2) CRUD, Kurzory

2.1) Sada replík a CRUD

I. Atomicita, transakcia, konzistencia a monotonicita

a) **Atomicita – nevykonané čítanie**

b) **Dvojfázové vykonanie a sémantika podobná transakcie**

c) **Trvanlivosť a eventuálna konzistencia**

d) **Monotonicita zápisu a čítania - Causal Consistency**

II. Dnešný stav

III. CRUD príkazy `(kurzor: forEach, toArray, next)`

- insert, update, delete

2.2) Dopytovanie

I. Dopytovanie **bez** polí a vnorených dokumentov

II. Dopytovanie polí a vnorených dokumentov

2.3) Príklady a kurzory

1) Kolekcia s poľom hodnôt

1a) Vytvorenie (insert) kolekcie maz1 s poľom/array hodnôt A

1b) Ktoré kľúče treba vrátiť

1c) Dopytovanie poľa

1d) Kurzor a `forEach`

2) Kolekcia s poľom vnorených dokumentov

2a) Kurzor - JSON.stringify

2b) Kurzor to array

3) Generovanie kolekcie

4) Kurzory a JavaScript

4a) Pole – `kur.next()`

2.1) Sada replík a CRUD

Úvod do MongoDB CRUD

MongoDB na **čítanie** a **zapisovanie** používa také **zámky**, ktoré umožňujú súbežným čitateľom **zdieľaný** (shared) prístup k zdroju, ale ktoré zabezpečujú **exkluzívny** prístup zápisu jedného dokumentu.

Mongod (Mongo Daemon) je základným procesom pre **správu** celého MongoDB servera (prijímanie požiadaviek, reagovanie na nich, riadenie požiadaviek na pamäť), ktorý beží v pozadí.

Mongo(**sh**) je JavaScript **rozhranie**, ktoré poskytuje interaktívnu komunikáciu s MongoDB, prijíma **užívateľské** príkazy, dopyty, spája sa s konkrétnou inštanciou mongod a potom príkazy spúšťa.

Sada replík a CRUD

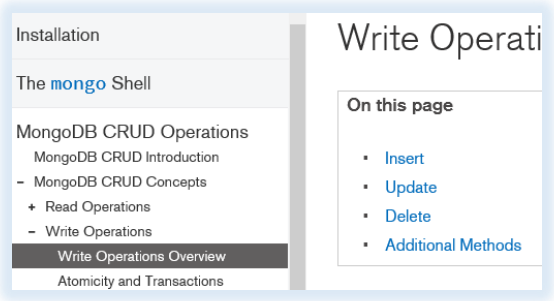
Sada replík (replica set) je skupina **inštancií** (členov) **mongod**, ktoré sa starajú o *rovnakú* sadu údajov.

Ak v danom momente **primarny** člen sa funkčne zlýha, zo **sekundárnyh** členov sa zvolí nový primárny, pozri 4. prednášku. V sade replík sekundárni členovia môžu spracovať **čítanie**, ale **zápis iba primárny** člen.

V MongoDB insert a update operácie prebiehajú **v poradí**:
- záznam u primárneho člena
- potom záznam u primárneho člena do **oplog**u (operation log)
- vytvorenie kópií u sekundárnych členov asynchrónne.

Write operácie

- Insert dokumenty
- Update
- Delete
- Ďalšie metódy



Read operácie

- Query Interface
- Query pravidlá
    - Dopyt v MongoDB sa vždy vzťahuje iba na jednu kolekciu
    - Za dopytom môžeme nastaviť `limit`, `skip` a `sort`
    - Okrem `sortu` nie je možné určit poradie výsledku dopytu
    - Dopytovať v MongoDB okrem **find** môžeme aj pomocou **kurzora** (forEach) a **agregácie**
- Query príkazy
- Projekcie (atrib.)

I. Atomicita, transakcia, konzistencia a monotonicita

a) **Atomicita – nevykonané čítanie**
b) **Dvojfázové vykonanie a sémantika podobná transakcie**
c) **Trvanlivosť a eventuálna konzistencia**
d) **Monotonicita zápisu a čítania - Causal Consistency**

a) Atomicita – nevykonané čítanie

- **Atomickosť/atomicita** zápisu **jedného** dokumentu znamená, že ak **zápis** v dokumente aktualizuje **viac** kľúčov (atribútov), **čiastočné aktualizácie** sa **nesmú načítať**, čitateľ ich nemôže uvidieť. Aj keď čitateľ nemôže vidieť čiastočne aktualizovaný jediný dokument, konkurenční čitatelia za isté okolnosti predsa môžu **uvidieť** aktualizovaný dokument predtým, ako zmeny sú **trvanlivé** (copy). To sa nazýva **nevykonané čítanie** (read uncommitted – čítanie *nevykonaného* zápisu).
- Keď jediná operácia **zápisu** ovplyvňuje **viac** dokumentov, úprava **jednotlivých** dokumentov je atomická, ale operácia ako **celok nie je atimická** - ďalšie operácie môžu zasahovať, prelínať. Jedinú operáciu **zápisu**, ktorá ovplyvňuje viac dokumentov, je možné **izolovať** pomocou operátora `$izolated`, avšak izolovaný zápis neposkytuje "všetko alebo nič" atomicitu.

Pre jedinú mongod inštanciu operácie čítania a zápisu do jediného dokumentu sú **serializovatelné**. V prípade skupiny **replík** iba v neprítomnosti **rollback**.

Dnes (po verzie 4.0) v MongoDB je zápis do jedného dokumentu atomárny (NETREBA dvojfázový commit – pozri nižšie). Príklad viacdokumentovej transakcie (Multi-document Transactions) je prevod peňazí z účtu A na účet B.

Historicky MongoDB transakcie nepodporovalo (do verzie 4.0). Vývojári vtedy museli ručne implementovať vzor nazývaný Two-Phase Commit (dvojfázové vykonanie) pomocou aplikačnej logiky v aplikácii. Dnes už MongoDB podporuje natívne ACID transakcie na vykonanie série operácií nad rôznymi dokumentmi či aj kolekciami.

