodný přístup.

### **2. Odstranění tombstonů a smazaných dat.**

### **3. Konsolidace SSTables** do jednoho souboru.

### **4. Smazání starých SSTable souborů:**

- Jakmile všechny čekající čtení dokončí práci s těmito soubory.



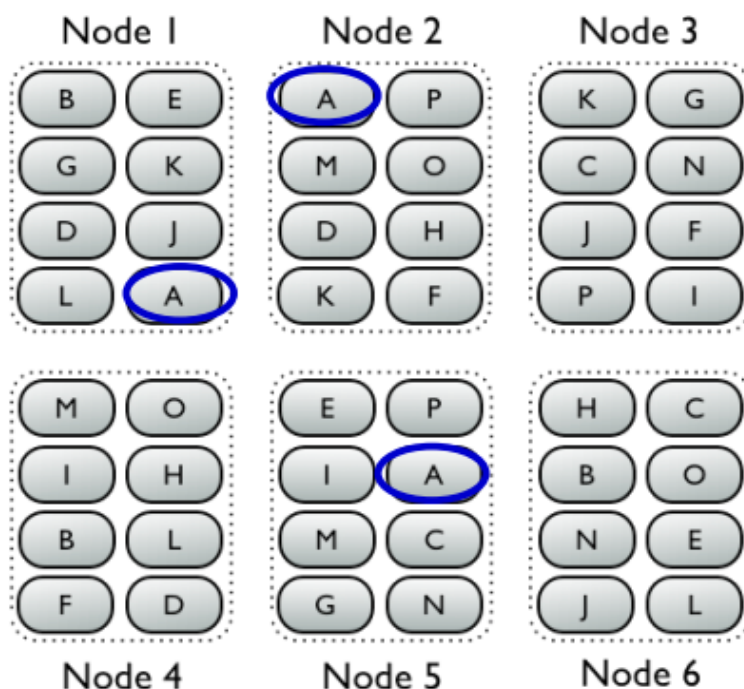
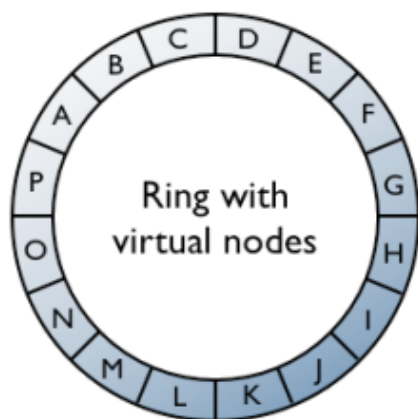
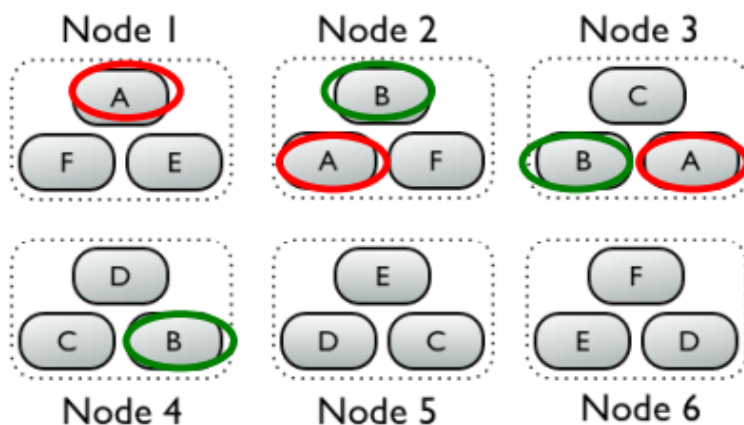
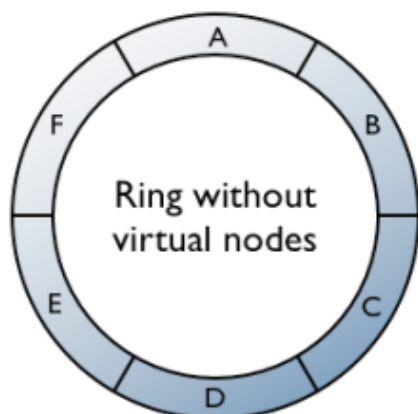
## Architektura

