Justin Huang

DB-Engines Ranking

356 systems in ranking, June 2020

Rank			DBMS	Database Model	Score		
Jun 2020	May 2020	Jun 2019			Jun 2020	May 2020	Jun 2019
1.	1.	1.	Oracle	Relational, Multi-model	1343.59	-1.85	+44.37
2.	2.	2.	MySQL	Relational, Multi-model	1277.89	-4.75	+54.26
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1067.31	-10.99	-20.45
4.	4.	4.	PostgreSQL	Relational, Multi-model	522.99	+8.19	+46.36
5.	5.	5.	MongoDB	Document, Multi-model	437.08	-1.92	+33.17
6.	6.	6.	IBM Db2	Relational, Multi-model	161.81	-0.83	-10.39
7.	7.	7.	Elasticsearch	Search engine, Multi-model	149.69	+0.56	+0.86
8.	8.	8.	Redis	Key-value, Multi-model	145.64	+2.17	-0.48
9.	9.	↑ 11.	SQLite	Relational	124.82	+1.78	-0.07
10.	↑ 11.	10.	Cassandra	Wide column	119.01	-0.15	-6.17
11.	↓ 10.	↓ 9.	Microsoft Access	Relational	117.18	-2.72	-23.83

DB-Engines Ranking of Document Store

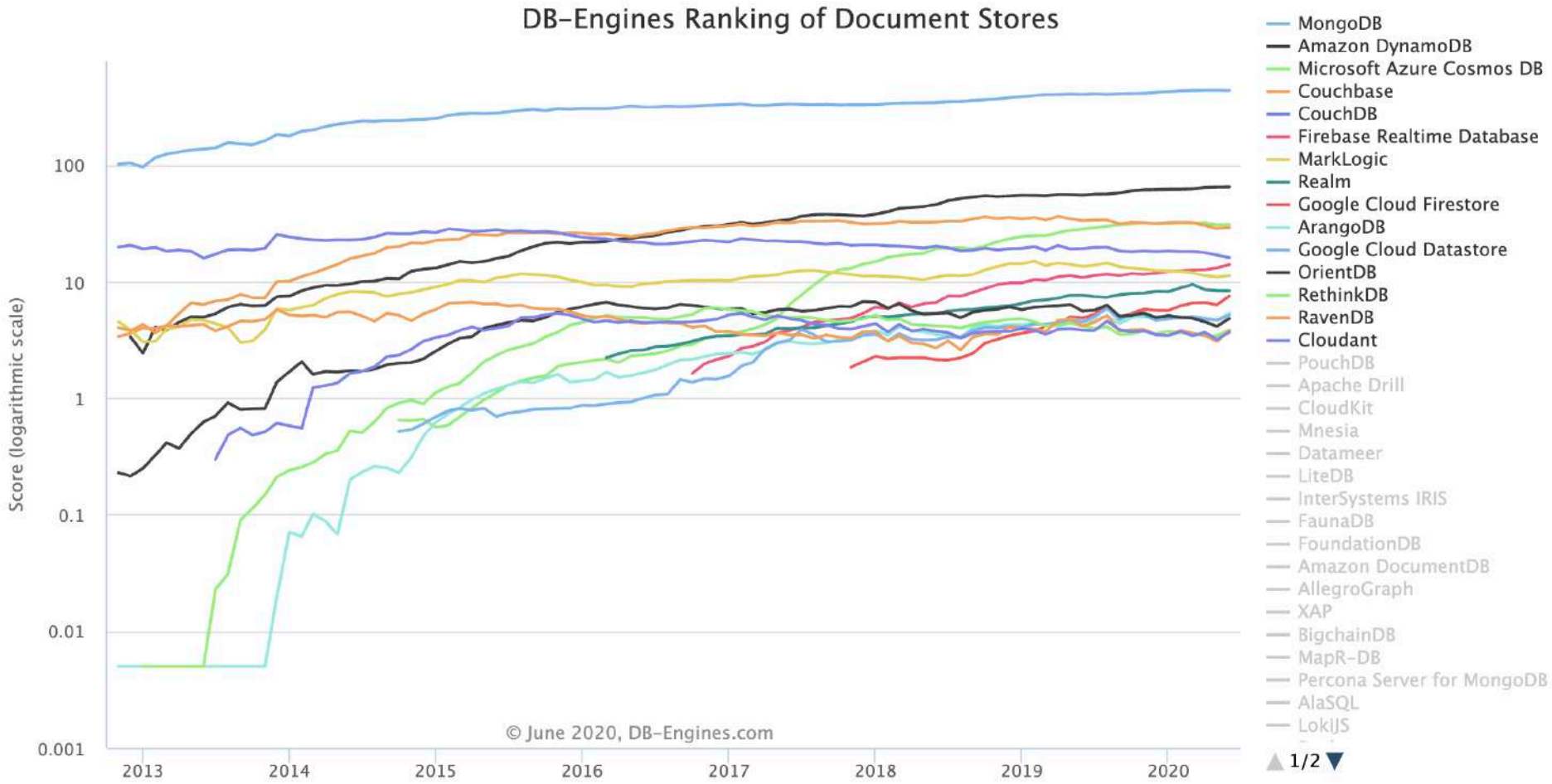
☐ include secondary database models

47 systems in ranking, June 2020

Rank			DBMS	Database Model	Score		
Jun 2020	May 2020	Jun 2019			Jun 2020	May 2020	Jun 2019
1.	1.	1.	MongoDB	Document, Multi-model	437.08	-1.92	+33.17
2.	2.	2.	Amazon DynamoDB	Multi-model	64.87	+0.15	+9.61
3.	3.	↑ 4.	Microsoft Azure Cosmos DB	Multi-model	30.80	+0.13	+2.56
4.	4.	↓ 3.	Couchbase	Document, Multi-model	29.14	+0.56	-4.22
5.	5.	5.	CouchDB	Document	16.07	-0.85	-3.15
6.	6.	↑ 7.	Firebase Realtime Database	Document	13.99	+0.87	+3.14
7.	7.	↓ 6.	MarkLogic	Multi-model	11.24	+0.28	-2.26
8.	8.	8.	Realm	Document	8.36	-0.02	+0.69
9.	9.	↑ 10.	Google Cloud Firestore	Document	7.52	+1.18	+2.64
10.	10.	↑ 11.	ArangoDB	Multi-model	5.38	+0.70	+0.81
11.	11.	↑ 12.	Google Cloud Datastore	Document	5.15	+0.49	+0.65
12.	12.	↓ 9.	OrientDB	Multi-model	4.82	+0.68	-0.77
13.	13.	↑ 14.	RethinkDB	Document	3.86	+0.41	-0.30
14.	↑ 15.	↓ 13.	RavenDB	Document, Multi-model	3.82	+0.74	-0.36
15.	↓ 14.	15.	Cloudant	Document	3.68	+0.49	-0.16
16.	16.	16.	PouchDB	Document	3.14	+0.29	+0.20
17.	17.	17.	Apache Drill	Multi-model	2.87	+0.38	+0.07
18.	18.	18.	CloudKit	Document	2.59	+0.52	+0.16
19.	↑ 21.	19.	Mnesia	Document	1.50	+0.37	+0.18
20.	↓ 19.	20.	Datameer	Document	1.47	-0.01	+0.24

DB-Engines Ranking of Document Store

☐ include secondary database models

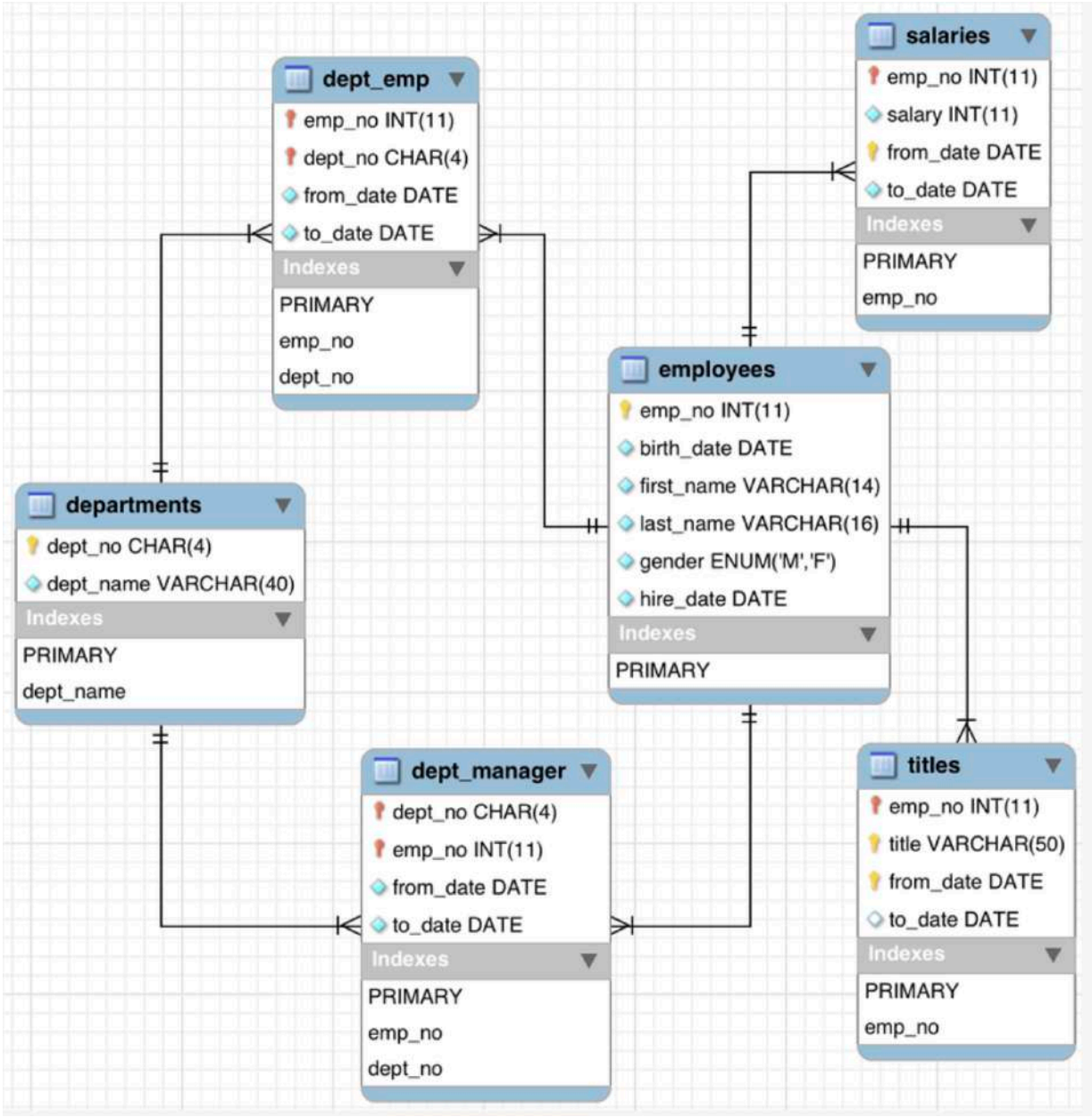


RDB

PubID	Publisher	PubAddress
03-4472822	Random House	123 4th Stree, New York
04-7733903	Wiley and Sons	45 Lincoln Blvd, Chicago
03-4859223	O'Reilly Press	77 Boston Ave, Cambridge
03-3920886	City Lights Books	99 Market, San Francisco

AuthorID	AuthorName	AuthorBDay
345-28-2938	Haile Selassie	14-Aug-92
392-48-9965	Joe Blow	14-Mar-15
454-22-4012	Sally Hemmings	12-Sep-70
663-59-1254	Hannah Arendt	12-Mar-06