- Konzistencia: Zaručuje, že dáta v oboch dokumentoch budú dávať zmysel (peniaze nezmiznú v medzipriestore).
- Obnoviteľnosť (Rollback): Ak zápis do účtu B zlyhá, zmeny na účte A sa automaticky vrátia späť (rollback), akoby sa nič nestalo.

b) **Dvojfázové vykonanie a sémantika podobná transakcie – oneskorená konzistencia**

Pretože jeden dokument často obsahuje **niekoľko vložených** dokumentov, **atomicita jedného** dokumentu je dostatočná pre veľa praktických prípadov. Pre viac **zápisov** naraz, ktoré by sa mali chovať ako jedna **transakcia**, je možné **vyžiadať dvojfázové vykonanie** (Two-Phase Commit 2PC), ktoré zaručuje **konzistenciu** dát a v prípade chyby stav pred transakciou je **obnoviteľný**. V priebehu vykonania sa však dokumenty a dáta môžu byť v stave **čakania** (pending). Preto **konzistencia** dát je zaručená *oneskorene*, lebo môže sa stať, že aplikácia počas dvojfázového commitu alebo rollbacku vráti prechodné dáta - to sa nazýva sémantika podobná transakcie (**transaction-like semantics**).

Čo znamenala oneskorená konzistencia pri simulovaní transakcie na úrovni aplikácie (do verzie 4.0)? Ak aplikácia vykonáva dvojfázový commit manuálne (v starších verziách), proces vyzerá takto:

- Fáza 1: Zapíše sa informácia "chystám sa urobiť zmenu" do dokumentu A.
- Fáza 2: Zapíše sa zmena do dokumentu B.
- Fáza 3: Označí sa to za dokončené.

Ak v momente medzi fázou 1 a 2 iný používateľ číta dáta, uvidí "rozpracovaný" stav. To je tá oneskorená konzistencia a prechodné dáta. Systém nie je v danom milisekundovom momente v rovnováhe.

Detailný rozbor fáz manuálnej transakcie

Pri manuálnom simulovaní transakcie sa zvyčajne používala pomocná kolekcia (napr. transactions), ktorá slúžila ako stavový stroj.

- Fáza 1: Príprava (Initial -> Pending)
  Aplikácia nevytvorí zmenu priamo, ale vytvorí záznam o zámere.
  - Vytvorenie transakčného dokumentu: Do špeciálnej kolekcie zapíšeš dokument so stavom initial, ID odosielateľa, ID príjemcu a sumu.
  - Zmena stavu na pending: Pomocou atomickej operácie findAndModify zmeníš stav transakcie na pending. Týmto si aplikácia "rezervuje" túto úlohu.
- Fáza 2: Aplikovanie zmien (Applying)
  Teraz začneš meniť cieľové dokumenty. Toto je moment, kedy vzniká dočasná nekonzistencia.
  - Update Dokumentu A: Odpočítaš sumu a pridáš referenciu na ID transakcie (aby si vedel, že táto zmena patrí k prebiehajúcemu procesu).
  - Update Dokumentu B: Pripočítaš sumu a opäť pridáš referenciu na ID transakcie.
  - V tomto bode, ak niekto číta dokumenty, vidí medzistav. Dáta sú "v procese".

- Fáza 3: Dokončenie (Committing -> Applied)
  Keď sú obe zmeny úspešne zapísané, musíš transakciu uzavrieť.
  - Zmena stavu transakcie na applied: Označíš transakciu v pomocnej kolekcii za úspešnú.
  - Cleanup (Voliteľné): Odstrániš referencie na ID transakcie z dokumentov A a B, aby boli "čisté".
  - Zmena stavu na done: Transakcia je definitívne ukončená.

Čo ak sa niečo pokazí? Nasleduje Rollback a Recovery.
Hlavný dôvod, prečo sa to volalo "oneskorená konzistencia", bol mechanizmus obnovy. Ak aplikácia spadla uprostred 2. fázy:

- Režijné procesy (Watchdogs): Na pozadí musel bežať skript, ktorý hľadal transakcie zaseknuté v stave pending príliš dlho.
- Dopredné dokončenie: Ak skript zistil, že zmena na A prebehla, ale na B nie, manuálne ju dokončil.
- Rollback: Ak zistil, že zmeny nezačali vôbec, transakciu zrušil a vrátil pôvodné hodnoty.

Dôležité upozornenie: Od verzie 4.0 (pre replika sety) a 4.2 (pre sharded clustery) MongoDB podporuje ACID transakcie pomocou syntaxe session.startTransaction(). Celý tento manuálny proces je dnes považovaný za prekonaný.

Termín sémantika podobná transakcii (Transaction-like) sa používa preto, lebo to nie je skutočná databázová transakcia. Je to len súbor pravidiel v kóde aplikácie, ktorý sa snaží transakciu napodobniť.

Dnes už tvrdenie o *oneskorenej konzistencii* pri transakciách neplatí, ak použijete natívne ACID transakcie (dostupné od verzie 4.0+). Moderné MongoDB transakcie používajú tzv. Snapshot Isolation:

- Kým transakcia nie je potvrdená (commit), nikto iný nevidí prechodné dáta.
- Ostatní používatelia vidia dáta v stave, v akom boli pred začatím transakcie.