- peer-to-peer distribuovaný systém
- **Coordinator** = jakýkoliv uzel zodpovědný za komunikaci s klientem

## Virtual Nodes

- každý uzel může vlastnit velké množství malých partií

## Example: replication factor = 3



## Gossip

- bezi každou sekundu
- výměna info s max 3 uzly
- každá **Gossip message** má informace o zdroji a verzi

## Partitioner

- **Úloha:** Rozděluje data mezi uzly (včetně replik).
- **Typy:**
  - **Murmur3Partitioner (výchozí):** Uniformní distribuce pomocí MurmurHash (rychlá, nešifrovaná).
  - **RandomPartitioner:** Uniformní distribuce pomocí MD5 (dřívější výchozí).

- **ByteOrderedPartitioner**: Řadí řádky lexikálně podle bajtů klíče, vhodné pro **ordered scans**, ale problémy s vyvažováním zátěže.

## Replikace

- Pokud je replikační faktor převysen, zápisy nejsou prováděny

### Replikace

- Pokud je replikační faktor překročen, **zápisy nejsou prováděny**.
- typicky 2-3 repliky

## Strategie pro umístění replik

### 1. SimpleStrategy

- **Vhodné pro jedno datové centrum.**
- **Pravidla:**
  1. První replika je umístěna na uzel určený partitionerem.
  2. Další repliky jsou umístěny na následující uzly ve směru hodinových ručiček v ringu.
- **Poznámka:** Uzel může patřit do datového centra a (volitelně) do racku.

### 2. NetworkTopologyStrategy

- **Vhodné pro více datových center.**
- **Pravidla:**
  1. První replika je umístěna podle partitioneru.
  2. Další repliky jsou umístěny:
    - **Preferenčně** na uzly v jiném racku (kvůli odolnosti proti výpadkům napájení, chlazení nebo síť).
    - Pokud uzel v jiném racku není k dispozici, replika se umístí na jiný uzel ve stejném racku.
- **Počet replik na datové centrum je konfigurovatelný.**

## Snitch

- komponenta Cassandra informující o síťové topologii

## Dokumentové databaze

---

- hodnoty jsou ukládány jako dokumenty
  - dokumenty = hierarchické formáty XML, JSON apod.
  - hodnota záznamu = dokument
- očekáváme podobnou strukturu dokumentu v kolekci

## Vhodná využití

- Event logging

- CMS, blogy
- Webova analytika
- E-Commerce

## Nevhodna vyuziti

- Koplexni transakce pres vice operaci
- Agregovane dotazy

## MongoDB (dokumentove)

---

- pouziva JSON dokumenty
- podpora indexace
- mapreduce popora
- vysoka dostupnost

Oracle	MongoDB
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	_id
join	DBRef