ISBN	AuthorID	PubID	Date	Title
1-34532-482-1	345-28-2938	03-4472822	1990	Cold Fusion for Dummies
1-38482-995-1	392-48-9965	04-7733903	1985	Macrame and Straw Tying
2-35921-499-4	454-22-4012	03-4859223	1852	Fluid Dynamics of Aquaducts
1-38278-293-4	663-59-1254	03-3920886	1967	Beads, Baskets & Revolution


dept_emp
emp_no INT(11)
dept_no CHAR(4)
from_date DATE
to_date DATE
Indexes
PRIMARY
emp_no
dept_no
salaries
emp_no INT(11)
salary INT(11)
from_date DATE
to_date DATE
Indexes
PRIMARY
emp_no
employees
emp_no INT(11)
birth_date DATE
first_name VARCHAR(14)
last_name VARCHAR(16)
gender ENUM('M','F')
hire_date DATE
Indexes
PRIMARY
departments
dept_no CHAR(4)
dept_name VARCHAR(40)
Indexes
PRIMARY
dept_name
dept_manager
dept_no CHAR(4)
emp_no INT(11)
from_date DATE
to_date DATE
Indexes
PRIMARY
emp_no
dept_no
titles
emp_no INT(11)
title VARCHAR(50)
from_date DATE
to_date DATE
Indexes
PRIMARY
emp_no

Semi-structured data (document)

```
{
    FirstName: "Bob",
    Address: "5 Oak St.",
    Hobby: "sailing"
}
```

A second document might be:

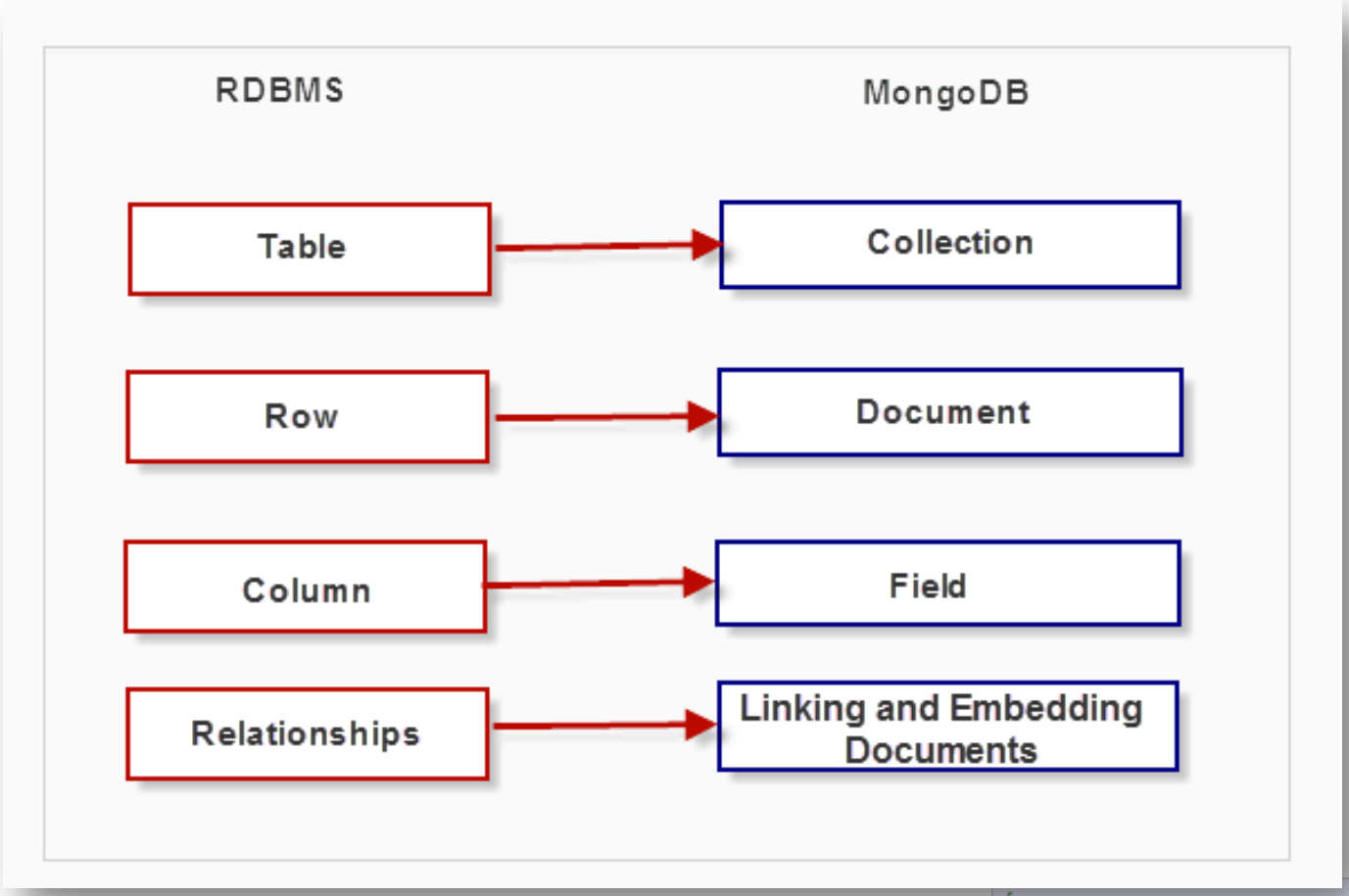
```
{
    FirstName: "Jonathan",
    Address: "15 Wanamassa Point Road",
    Children: [
        {Name: "Michael", Age: 10},
        {Name: "Jennifer", Age: 8},
        {Name: "Samantha", Age: 5},
        {Name: "Elena", Age: 2}
    ]
}
```

Introduction

- MongoDB is a ***document-oriented*** database
- Row in MySQL = Document in MongoDB (Record)
- **No join**.
- Developers in modern object-oriented languages think about their data.
- No predefined schemas
- a document's keys and values are not of fixed types or sizes.
- **Easy to Scale Out!**

JSON Example (Document)

```
{
    "id": 2,
    "name": "An ice sculpture",
    "price": 12.50,
    "tags": ["cold", "ice"],
    "dimensions": {
        "length": 7.0,
        "width": 12.0,
        "height": 9.5
    },
    "warehouseLocation": {
        "latitude": -78.75,
        "longitude": 20.4
    }
},
{
    "id": 3,
    "name": "A blue mouse",
    "price": 25.50,
    "dimensions": {
        "length": 3.1,
        "width": 1.0,
        "height": 1.0
    },
    "warehouseLocation": {
        "latitude": 54.4,
        "longitude": -32.7
    }
}
```



id	user_name	email	age	city
1	Mark Hanks	mark@abc.com	25	Los Angeles
2	Richard Peter	richard@abc.com	31	Dallas


```
{
    "_id": ObjectId("5146bb52d8524270060001f3"),
    "age": 25,
    "city": "Los Angeles",
    "email": "mark@abc.com",
    "user_name": "Mark Hanks"
}
{
    "_id": ObjectId("5146bb52d8524270060001f2"),
    "age": 31,
    "city": "Dallas",
    "email": "richard@abc.com",
    "user_name": "Richard Peter"
}
```

Document (JSON)

- *Document*: an **ordered set** of keys with associated values.
- Key/value pairs in documents are ordered: {"x" : 1, "y" : 2} is not the same as {"y" : 2, "x" : 1}.
- Cannot contain duplicate keys.

{"greeting" : "Hello, world!", "greeting" : "Hello, MongoDB!"}

Collection

- A *collection* is a group of documents.
- Collections have *dynamic schemas*.

  {"greeting" : "Hello, world!"}

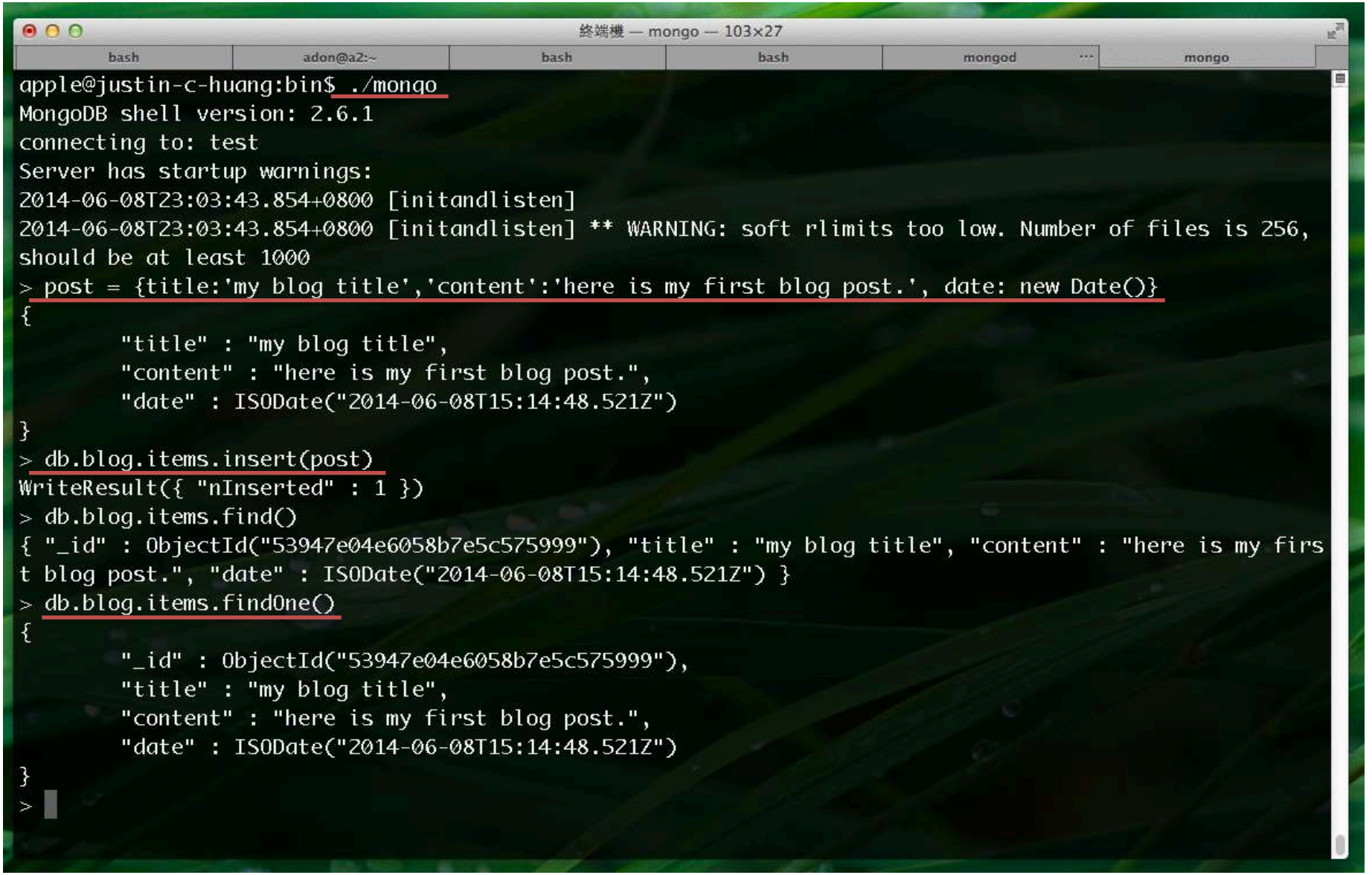
  {"foo" : 5}
- Naming: blog, blog.posts and blog.authors