Až v momente potvrdenia sa všetky zmeny prejavia naraz (atomicky).

c) Trvanlivosť a eventuálna konzistencia/nakoniec konzistentnosť

- Klienti v MongoDB môžu vidieť, čítať **výsledky zápisov** predtým, ako zápisy sú trvanlivé. Operácia zápisu je **trvanlivá** (durable), ak bude pretrvávať po vypnutí (alebo havárii) a reštarte jedného alebo viacerých serverových procesov:
- u jedného MongoDB servera, operácia zápisu je považovaná za **trvanlivú**, keď bola zapísaná do **súboru denníka** (journal file) servera (pozri 4. prednášku).
- pre skupinu replík, operácia zápisu je **značne trvanlivá**, akonáhle operácia zápisu je odolná na väčšine **hlasovacích uzlov** (voting nodes) v skupine replík; tzn. zapísané do *väčšiny* súborov denníka hlasovacích uzlov.

- MongoDB pri lokálnom (*local* - služobné slovo) **čítaní** vráti **najnovšie** údaje, ktoré sú k dispozícii v okamihu dotazu, a to **bez garancie**, že dáta boli trvanlivo zapísané na väčšine členov skupiny replík s možnosťou vrátenia stavu späť.
- Čítanie môže byť označené aj ako väčšinové (*majority*) alebo linearizable.

Eventuálna konzistencia / nakoniec konzistentnosť (eventual consistency) je vlastnosť **distribuovaného** systému, ktorá umožňuje, aby zmeny v systéme (viditeľne) nastali **postupne**. V databázovom systéme, to znamená, že čitateľní členovia nemusia (jednotne) odrážať najnovšie zápisy za všetkých okolností.

Uvažujme skupinu replík s jedným *primárnym* členom. Ak zápis je

- *local*, čítania z primárneho odrážajú najnovšie zápisy (za predpokladu, že nedošlo k zlyhaniu);
- *majority*, operácie čítania z primárnych alebo sekundárnych členov majú eventuálna konzistenciu.

Poznamenáme, že v MongoDB **kurzor** môže vrátiť ten istý dokument **viackrát** za isté okolnosti (napr. pri zmene indexovaného kľúča), lebo počas vrátenia dokumentov kurzorom, s dotazom sa môžu prelínať iné operácie.

Trvanlivosť v MongoDB hovorí o tom, či zapísané dáta skutočne "prežijú", ak napríklad vypadne prúd alebo spadne server. Pri manuálnom dvojfázovom commite bola trvanlivosť kritická, pretože ak by si "stratil" záznam o tom, že transakcia prebieha, skončil by si s poškodenými dátami.

Eventuálna konzistencia je stav, kedy systém v každom momente nie je v rovnováhe, ale zaručuje, že ak prestanú prichádzať nové zápisy, všetky čítania nakoniec vrátia rovnakú (správnu) hodnotu.

Pri manuálnom dvojfázovom commite nastávala nekonzistencia v týchto prípadoch:

- Priebeh transakcie: Medzi fázou 2 (zmena na dokumente B) a fázou 3 (označenie za dokončené) bol systém v nekonzistentnom stave. Čitateľ mohol vidieť sumu odpočítanú z účtu A, ale ešte nepripísanú na účet B.
- Čítanie z replík (Read Preference): Ak si čítal z "Secondary" uzlov, trvalo niekoľko milisekúnd (replikačný lag), kým sa tam zmena z primárneho uzla dostala.
- Zlyhanie procesu: Ak aplikácia spadla po zmene dokumentu A, ale pred zmenou dokumentu B, systém ostal nekonzistentný navždy, pokiaľ nenastúpil tvoj "recovery" skript, ktorý transakciu dodatočne dokončil alebo vrátil späť.

Problém: Ak aplikácia spadne uprostred procesu, databáza sama o sebe nevie, že má niečo vrátiť späť. Musí nastúpiť iný proces (cleanup script), ktorý to opraví.

Dôsledok: Dáta sú nakoniec konzistentné (eventually consistent), ale nie sú konzistentné v každom jednom okamihu.

Prehľad atomicity, konyistencie a izolácie pred pred verziou 4.0:

Atomicita	Zabezpečená len na úrovni jedného dokumentu. Viac dokumentov si musel *obaliť* pomocou stavového stroja (2PC).
Konzistencia	Bola *eventuálna*. Aplikácia musela byť navrhnutá tak, aby jej nevadilo, že chvíľu vidí neúplné dáta.
Izolácia	Prakticky neexistovala. Ostatné procesy videli čiastkové zápisy transakcie hneď, ako sa udiali.

d) **Monotonicita čítania a zápisu - Causal Consistency.**

Predpokladajme, že aplikácia vykoná postupnosť operácií, ktorá sa skladá

- z operácie **čítania** R1
- za ktorou nasleduje ďalšia operácia **čítania** R2.

Potom v prípade, že aplikácia vykonáva postupnosť operácií na **samostatnej inštancii** mongod, neskoršie čítanie R2 **nikdy** nevracia výsledky, ktoré odrážajú **skorší** stav, ako je vrátené z R1; tzn. R2 vracia dáta, ktoré rastú monotónne (**monotonically increasing**) na aktuálnosti od R1.

Kým MongoDB zaručuje

- **monotónne čítanie** iba pre samostatné inštancie mongod
- **monotónny zápis** pre samostatné inštancie mongod, ale aj skupiny replík a sharded klastre.

II. Dnešný stav

Hoci sú niektoré pojmy zastaralé, neznamená to, že zmizli, ale že ich už nemusíme programovať ručne.

a) Atomicita, nevykonané čítanie
b) Dvojfázové vykonanie, sémantika podobná transakcia
c) Trvanlivosť, eventuálna konzistencia
d) Monotonicita zápisu a čítania - Causal Consistency (kauzálna či príčinná konzistencia

a) Atomicita a Nevykonané čítanie (Dirty Read)

- **Atomicita: OSTOVALA (a rozšírila sa).**
  - *Predtým:* Garantovaná len pre jeden dokument.
  - *Dnes:* Od v4.0 je **natívna** pre viac dokumentov, kolekcií aj databáz. Ak zlyhá jeden krok v transakcii, MongoDB automaticky vráti späť všetky zmeny.
- **Nevykonané čítanie (Dirty Read): ZASTARALA (v rámci transakcií).**
  - *Predtým:* Pri manuálnom 2PC iné procesy videli "rozpracované" dáta (fáza 2).
  - *Dnes:* Vďaka **Snapshot Isolation** čitatelia nikdy nevidia nepotvrdené (dirty) dáta z prebiehajúcej transakcie. Vidia buď stav pred ňou, alebo po nej.