## Dokumenty v MongoDB

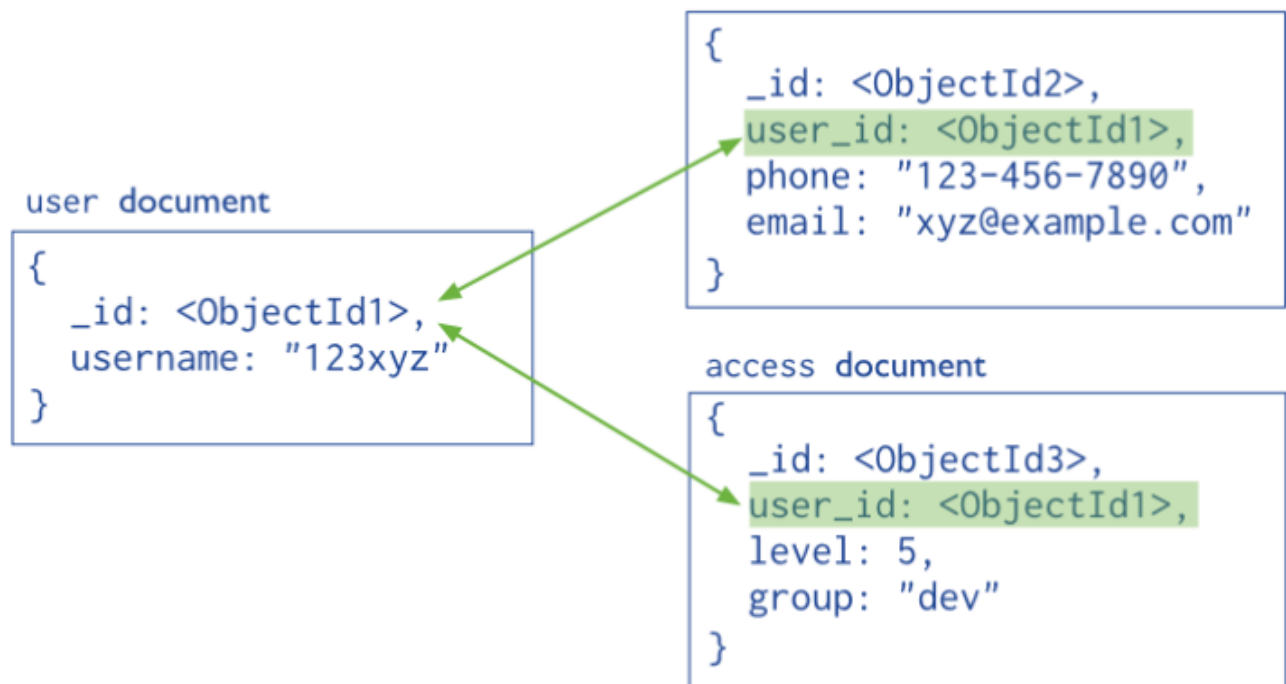
- vyuziva JSON
- ulozeno jako BSON - binarni JSON
- omezeni na nazvy prvku (\_id je rezervovano, \$ nemuze byt na zacatku, . nesmi byt vubec)

## Datovy model

- kolekce nevynucuji strukturu dat
- dulezite rozhodnuti je zda vyuzivat reference nebo embedovat dokumenty

## Reference

- normalizovany datovy model
- reference z jednoho dokumentu na dalsi
- vice flexibility nez embedding
- ochrana proti redundanci
- nevýhodou je možnost více roundtripu k serveru (follow up queries)



## Embedded data

- denormalizovany datovy model
- subdokumenty
- pribuzna data v jednom dokumentu
- mohou velmi narostat na velikosti

## Prace s MongoDB

- **Operace:** `insert`, `update`, `delete`.
  - **Kritéria:** Používají se pro výběr dokumentů, které se mají aktualizovat nebo odstranit.

---

## Vkládání dat

- **Vložení dokumentu:**

```
db.inventory.insert({ _id: 10, type: "misc", item: "card", qty: 15 });
```

- Vloží dokument s uživatelem definovaným `_id`.

- **Upsert (vložení nebo aktualizace):**

```
db.inventory.update(  
  { type: "book", item: "journal" },  
  { $set: { qty: 10 } },  
  { upsert: true }  
);
```

- Pokud dokument neexistuje, vytvoří nový.

- **Save (vlození nebo nahrazení):**

```
db.inventory.save({ type: "book", item: "notebook", qty: 40 });
```

---

## Mazání dat

- **Smazání všech odpovídajících dokumentů:**

```
db.inventory.remove({ type: "food" });
```

- **Smazání jednoho dokumentu:**

```
db.inventory.remove({ type: "food" }, 1);
```

---

## Aktualizace dat

- **Aktualizace více dokumentů:**

```
db.inventory.update(  
  { type: "book" },  
  { $inc: { qty: -1 } },  
  { multi: true }  
);
```

- **Nahrazení dokumentu:**

```
db.inventory.save({ _id: 10, type: "misc", item: "placard" });
```

---

## Dotazy

- **Základní dotazy:**

- Všechny dokumenty:

```
db.inventory.find({});
```

- Dokumenty, kde **type** = "snacks":

```
db.inventory.find({ type: "snacks" });
```

- Dokumenty s hodnotou v poli **type** buď "food" nebo "snacks":

```
db.inventory.find({ type: { $in: ["food", "snacks"] } });
```

- **Logické operátory:**

- Dokumenty, kde **qty** > 100 nebo **price** < 9.95:

```
db.inventory.find({
  $or: [
    { qty: { $gt: 100 } },
    { price: { $lt: 9.95 } }
  ]
});
```