Starting MongoDB

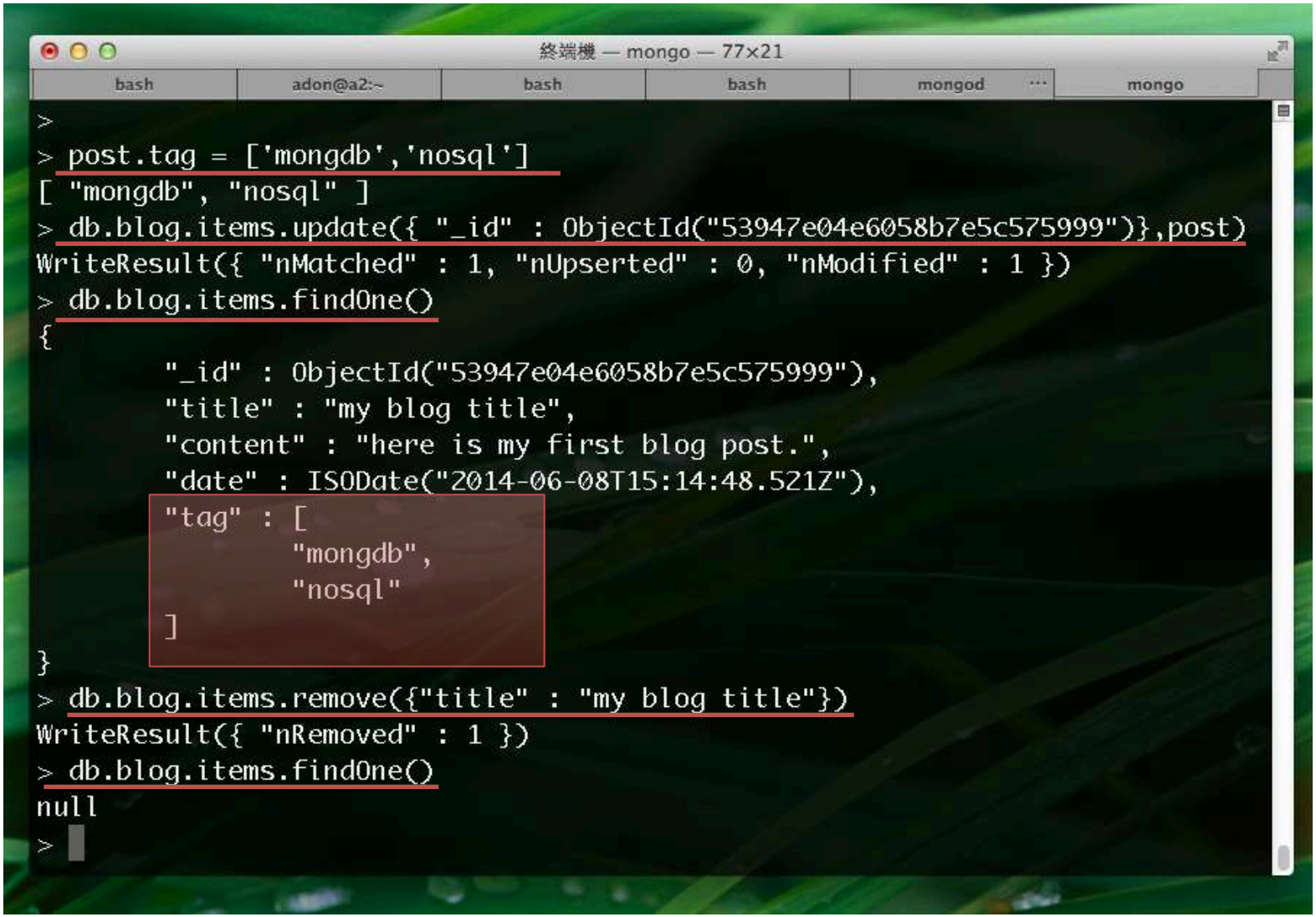
```
apple@justin-c-huang:bin$ pwd
/Users/apple/Desktop/mongodb-osx-x86_64-2.6.1/bin
apple@justin-c-huang:bin$ ls
bsondump    mongod      mongoexport  mongoimport  mongoperf    mongos      mongostat
mongo       mongodump   mongofiles   mongooplog   mongorestore mongosniff  mongotop
apple@justin-c-huang:bin$ ./mongod --dbpath /Volumes/newSpace/iii_mongodb_demo/iii_db
2014-06-08T23:03:43.854+0800 [initandlisten] MongoDB starting : pid=13257 port=27017 dbpath=/Volumes/newSpace/iii_
mongodb_demo/iii_db 64-bit host=justin-c-huang.local
2014-06-08T23:03:43.854+0800 [initandlisten]
2014-06-08T23:03:43.854+0800 [initandlisten] ** WARNING: soft rlimits too low. Number of files is 256, should be c
t least 1000
2014-06-08T23:03:43.854+0800 [initandlisten] db version v2.6.1
2014-06-08T23:03:43.854+0800 [initandlisten] git version: 4b95b086d2374bdcfcdf2249272fb552c9c726e8
2014-06-08T23:03:43.854+0800 [initandlisten] build info: Darwin mci-osx108-6.build.10gen.cc 12.5.0 Darwin Kernel V
ersion 12.5.0: Sun Sep 29 13:33:47 PDT 2013; root:xnu-2050.48.12~1/RELEASE_X86_64 x86_64 BOOST_LIB_VERSION=1_49
2014-06-08T23:03:43.854+0800 [initandlisten] allocator: system
2014-06-08T23:03:43.854+0800 [initandlisten] options: { storage: { dbPath: "/Volumes/newSpace/iii_mongodb_demo/iii
_db" } }
2014-06-08T23:03:43.855+0800 [initandlisten] journal dir=/Volumes/newSpace/iii_mongodb_demo/iii_db/journal
2014-06-08T23:03:43.855+0800 [initandlisten] recover : no journal files present, no recovery needed
2014-06-08T23:03:44.448+0800 [initandlisten] waiting for connections on port 27017
```

http://docs.mongodb.org/manual/reference/configuration-options/

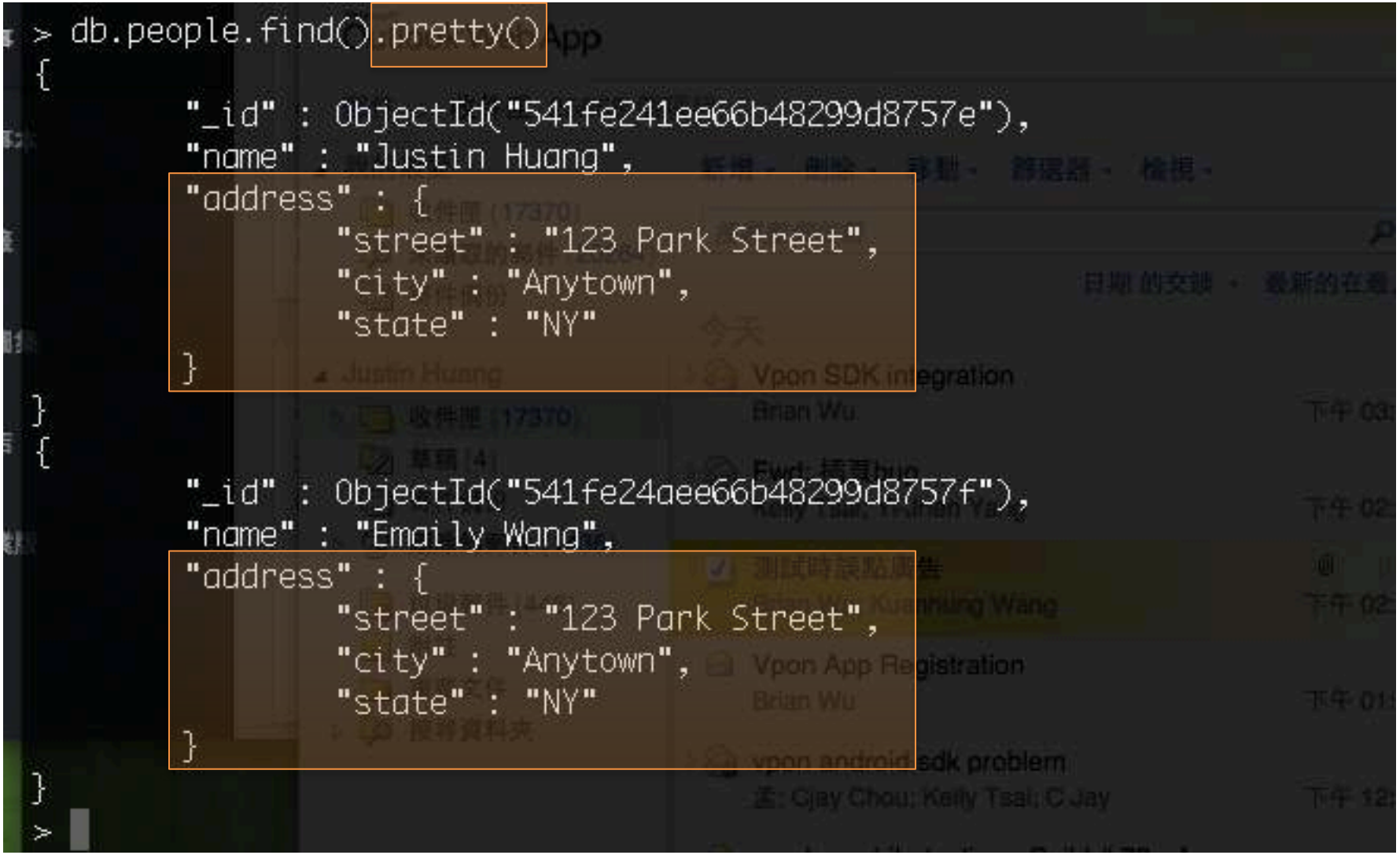
Basic Operations with the Shell



Update and remove



Embedded Document



```
> db.people.find().pretty()
{
        "_id" : ObjectId("541fe241ee66b48299d8757e"),
        "name" : "Justin Huang",
        "address" : {
                "street" : "123 Park Street",
                "city" : "Anytown",
                "state" : "NY"
        }
}
{
        "_id" : ObjectId("541fe24aee66b48299d8757f"),
        "name" : "Emaily Wang",
        "address" : {
                "street" : "123 Park Street",
                "city" : "Anytown",
                "state" : "NY"
        }
}
>
```

Ref: MongoDB: The Definitive Guide

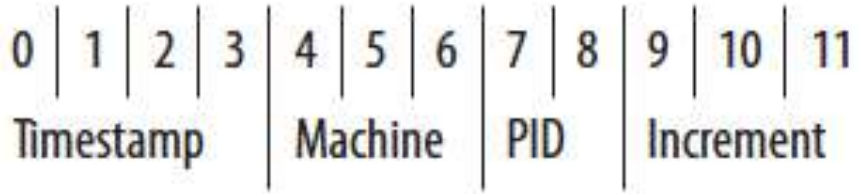
```
apple@justin-c-huang:bin$ ./mongod -f /Volumes/newSpace/iii_mongodb_demo/iii_config.cfg
about to fork child process, waiting until server is ready for connections.
forked process: 19726
child process started successfully, parent exiting
apple@justin-c-huang:bin$
```

```
apple@justin-c-huang:etc$ mongo
MongoDB shell version: 2.6.1
connecting to: test
Server has startup warnings:
2014-09-28T18:20:05.696+0800 [initandlisten]
2014-09-28T18:20:05.697+0800 [initandlisten] ** WARNING: soft rlimits too low. Number of files is 256, should be at least 1000
> db.adminCommand({shutdown:1})
2014-09-28T18:21:25.695+0800 DBClientCursor::init call() failed
2014-09-28T18:21:25.697+0800 Error: error doing query: failed at src/mongo/shell/query.js:81
2014-09-28T18:21:25.699+0800 trying reconnect to 127.0.0.1:27017 (127.0.0.1) failed
2014-09-28T18:21:25.699+0800 warning: Failed to connect to 127.0.0.1:27017, reason: errno:61 Connection refused
2014-09-28T18:21:25.699+0800 reconnect 127.0.0.1:27017 (127.0.0.1) failed failed couldn't connect to server 127.0.0.1:27017 (12
nnection attempt failed
```