b) Dvojfázové vykonanie a Sémantika podobná transakcia

- **Dvojfázové vykonanie (2PC): ZASTARALA (na úrovni aplikácie).**
  - Manuálne prepisovanie fáz `Initial -> Pending -> Applied` do dokumentov je dnes považované za "anti-pattern". Tento mechanizmus sa presunul z aplikácie priamo do databázového enginu (WiredTiger).
- **Sémantika podobná transakcii: ZASTARALA (nahradená SKUTOČNOU transakciou).**
  - Už nemusíme hovoriť o "simulácii" alebo "sémantike podobnej transakcii". MongoDB od v4.0 poskytuje **plnohodnotné ACID transakcie**, ktoré sa správajú rovnako ako v SQL databázach (Oracle, PostgreSQL).

c) Trvanlivosť a Eventuálna konzistencia

- **Trvanlivosť (Durability): OSTALA (a je kľúčová).**
  - Pojem ostal, ale dnes sa ovláda cez **Write Concern**. Aby bola transakcia skutočne trvanlivá aj pri páde servera, používa sa `w: "majority"` v kombinácii so zapnutým žurnálovaním (`j: true`).
- **Eventuálna konzistencia: OSTALA (ako voliteľná možnosť).**
  - MongoDB je stále distribuovaný systém. Ak čítaš z replík (Secondaries), stále narážaš na eventuálnu konzistenciu (dáta tam dorazia s oneskorením). Avšak pre transakcie a kritické operácie ju už vieš úplne eliminovať pomocou nastavenia `readConcern: "snapshot"`.

d) Monotonicita zápisu a čítania - Causal Consistency (kauzálna či príčinná konzistencia

- **Monotonicita: OSTALA (pod novým názvom Causal Consistency).**
  - Ide o pravidlo, že ak urobíš operáciu A a potom B, systém ich nikdy neuvidí v opačnom poradí.
  - **Monotonické čítanie:** Zaručuje, že ak raz uvidíš určitú verziu dát, nikdy sa ti nestane, že pri ďalšom obnovení uvidíš staršiu verziu (cestovanie v čase).
  - **Monotonický zápis:** Zaručuje, že tvoje zápisy sa vykonajú v poradí, v akom si ich poslal. V modernom MongoDB sa toto rieši cez **Client Sessions** a **Causal Consistency Guarantees**, ktoré sú od v3.6/4.0 štandardom.

III. CRUD príkazy (kurzor: forEach, toArray, next)

- insert vloží jeden alebo viac dokumentov do kolekcie
- update aktualizuje jeden alebo viac dokumentov
- delete maže jeden alebo viac dokumentov
- find vyhľadáva dokumenty v kolekcii
- getLastError vracia chybu poslednej operácie

Dokumenty a kolekcie

- find, findOne, distinct
- ~~insert,~~ insertOne, insertMany
- update, updateOne, updateMany, replaceOne
- ~~remove~~, deleteOne, deleteMany
- validate
- drop, dropIndex
- ~~copyTo~~
- aggregate, count, group
- mapReduce – filter + summary

Kurzor a dokumenty

- forEach, next, hasNext,
- limit, skip, size,
- count, min, max,
- map, sort, toArray

```
//use dbpred2 ;
db.tab1.drop();
db.kol1.drop(); ;
db.dropDatabase();

db.tab1.insertOne( { meno : "Fero", vaha : 82 } ) ;
db.tab1.insertMany([ { meno : "Jano", vaha : 88 }, { meno : "Stevo", vaha : 88 } ]) ;

db.tab1.find().forEach(  function(x) {  db.kol1.insertOne(x);  }  ); // vytvori kol1
// ⇔ rychlejsie:
db.tab1.aggregate([ { $match: {} }, { $out: "kol2" } ])
db.kol1.find();
```

https://www.mongodbmanager.com/clone-mongodb-collection#copyTo

Dokumenty

- ~~db.kol1.insert()~~
- db.kol1.insertOne()
- db.kol1.insertMany()

Delete

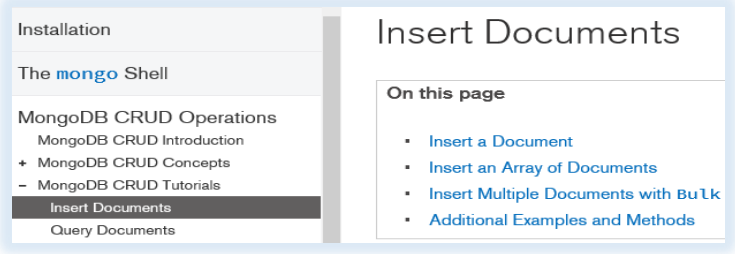
- db.kol1.deleteOne()
- db.kol1. deleteMany () - nemaž, kol1 sa používa