---

## Práce s poddokumenty

- **Dotaz na přesnou strukturu poddokumentu:**

```
db.inventory.find({
  producer: {
    company: "ABC123",
    address: "123 Street"
  }
});
```

- **Dotaz na konkrétní pole v poddokumentu:**

```
db.inventory.find({ "producer.company": "ABC123" });
```



## Práce s poli

- Pole s přesnou hodnotou a pořadím:

```
db.inventory.find({ tags: ["fruit", "food", "citrus"] });
```

- Pole obsahující prvek:

```
db.inventory.find({ tags: "fruit" });
```

---

## Omezení a třídění výsledků

- Omezení polí výsledku:

- Pouze `item` a `qty`:

```
db.inventory.find({ type: "food" }, { item: 1, qty: 1 });
```

- Vyloučení pole `type`:

```
db.inventory.find({ type: "food" }, { type: 0 });
```

- Třídění:

- Podle `age` sestupně:

```
db.collection.find().sort({ age: -1 });
```

- Podle `last` a následně `first` vzestupně:

```
db.bios.find().sort({ "name.last": 1, "name.first": 1 });
```

## Vyuziti indexu

- Bez indexů:

- MongoDB musí skenovat každý dokument v kolekci, aby našla odpovídající dokumenty.

- Indexy:

- Ukládají část dat kolekce v snadno prohledávatelné formě.
- Hodnoty konkrétních polí (nebo sad polí) jsou ukládány a tříděny.

- Používají struktury podobné **B-stromům**.
- **Účel:**
  - Zrychlení běžných dotazů.
  - Optimalizace výkonu v konkrétních situacích.

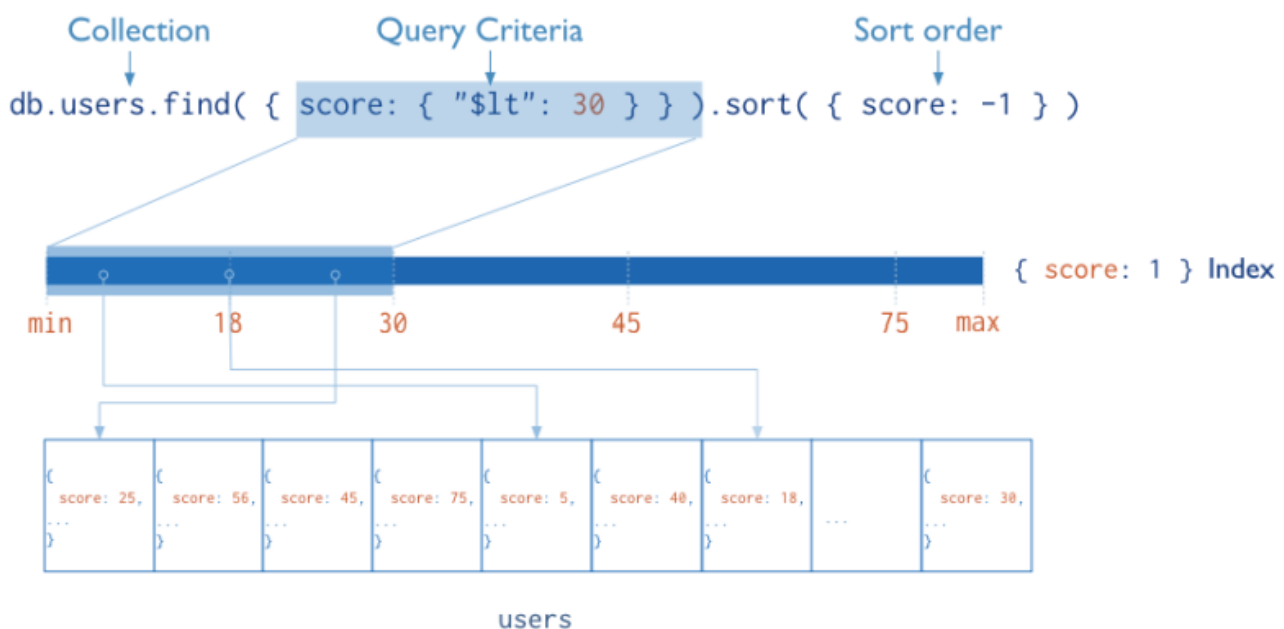
## Použití indexů

### 1. Pro tříděné výsledky:

- MongoDB traversuje index přímo (vzestupně/sestupně), aniž by musela data třídit.
- Data mimo index se nekontrolují.