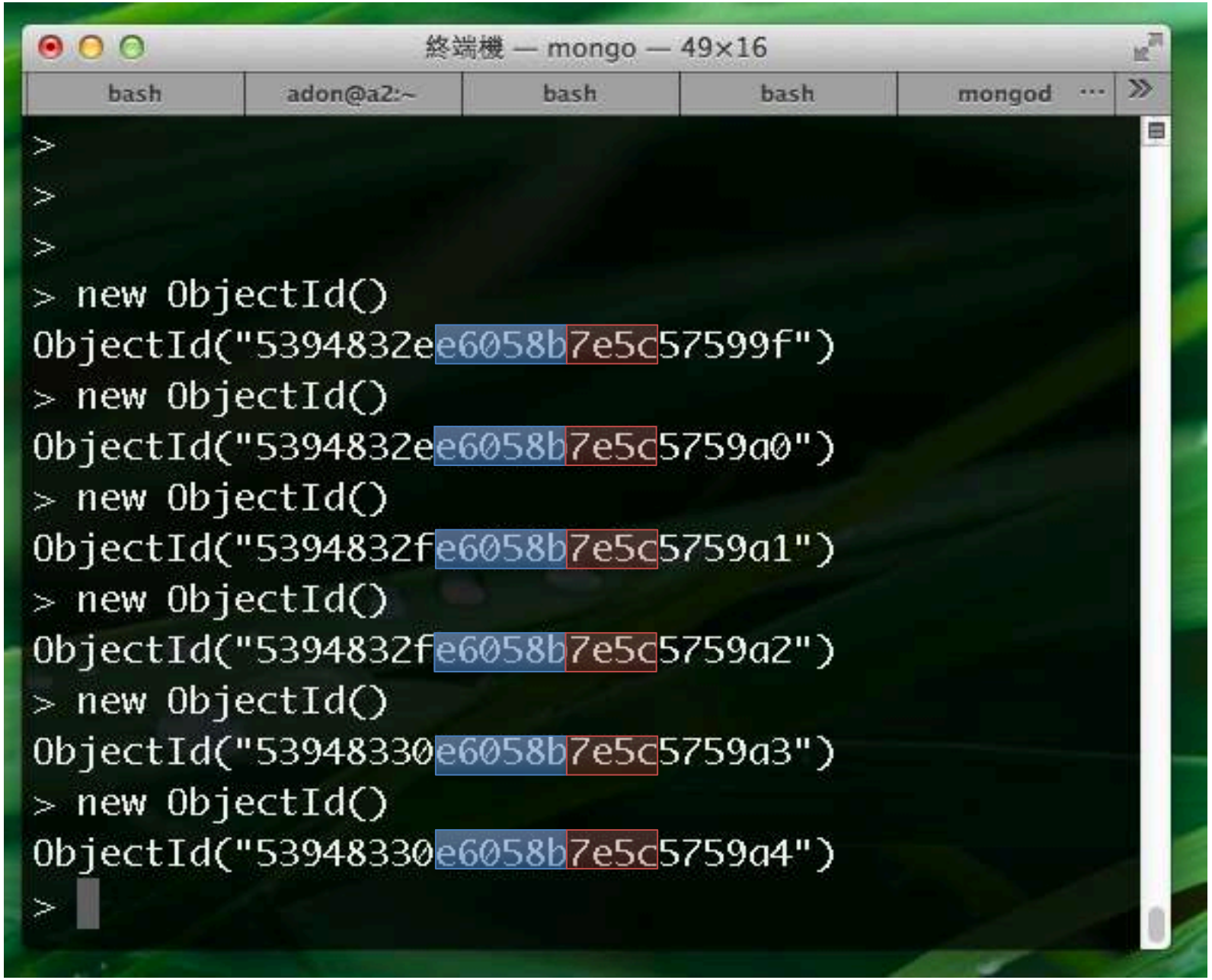
```
systemLog:
   destination: file
   path: "/Volumes/newSpace/iii_mongodb_demo/iii_2016_1_mongodb.log"
   quiet: true
   logAppend: true
storage:
   dbPath: "/Volumes/newSpace/iii_mongodb_demo/iii_db_2016"
   journal:
      enabled: true
processManagement:
   fork: true
net:
   http:
     enabled: true
     JSONPEnabled: true
     RESTInterfaceEnabled: true
   bindIp: 127.0.0.1
   port: 27017
```

_id and ObjectId

- Every document must have an "**_id**" key.
- The "**_id**" key's value can be any type, but it defaults to an ObjectId.
- In a single collection, every document must have a unique value for "**_id**"
- ObjectIds use 12 bytes of storage.



Ref: MongoDB: The Definitive Guide


終端機 — mongo — 49×16
bash
adon@a2:~
bash
bash
mongod
>
>
>
> new ObjectId()
ObjectId("5394832ee6058b7e5c57599f")
> new ObjectId()
ObjectId("5394832ee6058b7e5c5759a0")
> new ObjectId()
ObjectId("5394832fe6058b7e5c5759a1")
> new ObjectId()
ObjectId("5394832fe6058b7e5c5759a2")
> new ObjectId()
ObjectId("53948330e6058b7e5c5759a3")
> new ObjectId()
ObjectId("53948330e6058b7e5c5759a4")
>

Bulk insert

```
> db.testData.insert([{a:1},{b:2},{c:3}])
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 3,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})
> db.testData.find()
{ "_id" : ObjectId("539476f6f0d60812ea89f25b"), "name" : "mongo" }
{ "_id" : ObjectId("539476fcf0d60812ea89f25c"), "x" : 3 }
{ "_id" : ObjectId("53947a978f88712a99ad3ed5"), "bar1" : "hello world2" }
{ "_id" : ObjectId("53947a9ac6f75713a28959da"), "bar1" : "hello world2" }
{ "_id" : ObjectId("53948928e6058b7e5c5759a5"), "a" : 1 }
{ "_id" : ObjectId("53948928e6058b7e5c5759a6"), "b" : 2 }
{ "_id" : ObjectId("53948928e6058b7e5c5759a7"), "c" : 3 }
>
```

Document Insert Validation

- Adds an "_id" field if one does not exist.
- All documents must be smaller than 16 MB.
- To see the BSON size (in bytes) of the document *doc*, run Object.bsonsize(*doc*) from the shell.

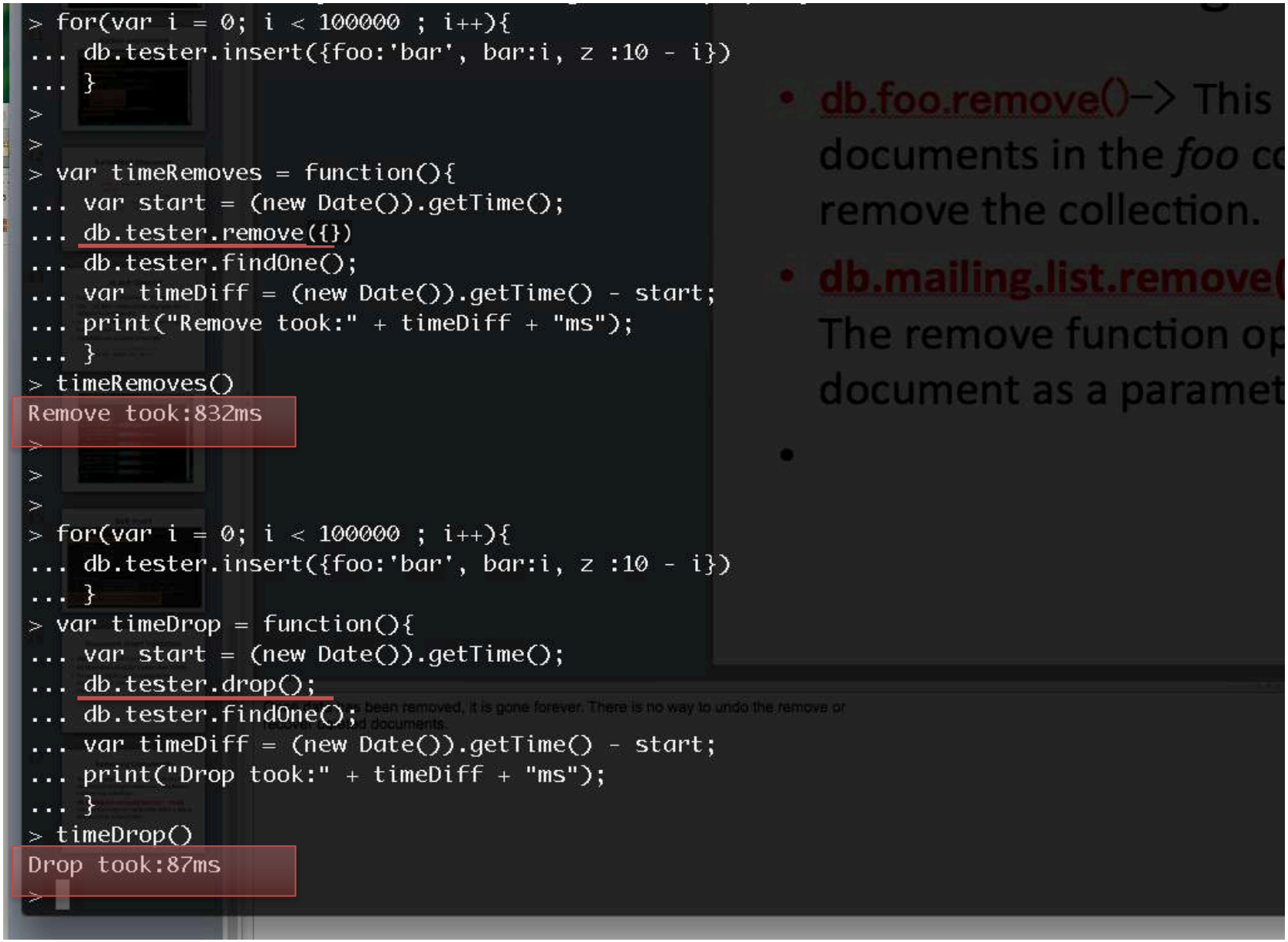
Ref: MongoDB: The Definitive Guide

Removing Documents

- **db.foo.remove({})**-> This will remove all of the documents in the *foo* collection. This doesn't remove the collection.
- **db.mailing.list.remove({"opt-out" : true})** The remove function optionally takes a query document as a parameter.

Ref: MongoDB: The Definitive Guide

Remove Speed



```
db.runCommand(
   {
      delete: "orders",
      deletes: [ { q: { status: "D" }, limit: 0 } ],
      writeConcern: { w: "majority", wtimeout: 5000 }
   }
)
```

```
db.runCommand(
   {
      delete: "orders",
      deletes: [ { q: { }, limit: 0 } ],
      writeConcern: { w: "majority", wtimeout: 5000 }
   }
)
```

```
db.runCommand(
   {
      insert: "users",
      documents: [
         { _id: 2, user: "ijk123", status: "A" },
         { _id: 3, user: "xyz123", status: "P" },
         { _id: 4, user: "mop123", status: "P" }
      ],
      ordered: false,
      writeConcern: { w: "majority", wtimeout: 5000 }
   }
)
```

Using Modifiers

```
> db.webs.findOne()
{
        "_id" : ObjectId("53971cf349c30b76a63cb6b0"),
        "url" : "www.example.com",
        "pageviews" : 52
}
> db.webs.update({url:'www.example.com'},{$inc: {pageviews:1}})
> db.webs.findOne()
{
        "_id" : ObjectId("53971cf349c30b76a63cb6b0"),
        "url" : "www.example.com",
        "pageviews" : 53
}
> db.webs.update({url:'www.example.com'},{$inc: {pageviews:7}})
> db.webs.findOne()
{
        "_id" : ObjectId("53971cf349c30b76a63cb6b0"),
        "url" : "www.example.com",
        "pageviews" : 60
}
>
```

Using Modifiers

```
> db.users.findOne()
{
        "_id" : ObjectId("53971e517a442f47069bca4b"),
        "name" : "joe",
        "age" : 30,
        "sex" : "male"
}
>
> db.users.update({"_id" : ObjectId("53971e517a442f47069bca4b")},
... {$set:{'favorite book': 'war and peace'}})
>
> db.users.findOne()
{
        "_id" : ObjectId("53971e517a442f47069bca4b"),
        "name" : "joe",
        "age" : 30,
        "sex" : "male",
        "favorite book" : "war and peace"
}
>
> db.users.update({"_id" : ObjectId("53971e517a442f47069bca4b")},
... {$set:{'favorite book': 'Green Eggs and Ham'}})
>
> db.users.findOne()
{
        "_id" : ObjectId("53971e517a442f47069bca4b"),
        "name" : "joe",
        "age" : 30,
        "sex" : "male",
        "favorite book" : "Green Eggs and Ham"
}
```