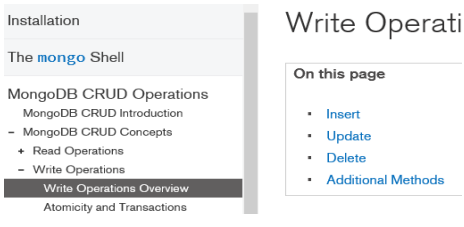
- ~~db.kol1.update()~~
- db.kol1.updateOne()
- db.kol1.updateMany()
- db.kol1.replaceOne()

update operátory

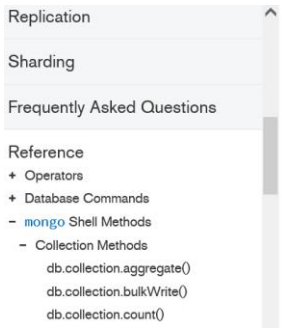
- kľúča
- poľa

- **Dokumenty**





update, delete




```
db.uuu.drop();// odstrani celu kolekciu uuu, tu este nic
db.uuu.insert({ "_id" : 1, "vaha" : 70, vyska : 174 });
var ii = { "_id" : 2, "vaha" : 82, vyska : 175 };
db.uuu.insert(ii);
db.uuu.find();
```

update

```
db.uuu.updateOne( { vaha : 82 }, { $set: { vyska : 177 }
} );
```

upsert – ak neexistuje pre update**, potom insertuj**

```
try {
  db.uuu.updateOne(
    { vaha : 75 },
    { $set: { vyska : 175 } }
   ,{ upsert: true }
  );
} catch (e) {
  print(e);
};  db.uuu.find();
```

```
try {
  db.uuu. deleteMany ( { "vaha" : 74} );  // odstrani dokumenty, splnujuce kriterium
} catch (e) {
  print(e);
};   db.uuu.find();
```

justOne : false – delete viac

```
db.uuu.insert({ "_id" : 4, "vaha" : 77, vyska : 177 });
try {
  db.uuu. deleteMany ( { vyska : 100}, {justOne: false} );
} catch (e) {
  print(e);
};   print(db.uuu.find( ).toArray()); // db.uuu.deleteMany({})   print nie je nutný
```

```
[
  { _id: 1, vaha: 70, vyska: 174 },
  { _id: 2, vaha: 82, vyska: 177 },
  { _id: ObjectId('69f11619dc2d5d2435d54e5d'), vaha: 75, vyska: 175 },
  { _id: 4, vaha: 77, vyska: 177 }
]
```

2.2) Dopytovanie (poradie!)

`db.kol1.find( {Horiz.filt}, {Vert.filt.} );`

Poznámka: RDB pojmy *horizontálna* a *vertikálna filtrácia* sa v MongoDB nepoužívajú. Namiesto nich sa hovorí o dopytovaní, načítaní dokumentov a vrátení vybraných kľúčov (atrib.).

`{Vert.filt.}` ⇔ `{_id : 01, …, klucJ : 1~~, klucK : 1~~},`

Kde ak

- 01 sa rovná 1, potom sa hodnoty zodpovedajúceho kľúča vrátia

- 01 sa rovná 0, potom sa hodnoty zodpovedajúceho kľúča nevrátia

```
db.uuu.find( {}, {_id : 0,   "vaha": 1 } );   // vrat iba kluce  vaha, vyska nie
```

Najprv sa **heslovite** pozreme na `{Horiz.filt}`

I. Dopytovanie **bez** vnorených dokumentov a polí
   A. Dopytovanie **bez vonkajšieho** modifikátora
      a) **Jednoduché** dopytovanie **:** `{ kluc1: hod1, …, klucM : hodM}`
      b) **$** dopytovanie : `hodnotaJ ⇔{$oper1: hod1, …, $operN: hodN }`
   B. Dopytovanie **s vonkajším** modifikátorom ($or)

II. Dopytovanie
   C. **polí**
   D. **vnorených** dokumentov

kde operátor `$oper:` hodnota môže byť napr. `db.uuu.find({"vaha": {$gt:1} });`

Výrazový op.

`{ $gt :1, $lt :5, $in : [2, 3, 4] },`

pozri nižšie.

I. Dopytovanie bez vnorených dokumentov a polí

A. Dopytovanie bez vonkajšieho modifikátora
B. Dopytovanie s vonkajším modifikátorom

A. Dopytovanie bez vonkajšieho modifikátora

a) Jednoduché dopytovanie

`{Horiz.filt} ⇔ {kluc1 : hodnota1, …, klucM : hodnotaM}`

na celú hodnotu kľúča, kde hodnotaJ môže byť:

- skalárna veličina, ako číslo, reťazec alebo datum: 123, "Fero", "2016.4.4"

```
db.kol1.find( {vaha: 88, meno:"Jano"}, {_id : 0, meno : 1,   "vaha": 1 } );
{"meno" : "Jano", "vaha" : 88 }
```

- pole skalárnych veličín: [1, 12.1, 5], ["Fero", "Jano"], ale aj [1, "Fero"]

```
db.kol1.deleteOne({_id:4});
db.kol1.deleteOne({_id:5});
db.kol1.insert({_id:4, vaha:[1,2,3], dtm:"2016.9.9", meno:["Fero", "Jano"]});
db.kol1.insert({ _id : 5, vaha : [1,2,3], dtm:"2017.9.9" });
```

Dopyt na hodnotu celého vektor-kľúča z

```
db.kol1.find({vaha:[1,2,3], dtm: {$gt : "2017.9.8" } });
{ "_id" : 5, "vaha" : [ 1, 2, 3 ], "meno" : "2017.9.9" }
```

b) $ dopytovanie

`{Horiz.filt} ⇔ {$oper1: dok1, …, $operN: dokN }`

```
db.kol1.find( {dtm: {$gt:"2015.9.9", $lt:"2018.9.9"},
               meno:{$in: ["Fero", "Jano"] }} );
{ "_id" : 4, "vaha" : [ 1, 2, 3 ], "dtm" : "2016.9.9", "meno" : [ "Fero", "Jano" ] }
```

$ Operátory dopytovania a projektovania

- porovnávacie operátory (pozri predchádzajúce príklady)
  - $lt, $lte, $gt, $gte s hodnotami typu číslo alebo dátum
  - $eq, $ne s hodnotami ľubovoľného typu
  - $in, $nin
- $not
- vonkajšie modifikátory $or, $nor, $and
- $exists, $type
- pre pole
  - $elemMatch
  - $all
  - $size
  - $slice
  - $ (projection)
- $where klauzula

