FIRST - Jövő Internet kutatások az elmélettől az alkalmazásig

TÁMOP-4.2.2.C-11/1/KONV-2012-0001

A MongoDB adatbázis-kezelő rendszer

Szathmáry László

2016

A projektek az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósulnak meg.

MongoDB

http://www.mongodb.org/

A MongoDB egy nem-relációs, dokumentum-orientált adatbázis-kezelő rendszer, mely a NoSQL családba tartozik.
Az első változat 2009-ben jött ki, vagyis relatíve új.

Legfontosabb eltérések a tradicionális relációs adatbázis-kezelőkkel szemben:

1. nem támogatja a join műveletet
2. nincsenek tranzakciók
3. a sémák flexibilisek, vagyis egy kollekcióban nem kell minden dokumentumnak egyazon séma szerint felépülnie

Nincs join, de vannak alternatívák, pl.: beágyazás. A join-t természetesen az alkalmazáskódban is meg lehet valósítani több lekérdezés használatával.

MongoDB

Új mezőket (jellemzők, *properties*) bármikor, dinamikusan hozzá lehet adni egy „rekordhoz”, nincs szükség ALTER TABLE utasításokra.

Ha ismerjük a JSON szerializációs formátumot, akkor azonnal ismerősnek fognak tűnni a MongoDB dokumentumai és lekérdezései.

A gyorsabb lekérdezés végett itt is ki lehet alakítani indexeket bizonyos mezőkre.

SQL adatbázis	MongoDB
adatbázis	adatbázis
tábla	kollekció
sor / rekord	dokumentum
oszlop	mező / jellemző (property)

MongoDB terminológia

JSON

A JSON sok esetben használható az XML alternatívájaként

- szintén szöveg-alapú, az ember számára is jól olvasható
- szintén hierarchikus felépítésű
- szintén nagyszerűen használható alkalmazások közti kommunikációra
- egyszerűbb mint az XML
- nem annyira bőbeszédű, rövidebb
- gyorsabban lehet írni/olvasni

XML:

```
<Person>
   <FirstName>Homer</FirstName>
   <LastName>Simpson</LastName>
   <Relatives>
      <Relative>Grandpa</Relative>
      <Relative>Marge</Relative>
      <Relative>The Boy</Relative>
      <Relative>Lisa</Relative>
   </Relatives>
</Person>
```

JSON:

```
{
   "firstName": "Homer",
   "lastName": "Simpson",
   "relatives": [ "Grandpa", "Marge", "The Boy", "Lisa" ]
}
```

XML:

```
<persons>
 <person>
  <name>Ford Prefect</name>
  <gender>male</gender>
 </person>
 <person>
  <name>Arthur Dent</name>
  <gender>male</gender>
 </person>
 <person>
  <name>Tricia McMillan</name>
  <gender>female</gender>
 </person>
</persons>
```

JSON:

```
[
  {
   "name": "Ford Prefect",
   "gender": "male"
  },
  {
   "name": "Arthur Dent",
   "gender": "male"
  },
  {
   "name": "Tricia McMillan",
   "gender": "female"
  }
]
```

XML:

```
<settings>
    <path>/home/luke/Dropbox/Public</path>
    <user_id>123456</user_id>
    <auto_sync>True</auto_sync>
</settings>
```

JSON:

```
{
   "path": "/home/luke/Dropbox/Public",
   "user_id": 123456,
   "auto_sync": true
}
```

JSON szintaktikai szabályok

- az adatok kulcs/érték párok
- az adatok vesszővel vannak elválasztva
- a kapcsos zárójelek objektumot jelölnek
- a szögletes zárójelek tömböt jelölnek

JSON értékek

Egy JSON érték a következő lehet:

- szám (egész vagy lebegőpontos)
- sztring (idézőjelek között)
- boolean (true vagy false)
- tömb (szögletes zárójelek között)
- objektum (kapcsos zárójelek között)
- null

Telepítés Ubuntu 14.04 alatt #1

A MongoDB telepítését Linux alatt nézzük meg, de van bináris telepítő Windows illetve Mac OS X rendszerekhez is.

A 32 bites verzió esetén az adatok mérete max. 2 GB lehet. Ezen korlátozás miatt érdemesebb inkább a 64 bites verziót telepíteni.

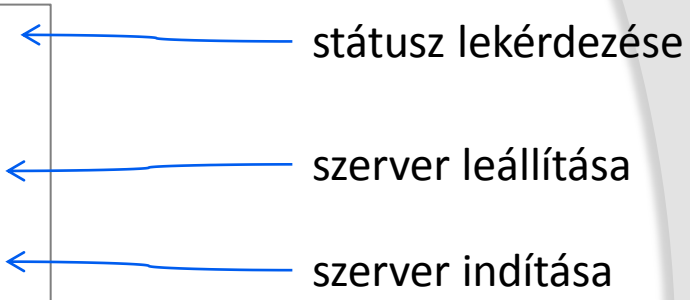
Ha Ubuntu alatt a legfrissebb verziót szeretnénk feltenni, akkor ne a hivatalos Ubuntu repository-ból telepítsük, hanem közvetlenül a fejlesztőktől:

https://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/

Telepítés Ubuntu 14.04 alatt #2

A telepítő el is indítja a szervert. Erről meg is győződhetünk:

```
jabba@montreal:~$ sudo service mongod status
[sudo] password for jabba:
mongod start/running, process 6408
jabba@montreal:~$ sudo service mongod stop
mongod stop/waiting
jabba@montreal:~$ sudo service mongod start
mongod start/running, process 6645
```



A szerver alapértelmezés szerint a 27017-es porton figyel.

Ha nem fut a szerver és nem tudjuk elindítani, akkor elképzelhető, hogy egy nem szabályos leállás során nem törlődött a lock állomány. A lock állomány helye:

`/var/lib/mongodb/mongod.lock`

Telepítés Manjaro alatt

A MongoDB telepítését Ubuntu mellett nézzük meg egy másik (egyre népszerűbb) disztribúción, a Manjaro Linuxon is.

Itt könnyebb dolgunk van, ui. a repository-k gyorsabban frissülnek, mint az Ubuntu esetében. A telepítést a következő paranccsal tudjuk megtenni:

```
$ sudo pacman -S mongodb
```

A MongoDB itt nem indul el automatikusan. Ezt a következőképpen tudjuk engedélyezni:

```
$ systemctl enable mongodb.service
$ systemctl start mongodb.service
```

Újraindításkor mostantól automatikusan el fog indulni a MongoDB.