Using Modifiers

```
> db.users.update({"_id" : ObjectId("53971e517a442f47069bca4b")},
... {$set:{'favorite book':
... ['book1','book2','book3']}})
> db.users.findOne()
{
        "_id" : ObjectId("53971e517a442f47069bca4b"),
        "name" : "joe",
        "age" : 30,
        "sex" : "male",
        "favorite book" : [
                "book1",
                "book2",
                "book3"
        ]
}
> db.users.update({"name" : "joe"},{$unset:{'favorite book':1}})
> db.users.findOne()
{
        "_id" : ObjectId("53971e517a442f47069bca4b"),
        "name" : "joe",
        "age" : 30,
        "sex" : "male"
}
```

Using Modifiers

```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "title" : "A Blog Post",
    "content" : "...",
    "author" : {
        "name" : "joe",
        "email" : "joe@example.com"
    }
}
> db.blog.posts.update({"author.name" : "joe"},
... {"$set" : {"author.name" : "joe schmoe"}})
```

Ref: MongoDB: The Definitive Guide

```
> db.blog.posts.findOne()
{
        "_id" : ObjectId("5397240b08cc5aa45ccb5be9"),
        "title" : "A blog post",
        "comment" : [ ]
}
>
> db.blog.posts.update({"title" : "A blog post"},
... {$push:{comment:
... {name:'justin', email:'justin@gmail',content:'nice post.'}}})
>
> db.blog.posts.findOne()
{
        "_id" : ObjectId("5397240b08cc5aa45ccb5be9"),
        "title" : "A blog post",
        "comment" : [
                {
                        "name" : "justin",
                        "email" : "justin@gmail",
                        "content" : "nice post."
                }
        ]
}
> db.blog.posts.update({"title" : "A blog post"},
... {$push:{comment:
... {name:'alan', email:'alan@gmail',content:'very good.'}}})
>
> db.blog.posts.findOne()
{
        "_id" : ObjectId("5397240b08cc5aa45ccb5be9"),
        "title" : "A blog post",
        "comment" : [
                {
                        "name" : "justin",
                        "email" : "justin@gmail",
                        "content" : "nice post."
                },
                {
                        "name" : "alan",
                        "email" : "alan@gmail",
                        "content" : "very good."
                }
        ]
}
```

```
> db.tmp.findOne()
{
        "_id" : ObjectId("5397b8f60b5fc01b0a6fe301"),
        "votes" : [
                5,
                7,
                8,
                12
        ]
}
> db.tmp.update({},{$addToSet:{votes:3}})
> db.tmp.findOne()
{
        "_id" : ObjectId("5397b8f60b5fc01b0a6fe301"),
        "votes" : [
                5,
                7,
                8,
                12,
                3
        ]
}
> db.tmp.update({},{$addToSet:{votes:3}})
> db.tmp.findOne()
{
        "_id" : ObjectId("5397b8f60b5fc01b0a6fe301"),
        "votes" : [
                5,
                7,
                8,
                12,
                3
        ]
}
```

```
> db.users.update({"_id" : ObjectId("4b2d75476cc613d5ee930164")}, {"$addToSet" :
... {"emails" : {"$each" :
...     ["joe@php.net", "joe@example.com", "joe@python.org"]}}})
> db.users.findOne({"_id" : ObjectId("4b2d75476cc613d5ee930164")})
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "username" : "joe",
    "emails" : [
        "joe@example.com",
        "joe@gmail.com",
        "joe@yahoo.com",
        "joe@hotmail.com"
        "joe@php.net"
        "joe@python.org"
    ]
}
```

Modifier speed

```
> db.tmp.insert({x:"a"})
WriteResult({ "nInserted" : 1 })
> db.tmp.insert({x:"b"})
WriteResult({ "nInserted" : 1 })
> db.tmp.insert({x:"c"})
WriteResult({ "nInserted" : 1 })
> db.tmp.find()
{ "_id" : ObjectId("5944951e3bab03bfff1d3ea9"), "x" : "a" }
{ "_id" : ObjectId("594495243bab03bfff1d3eaa"), "x" : "b" }
{ "_id" : ObjectId("5944952c3bab03bfff1d3eab"), "x" : "c" }
> db.tmp.update({x:"b"},{$set:{x:"bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.tmp.find()
{ "_id" : ObjectId("5944951e3bab03bfff1d3ea9"), "x" : "a" }
{ "_id" : ObjectId("5944952c3bab03bfff1d3eab"), "x" : "c" }
{ "_id" : ObjectId("594495243bab03bfff1d3eaa"), "x" : "bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bb" }
```

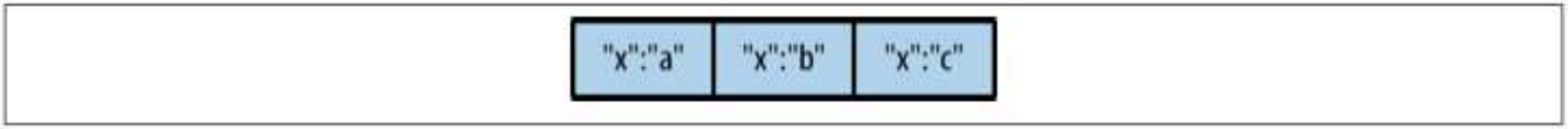


Figure 3-1. Initially, documents are inserted with no space between them

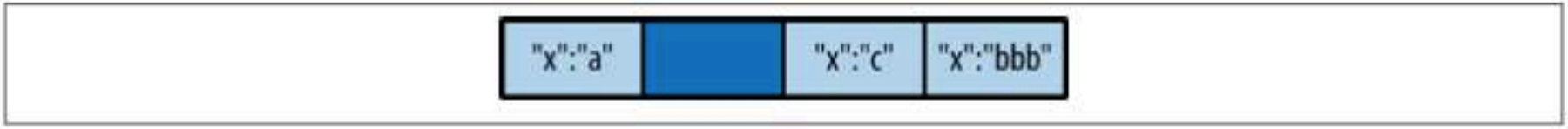


Figure 3-2. If a document grows and must be moved, free space is left behind and the padding size in increased

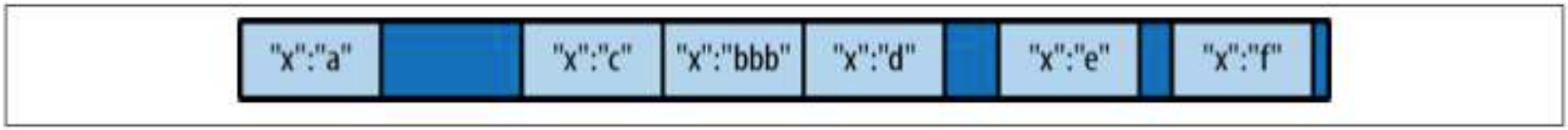


Figure 3-3. Subsequent documents are inserted with the padding factor in free space between them. If moves do not occur on subsequent inserts, this padding factor will decrease.

Ref: MongoDB: The Definitive Guide

Upsert

- An *upsert* is a special type of update. If **no document** is found that matches the update criteria, a new document will be created by combining the criteria and updated documents.

Ref: MongoDB: The Definitive Guide

```
> db.webs.find()
{ "_id" : ObjectId("53971cf349c30b76a63cb6b0"), "url" : "www.example.com", "pageviews" : 60 }
>
>
>
>
>
>
>
>
> db.webs.update({url:'www.google.com'},{$inc:{pageviews:1}},true)
WriteResult({
        "nMatched" : 0,
        "nUpserted" : 1,
        "nModified" : 0,
        "_id" : ObjectId("5399cb4eab25927a80fe0232")
})
>
>
> db.webs.find()
{ "_id" : ObjectId("53971cf349c30b76a63cb6b0"), "url" : "www.example.com", "pageviews" : 60 }
{ "_id" : ObjectId("5399cb4eab25927a80fe0232"), "url" : "www.google.com", "pageviews" : 1 }
>
```

Save

```
> db.foo.find()
> var x = {num:33}
> db.foo.save(x)
WriteResult({ "nInserted" : 1 })
> db.foo.findOne()
{ "_id" : ObjectId("5399cf655323ac617182a4e3"), "num" : 33 }
> var y = db.foo.findOne()
> y.num = 66
66
> db.foo.save(y)
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.foo.findOne()
{ "_id" : ObjectId("5399cf655323ac617182a4e3"), "num" : 66 }
>
```

Updating Multiple Documents

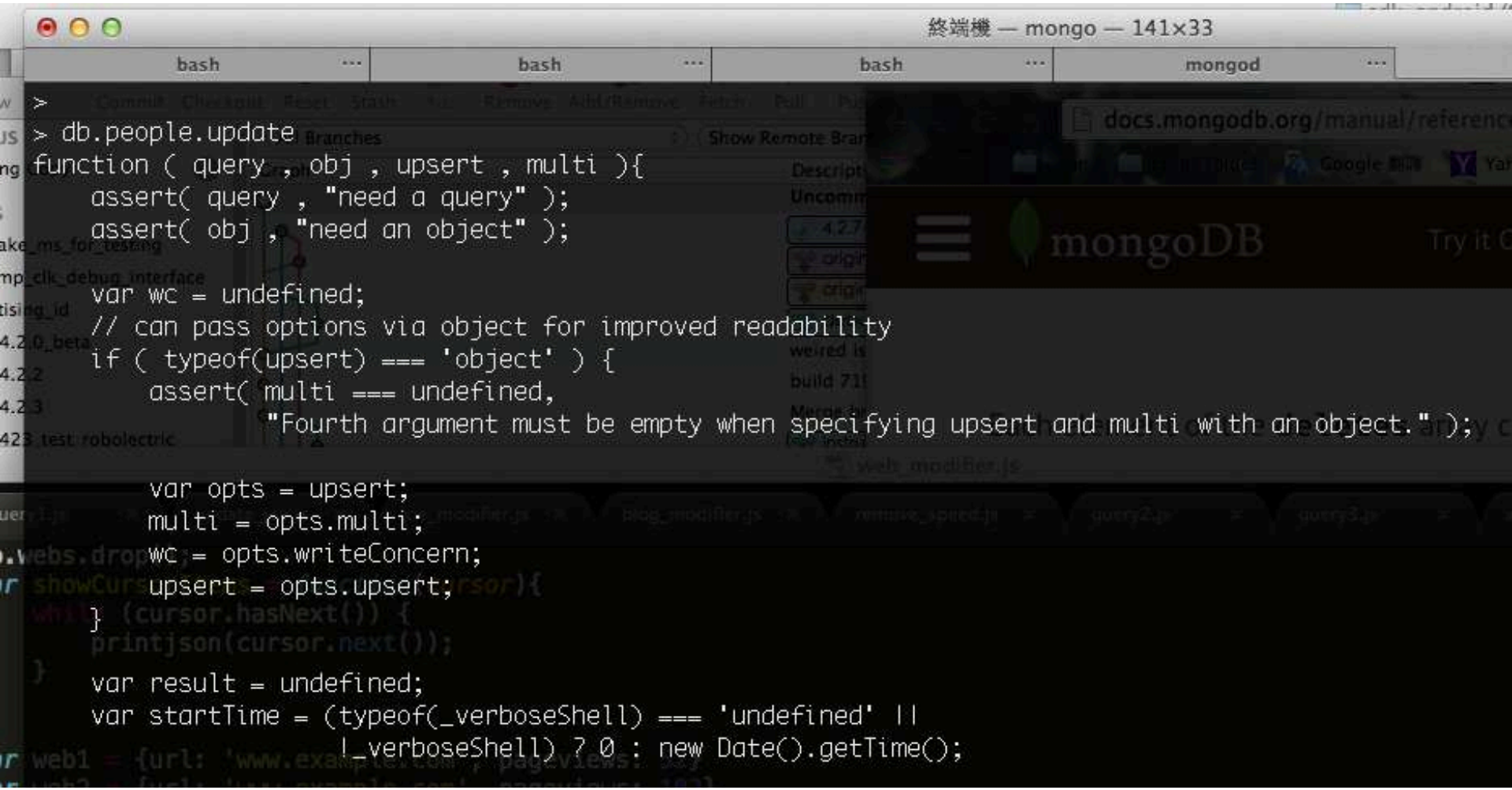
```
> db.users.update({"birthday" : "10/13/1978"},
... {"$set" : {"gift" : "Happy Birthday!"}}, false, true)
```

```
> db.count.update({x : 1}, {$inc : {x : 1}}, false, true)
> db.runCommand({getLastError : 1})
{
    "err" : null,
    "updatedExisting" : true,
    "n" : 5,
    "ok" : true
}
```

"n" is 5, meaning that five documents were affected by the update.