### 2. Pro pokryté výsledky:

- Pokud index obsahuje všechna pole potřebná pro dotaz, MongoDB vrátí výsledky pouze z indexu, což zrychlí dotaz.



## Typy indexů

### 1. Výchozí `_id`:

- Automaticky vytvořený.
- Unikátní.

### 2. Jednopolní indexy:

- Uživatelem definované na jednom poli dokumentu.

```
db.people.ensureIndex({ "phone-number": 1 });
```

### 3. Složené indexy (Compound):

- Uživatelem definované na více polích.

```
db.products.ensureIndex({ item: 1, category: 1, price: 1 });
```

#### 4. Multikey indexy:

- Indexují obsah uložený v polích typu pole.
- Vytváří samostatné indexové záznamy pro každý prvek pole.

```
db.collection.ensureIndex({ "tags": 1 });
```

#### 5. Geoprostorové indexy:

- **2d indexy:** Pro data na dvourozměrné rovině.
- **2sphere indexy:** Pro data reprezentující zeměpisnou délku a šířku.

#### 6. Textové indexy:

- Pro hledání textového obsahu v kolekci.

#### 7. Hash indexy:

- Indexuje hash hodnoty pole.
- Podporuje pouze rovnostní dotazy, ne rozsahové.

---

## Příklady

- **Jednopolní index:**

```
db.people.ensureIndex({ "phone-number": 1 });
```

- **Složený index:**

```
db.products.ensureIndex({ item: 1, category: 1, price: 1 });
```

- **Unikátní index:**

```
db.accounts.ensureIndex({ "tax-id": 1 }, { unique: true });
```

- **Hash index:**

```
db.collection.ensureIndex({ _id: "hashed" });
```

## Replikace

- master/slave
- **replica set** = skupina instanci hostující stejný dataset (každý má svoji kopii)

### Primární uzel

- master
- přijímá všechny write operace
- zapisuje do **oplogu**

### Sekundární uzly

- slave
- čte z **oplogu** mastera
- aplikuje operace z oplogu ve stejném pořadí pro udržení aktuálních dat
- můžeme ho dále nastavit:
  - **Priority 0** - nemůže být primary ve volbě
  - **Hidden** - nelze z něj číst
  - **Delayed** - běží na něm historická data, např. z důvodu recovery

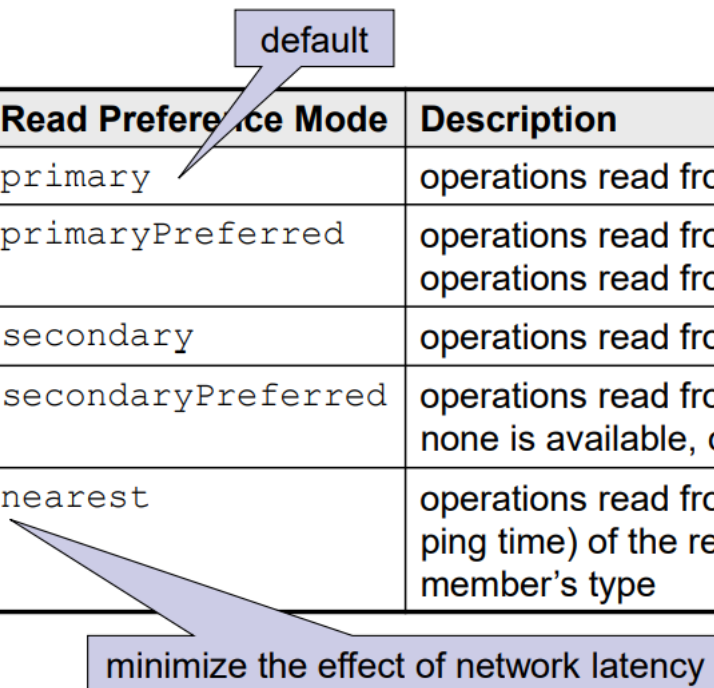
## Zápis

1. MongoDB aplikuje zapisovací operace na **primární uzel** (primary).
2. Operace jsou zaznamenány do **oplogu primárního uzlu**.
3. Sekundární uzly (secondary members):
  - Replikují obsah oplogu.
  - Aplikují operace z oplogu na svá data, aby byla aktuální.