Telepítés után

Telepítés után a következőképpen tudunk meggyőződni arról, hogy tudunk-e kapcsolódni a szerverhez:

```
$ mongo
MongoDB shell version: 3.2.4
connecting to: test
>
```

nem kaptunk hibát, minden rendben

Lényeges könyvtárak / állományok:

```
/var/lib/mongodb

/var/lib/mongodb/mongod.lock

/etc/mongodb.conf
```

A bináris adatbázisok alapértelmezett helye.

lock állomány

konfigurációs állomány

Telepítés után

A MongoDB elég nagyvonalúan allokál tárterületet az adatbázisok számára. Ez éles környezetben rendben van, de egy fejlesztői gépen valószínűleg szeretnénk spórolni a tárhellyel.

Adjuk hozzá a következő sorokat a `/etc/mongodb.conf` végéhez:

```
# disable data file preallocation
noprealloc = true
# use smaller files
smallfiles = true
```

Majd indítsuk újra a MongoDB szervert. Az új beállítások csak az újonnan létrehozott adatbázisokra fognak vonatkozni!

Webes adminisztráció

A RockMongo egy PHP-ban írt webes adminisztrációs felület MongoDB adatbázisokhoz. Szerepét tekintve a PHPMyAdmin-hoz lehetne hasonlítani.

https://github.com/iwind/rockmongo

Telepítés Ubuntu alatt

Töltsük le a tömörített archívumot, majd bontsuk ki például a következő könyvtárba: `/home/USER/public_html/rockmongo/`

Majd:

```
sudo apt-get install php-pear
sudo pecl install mongo
```

Webes adminisztráció

Nyissuk meg a `php.ini` állományt (`/etc/php5/apache2/php.ini`) s adjuk a végéhez a következő sort:

```
extension=mongo.so
```

Majd indítsuk újra a webszervert.

Végül látogassuk meg a `http://localhost/~USER/rockmongo` webhelyet. Alapértelmezett felhasználói név / jelszó:
admin / *admin* .

Mindehhez természetesen az is kell, hogy a PHP szkriptek futtatása aktiválva legyen a `HOME` könyvtárunkban lévő `public_html` könyvtárban.

Webes adminisztráció

Telepítés Manjaro alatt

Először is tegyük fel az Apache webszervert, ill. a hozzá tartozó PHP támogatást:

```
$ yaourt apache22
$ yaourt php-apache22
```

Majd indítsuk el a webszervert:

```
$ systemctl enable httpd
$ systemctl start httpd
```

Ellenőrzésképpen látogassuk meg a http://localhost címet. Ha minden jól ment, akkor hibaüzenet helyett egy üres könyvtárat kell kapnunk.

Webes adminisztráció

Aktiváljuk a PHP támogatást. Nyissuk meg az `/etc/httpd/conf/httpd.conf` állományt, majd a végére illesszük be az alábbi sorokat:

```
LoadModule php5_module        modules/libphp5.so
AddHandler php5-script php
Include conf/extra/php5_module.conf
```

Indítsuk újra a webszervert (`systemctl restart httpd`).

A HOME könyvtárunkban hozzunk létre egy `public_html` nevű könyvtárat, majd ebben egy teszt PHP állományt `index.php` néven az alábbi tartalommal:

```
<?php
    echo phpinfo();
?>
```

Nyissuk meg ezt a file-t az alábbi címen:

http://localhost/~USER/index.php

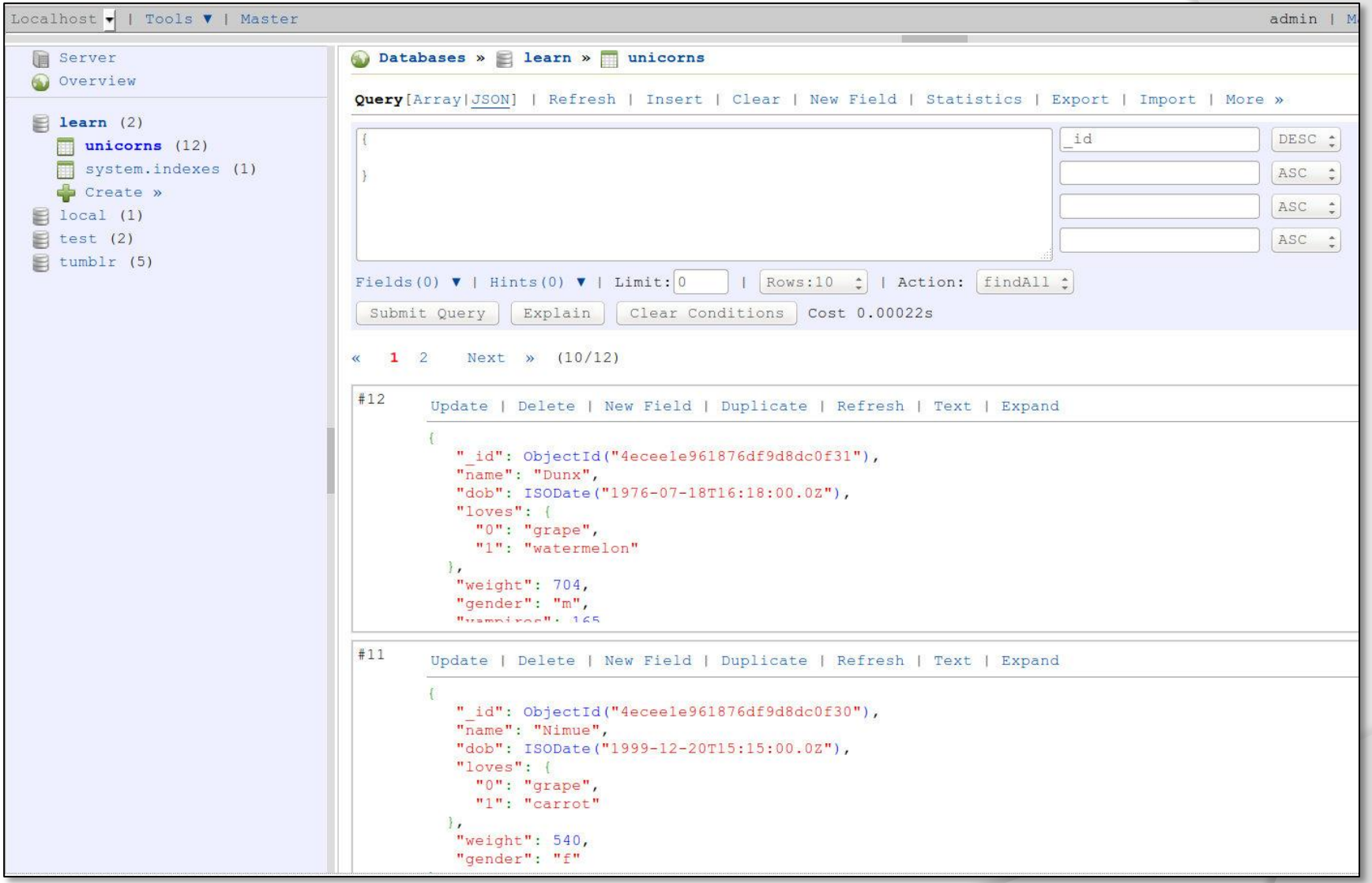
Itt a PHP telepítésünkról szóló részletes listát kell kapnunk.