Ref: MongoDB: The Definitive Guide

Using Modifiers



http://docs.mongodb.org/manual/reference/command/findAndModify

findAndModify

```
db.people.findAndModify({
    query: { name: "Pascal", state: "active", rating: 25 },
    sort: { rating: 1 },
    update: { $inc: { score: 1 } },
    upsert: true,
    new: true
})
```

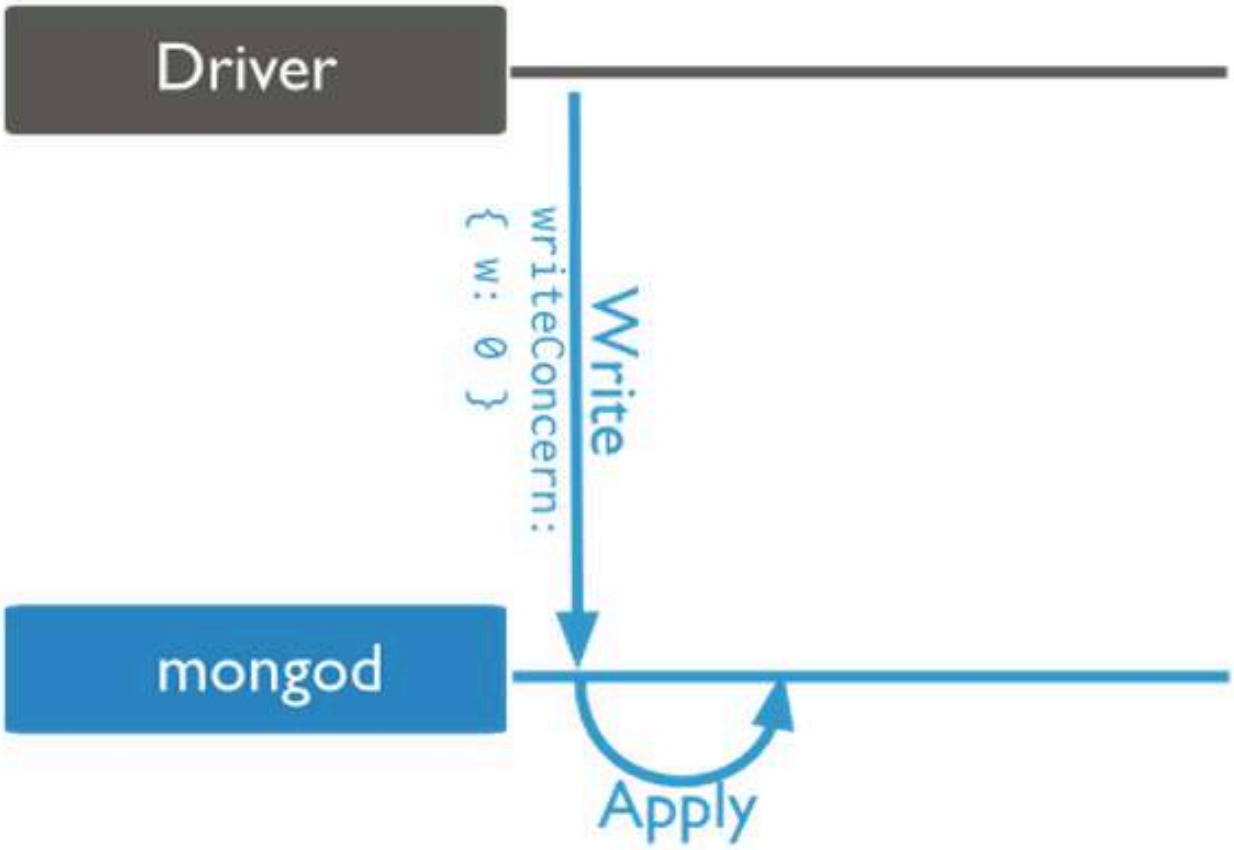
The method returns the newly inserted document:

```
{
   "_id" : ObjectId("50f49ad6444c11ac2448a5d6"),
   "name" : "Pascal",
   "rating" : 25,
   "score" : 1,
   "state" : "active"
}
```

Write Concern

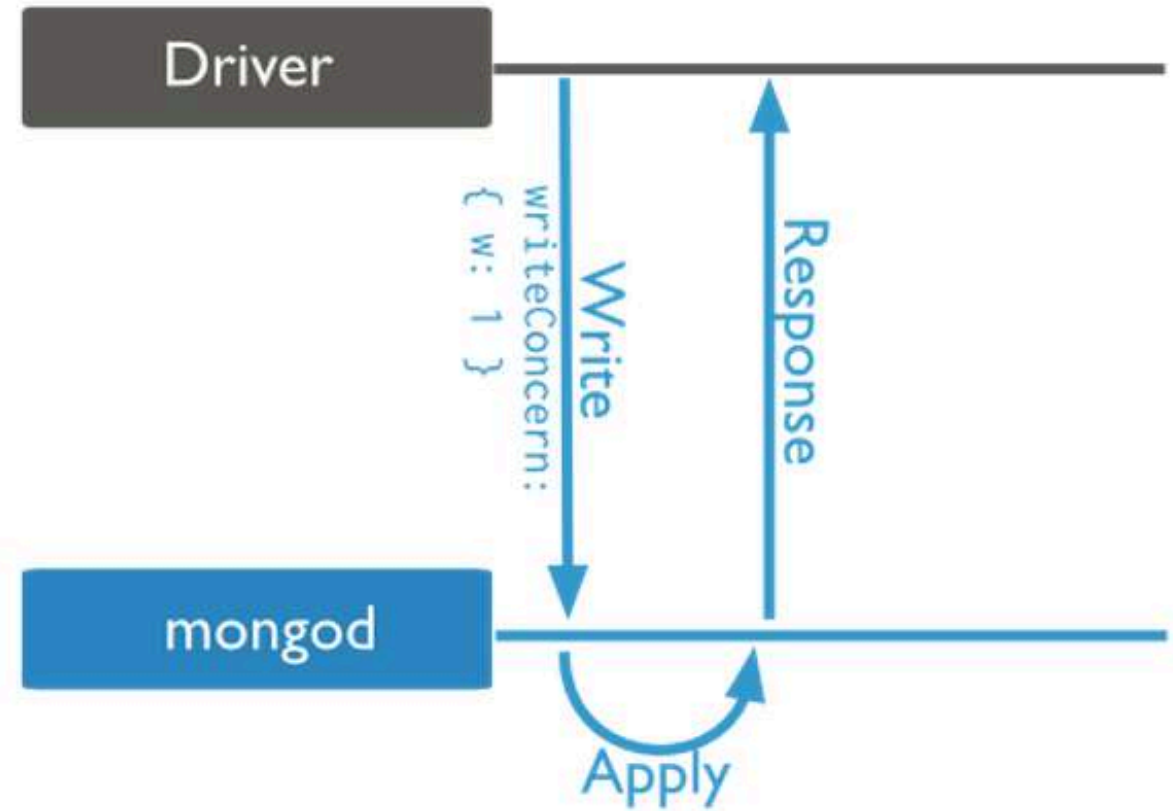
- The two basic write concerns are *acknowledged(應答式)* or *unacknowledged(非應答式)* writes.
- Acknowledged writes are the default: **you get a response** that tells you whether or not the database successfully processed your write.
- Unacknowledged writes **do not return any response**, so you do not know if the write succeeded or not.

Write Concern



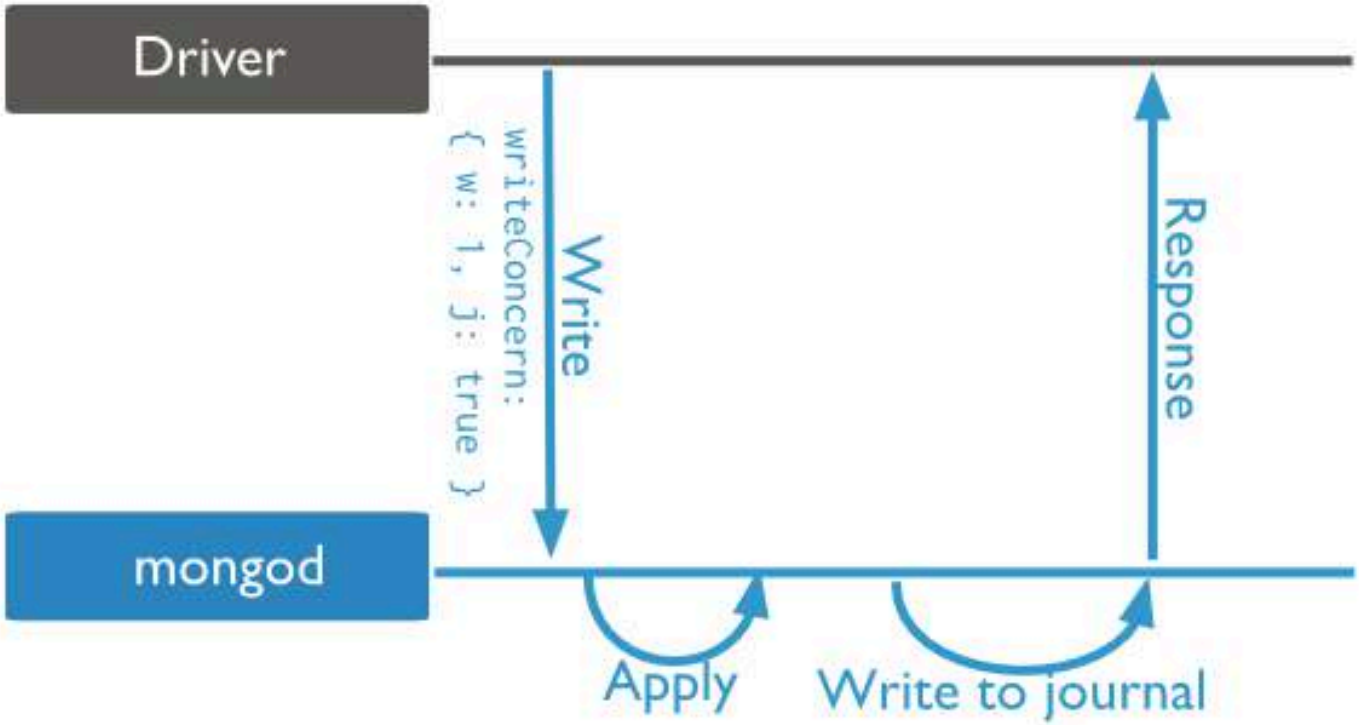
Write operation to a **mongod** instance with write concern of **unacknowledged**. The client does not wait for any acknowledgment.