---

## Čtení

1. **Všichni členové replica setu** mohou přijímat požadavky na čtení.
2. **Výchozí chování:**
  - Aplikace směřuje čtení na **primární uzel**.
  - Zaručuje, že čtení vrátí nejaktuálnější verzi dokumentu.
  - Snižuje propustnost pro čtení na sekundárních uzlech.
3. **Read preference mode:**
  - Lze nastavit, aby čtení probíhalo také ze sekundárních uzlů.
  - Například:
    - **primaryPreferred:** Primární uzel je preferován, ale sekundární uzly jsou použity, pokud primární není dostupný.
    - **secondary:** Čtení probíhá pouze ze sekundárních uzlů.



Read Preference Mode	Description
primary	operations read from the current replica set primary
primaryPreferred	operations read from the primary, but if unavailable, operations read from secondary members
secondary	operations read from the secondary members
secondaryPreferred	operations read from secondary members, but if none is available, operations read from the primary
nearest	operations read from the nearest member (= shortest ping time) of the replica set, irrespective of the member's type

## Replica Set Elections v MongoDB

### Principy voleb

#### 1. Primární uzel:

- V replica setu může být **maximálně jeden primární uzel**.
- Pokud primární uzel není dostupný, proběhne volba nového primárního uzlu.

#### 2. Trvání voleb:

- Volba obvykle trvá přibližně **1 minutu**.
- Během této doby není k dispozici žádný primární uzel → nejsou možné zápisy.

---

### Faktory ovlivňující volby

#### 1. Heartbeat (ping):

- Uzel posílá heartbeat ostatním každé **2 sekundy**.
- Pokud odpověď nepříjde do **10 sekund**, uzel je považován za nedostupný.

#### 2. Priority uzlů:

- Uzel s vyšší prioritou má přednost při volbě primárního uzlu.
- **Priority = 0:**
  - Uzel nemůže být primární.
  - Nemůže zahájit volbu, ale může hlasovat.
- Aktuální primární uzel:
  - Musí být v **souladu s nejnovějším oplogem** do **10 sekund**.
- Sekundární uzel s vyšší prioritou:
  - Pokud dožene synchronizaci do **10 sekund**, může vyvolat volbu.

### 3. Spojení:

- Uzel může být zvolen primárním pouze tehdy, pokud je připojen k **většině členů** replica setu.
- 

## Mechanismus voleb

### 1. Volby se spustí, když:

- Replica set je inicializován.
- Sekundární uzel ztratí kontakt s primárním.
- Primární uzel "ustoupí" (step down) nebo ztratí spojení s většinou členů.

### 2. Primární uzel se vzdá role:

- Po obdržení příkazu `replSetStepDown`.
- Pokud má sekundární uzel vyšší prioritu.
- Pokud nemůže kontaktovat většinu členů replica setu.

### 3. Volba nového primárního uzlu:

- Člen s **nejvyšší prioritou**, který je aktuální, se stane kandidátem.
  - První člen, který získá **většinu hlasů**, je zvolen primárním.
- 

## Speciální případy:

### 1. Nehlasující členové (non-voting members):

- Mají kopii dat, ale nemohou hlasovat.
- Mohou být zvoleni primárním, ale toto nastavení není doporučeno.

### 2. Veto členů:

- Každý člen může volbu vetovat, pokud:
  - Kandidát není synchronizován s nejnovější operací v replica setu.
  - Kandidát má nižší prioritu než jiný oprávněný člen.