Webes adminisztráció

Most már csak a PHP és a MongoDB közötti kommunikációt kell megteremteni. Ehhez telepítsük fel a `php-mongo` csomagot, majd indítsuk újra a webszervert.

A https://github.com/iwind/rockmongo címről töltsük le a RockMongo-t tömörített formában majd bontsuk ki a következő könyvtárba: `/home/USER/public_html/rockmongo/`

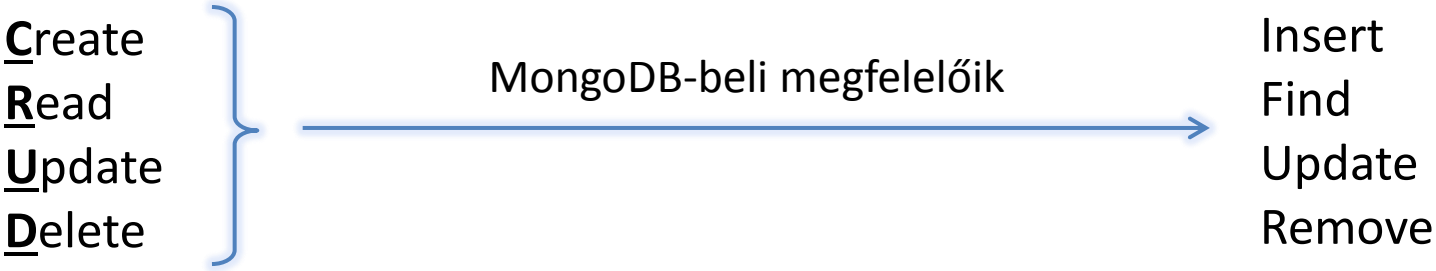
Végül látogassuk meg a `http://localhost/~USER/rockmongo` webhelyet. Alapértelmezett felhasználói név / jelszó: *admin* / *admin* .



A RockMongo webes felülete

CRUD műveletek mongo shellben

CRUD alatt a következő alapvető műveleteket értjük:



Ezeket a műveleteket először a mongo shellben, majd pedig Python alkalmazásokban nézzük meg.

```
$ mongo
MongoDB shell version: 3.2.4
connecting to: test
> show collections
>
```

A mongo shellt a „mongo” parancs kiadásával tudjuk elindítani. Ez tulajdonképpen egy JavaScript interpreter. Alapértelmezés szerint a „test” nevű adatbázishoz csatlakozik. A „show collections” segítségével tudjuk lekérdezni egy adatbázis „tábláit”, vagyis kollekcióit. Jelenleg még nincs egyetlen kollekciónk sem.

```
$ mongo
MongoDB shell version: 3.2.4
connecting to: test
> show collections
> doc1 = {name: "John", age: 30, profession: "hacker"}
{ "name" : "John", "age" : 30, "profession" : "hacker" }
> db.people.insert(doc1)
WriteResult({ "nInserted" : 1 })
> doc2 = {name: "Jessie", age: 35, profession: "programmer"}
{ "name" : "Jessie", "age" : 35, "profession" : "programmer" }
> db.people.insert(doc2)
WriteResult({ "nInserted" : 1 })
> show collections
people
system.indexes
>
```

Létrehozunk egy „doc1” nevű JSON objektumot. Egy kollekció JSON objektumok halmaza. A dokumentum 3 mezővel rendelkezik: név, életkor, foglalkozás. Ezt egy másik dokumentummal együtt beszúrjuk a „people” nevű kollekcióba. A „db”-vel az aktuális adatbázisra hivatkozunk, mely most a „test” nevű adatbázis. Beszúrás után a shell vissza is jelez, hogy sikeres volt a beszúrás. A „show collections” parancs hatására látható, hogy a „people” kollekció automatikusan létrejött. Vagyis ha egy nem létező kollekcióba szúrunk be, a MongoDB automatikusan létrehozza a kollekciót.
Létrejött egy „system.indexes” kollekció is, mely az indexeket tárolja, de erről majd később lesz szó.

```
> db.people.find()
{ "_id" : ObjectId("54d1e6566b6fa7670efaeb20"), "name" : "John", "age" : 30,
"profession" : "hacker" }
{ "_id" : ObjectId("54d1e69c6b6fa7670efaeb21"), "name" : "Jessie", "age" :
35, "profession" : "programmer" }
>
> db.people.findOne()
{
        "_id" : ObjectId("54d1e6566b6fa7670efaeb20"),
        "name" : "John",
        "age" : 30,
        "profession" : "hacker"
}
```

A find() függvény segítségével egy kollekció összes dokumentumát tudjuk lekérdezni. A findOne() függvény csupán egyetlen dokumentumot ad vissza. Ezt inkább akkor használjuk, amikor a dokumentumok felépítésére vagyunk kíváncsiak (pl. milyen mezők vannak benne).
Mint látható, minden dokumentum rendelkezik egy „_id” mezővel. Ez az elsődleges kulcs. Ha ezt beszúráskor nem adjuk meg, akkor a MongoDB automatikusan létrehoz a dokumentumban egy ilyen mezőt, melynek az értéke egy ObjectId objektum lesz, ami egy 96 bites hash érték.

(A hash érték előállításakor figyelembe veszi az időt, a futtató gépet, processz ID-t, ill. egy globális számlálót a processzen belül. Ezek miatt az ütközés veszélye nagyon kicsi.)

```
> db.people.find({name: "John"})
```

Az SQL **WHERE** részének felel meg. Vagyis: keressük azokat a dokumentumokat, ahol a „name” mező értéke „John”.

```
> db.people.find({name: "John"}, {name: true})
```

Az SQL **SELECT** részének felel meg. Vagyis: az eredmény rekordoknak csak a „name” mezőjét akarjuk látni.

1
2
3
4

```
> db.people.find({name: "John"}, {name: true})
{ "_id" : ObjectId("54d1e6566b6fa7670efaeb20"), "name" : "John" }
> db.people.find({name: "John"}, {name: true, _id: false})
{ "name" : "John" }
> db.people.find({}, {name: true, _id: false})
{ "name" : "John" }
{ "name" : "Jessie" }
> db.people.find({}, {name: 1, _id: 0})
{ "name" : "John" }
{ "name" : "Jessie" }
>
```

Az első példában beállítottuk, hogy csak a „name” mezőt akarjuk megtartani az eredményben. Ennek ellenére a Mongo megjelenítette az „_id”-t is. Ez az alapértelmezett működési mód, így ha az „_id”-re nem vagyunk kíváncsiak, akkor ezt explicit módon el kell rejtenünk (lásd 2. példa).

