

NoSQL adatbáziskezelők

Szathmáry László
Debreceni Egyetem
Informatikai Kar

Utolsó gyakorlat

- sématervezés
- indexek használata
- konklúzió

(utolsó módosítás: 2017. nov. 30.)

Sématervezés

A MongoDB használata során a nagy kérdés: az adatokat ágyazzuk be, vagy tegyük őket külön kollekcióba?

Relációs adatbázis-kezelő rendszerek esetén a 3NF használata preferált. A MongoDB esetében viszont azt kell átgondolni, hogy az alkalmazásban hogyan akarjuk felhasználni az adatokat. Vagyis az alkalmazáshoz igazítjuk hozzá a sémát (*Application-Driven Schema*).

Milyen adatokat használunk együtt?

Mely adatokat olvassuk sokat?

Mely adatokat módosítjuk sokat?

} ezeket a szempontokat is
figyelembe kell venni

Jellemzői:

- beágyazott adatok használata
- nincs join művelet
- atomiak a műveletek (de nincs tranzakció)
- egy kollekciónak nincs deklarált sémája (habár a dokumentumok általában eléggé hasonlóak)

Ha a sémánk úgy néz ki, mint egy RDBMS séma, akkor a MongoDB-t valószínűleg nem megfelelően használjuk.

Vegyünk pl. egy blogot. Egy RDBMS rendszerben lenne egy *post* táblánk, ami a blogbejegyzéseket tartalmazza. Lehetséges mezők: cím, tartalom, létrehozási dátum. Egy post-hoz tartoznak kulcsszavak is, de a tag-eket már egy másik táblába kellene tenni (*tags*). Egy post-hoz tartoznak kommentek is, melyek egy harmadik táblába kerülnének (*comments*). Vagyis ha meg akarunk jeleníteni egy blogposztot, akkor ehhez legalább 3 táblát kellene felhasználnunk, ill. join művelettel összekapcsolni.

MongoDB esetében egy post dokumentumba be lehet illeszteni a tag-eket egy listába. Ugyanígy a kommentek is beágyazhatók az adott posztba, elvégre egy post-hoz nem szokott túl sok komment tartozni. Itt a post-ok megjelenítéséhez csak egyetlen kollekció dokumentumait kell lekérdezni, s egy dokumentum minden szükséges adatot tartalmaz.

1:1 kapcsolat (One to One)

Példa 1:1 kapcsolatra: *Employee: Resume*. Egy alkalmazottnak egy önéletrajza van, ill. egy önéletrajz egy alkalmazotthoz tartozik.

Két lehetőségünk is van:

- 1) Különválasztjuk őket két kollekcióba. Az Alkalmazott tartalmaz egy *resume_id*-t, míg az Önéletrajz tartalmaz egy *employee_id*-t.
- 2) Beágyazás. Az önéletrajzt beágyazzuk az alkalmazott dokumentumába.

Melyiket válasszuk?

- Egy dokumentum mérete a MongoDB-ben max. 16 MB lehet. Ha az alkalmazott és az önéletrajz összmérete ezt meghaladja, akkor nem lehetnek együtt.
- Ha az önéletrajz nagy (pl. képeket is tartalmaz), de ritkán van rá szükség, az alkalmazottat viszont gyakran olvassuk, akkor szét lehet őket választani.
- Ha az önéletrajz nagyon gyakran frissül, akkor is lehet külön.
- 1:1 kapcsolat esetében viszont a legtermészetesebb a beágyazás. Ha az előző speciális esetek nem állnak fenn, akkor nyugodtan be lehet ágyazni az önéletrajzt az alkalmazott dokumentumába.

1:N kapcsolat (One to Many / One to Few)

Példa 1:N kapcsolatra: *City: Person*. Egy városnak sok lakója van, ill. egy személynek csak egy állandó lakhelye van.