## Arbiter

- specialni uzel
- neuchovava dataset
- nemuze byt primary
- je pouze pro hlasovani ve volbach (pro repliky se sudym poctem hlasujicich)

## Sharding

### Sharded Clusters v MongoDB

- Složení:

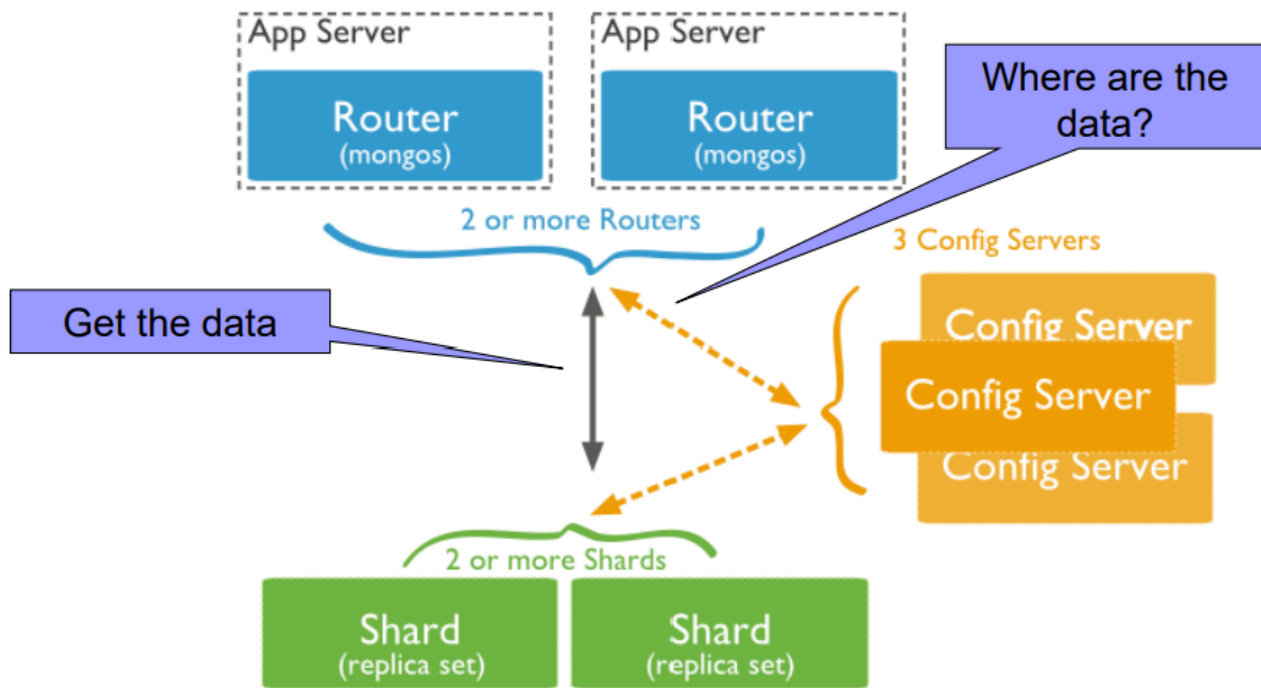
1. **Shards:** Uchovávají data.
  - Každý shard je replica set.

## 2. Query Routers: Rozhraní pro klientské aplikace.

- Routují operace na správné shard(y) a vracejí výsledky.
- Více routerů pro rozložení zátěže.

## 3. Config Servers: Uchovávají metadata clusteru.

- Mapují datovou sadu na shardy.
- Doporučený počet: 3.



## Partitioning dat

### Data Partitioning v MongoDB

- **Rozdělení dat:** Data kolekce se dělí podle **shard key**.
  - **Shard key:**
    - Indexované pole (možné složené), které je přítomné v každém dokumentu.
    - **Neměnný** (immutable).
  - Data se dělí na **chunks**, které se distribuují mezi shardy.
- **Rozdělení dat:**
  - **Range-Based Partitioning:**
    - Hodnoty shard key se rozdělí na kontinuální úseky (chunks).
    - Efektivnější pro range dotazy, ale může vést k nerovnoměrné distribuci dat.
  - **Hash-Based Partitioning:**
    - Hodnota shard key se zahashuje a výsledné hashe tvoří chunks.
    - Rovnoměrnější distribuce dat, ale range dotazy mohou zasáhnout více shardů.
- **Chunk velikost a migrace:**
  - Chunk se rozdělí, pokud přeroste nastavenou velikost (**výchozí 64MB**).
  - **Malé chunky:** Rovnoměrnější distribuce, ale častější migrace.

- **Velké chunky:** Méně migrací, ale nerovnoměrné zatížení.