Ha a WHERE részben egy üres keresési feltételt adunk meg, akkor ez definíció szerint minden dokumentumra illeszkedni fog. Vagyis a 3. példát a következőképpen tudnánk megfogalmazni: listázd ki az összes dokumentumot, de csak a „name” mezőt jelenítsd meg! A 4. példában azt látjuk, hogy a „true” és „false” értékek helyett az 1 / 0 konstansok is használhatók.

Hasonlító operátorok

$gt	>
$gte	≥
$lt	<
$lte	≤

```
> db.people.find({age: {$gt: 20}, profession: "hacker"})
```

Keressük azon dokumentumokat, ahol az „age” értéke nagyobb, mint 20 ÉS az illető foglalkozása hacker. Látható, hogy a WHERE részben több kifejezés is megadható egymástól vesszővel elválasztva, s ilyenkor a kifejezések logikai ÉS művelettel lesznek összekapcsolva.

A fenti lekérdezés SQL megfelelője így nézne ki:

```
SELECT * FROM people
WHERE age > 20 AND profession = „hacker”
```

További operátorok

$exists — a dokumentum rendelkezik-e az adott mezővel

```
> db.people.find({profession: {$exists: true}})
```

Azon dokumentumokra vagyunk kíváncsiak, amelyek rendelkeznek a „profession” mezővel. (Emlékezzünk, a MongoDB séma nélküli, vagyis egy kollekcióban nyugodtan lehetnek egymástól eltérő felépítésű dokumentumok).

$regex — mintaillesztés

```
> db.people.find({name: {$regex: "^J"}})
```

Keressük azon személyeket, akiknek a neve „J”-vel kezdődik.

$regex — mintaillesztés (folyt.)

Egy reguláris kifejezést egyszerűbben is írhatunk:

```
> db.people.find({name: /h/})
```

Azaz: keressük azon személyeket, akiknek a neve tartalmazza a „h” betűt.

Mi a helyzet akkor, ha a minta tagadására van szükségünk? Pl. azon személyekre vagyunk kíváncsiak, akiknek a neve NEM tartalmazza a „h” betűt:

```
> db.people.find({name: {$not: /h/}})
```

$or — logikai VAGY, *prefix operátor*

```
> db.people.find({$or: [{name: /e$/}, {age: {$exists: true}}]})
```

A $or prefix operátor, ezért előre kerül. Az egyes feltételeket egy listában („[” és „]” jelek között) fogjuk össze. Azon dokumentumokat akarjuk kiíratni, ahol az illető neve „e”-re végződik VAGY ha létezik a dokumentumban az „age” mező.

$and

logikai ÉS, *prefix operátor*

Legyen adott a köv. három dokumentum:

```
> db.people.find()
{ "_id" : ..., "name" : "John", "age" : 30, "profession" : "hacker" }
{ "_id" : ..., "name" : "Jessie", "age" : 35, "profession" : "programmer" }
{ "_id" : ..., "name" : "Andrea", "age" : 27, "profession" : "student" }
```

Kérjük le azokat a személyeket, akiknek a neve lexikografikusan nagyobb, mint a „C”, ÉS a nevük tartalmazza az „e” betűt.

A $and operátorra tulajdonképpen ritkán van szükség, ui. a fenti lekérdezést nélküle is meg lehet fogalmazni. Ehhez elegendő csupán több WHERE feltételt felsorolni:

```
> db.people.find({name: {$gt: "C", $regex: "e"}})
{ "_id" : ..., "name" : "Jessie", "age" : 35, "profession" : "programmer" }
```

Ugyanez a $and operátorral:

```
> db.people.find({$and: [{name: {$gt: "C"}}, {name: {$regex: "e"}}]})
```

$and

logikai ÉS, *prefix operátor* (folyt.)

VIGYÁZAT! Csábító lehet, hogy az előző lekérdezés esetében a két feltételt szétbontsuk:

```
> db.people.find({name: {$gt: "C"}, name: {$regex: "e"}})
{ "_id" : ..., "name" : "Jessie", "age" : 35, "profession" : "programmer" }
{ "_id" : ..., "name" : "Andrea", "age" : 27, "profession" : "student" }
```

Mint látható, itt már két dokumentumot kaptunk eredményül, s az „Andrea” dokumentum esetében nem is teljesül, hogy lexikografikusan a „C” után következne.

Ennek az az oka, hogy a névre vonatkozó második feltétel felüldefiniálta az első feltételt! A fenti lekérdezés így azokat a dokumentumokat adja vissza, ahol a név mező tartalmazza az „e” betűt. Az első feltétel felül lett írva.

Ilyen esetekben válasszunk az előző oldalon látott megoldások közül.

Keresés listában

Legyenek adottak a köv. dokumentumok:

```
> db.students.find()
{ "_id" : ..., "name" : "Andrew", "movies" : [ "Star Wars", "Indiana Jones" ] }
{ "_id" : ..., "name" : "Brian", "movies" : [ "Star Trek", "Star Wars" ] }
```

Próbáljuk ki az alábbi lekérdezést:

```
> db.students.find({movies: "Star Wars"})
```

Ez mindkét rekordot vissza fogja adni!

A lekérdezés kiértékelése a következőképpen történik:

1) Megnézi, hogy van-e ilyen nevű sztring típusú mező.
2) Ha nincs, akkor van-e ilyen nevű lista. Ebben a példában a válasz pozitív, hiszen a „movies” mező típusa lista. Ekkor megnézi, hogy a lista tartalmazza-e a keresett elemet.

VIGYÁZAT! Csak a lista legfelső szintjét nézi végig! Ha a lista tartalmaz allistákat is, azokba nem megy bele rekurzívan!

$all

contains all ...

Egy lista tartalmazza-e az általunk felsorolt összes elemet? (A sorrend nem számít)

```
> db.students.find({movies: {$all: ["Star Wars", "Star Trek"]}})
{ "_id" : ..., "name" : "Brian", "movies" : [ "Star Trek", "Star Wars" ] }
```

Keressük azokat, akiknek a kedvenc filmjei között szerepel mind a Star Wars, mind pedig a Star Trek. Látható, hogy a sorrend lényegtelen.

$in

contains either ... or ...

Egy lista tartalmazza-e az általunk felsorolt elemek valamelyikét?

```
> db.students.find({movies: {$in: ["Star Trek", "Terminator"]}})
{ "_id" : ..., "name" : "Brian", "movies" : [ "Star Trek", "Star Wars" ] }
```

Keressük azokat, akiknek a kedvenc filmjei között szerepel a Star Trek vagy a Terminator.