B. Dopytovanie s vonkajším modifikátorom

Kým posledný dopyt je automaticky AND dopyt, OR dopyt sa určuje explicitne ako vonkajší modifikátor:

```
db.kol1.find({$or:[{dtm: {$gt:"2015.9.9",$lt:"2018.9.9" }},
                   {meno:{$in:["Fero", "Jano"] }} ] } );
{ "_id" : ObjectId("5ad5a545452815f627de2b24"), "meno" : "Fero", "vaha" : 82 }
{ "_id" : ObjectId("5ad5a590452815f627de2b25"), "meno" : "Jano", "vaha" : 88 }
{ "_id" : 4, "vaha" : [ 1, 2, 3 ], "dtm" : "2016.9.9", "meno" : [ "Fero", "Jano" ] }
{ "_id" : 5, "vaha" : [ 1, 2, 3 ], "dtm" : "2017.9.9" }
```

II. Dopytovanie polí a polí vnorených dokumentov

https://www.tutorialsteacher.com/mongodb/documents?utm_content=cmp-true

```
use dbpred2;
db.maz1.drop();
db.maz1.insert( { x: "ab", A: [ 2, 3, 4 ] } );
db.maz1.insert(
  [
        { x: "aa", A: [ 2, 4 ] },
        { x: "aa", A: [ 3, 4, 2 ] }
  ]
);
```

```
db.maz2.drop();
db.maz2.insert(
[
 { x: "ab", A: [ {je:2, ke:3}, {je:2, ke:4} ] },
 { x: "aa", A: [ {je:2, ke:4}, {je:3, ke:4} ] },
 { x: "aa", A: [ {je:3, ke:4}, {je:4, ke:5} ] }
]
); // Run on: https://www.mongodb.com/docs/v4.0/tutorial/query-array-of-documents/
```

Prvý prvok (nultý index) **poľa** A sa rovná 2:

```
db.maz1.find( { 'A.0': 2 } , {_id:0});   //!!! 'A.0' alebo aj uvodzovky:
[{ "x" : "ab", "A" : [ 2, 3, 4 ] }, { "x" : "aa", "A" : [ 2, 4 ] }]
```

```
db.maz2.findOne({}, {_id:0}.toArray(); // vrat bez _id
```

Dopyt

```
db.maz2.find({}, {_id:0}.toArray();
```

vráti pochopiteľne výsledok

```
{ "x" : "ab", "A" : [ { "je" : 2, "ke" : 3 }, { "je" : 2, "ke" : 4 } ] }
{ "x" : "aa", "A" : [ { "je" : 2, "ke" : 4 }, { "je" : 3, "ke" : 4 } ] }
{ "x" : "aa", "A" : [ { "je" : 3, "ke" : 4 }, { "je" : 4, "ke" : 5 } ] }
```

kde kľúč "A" obsahuje pole **vnorených** dokumentov, preto

```
db.maz2.find( { 'A.ke': 4, "A.je": 4 }, {_id : 0}).toArray();  // vrati:
 [ { "x" : "aa", "A" : [ { "je" : 3, "ke" : 4 }, { "je" : 4, "ke" : 5 } ] }  ]
```

2.3) Príklady a kurzory

1) Kolekcia s poľom hodnôt

- **1a) Vytvorenie (insert) kolekcie maz1 s poľom/array hodnôt** A
- **1b) Ktoré kľúče treba vrátiť**
- **1c) Dopytovanie poľa**
- **1d) Kurzor a** forEach

2) Kolekcia s poľom vnorených dokumentov

- **2a) Kurzor -** JSON.stringify
- **2b) Kurzor to array**

3) Generovanie kolekcie

4) Kurzory a JavaScript

- **4a) Pole** – `kur.next()`

1) Kolekcia s poľom hodnôt

1a) Vytvorenie (insert) kolekcie maz1 s poľom/array hodnôt A

A: [2, 3, 4]

Vkladanie jedného dokumentu alebo viac dokumentov naraz pomocou poľa [ ]

```
// ⇔ vyššie
db.maz1.drop();
db.maz1.insert( { x: "ab", A: [ 2, 3, 4 ] } );
db.maz1.insert(
  [
        { x: "aa", A: [ 2, 4 ] },
        { x: "aa", A: [ 3, 4, 2 ] }
  ]
);
```

```
db.maz1.findOne();
{
        "_id" : ObjectId("5707ed009fed16950670b068"),
        "x" : "ab",
        "A" : [
                2,
                3,
                4
        ]
}
```

```
db.maz1.find()
{ "_id" : ObjectId("5707ed009fed16950670b068"), "x" : "ab", "A" : [ 2, 3, 4 ] }
{ "_id" : ObjectId("5707ed009fed16950670b069"), "x" : "aa", "A" : [ 2, 4 ] }
{ "_id" : ObjectId("5707ed009fed16950670b06a"), "x" : "aa", "A" : [ 3, 4, 2 ] }
```

1b) Ktoré kľúče treba vrátiť

```
db.maz1.findOne( { }, {_id: 1} ).toArray();
"_id" : ObjectId("5707ed009fed16950670b068")
```

```
db.maz1.findOne( { }, {_id: 0} );
{ "x" : "ab", "A" : [ 2, 3, 4 ] }
```

```
db.maz1.findOne( { }, {"A":1, _id:0} );
{ "A" : [ 2, 3, 4 ] }
db.maz1.find( { }, {A:1, _id:0} );
[ { "A" : [ 2, 3, 4 ] }, { "A" : [ 2, 4 ] }, { "A" : [ 3, 4, 2 ] } ]
```

1c) Dopytovanie poľa

- Dopytujeme na cele pole

```
db.maz1.find( { A: [ 2, 4 ] } ).toArray();  // nie  A: [ 2, 3 ]  ale  A: [ 2, 3, 4 ]
[ {"_id" : ObjectId("5707ed009fed16950670b069"), "x" : "aa", "A" : [ 2, 4 ] } ]
```