## Journaling

- **Co dělá:** Ukládá zápisové operace do paměti a journalu před aplikací na data.
- **Účel:** Obnovení konzistence databáze po tvrdém vypnutí.
- **Journal file:**
  - Append-only log (write-ahead redo log).
  - Smazán, když jsou všechny zápisy provedeny.
  - Nový soubor vytvořen při 1 GB (velikost lze upravit).
- **Clean shutdown:** Smaže všechny journal soubory.

## Two-Phase Commit v MongoDB

Transakce přes více dokumentů:

- Když operace zahrnuje více dokumentů (např. změny v několika kolekcích), MongoDB podporuje multi-document transactions, které zajišťují transakční vlastnosti (ACID) pro více dokumentů najednou.

### Co je Two-Phase Commit?

- Proces umožňující transakce zahrnující více dokumentů s **transaction-like** vlastnostmi.
- Data jsou ukládána a spravována pomocí speciální kolekce transakcí.

---

## Kroky Two-Phase Commit

### Příklad: Převod peněz mezi účty A a B

#### 1. Inicializace transakce:

- Vytvoříme účty:

```
db.accounts.save({name: "A", balance: 1000, pendingTransactions: []});
db.accounts.save({name: "B", balance: 1000, pendingTransactions: []});
```

- Vytvoříme transakci:

```
db.transactions.save({source: "A", destination: "B", value: 100, state:
"initial"});
```

- Transakce má stav **initial**.

#### 2. Změna stavu transakce na **pending**:

```
t = db.transactions.findOne({state: "initial"});
db.transactions.update({_id: t._id}, { $set: {state: "pending"} });
```



### 3. Aplikace transakce na účty:

- Odepíšeme z účtu A a připsíme na účet B:

```
db.accounts.update({ name: t.source, pendingTransactions: {$ne: t._id}
},
{ $inc: {balance: -t.value}, $push: {pendingTransactions: t._id} });
db.accounts.update({ name: t.destination, pendingTransactions: {$ne:
t._id} },
{ $inc: {balance: t.value}, $push: {pendingTransactions: t._id} });
```

### 4. Změna stavu transakce na **applied**:

```
db.transactions.update({_id: t._id}, { $set: {state: "applied"} });
```

### 5. Odstranění **pendingTransactions** z účtů:

```
db.accounts.update({name: t.source}, { $pull: {pendingTransactions: t._id}
});
db.accounts.update({name: t.destination}, { $pull: {pendingTransactions:
t._id} });
```

### 6. Změna stavu transakce na **done**:

```
db.transactions.update({_id: t._id}, { $set: {state: "done"} });
```

---

## Řešení chyb

- **Mezi kroky 1 a 3 (před aplikací transakce):**
  - Pokračujeme od kroku 2: nastavíme stav transakce na **pending**.
- **Mezi kroky 3 a 6 (po aplikaci transakce):**
  - Pokračujeme od kroku 5: odstraníme **pendingTransactions** a dokončíme transakci.

---

## Rollback transakce

### 1. Nastavení stavu na **cancelling**:

```
db.transactions.update({_id: t._id}, { $set: {state: "cancelling"} });
```

## 2. Vrácení změn na účtech:

```
db.accounts.update({name: t.source, pendingTransactions: t._id},  
  { $inc: {balance: t.value}, $pull: {pendingTransactions: t._id} });  
db.accounts.update({name: t.destination, pendingTransactions: t._id},  
  { $inc: {balance: -t.value}, $pull: {pendingTransactions: t._id} });
```

## 3. Změna stavu na **cancelled**:

```
db.transactions.update({_id: t._id}, { $set: {state: "cancelled"} });
```

---

## Více aplikací a správa transakcí

- **Požadavek:** Jen jedna aplikace může spravovat konkrétní transakci.
- **Řešení:** Použití metody `findAndModify`:

```
t = db.transactions.findAndModify(  
  {query: {state: "initial", application: {$exists: false}},  
  update: {$set: {state: "pending", application: "A1"}},  
  new: true});
```

- Tímto způsobem se transakce přiřadí konkrétní aplikaci a zajistí se atomická změna.

---

## Grafove databaze

---

## Multimodel databaze

---

## Polystores

---

## Advanced