Pontozott jelölés (*dot notation*)

Tegyük fel, hogy a dokumentumaink a következőképpen épülnek fel:

```
{
    "_id" : ObjectId("54d6564f17c22d7c020604a9"),
    "product" : "Google Nexus 7",
    "price" : 145,
    "reviews" : [ { "user" : "alan", "score" : 4 },
                  { "user" : "anna", "score" : 5 } ]
}
```

Keressük azokat a termékeket, amelyek 100 dollárnál drágábbak, s van 5-ös vagy jobb értékelésük:

```
> db.products.find({price: {$gt: 100}, "reviews.score": {$gte: 5}})
```

A „reviews” mező lista típusú, de az elemei objektumok. Veszi a lista egyes elemeit, s megnézi, hogy az objektum „score” mezője ≥ 5. Ha igen, akkor teljesül a feltétel.

sort, skip, limit

Legyenek adottak a következő dokumentumok:

```
> db.names.find()
{ "_id" : ..., "name" : "E" }
{ "_id" : ..., "name" : "D" }
{ "_id" : ..., "name" : "A" }
{ "_id" : ..., "name" : "B" }
{ "_id" : ..., "name" : "C" }
```

Írassuk ki a neveket növekvő, ill. csökkenő sorrendben:

```
> db.names.find().sort({name: 1})    # kimeneti sorrend: A, B, C, D, E
> db.names.find().sort({name: -1})   # kimeneti sorrend: E, D, C, B, A
```

Megadjuk, hogy melyik mező szerint szeretnénk rendezni a dokumentumokat, ill. rögzítjük a rendezés irányát (1: növekvő sorrend, -1: csökkenő sorrend).

```
> db.names.find().sort({name: 1}).limit(2)  # kimeneti sorrend: A, B
```

Rendezzük az elemeket, de az eredményt csak az első két dokumentumra limitáljuk.

sort, skip, limit (folyt.)

```
> db.names.find().sort({name: 1}).skip(2).limit(2)  # kimenet: C, D
```

Rendezzük az elemeket (A, B, C, D, E), ugorjuk át az első kettőt (marad: C, D, E), majd írassuk ki az első kettőt: C, D.

A sort, skip és limit hívások felírási sorrendjétől függetlenül a rendszer ebben a sorrendben fogja őket végrehajtani: 1) sort, 2) skip, 3) limit. Vagyis egy `db.names.find().limit(2).skip(2).sort({name: 1})` hívás kimenete is C, D lesz!

count

Állapítsuk meg egy kollekcióban lévő dokumentumok számát. Ezt kétféleképpen is megtehetjük:

```
> db.names.find().count()
5
> db.names.count()
5
```

Update

Dokumentumok módosítására szolgál. Példa:

```
> db.people.find({name: "Andrea"})
{ "_id" : ..., "name" : "Andrea", "age" : 27, "profession" : "student" }
> db.people.update({name: "Andrea"}, {name: "Sarah", age: 30})
```

WHERE rész
melyik dokumentumot akarjuk módosítani

új dokumentum
a talált dokumentumot ezzel *felülírjuk*

```
> db.people.find({name: "Sarah"})
{ "_id" : ..., "name" : "Sarah", "age" : 30 }
```

VIGYÁZAT! Mint látható, a dokumentumból eltűnt a „profession” mező! Ennek az az oka, hogy az update esetében a talált dokumentumot egy új dokumentumra cseréltük le, amelyben csak a nevet és az életkort állítottuk be.
Az „_id” mezőt ilyenkor változatlanul hagyja.

$set

mező értékének beállítása / új mező hozzáadása

```
> db.people.find({name: "Sarah"})
{ "_id" : ..., "name" : "Sarah", "age" : 30 }
> db.people.update({name: "Sarah"}, {$set: {age: 33}})
> db.people.update({name: "Sarah"}, {$set: {profession: "protector"}})
> db.people.find({name: "Sarah"})
{ "_id" : ..., "name" : "Sarah", "age" : 33, "profession" : "protector" }
```

Először beállítjuk az „age” mező értékét 33-ra. Mivel már volt ilyen mező, ezért csak frissül a mező értéke. Ezután beállítjuk a „profession” mező értékét. Mivel nem volt ilyen mező, ezért beszúrja ezt az új mezőt. Mint látható, itt már nem adjuk meg a teljes dokumentumot.

$unset

mező törlése

```
> db.people.find({name: "Sarah"})
{ "_id" : ..., "name" : "Sarah", "age" : 33, "profession" : "protector" }
> db.people.update({name: "Sarah"}, {$unset: {age: 1}})
> db.people.find({name: "Sarah"})
{ "_id" : ..., "name" : "Sarah", "profession" : "protector" }
```

Az „age” mezőt eltávolítjuk a dokumentumból. Az „age” után itt 1-es értéket adtunk meg, de tetszőleges érték beírható, figyelmen kívül hagyja.

$inc

mező értékének növelése / csökkentése

```
> db.people.find({name: "Sarah"})
{ "_id" : ..., "name" : "Sarah", "profession" : "protector", "age" : 33 }
> db.people.update({name: "Sarah"}, {$inc: {age: 2}})
> db.people.find({name: "Sarah"})
{ "_id" : ..., "name" : "Sarah", "profession" : "protector", "age" : 35 }
```

Az „age” mező értékét megnöveljük 2-vel.

Ha nem létezett a mező, akkor létrehozza 0 kezdőértékkel, s azt növeli meg.

```
> db.people.find({name: "David"})
{ "_id" : ..., "name" : "David" }
> db.people.update({name: "David"}, {$inc: {awards: 1}})
> db.people.find({name: "David"})
{ "_id" : ..., "name" : "David", "awards" : 1 }
```

Egy mező értékének a csökkentésére nincs külön operátor. Megoldás: használjuk a $inc operátort negatív értékkel:

```
> db.people.find({name: "Sarah"})
{ "_id" : ..., "name" : "Sarah", "profession" : "protector", "age" : 35 }
> db.people.update({name: "Sarah"}, {$inc: {age: -5}})
> db.people.find({name: "Sarah"})
{ "_id" : ..., "name" : "Sarah", "profession" : "protector", "age" : 30 }
```

multi-update

Mi a helyzet akkor, ha az update feltétele több dokumentumra is illeszkedik? Az alapértelmezés szerint az update csak az első dokumentumot fogja frissíteni, vagyis azt, amelyiket a keresés először visszaadja.