- Dopytujeme na hodnotu prvku

```
db.maz1.find( { A: 3 } ).toArray();  // ⇔
db.maz1.find( { A: { $elemMatch: { $eq: 3 } } } ).toArray();
{ "_id" : ObjectId("5707ed009fed16950670b068"), "x" : "ab", "A" : [ 2, 3, 4 ] }
{ "_id" : ObjectId("5707ed009fed16950670b06a"), "x" : "aa", "A" : [ 3, 4, 2 ] }
```

- Dopytujeme pomocou indexu

```
db.maz1.find( { 'A.1': 4 } ).toArray();  // !!! 'A.1' druhy prvok A sa rovna 4;  nutny apostrof
{ "_id" : ObjectId("5707ed009fed16950670b069"), "x" : "aa", "A" : [ 2, 4 ] }
{ "_id" : ObjectId("5707ed009fed16950670b06a"), "x" : "aa", "A" : [ 3, 4, 2 ] }

db.maz1.find( { A: { $elemMatch: { $gt: 2, $lt: 4 } } } ).toArray();
{ "_id" : ObjectId("5707ed009fed16950670b068"), "x" : "ab", "A" : [ 2, 3, 4 ] }
{ "_id" : ObjectId("5707ed009fed16950670b06a"), "x" : "aa", "A" : [ 3, 4, 2 ] }
```

1d) Kurzor a forEach

```
var kurzor = db.maz1.find();
kurzor.forEach(function(e) {print(e.x, e.A);});  // e  je dokument/riadok
ab 2,3,4
aa 2,4
aa 3,4,2

var kurzor = db.maz1.find().limit(2);
kurzor.forEach(function(e) {print(e.x, e.A[1]);});  // !!!
ab 3
aa 4
```

2) Kolekcia maz2 s poľom vnorených dokumentov A

A : [{je : 2, ne : 3}, {je : 2, ne : 4}]

```
db.maz2.drop();
db.maz2.insert(
[
 { x: "ab", A: [ {je:2, ke:3}, {je:2, ke:4} ] },
 { x: "aa", A: [ {je:2, ke:4}, {je:3, ke:4} ] },
 { x: "aa", A: [ {je:3, ke:4}, {je:4, ke:5} ] }
]
);
```

- Dopytujeme pomocou indexu a kľúča

```
db.maz2.find( { 'A.1.ke': 4 }, {_id:0} ).toArray();
[  { "x" : "ab", "A" : [ { "je" : 2,"ke" : 3 }, { "je" : 2, "ke" : 4 } ] },
   { "x" : "aa", "A" : [ { "je" : 2,"ke" : 4 }, { "je" : 3, "ke" : 4 } ] }  ]
```

- Dopytujeme iba pomocou kľúča

```
db.maz2.find( { 'A.ke': 4 } ).toArray();
{ "_id" : ObjectId("...5e"), "x" : "ab", "A" : [ { "je" : 2,"ke" : 3 }, { "je" : 2, "ke" : 4 } ] }
{ "_id" : ObjectId("...5f"), "x" : "aa", "A" : [ { "je" : 2,"ke" : 4 }, { "je" : 3, "ke" : 4 } ] }
{ "_id" : ObjectId("...60"), "x" : "aa", "A" : [ { "je" : 3,"ke" : 4 }, { "je" : 4, "ke" : 5 } ] }
```

```
db.maz2.find( { 'A.ke': 4, "A.je": 4 } ).toArray();
{ "_id" : ObjectId("...60"), "x" : "aa", "A" : [ { "je" : 3, "ke" : 4 }, { "je" : 4, "ke" : 5 } ] }
```

```
db.maz2.find( { A: { $elemMatch: { "je": 2, "ke": 4 } } } ).toArray();
{ "_id" : ObjectId("...05e"), "x" : "ab", "A" : [ { "je" : 2, "ke" : 3 }, { "je" : 2, "ke" : 4 } ] }
{ "_id" : ObjectId("...05f"), "x" : "aa", "A" : [ { "je" : 2, "ke" : 4 }, { "je" : 3, "ke" : 4 } ] }
```

2a) Kurzor - JSON.stringify

```
var kurzor = db.maz2.find();
kurzor.forEach(function(e) { print(e.x, e.A); });  //  uz sa to vyriesilo!
```

```
ab [object Object],[object Object]
aa [object Object],[object Object]
aa [object Object],[object Object]
```

```
ab [ { je: 2, ke: 3 }, { je: 2, ke: 4 } ]
aa [ { je: 2, ke: 4 }, { je: 3, ke: 4 } ]
aa [ { je: 3, ke: 4 }, { je: 4, ke: 5 } ]
```

Riešenie: JSON.stringify

```
ab [{"je":2,"ke":3},{"je":2,"ke":4}]
aa [{"je":2,"ke":4},{"je":3,"ke":4}]
aa [{"je":3,"ke":4},{"je":4,"ke":5}]
```

```
var kurzor = db.maz2.find();
kurzor.forEach(function(e) { print(e.x, e.A[0]); });
//kurzor.forEach(function(e) {print(e.x, JSON.stringify(e.A[0])); });
```

```
ab {"je":2,"ke":3}
aa {"je":2,"ke":4}
aa {"je":3,"ke":4}
```