A város lakóit be lehetne tenni egy listába. Viszont egy város lakossága nagyon sok lehet, így ez a lista nagyon nagy méretűvé válna. Ebben az esetben a szétválasztás tűnik jó megoldásnak.

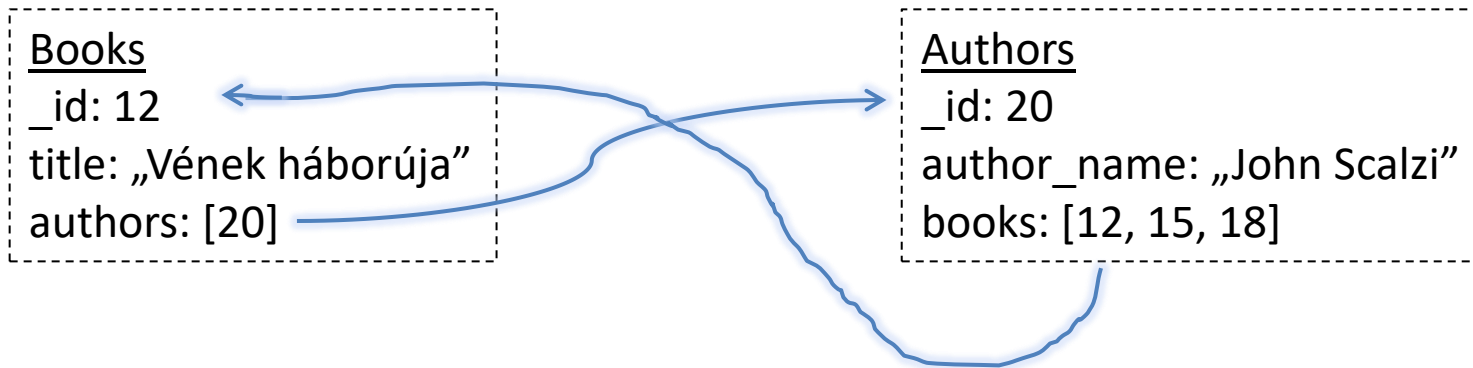
Viszont mi a helyzet akkor, ha nem *One to Many*, hanem csak *One to Few* kapcsolatról van szó, vagyis amikor egy elemhez csak néhány elem tartozik? Gondoljunk vissza a blogposztos példára: egy poszthoz általában nem szokott túl sok komment tartozni. Ebben az esetben a kommentek beágyazhatók a blogposztba egy lista formájában.

Összefoglalva:

<i>One to Many:</i>	két kollekció használata lesz talán a legjobb megoldás
<i>One to Few:</i>	beágyazás

N:M kapcsolat (Many to Many)

Példa N:M kapcsolatra: *Books: Authors*. Egy könyvnek több szerzője is lehet, ill. egy szerzőnek lehet több könyve is.



Egy másik megoldás lehetne a beágyazás, de ez anomáliákhoz vezethet. Pl. az **Authors** alá tesszük be a könyveket. Ekkor ugyanaz a könyv több szerző alatt is ugyanúgy jelenne meg. Mi van akkor, ha módosítani akarjuk egy könyv adatait? Az összes helyen frissíteni kellene...

Indexek használata

Alapesetben, ha keresünk valamit, akkor végigmegyünk az egész kollekción (*collection scan*). Ha nagy az adathalmaz, akkor ez nagyon lassú lesz.

Ötlet: használjunk indexeket. Az indexben a kulcsok rendezetten szerepelnek. Ezen már gyorsan lehet keresni, ill. hatékony keresési algoritmusok is használhatók, pl. bináris keresés (a MongoDB valójában B-fát használ). Az indexben lévő kulcsot gyorsan meg tudjuk találni, majd a mutatót követve megvan a keresett dokumentum is.

Ha van index:

- Az olvasás nagyon gyors.
- Az írás kicsivel lassabb lesz, mivel beszúráskor az indexet frissíteni kell. Vagyis nem kell mindenre indexet tenni! Csak arra tegyünk indexet, amit a leggyakrabban lekérdezőnk.