Ha több dokumentumot is szeretnénk módosítani, akkor ezt explicit módon jelezni kell.

```
> db.people.find()
{ "_id" : ..., "name" : "John", "age" : 30, "profession" : "hacker" }
{ "_id" : ..., "name" : "Jessie", "age" : 35, "profession" : "programmer" }
> db.people.update({}, {$set: {age: 28}})
> db.people.find()
{ "_id" : ..., "name" : "John", "age" : 28, "profession" : "hacker" }
{ "_id" : ..., "name" : "Jessie", "age" : 35, "profession" : "programmer" }

> db.people.update({}, {$set: {age: 28}}, {multi:true})
> db.people.find()
{ "_id" : ..., "name" : "John", "age" : 28, "profession" : "hacker" }
{ "_id" : ..., "name" : "Jessie", "age" : 28, "profession" : "programmer" }
```

Az üres keresési feltétel (`{}`) valamennyi dokumentumra illeszkedik.
Az első update során csak az egyik dokumentum módosult. A második esetben, amikor megadtuk a `{multi: true}` kapcsolót, már az összes.

upsert

Az upsert művelet az update/insert műveleteket takarja. Ha a keresett dokumentum nem létezik, akkor insert-ként, ha létezik, akkor pedig update-ként funkcionál.

```
> db.people.find()
{ "_id" : ..., "name" : "John", "age" : 28, "profession" : "hacker" }

> db.people.update({name: "George"}, {$set: {age: 22}}, {upsert: true})
> db.people.find()
{ "_id" : ..., "name" : "John", "age" : 28, "profession" : "hacker" }
{ "_id" : ..., "name" : "George", "age" : 22 }

> db.people.update({name: "George"}, {$set: {age: 20}}, {upsert: true})
> db.people.find()
{ "_id" : ..., "name" : "John", "age" : 28, "profession" : "hacker" }
{ "_id" : ..., "name" : "George", "age" : 20 }
```

Az upsert műveletet egy külön kapcsolóval (`{upsert: true}`) tudjuk jelezni.
Az első upsert során még nem létezett a George nevű személy, ezért egy insert művelet hajtódott végre.
A második esetben már létezett egy ilyen nevű illető, így a hozzá tartozó dokumentum csupán frissült (hagyományos update művelet).

műveletek tömbökkel

(A dokumentumokban szereplő tömbök dinamikus tömbök, így a „tömb” és a „lista” elnevezéseket azonos értelemben fogjuk használni.)

Hozzuk létre az alábbi dokumentumot:

```
> db.arrays.insert({_id: 0, a: [1, 2, 3, 4, 5]})
```

Az alábbiakban bemutatunk néhány lista-műveletet. A műveleteket inkrementálisan hajtjuk végre, vagyis a bemenet mindig az előző lépés kimenete lesz.

lista adott pozíciójú elemének módosítása

```
> db.arrays.update({_id: 0}, {$set: {"a.2": 9}})
> db.arrays.find()
{ "_id" : 0, "a" : [ 1, 2, 9, 4, 5 ] }
```

A lista indexelése 0-tól indul. A 2-es indexű elem értékét 9-re módosítjuk.

$push — lista bővítése jobbról

```
> db.arrays.update({_id: 0}, {$push: {a: 6}})
> db.arrays.find()
{ "_id" : 0, "a" : [ 1, 2, 9, 4, 5, 6 ] }
```

$pop

jobb szélső / bal szélső elem eltávolítása

Jobb szélső elem törlése:

```
> db.arrays.update({_id: 0}, {$pop: {a: 1}})
> db.arrays.find()
{ "_id" : 0, "a" : [ 1, 2, 9, 4, 5 ] }
```

Bal szélső elem törlése:

```
> db.arrays.update({_id: 0}, {$pop: {a: -1}})
> db.arrays.find()
{ "_id" : 0, "a" : [ 2, 9, 4, 5 ] }
```

$pushAll

bővítés jobbról egy lista elemeivel

```
> db.arrays.update({_id: 0}, {$pushAll: {a: [10, 20, 30]}})
> db.arrays.find()
{ "_id" : 0, "a" : [ 2, 9, 4, 5, 10, 20, 30 ] }
```

$pull

adott elem törlése (érték szerint megadva)

Töröljük pl. a 20-as elemet (bárhol is van):

```
> db.arrays.update({_id: 0}, {$pull: {a: 20}})
> db.arrays.find()
{ "_id" : 0, "a" : [ 2, 9, 4, 5, 10, 30 ] }
```

$pullAll

felsorolt elemek törlése

```
> db.arrays.update({_id: 0}, {$pullAll: {a: [10, 5, 9]}})
> db.arrays.find()
{ "_id" : 0, "a" : [ 2, 4, 30 ] }
```

$addToSet

halmaz bővítése

```
> db.arrays.update({_id: 0}, {$addToSet: {a: 4}})
> db.arrays.find()
{ "_id" : 0, "a" : [ 2, 4, 30 ] }
> db.arrays.update({_id: 0}, {$addToSet: {a: 35}})
> db.arrays.find()
{ "_id" : 0, "a" : [ 2, 4, 30, 35 ] }
```

Halmazként kezeli a listát. Ha az adott elem már benne van (itt a 4-es), akkor nem teszi bele. Ha az elem nincs benne (lásd 35-ös elem), akkor beleteszi. A duplikátumok így egyszerűen elkerülhetők.

remove

Egy kollekcióból a következőképpen tudunk törölni:

```
> db.people.remove( feltétel )
```

A feltételt ugyanúgy adjuk meg, mint a find() esetében. Ennek hatására törlődnek azok a dokumentumok, amelyek megfelelnek a feltételnek.

Hogyan tudjuk kitörölni egy kollekció összes dokumentumát?

1) `db.collection.remove({})`
Az üres feltétel minden dokumentumra illeszkedik.

2) `db.collection.drop()`
Ez törli az egész kollekciót.

Az első esetben a dokumentumokat egyenként törli, de meghagyja az indexeket.

A második esetben az egész kollekciót töröljük, így ez gyorsabb lesz, mint ha az elemeket egyenként törölnénk. Ez a második változat az indexeket is törli (míg az első nem).

Néhány példa

Növeljük meg a pontszámát 20-szal minden olyan hallgatónak, aki 70-nél kevesebb ponttal rendelkezik.

Először nézzük meg az érintett hallgatókat:

```
> db.students.find({score: {$lt: 70}})
{ "_id" : ..., "name" : "Howard", "type" : "exam", "score" : 65 }
{ "_id" : ..., "name" : "Patrick", "type" : "exam", "score" : 55 }
```

Majd növeljük meg a pontszámukat. Vegyük figyelembe, hogy egynél több dokumentumot kell módosítani!

```
> db.students.update({score: {$lt: 70}}, {$inc: {score: 20}}, {multi: true})
```

A végén ellenőrizzük le a módosításokat:

```
> db.students.find()
{ "_id" : ..., "name" : "George", "type" : "exam", "score" : 75 }
{ "_id" : ..., "name" : "Howard", "type" : "exam", "score" : 85 }
{ "_id" : ..., "name" : "Patrick", "type" : "exam", "score" : 75 }
```

Töröljük a George nevű hallgatót.

```
> db.students.remove({name: "George"})
WriteResult({ "nRemoved" : 1 })
```

CRUD műveletek Python-ban

Az előzőekben megnéztük a CRUD műveleteket a mongo shell-ben. Most nézzük át, hogy egy Python alkalmazásból hogyan lehet meghívni ezeket a műveleteket.