2b) Kurzor to array

Kolekciu Autori sme vytvorili na prvej prednáške o MongoDB.

```
use knihy
var kurz = db.Autori.find(); kurz;
var kurz = db.Autori.find(); kurz.toArray();  //        (prehladne)

var kurz = db.Autori.find();
//NO kurz[1] ;
kurz.toArray()[1] ;  // ⇔

var kurz = db.Autori.find();
var dcs = kurz.toArray(); dcs[1];
```

```
{
  _id: ObjectId("661e4656c31e80b5c9933a76"),
  meno: 'Imro',
  adresa: 'AL',
  dat_nar: '1990',
  knihy: [
    { nazov: 'RDBS', zaner: 'PC', rok: 2005, cena: 45 },
```

```
    { nazov: 'NoSQL', zaner: 'PC', rok: 2010, cena: 45 },
    { nazov: ' C# ', zaner: 'PC', rok: 2016, cena: 50 }
  ]
}
```

a[1].A[1].je ak je potrebne, vratme sa k def. maz2

```
use dbpred2
//   db.maz2.find().toArray().je;  // nie
a = db.maz2.find().toArray(); a[1].A[1].je;
3
```

s využitím pomocnej kolekcie MAZ

```
a = db.maz2.find().toArray();  db.MAZ.insert(a[1].A[1]);  db.MAZ.find( {je:3} );
{ "_id" : ObjectId("570a22197a0780fe760ea78b"), "je" : 3, "ke" : 4 }
```

3) Generovanie kolekcie

Math.random() vráti pseudonáhodné číslo z intervalu [0; 1], teda ide o rovnomerné rozdelenie.

```
function plusKdni(datum, k) {
   var d = new Date(datum);
   d.setDate(d.getDate( ) + k);
   return d;
};

function plusKrokov(datum, k) {
   var d = new Date(datum);
   d.setYear(d.getFullYear( ) + k);
   return d;
};

function randomAB(A, B) {
   return Math.floor( A+(Math.random( )*(B-A+1)) );
};
```

```
new Date(2016, 4-1, 4+1)
ISODate("2016-04-04T22:00:00Z")
```

```
plusKrokov(new Date(), 1);
ISODate("2017-04-18T20:32:55.191Z")
```

```
plusKdni(new Date(), 1);
ISODate("2016-04-19T20:30:09.020Z")
```

```
for ( i=1; i<=5; i++) { print(randomAB(10,20)) }; // vypise 5 cel.cisel z intervalu [10;20]
```

Vytvorme kolekciu *studenti* s 29-mi dokumentami

```
db.studenti.drop();
for (i=1; i<=29; i++){db.studenti.insert( { "i":i, "meno": "student" + randomAB(10,20) } ) };
db.studenti.find({meno:"student18"},{i:1, meno:1,_id:0}).count();  // vykonajme viackrat
```

```
db.studenti.find({meno:"student18"},{i:1, meno:1,_id:0}).toArray();
```
```
{ "i" : 7, "meno" : "student18" }
{ "i" : 99, "meno" : "student18" }
```

Dopyt štandardne vráti iba prvých dvadsať dokumentov. Aby sme uvideli ďalšie dokumenty, systém po vykonaní limit dopytu nás upozorňuje, aby sme zadali `it`

```
db.studenti.drop();
for (i=1; i<=79; i++){db.studenti.insert( { "_id":i, "meno": "student" + randomAB(10,20) } ) };
db.studenti.find().limit(41).toArray();
```

4) Kurzory a JavaScript

```
var kur = db.studenti.find({},{i:1, meno:1,_id:0});
kur;
kur.count( );  // 79
```

```
var kur = db.studenti.find({},{i:1, meno:1,_id:0});
kur.limit(50);
```

```
var kur = db.studenti.find({},{i:1, meno:1,_id:0}); // kur
while(kur.hasNext()){
     var k = kur.next();
     //print(k.i,  k.meno);
     if(k.meno=="student18"){print(k.i, k.meno);}
};
```

```
6 student18
17 student18
...
67 student18
```

⇔

```
var kur = db.studenti.find({},{i:1, meno:1,_id:0}); // kur
var A= kur.toArray();
for( k = 0; k <  kur.count( ); k++){
          //print(A[k].i, A[k].meno);
         if( A[k].meno =="student18"){ print(A[k].i, A[k].meno);}
};
```

4a) Pole – `kur.next()`

```
var kur = db.maz2.find({}, {_id:0}); //kur;
var B= kur.toArray(); B
print( JSON.stringify(B) );
```

```
[
  { x: 'ab', A: [ { je: 2, ke: 3 }, { je: 2, ke: 4 } ] },
  { x: 'aa', A: [ { je: 2, ke: 4 }, { je: 3, ke: 4 } ] },
  { x: 'aa', A: [ { je: 3, ke: 4 }, { je: 4, ke: 5 } ] }
]
```

[{"x":"ab","A":[{"je":2,"ke":3},{"je":2,"ke":4}]},{"x":"aa","A":[{"je":2,"ke":4}
,{"je":3,"ke":4}]},{"x":"aa","A":[{"je":3,"ke":4},{"je":4,"ke":5}]}]

```
var kur = db.maz2.find();
while(kur.hasNext()){
     var k = kur.next();
     print(k.x,   JSON.stringify(k.A[0]) );
};
```

```
ab {"je":2,"ke":3}
aa {"je":2,"ke":4}
aa {"je":3,"ke":4}
```

```
var kur = db.maz2.find();
while(kur.hasNext()){
     var k = kur.next();
     print(k.x, JSON.stringify(k.A[0].je));
};
```

```
ab 2
aa 2
aa 3
```

```
var kur = db.maz2.find();
while(kur.hasNext()){
     var k = kur.next();
     print(k.x, JSON.stringify(k.A));
};
```

```
ab [{"je":2,"ke":3},{"je":2,"ke":4}]
aa [{"je":2,"ke":4},{"je":3,"ke":4}]
aa [{"je":3,"ke":4},{"je":4,"ke":5}]
```

```
var kur = db.maz2.find({}, {_id:0});
while(kur.hasNext()){
     var k = kur.next();
     var x;
     var s = "";
     for (x in k) {
        s += JSON.stringify(k[x])+",";
     }
     print(s);
};
```

```
"ab",[{"je":2,"ke":3},{"je":2,"ke":4}],
"aa",[{"je":2,"ke":4},{"je":3,"ke":4}],
"aa",[{"je":3,"ke":4},{"je":4,"ke":5}],
```

Optimalizácia dopytu

- indexy
- `db.kol1.explain()`