Egy adatbázis teljesítményére az indexek használata van a legnagyobb hatással. Ha jól választjuk meg az indexeket, akkor csak nagyon kevés lekérdezésnek kell teljesen végigszkennelni egy kollekciót.

Futtassuk le az alábbi szkriptet. Ez a students kollekcióba betesz 5 millió dokumentumot:

```
4  import pymongo
5
6  conn = pymongo.MongoClient()
7  db = conn['test']
8  students = db['students']
9
10 def insert():
11     students.drop()
12     for i in xrange(1, 5000000+1):
13         students.insert({"student_id": i})
14
15 def main():
16     insert()
```

Végezzünk el néhány lekérdezést:

```
> db.students.find({'student_id': 4000000})
```

Ez eltart néhány másodpercig. A find() az összes előfordulást megkeresi, nem csak az elsőt, vagyis az egész kollekciót végignézi.

További lekérdezések:

```
> db.students.findOne({student_id: 10})
```

A `findOne()` csak az első találatig keres. Mivel ez a dokumentum ott van a kollekció elején, ezért nagyon gyorsan végezni fog.

```
> db.students.findOne({student_id: 3000000})
```

Ez is lassú lesz. Igaz, hogy csak az első találatig keres, de ez a dokumentum nem a kollekció elején szerepel.

index létrehozása

```
> db.students.createIndex({student_id: 1})
```

Ez a `students` kollekción létrehoz egy indexet. Mivel 5 millió dokumentumunk van, az index felépítése eltart egy pár percre. Az „1”-es érték jelentése: a `student_id`-ra növekvő sorrendben építjük fel az indexet (a -1 jelentése csökkenő sorrend lenne).

Egy új index létrehozásakor visszajelzést is kapunk:

```
> db.students.createIndex({student_id: 1})
{
  "createdCollectionAutomatically" : false,
  "numIndexesBefore" : 1,
  "numIndexesAfter" : 2,
  "ok" : 1
}
```

Vagyis: a parancs kiadása előtt egy index volt, majd végrehajtás után két indexünk lett. Minden kollekción van egy index, ami az „_id” mezőre épül fel.

Ha most kipróbáljuk ismét a find() függvényt, akkor azt fogjuk tapasztalni, hogy szinte azonnal választ kapunk a lekérdezésre (pl. egy lapon tesztelve index nélkül 2114 ms-ig tartott a lekérdezés, míg ugyanez index használatával 34 ms-ra csökkent).

index törlése


```
> db.students.dropIndex({student_id: 1})
```

Az index törlése a létrehozáshoz hasonlóan történik, csak itt a `dropIndex()` függvényt kell meghívni.

indexek felfedezése


```
> db.system.indexes.find()
```

ez az összes kollekciónál
ad információt



```
> db.students.getIndexes()
[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },
    "name" : "_id_",
    "ns" : "test.students"
  },
  {
    "v" : 1,
    "key" : {
      "student_id" : 1
    },
    "name" : "student_id_1",
    "ns" : "test.students"
  }
]
```

csak az adott kollekció
indexeit mutatja



integritási megszorítások

Tegyük fel, hogy az egyik mezőben csak egyedi (unique) értékeket akarunk engedélyezni. Ezt a megszorítást úgy tudjuk elérni, hogy az adott mezőre (vagy mezőkre) létrehozunk egy indexet, s unique-nak jelöljük meg. Pl. két diáknak ne lehessen azonos beceneve:

```
> db.students.createIndex({nickname: 1}, {unique: true})
```

Ha egy duplikátumot akarunk beszúrni, akkor hibát kapunk.

Ha a kollekció már tartalmaz duplikátumokat egy mezőn, s arra unique indexet akarunk tenni, akkor hibát kapunk.