Python-hoz a „pymongo” csomag a hivatalos meghajtó (driver) . A régebbi verziókban a Connection osztály használatát javasolták, de ez ma már elavult (deprecated). Az új verziók a MongoClient osztály használatát javasolják.
A régi változat esetén íráskor a szervertől nem kaptunk visszaigazolást, minden aszinkron módon ment. Az új alapértelmezés szerint íráskor megvárjuk, míg a szerver visszaigazol. A MongoClient osztály már így működik.

```
 4  import pymongo
 5
 6  conn = pymongo.MongoClient()
 7  db = conn['test']
 8  people = db['people']
 9
10  def find_one():
11      query = {'name': 'George'}
12      doc = people.find_one(query)
13      print doc
```

A find_one() függvény egy konkrét dokumentumot ad vissza. Ennek Python-beli típusa: szótár.

find

Tekintsük az alábbi kollekciót:

```
> db.students.find()
{ "_id" : ..., "name" : "Howard", "type" : "exam", "score" : 85 }
{ "_id" : ..., "name" : "Patrick", "type" : "exam", "score" : 75 }
```

Python-ban a find() egy kurzort ad vissza. Ha ezen a kurzoron egy *for* ciklussal végigmegyünk, akkor megkapjuk az egyes dokumentumokat:

```
 4  import pymongo
 5
 6  conn = pymongo.MongoClient()
 7  db = conn['test']
 8  students = db['students']
 9
10  def find():
11      query = {'type': 'exam'}
12      cursor = students.find(query)
13      for doc in cursor:
14          print doc
```

Ez a kód a fenti két dokumentumot fogja kiírni.

projekció

A projekció során az eredményként kapott dokumentumoknak csak bizonyos mezőit szeretnénk megtartani.

```
 4  import pymongo
 5
 6  conn = pymongo.MongoClient()
 7  db = conn['test']
 8  students = db['students']
 9
10  def find():
11      query = {'type': 'exam'}
12      projection = {'name': 1, '_id': 0}
13
14      cursor = students.find(query, projection)
15      for doc in cursor:
16          print doc
```

A fenti kód kimenete (az előző oldalon látható kollekción futtatva):

```
{u'name': u'Howard'}
{u'name': u'Patrick'}
```

(Python-ban a sztringek előtti u'...' prefix Unicode sztringet jelöl).

$gt $lt

```
> db.students.find()
{ "_id" : ..., "name" : "Patrick", "type" : "exam", "score" : 75 }
{ "_id" : ..., "name" : "Jones", "type" : "exam", "score" : 80 }
{ "_id" : ..., "name" : "Howard", "type" : "exam", "score" : 85 }
{ "_id" : ..., "name" : "Adam", "type" : "exam", "score" : 90 }
```

Kik azok, akiknek a vizsgajegyük pontszáma nagyobb, mint 75, de kisebb, mint 90?

```
10  def find():
11      query = {'type': 'exam', 'score': {'$gt': 75, '$lt': 90}}
12
13      cursor = students.find(query)
14      for doc in cursor:
15          print doc
```

A fenti lekérdezés a Jones és Howard nevű diákok dokumentumait adja vissza.

sort, skip, limit

A rendezés szintaxisa egy kicsit eltér a mongo shell-ben megszokottól:

```
find(...).sort({title: 1})     # mongo shell

find(...).sort('title', 1)     # Python
```

A *sort* második paraméterével tudjuk beállítani a rendezési sorrendet. Az 1 növekvő, míg a -1 csökkenő sorrendet jelent. Python-ban ezek helyett alkalmazhatók még a `pymongo.ASCENDING` és `pymongo.DESCENDING` konstansok is.

Vegyük az alábbi példát, s futtassuk le az előző oldalon látható kollekción:

```
10  def find():
11      cursor = students.find().sort('name', 1).skip(2).limit(1)
12      for doc in cursor:
13          print doc
```

A négy hallgató név szerint sorba rendezve: Adam, Howard, Jones, Patrick. Ha az első kettőt átugorjuk, s a maradékból vesszük az elsőt, akkor a fenti kódrészlet a Jones-hoz tartozó dokumentumot fogja kiírni.

sort, skip, limit (folyt.)

A sort, skip és limit függvényeket tetszőleges sorrendben hívhatjuk meg, viszont a MongoDB ezeket mindig a következő sorrendben fogja végrehajtani: 1) sort, 2) skip, 3) limit. Az adatbázisból az elemek lekérdezése majd csak akkor történik meg, amikor a kurzor objektumot elkezdjük iterálni.

Mi a helyzet akkor, ha több mező szerint akarjuk rendezni a dokumentumokat? Előbb az egyik mező szerint, majd azon belül egy másik mező értéke alapján. A Python szintaxis itt is eltér egy kicsit a mongo shellben megszokottól:

```
find(...).sort({student_id: 1, score: -1})            # mongo shell

find(...).sort([('student_id', 1), ('score', -1)])    # Python
```

Python esetében ilyenkor tuple-ök listáját kell megadni. Miért ez a szintaktikai eltérés? JSON-ban a kulcs/érték pároknak van egy sorrendje, viszont egy Python szótárban az elemeknek nincs kötött sorrendje. Ha több mező alapján szeretnénk rendezni, s Python szótárat használnánk, akkor az nem definiálná egyértelműen a mezők sorrendjét.

save

A *save* segítségével egy Python szótárat tudunk dokumentumként elmenteni az adatbázisba:

```
 6  conn = pymongo.MongoClient()
 7  db = conn['test']
 8  students = db['students']
 9
10  def insert():
11      doc = {'name': 'Joe'}
12      print doc
13      students.save(doc)
14      print doc
```

A program kimenete:

```
{'name': 'Joe'}
{'_id': ObjectId('54ee30eba541471870e93f9c'), 'name': 'Joe'}
```

Amikor egy szótárat beszúrunk az adatbázisba, akkor a driver megnézi, hogy van-e „_id” kulcsa. Ha nincs, akkor módosítja a szótárat és beletesz egy „_id”-t, ami egy ObjectId objektum lesz. A második print utasítás kimenete ezért tér el az első kimenettől.

save (folyt.)