Write Concern



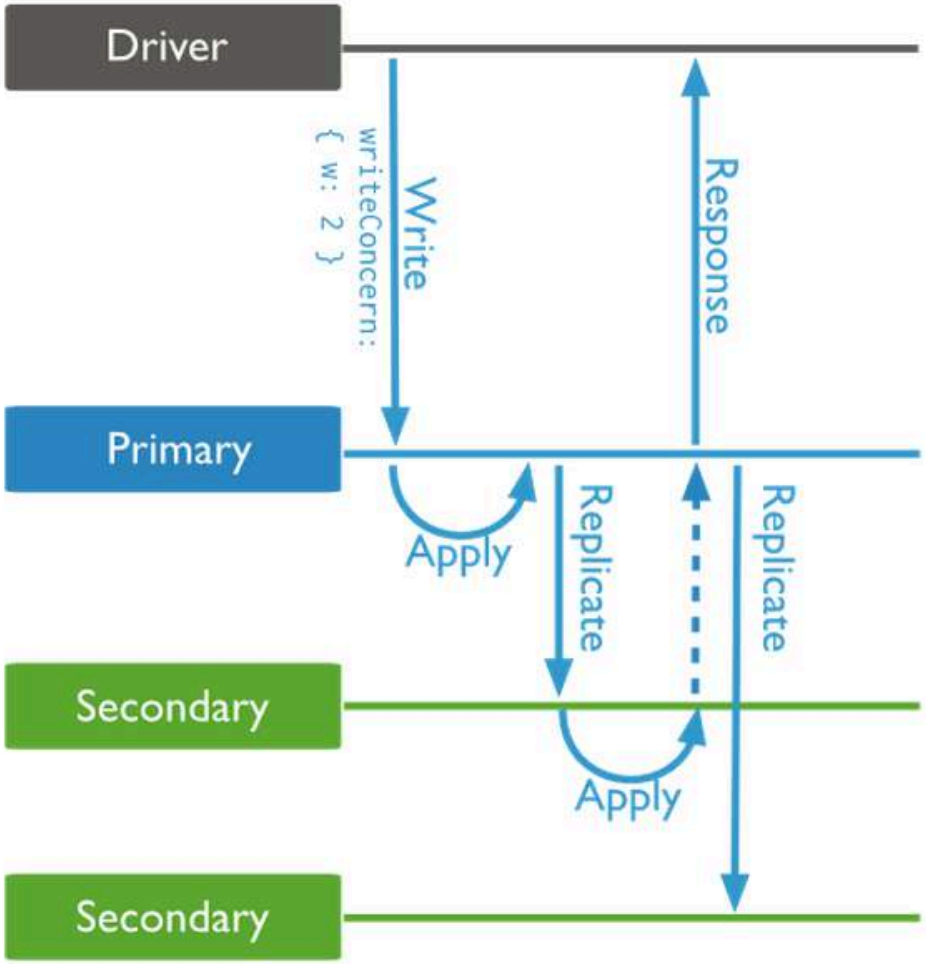
Write operation to a **mongod** instance with write concern of **acknowledged**. The client waits for acknowledgment of success or exception.

Write Concern



Write operation to a **mongod** instance with write concern of **journaled**. The **mongod** sends acknowledgment after it commits the write operation to the journal.

Write Concern



Write operation to a replica set with write concern level of **w:2** or write to the primary and at least one secondary.

```
db.products.insert(
    { item: "envelopes", qty : 100, type: "Clasp" },
    { writeConcern: { w: "majority", wtimeout: 5000 } }
)
```

```
db.runCommand(
   {
      update: "users",
      updates: [
         {
           q: { user: "abc123" }, u: { $set: { status: "A" }, $inc: { points: 1 } }
         }
      ],
      ordered: false,
      writeConcern: { w: "majority", wtimeout: 5000 }
   }
)
```

Specifying Which Keys to Return

```
> db.users.find({}, {"username" : 1, "email" : 1})
{
    "_id" : ObjectId("4ba0f0dfd22aa494fd523620"),
    "username" : "joe",
    "email" : "joe@example.com"
}
```

```
> db.users.find({}, {"username" : 1, "_id" : 0})
{
    "username" : "joe",
}
```

Ref: MongoDB: The Definitive Guide

Query Criteria

```
var showCursorItems = function(cursor){
    while (cursor.hasNext()) {
        printjson(cursor.next());
    }
}

var db = db.getSisterDB("iii");

db.users.drop();

for(var i = 0; i < 100; i++){
    db.users.insert({age:i});
}

cursor = db.users.find({},{age:1,_id:0});
showCursorItems(cursor);

print('------------between the ages of 18 and 30----------------');

cursor = db.users.find({age:{$gte:18, $lte:30}},{age:1,_id:0});
showCursorItems(cursor);
```

$in and $nin

```
print('------------about $in ------------------');
cursor = db.users.find(
    {age:{$in:[20,30,40]}},
    {age:1,_id:0}
);
showCursorItems(cursor);

print('------------about $nin-----------------');
cursor = db.users.find(
    {age:{$nin:[20,21,22,23,24,25,26,27,28,29,30]}},
    {age:1,_id:0}
);
showCursorItems(cursor);
```

$or

```
var productA = {name: 'A', quantity: 50, price: 10}
var productB = {name: 'B', quantity: 20, price: 20}
var productC = {name: 'C', quantity: 50, price: 30}
var productD = {name: 'D', quantity: 30, price: 10}
var productE = {name: 'E', quantity: 60, price: 10}
var productF = {name: 'F', quantity: 80, price: 20}
var productG = {name: 'G', quantity: 10, price: 90}
var productH = {name: 'H', quantity: 60, price: 60}

db.inventory.insert([productA,productB,productC,productD,productE,productF,productG,productH]);

cursor = db.inventory.find( {$or:[
                                {quantity:10},
                                {price:10}
                              ]
                            },
                            {_id:0}
);

showCursorItems(cursor);
```

$not

```
cursor = db.inventory.find( {
                                price:{
                                        $not:{
                                                $gte:20
                                              }
                                      }
                            },
                            {_id:0}
);
showCursorItems(cursor);
```

null

```
var productA = {name: 'A', quantity: 50, price: 10, level: 10}
var productB = {name: 'B', quantity: 20, price: 20, level: 30}
var productC = {name: 'C', quantity: 50, price: 30, level: 20}
var productD = {name: 'D', quantity: 30, price: 10, level: 50}
var productE = {name: 'E', quantity: 60, price: 10, level: 60}
var productF = {name: 'F', quantity: 80, price: 20}
var productG = {name: 'G', quantity: 10, price: 90}
var productH = {name: 'H', quantity: 60, price: 60}

db.inventory.insert([productA,productB,productC,productD,productE,productF,productG,productH]);

cursor = db.inventory.find({
                            level:null
                          },
                          {_id:0}
);
showCursorItems(cursor);
```

```
var productA = {name: 'A', quantity: 50, price: 10, level: 10}
var productB = {name: 'B', quantity: 20, price: 20, level: 30}
var productC = {name: 'C', quantity: 50, price: 30, level: 20}
var productD = {name: 'D', quantity: 30, price: 10, level: 50}
var productE = {name: 'E', quantity: 60, price: 10, level: 60}
var productF = {name: 'F', quantity: 80, price: 20, level: null}
var productG = {name: 'G', quantity: 10, price: 90, level: null}
var productH = {name: 'H', quantity: 60, price: 60}

db.inventory.insert([
    productA,
    productB,
    productC,
    productD,
    productE,
    productF,
    productG,
    productH
]);

cursor = db.inventory.find({
                            level:{
                                    $in:[null],
                                    $exists:true
                                  }
                          },
                          {_id:0}
);
showCursorItems(cursor);
```

Array

```
db.food.insert({_id:1,fruit:['apple','banana','peach']});
db.food.insert({_id:2,fruit:['apple','kumpuat','orange']});
db.food.insert({_id:3,fruit:['cherry','banana','apple']});

cursor = db.food.find({fruit:'banana'});
showCursorItems(cursor);
```

```
db.food.insert({_id:1,fruit:['apple','banana','peach']});
db.food.insert({_id:2,fruit:['apple','kumpuat','orange']});
db.food.insert({_id:3,fruit:['cherry','banana','apple']});

cursor = db.food.find({'fruit.2':'orange'});
showCursorItems(cursor);
```

$all

```
db.food.insert({_id:1,fruit:['apple','banana','peach']});
db.food.insert({_id:2,fruit:['apple','kumpuat','orange']});
db.food.insert({_id:3,fruit:['cherry','banana','apple']});

cursor = db.food.find({
                        fruit:{
                            $all:['apple','banana']
                        }
                      }
);

showCursorItems(cursor);
```

$size

```
db.food.insert({_id:1,fruit:['apple','banana','peach']});
db.food.insert({_id:2,fruit:['apple','kumpuat','orange']});
db.food.insert({_id:3,fruit:['cherry','banana','apple']});
db.food.insert({_id:4,fruit:['banana','apple']});


cursor = db.food.find({fruit:{$size:2}});
showCursorItems(cursor);
```

About array's size

```
> db.food.update(criteria, {"$push" : {"fruit" : "strawberry"}})
```

it can simply be changed to this:

```
> db.food.update(criteria,
... {"$push" : {"fruit" : "strawberry"}, "$inc" : {"size" : 1}})
```

```
> db.food.find({"size" : {"$gt" : 3}})
```

Ref: MongoDB: The Definitive Guide

$slice

```
db.food.insert({_id:1,fruit:['apple','banana','peach']});
db.food.insert({_id:2,fruit:['apple','kumpuat','orange']});
db.food.insert({_id:3,fruit:['cherry','banana','apple']});
db.food.insert({_id:4,fruit:['banana','apple']});

cursor = db.food.find({},{fruit:{$slice:1}});
showCursorItems(cursor);
```

```
{ "_id" : 1, "fruit" : [ "apple" ] }
{ "_id" : 2, "fruit" : [ "apple" ] }
{ "_id" : 3, "fruit" : [ "cherry" ] }
{ "_id" : 4, "fruit" : [ "banana" ] }
```

$slice

```
db.food.insert({_id:1,fruit:['apple','banana','peach']});
db.food.insert({_id:2,fruit:['apple','kumpuat','orange']});
db.food.insert({_id:3,fruit:['cherry','banana','apple']});
db.food.insert({_id:4,fruit:['banana','apple']});


cursor = db.food.find({},{fruit:{$slice:[2,1]}});
showCursorItems(cursor);
```

```
{ "_id" : 1, "fruit" : [ "peach" ] }
{ "_id" : 2, "fruit" : [ "orange" ] }
{ "_id" : 3, "fruit" : [ "apple" ] }
{ "_id" : 4, "fruit" : [ ] }
```

$where

```
> db.foo.insert({"apple" : 1, "banana" : 6, "peach" : 3})
> db.foo.insert({"apple" : 8, "spinach" : 4, "watermelon" : 4})
```

We'd like to return documents where any two of the fields are equal.

```
> db.foo.find({"$where" : function () {
... for (var current in this) {
...     for (var other in this) {
...         if (current != other && this[current] == this[other]) {
...             return true;
...         }
...     }
... }
... return false;
... }});
```

Ref: MongoDB: The Definitive Guide

Injection attacks

```
> func = "function() { print('Hello, "+name+"!'); }"
```

If name is a user-defined variable, it could be the string `"'); db.dropDatabase(); print('"`, which would turn the code into this:

```
> func = "function() { print('Hello, '); db.dropDatabase(); print('!'); }"
```

Now, if you run this code, your entire database will be dropped!