A dropDups kapcsolóval eltávolíthatók a duplikátumok. Viszont **ez egy veszélyes művelet!** Nem lehet tudni, hogy melyik dokumentumot tartja meg egyedinek!
Használata:

```
> db.students.createIndex({nickname: 1}, {unique: true, dropDups: true})
```

explain

Az `explain()` függvény egy lekérdezésről ad plusz információkat. Innen tudjuk kideríteni például, hogy a lekérdezés során mely indexeket használta a rendszer.

```
> db.students.find({student_id: 400}).explain()
{
  "cursor" : "BtreeCursor student_id_1",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 1,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 1,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 1,
  "nChunkSkips" : 0,
  "millis" : 27,
  "indexBounds" : {
    "student_id" : [
      [
        400,
        400
      ]
    ]
  },
  "server" : "pybox:27017",
  "filterSet" : false
}
```

BasicCursor: nem használt indexet

BtreeCursor: használt indexet (itt a `student_id` mezőre felépítettet)

n: találatok száma

nscanned: hány dokumentumot kellett végignézni (itt az index miatt egyből megtaláltuk a keresett elemet)

millis: a lekérdezés mennyi ideig tartott msec-ban

index mérete

Egy kollekció, ill. a hozzá tartozó index(ek) méretét a következőképpen tudjuk lekérdezni:

```
> db.students.stats()
{
  "ns" : "test.students",
  "count" : 5000000,
  "size" : 240000208,
  "avgObjSize" : 48,
  "storageSize" : 410312704,
  "numExtents" : 14,
  "nindexes" : 2,
  "lastExtentSize" : 114012160,
  "paddingFactor" : 1,
  "systemFlags" : 1,
  "userFlags" : 1,
  "totalIndexSize" : 288032304,
  "indexSizes" : {
    "_id_" : 162228192,
    "student_id_1" : 125804112
  },
  "ok" : 1
}
```

count: dokumentumok száma
(itt most: 5 millió elem)
avgObjSize: egy dokumentum
átlagos mérete (itt: 48 byte)
storageSize: a merevelemezen
mennyi helyet foglal a kollekció
(itt: 410 MB)
totalIndexSize: indexek
összmérete (itt: 288 MB)

Vagyis: az index nincs ingyen! Jól
át kell gondolni, hogy mire
teszünk indexet.

Full Text Search index

Tegyük fel, hogy az adatbázisunkban nagyméretű szövegeket tárolunk, s ezekben hatékonyan szeretnénk tudni keresni. Ekkor a szöveget tartalmazó mezőre (itt: words mező) egy *text* típusú indexet érdemes tenni:

```
> db.sentences.createIndex({'words': 'text'})
```

Keresés:

```
> db.sentences.find({'$text': {'$search': 'herceg'}})
```

A keresett 'herceg' szó a szövegben bárhol előfordulhat, meg fogja találni.

```
> db.sentences.find({'$text': {'$search': 'herceg macska kulcs'}})
```

Ha több keresési kifejezést is megadunk, akkor ezeket ilyenkor logikai VAGY operátorral kapcsolja össze. Vagyis: keressük azokat a dokumentumokat, melyekben a herceg, macska, vagy kulcs szavak bármelyike előfordul.

Konklúzió

- Bepillantást nyertünk a MongoDB használatába, ami a legismertebb NoSQL adatbáziskezelő-rendszer.
- Láthattuk, hogy nem csak SQL-ben, ill. táblákban lehet gondolkodni, hanem (JSON) dokumentumokban is.
- Láttuk a parancssoros shell használatát.
- A MongoDB-hez léteznek grafikus adminisztrációs felületek is (pl. Studio 3T).
- Láttuk, hogy hogyan lehet használni a Mongo-t egy alkalmazásból (Python).
- Ideális választás gyors prototípusfejlesztéshez, amikor menet közben alakul ki az adatbázis szerkezete (dinamikus sémák).
- Kik használják a MongoDB-t: <https://www.mongodb.com/who-uses-mongodb>