A *save* -vel nem csak beszúrni lehet, hanem egy adatbázisban lévő dokumentumot is tudunk vele módosítani:

```
10  def save():
11      doc = students.find_one({'name': 'Joe'})
12      print doc
13      doc['age'] = 35
14      students.save(doc)
15
16      doc = students.find_one({'name': 'Joe'})
17      print doc
```

A program kimenete:

```
{u'_id': ObjectId('54ee30eba541471870e93f9c'), u'name': u'Joe'}
{u'age': 35, u'_id': ObjectId('54ee30eba541471870e93f9c'), u'name': u'Joe'}
```

A `find_one()` egy konkrét dokumentumot ad vissza, amit a pymongo driver Python szótárrá alakít át. Ezt a szótárat kibővítjük egy új mezővel, majd a `save` függvénnyel elmentjük az adatbázisba. Mivel a `doc` objektum már rendelkezett egy „_id” mezővel, azért mentéskor az ezzel az „_id”-vel rendelkező dokumentumot frissíti. A `save` ilyenkor az adatbázisban lévő elemet felülírja.

update

$set — mező értékének beállítása / új mező hozzáadása

```
10  def update():
11      doc = students.find_one({'name': 'Joe'})
12      print doc
13      students.update({'name': 'Joe'}, {'$set': {'age': 33}})
14      doc = students.find_one({'name': 'Joe'})
15      print doc
```

A program kimenete:

```
{u'age': 35, u'_id': ObjectId('54ee30eba541471870e93f9c'), u'name': u'Joe'}
{u'age': 33, u'_id': ObjectId('54ee30eba541471870e93f9c'), u'name': u'Joe'}
```

Itt egy mező értékét módosítjuk. Ez a megoldás hatékonyabb, mint az előző *save* függvény, mivel itt a BSON elemnek csak egy részét módítjuk, míg a *save* a teljes dokumentumot fölülírta.

(BSON – bináris JSON formátum. A MongoDB ilyen formában tárolja a kollekciók dokumentumait.)

upsert

Az *upsert* művelet, mint már korábban is láttuk, az *update* és *insert* műveleteket vonja össze. Ha a WHERE feltételnek a megfelelője nem teljesül (vagyis nincs ilyen dokumentum), akkor egy *insert*, ellenkező esetben pedig egy *update* művelet fog végrehajtódni.

```
 6  conn = pymongo.MongoClient()
 7  db = conn['test']
 8  people = db['people']
 9
10  def upsert():
11      people.drop()
12      print people.count()
13      people.update({"name": "George"}, {"$set": {"age": 22}}, upsert=True)
14      print people.find_one({"name": "George"})
15      people.update({"name": "Joe"}, {"age": 30}, upsert=True)
16      print people.find_one({"age": 30})
```

A program kimenete:

```
0
{u'age': 22, u'_id': ObjectId(...), u'name': u'George'}
{u'age': 30, u'_id': ObjectId(...)}
```

upsert (folyt.)

Az előző oldalon látható kimenet magyarázata:

Első lépésben a kollekciót töröljük, így a benne lévő dokumentumok száma 0.

Ezután keresünk egy George nevű illetőt. Mivel nem létezik ilyen dokumentum, ezért az upsert egy insert műveletnek fog megfelelni. A létrejövő dokumentumnak van *name* és *age* mezője, s természetesen *_id* mezője is.

Vegyük észre, hogy az upsert művelet beállítására Python-ban más szintaxist kell alkalmazni!

A második esetben egy Joe nevű illetőt keresünk. Ilyen dokumentumunk sincs, ezért ismét egy insert művelet hajtódik végre. Viszont az eredmény most csak két mezővel fog rendelkezni: *_id* és *age*. Ennek az az oka, hogy a WHERE rész után megadott `{„age”: 30}` lesz a dokumentum tartalma. Az első esetben a `$set` –tel hozzáadtunk egy új mezőt, itt viszont a dokumentum tartalmát a megadott értékre állítjuk be, ami csupán egyetlen mezőt (*age*) tartalmaz. A pymongo driver az *_id* mezőt automatikusan teszi be minden új dokumentumba.

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 önéletrajzot 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 önéletrajzot 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.

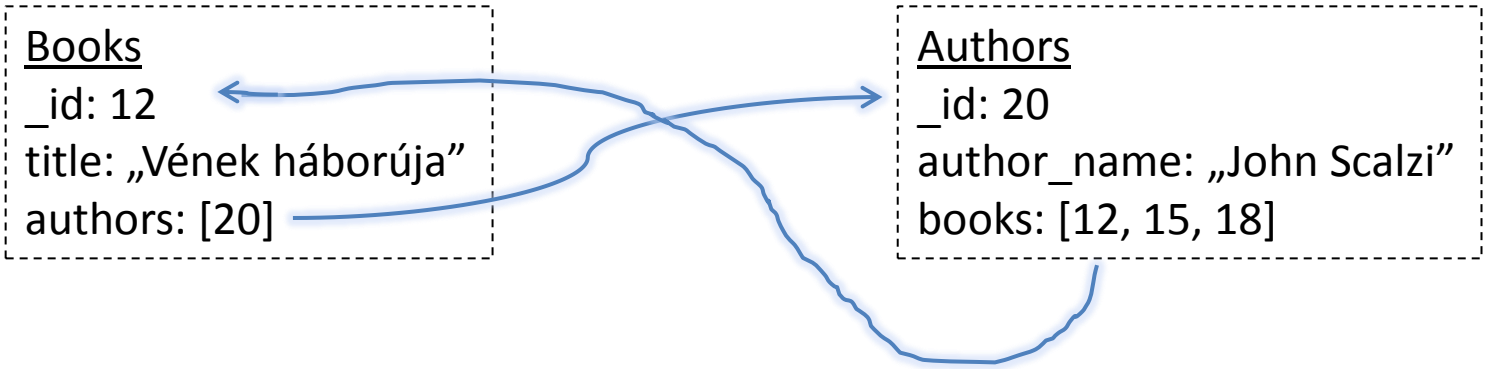
Viszont mi a helyzet akkor, ha nem *One to Many*, hanem csak *One to Few* kapcsolatról van szó, vagyis amikor egy elemhez csak néhány elem tartozik? Gondoljunk vissza a blogposztos példára: egy poszthoz általában nem szokott túl sok komment tartozni. Ebben az esetben a kommentek beágyazhatók a blogposztba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

Összefoglalva:

One to Many:	két kollekció használata lesz talán a legjobb megoldás
One to Few:	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 lekérdezünk.

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 dokuemtum 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 percig. 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 laptopon 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 kollekcióról 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. RockMongo).
• 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).
• Hivatalos (és ingyenes) kurzusok: https://university.mongodb.com/
• Kik használják a MongoDB-t: https://www.mongodb.com/who-uses-mongodb