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

UserId:

105 or 1=1

Server Result

```
SELECT * FROM Users WHERE UserId = 105 or 1=1
```


ZU 0666', 0, 0); DROP DATABASE TABLICE;

Cursor

- The database returns results from find using a *cursor*.
- The Cursor class implements JavaScript's iterator interface

```
db.food.insert({_id:1,fruit:['apple','banana','peach']});
db.food.insert({_id:2,fruit:['apple','kumpuat','orange']});
db.food.insert({_id:3,fruit:['cherry','banana','apple']});
db.food.insert({_id:4,fruit:['banana','apple']});

var cursor = db.food.find();
cursor.forEach(function(x){
    printjson(x.fruit);
});
```

Limits, Skips, and Sorts

```
var user = {name:'justin',age:20};
db.test.insert(user);

user.name = 'Alan'; user.age = 32; db.test.insert(user);
user.name = 'Kelly'; user.age = 25; db.test.insert(user);
user.name = 'Hopper'; user.age = 12; db.test.insert(user);
user.name = 'James'; user.age = 51; db.test.insert(user);
user.name = 'Lisa'; user.age = 65; db.test.insert(user);

var cursor = db.test.find().skip(5);
showCursorItems(cursor);
```

```
{
        "_id" : ObjectId("539be8fa5e50753e6203ba88"),
        "name" : "Lisa",
        "age" : 65
}
```

Limits, Skips, and Sorts

```
var user = {name:'justin',age:20};
db.test.insert(user);

user.name = 'Alan'; user.age = 32; db.test.insert(user);
user.name = 'Kelly'; user.age = 25; db.test.insert(user);
user.name = 'Hopper'; user.age = 12; db.test.insert(user);
user.name = 'James'; user.age = 51; db.test.insert(user);
user.name = 'Lisa'; user.age = 65; db.test.insert(user);

var cursor = db.test.find().limit(3).sort({age:1});
showCursorItems(cursor);
```

Finding a random document

```
> // do not use
> var total = db.foo.count()
> var random = Math.floor(Math.random()*total)
> db.foo.find().skip(random).limit(1)
```

```
> db.people.insert({"name" : "joe", "random" : Math.random()})
> db.people.insert({"name" : "john", "random" : Math.random()})
> db.people.insert({"name" : "jim", "random" : Math.random()})
```

```
> var random = Math.random()
> result = db.foo.findOne({"random" : {"$gt" : random}})
```

```
> if (result == null) {
...     result = db.foo.findOne({"random" : {"$lt" : random}})
... }
```

Database Commands

```
db.tmp.insert({_id:1,x:15});
db.tmp.insert({_id:1,x:25});

var err = db.runCommand({getLastError:1});
printjson(err);
```

```
{
        "connectionId" : 74,
        "err" : "insertDocument :: caused by :: 11000 E11000 duplicate key error index: iii.tmp.$_id_  dup key: { : 1.0 }",
        "code" : 11000,
        "n" : 0,
        "ok" : 1
}
```

Index

```
for(i = 0; i<1000000; i++){
    db.users.insert(
        {
            i:i,
            username:'user'+i,
            age:Math.floor(Math.random()*120),
            created: new Date()
        }
    );
}
print('-----------------------------------------------------------');
var explainObj = db.users.find({username:'user101'}).explain();
printjson(explainObj);
```

Ref: MongoDB: The Definitive Guide

```
{
    "executionSuccess" : true,
    "nReturned" : 1,
    "executionTimeMillis" : 412,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 1000000,
    "executionStages" : {
            "stage" : "COLLSCAN",
            "filter" : {
                    "username" : {
                            "$eq" : "user101"
                    }
            },
            "nReturned" : 1,
            "executionTimeMillisEstimate" : 352,
            "works" : 1000002,
            "advanced" : 1,
            "needTime" : 1000000,
            "needYield" : 0,
            "saveState" : 7823,
            "restoreState" : 7823,
            "isEOF" : 1,
            "invalidates" : 0,
            "direction" : "forward",
            "docsExamined" : 1000000
    }
}
```

Index

```
db.users.ensureIndex({username:1})

print('-----------------------------------------------------------');
var explainObj = db.users.find({username:'user101'}).explain();
printjson(explainObj);
```

```
{
    "executionSuccess" : true,
    "nReturned" : 1,
    "executionTimeMillis" : 0,
    "totalKeysExamined" : 1,
    "totalDocsExamined" : 1,
    "executionStages" : {
            "stage" : "FETCH",
            "nReturned" : 1,
            "executionTimeMillisEstimate" : 0,
            "works" : 2,
            "advanced" : 1,
            "needTime" : 0,
            "needYield" : 0,
            "saveState" : 0,
            "restoreState" : 0,
            "isEOF" : 1,
            "invalidates" : 0,
            "docsExamined" : 1,
            "alreadyHasObj" : 0,
            "inputStage" : {
                    "stage" : "IXSCAN",
                    "nReturned" : 1,
                    "executionTimeMillisEstimate" : 0,
                    "works" : 2,
                    "advanced" : 1,
                    "needTime" : 0,
                    "needYield" : 0,
                    "saveState" : 0,
                    "restoreState" : 0,
                    "isEOF" : 1,
                    "invalidates" : 0,
                    "keyPattern" : {
                            "username" : 1
                    },
                    "indexName" : "username_1",
                    "isMultiKey" : false,
```

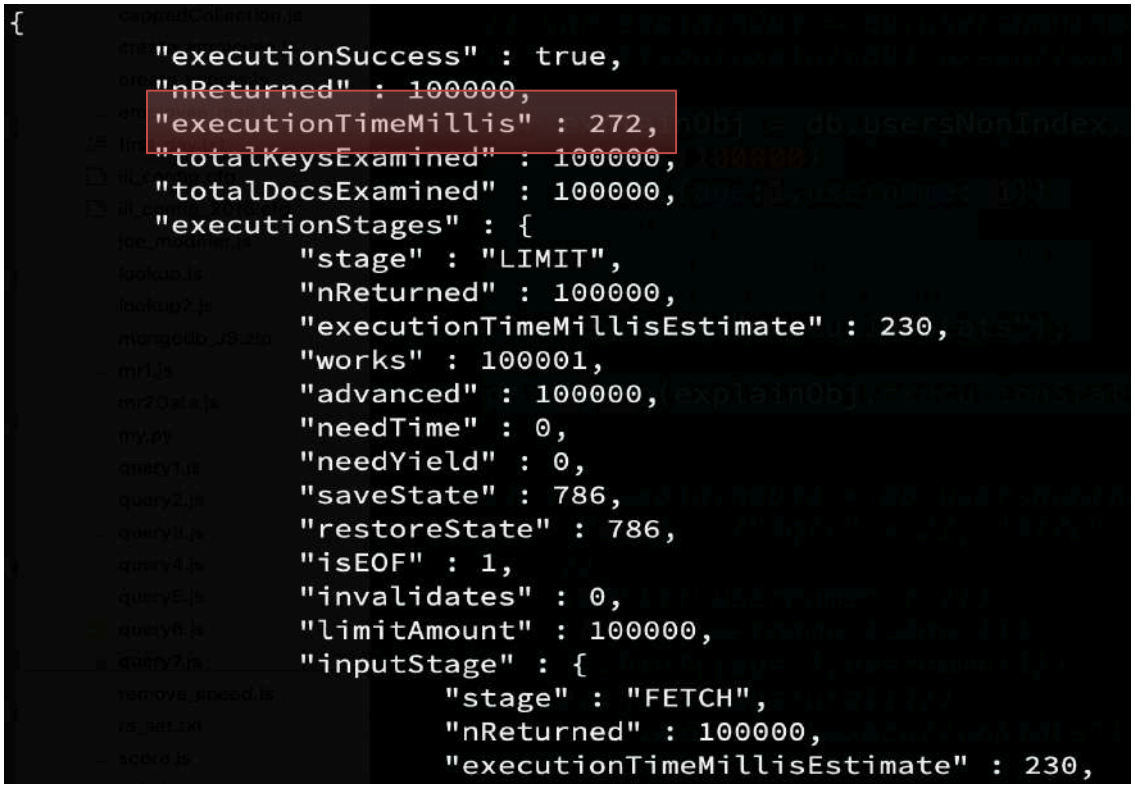
index

- Indexes have their price: every write (insert, update, or delete) will take longer for every index you add.
- MongoDB limits you to 64 indexes per collection.
- To choose which fields to create indexes for, look through your common queries and queries that need to be fast and try to find a common set of keys from those.

```
var explainObj = db.users.find({})
    .limit(100000)
    .sort({age:1,username: 1})
    .explain();

printjson(explainObj);
```

```
{
        "executionSuccess" : true,
        "nReturned" : 100000,
        "executionTimeMillis" : 3987,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 1000000,
        "executionStages" : {
                "stage" : "SORT",
                "nReturned" : 100000,
                "executionTimeMillisEstimate" : 3813,
                "works" : 1100004,
                "advanced" : 100000,
                "needTime" : 1000003,
                "needYield" : 0,
                "saveState" : 8698,
                "restoreState" : 8698,
                "isEOF" : 1,
                "invalidates" : 0,
                "sortPattern" : {
                        "age" : 1,
                        "username" : 1
                },
                "memUsage" : 9588867,
                "memLimit" : 33554432,
                "limitAmount" : 100000,
```

```
db.users.ensureIndex({age:1,username:1})

var explainObj = db.users.find({})
    .limit(100000)
    .sort({age:1,username: 1})
    .explain();

printjson(explainObj);
```



```
{
        "executionSuccess" : true,
        "nReturned" : 100000,
        "executionTimeMillis" : 272,
        "totalKeysExamined" : 100000,
        "totalDocsExamined" : 100000,
        "executionStages" : {
                "stage" : "LIMIT",
                "nReturned" : 100000,
                "executionTimeMillisEstimate" : 230,
                "works" : 100001,
                "advanced" : 100000,
                "needTime" : 0,
                "needYield" : 0,
                "saveState" : 786,
                "restoreState" : 786,
                "isEOF" : 1,
                "invalidates" : 0,
                "limitAmount" : 100000,
                "inputStage" : {
                        "stage" : "FETCH",
                        "nReturned" : 100000,
                        "executionTimeMillisEstimate" : 230,
```

Index

If we index this collection by `{"age" : 1, "username" : 1}`, the index will look roughly like this:

```
[0, "user100309"] -> 0x0c965148
[0, "user100334"] -> 0xf51f818e
[0, "user100479"] -> 0x00fd7934
...
[0, "user99985" ] -> 0xd246648f
[1, "user100156"] -> 0xf78d5bdd
[1, "user100187"] -> 0x68ab28bd
[1, "user100192"] -> 0x5c7fb621
...
[1, "user999920"] -> 0x67ded4b7
[2, "user100141"] -> 0x3996dd46
[2, "user100149"] -> 0xfce68412
[2, "user100223"] -> 0x91106e23
...
```

Ref: MongoDB: The Definitive Guide

Query type

- db.users.find({"age" : 21}).sort({"username" : -1})
- db.users.find({"age" : {"$gte" : 21, "$lte" : 30}})
- db.users.find({"age" : {"$gte" : 21, "$lte" : 30}}).sort({"username" : 1})

Ref: MongoDB: The Definitive Guide

```
var explainObj = db.users.find({
    "age" : {"$gte" : 21, "$lte" : 50}
})
.sort({"username" : 1})
.explain();

printjson(explainObj);
```

```
{
    "cursor" : "BtreeCursor age_1_username_1",
    "isMultiKey" : false,
    "n" : 250142,
    "nscannedObjects" : 250142,
    "nscanned" : 250142,
    "nscannedObjectsAllPlans" : 250142,
    "nscannedAllPlans" : 250142,
    "scanAndOrder" : true,
    "indexOnly" : false,
    "nYields" : 3909,
    "nChunkSkips" : 0,
    "millis" : 2322,
    "indexBounds" : {
        "age" : [
            [
                21,
                50
            ]
        ],
        "username" : [
            [
                {
                    "$minElement" : 1
                },
                {
                    "$maxElement" : 1
                }
            ]
        ]
    },
    "server" : "justin-c-huang.local:27017",
    "filterSet" : false
}
```

{"username" : 1, "age" : 1}

```
["user0", 69]
["user1", 50]
["user10", 80]
["user100", 48]
["user1000", 111]
["user10000", 98]
["user100000", 21] -> 0x73f0b48d
["user100001", 60]
["user100002", 82]
["user100003", 27] -> 0x0078f55f
["user100004", 22] -> 0x5f0d3088
["user100005", 95]
...
```

```
db.users.ensureIndex({username:1,age:1})
```

```
var explainObj = db.users.find({
    "age" : {"$gte" : 21, "$lte" : 50}
})
.sort({"username" : 1})
.explain();

printjson(explainObj);
```

```
{
    "cursor" : "BtreeCursor username_1_age_1",
    "isMultiKey" : false,
    "n" : 250142,
    "nscannedObjects" : 1000000,
    "nscanned" : 1000000,
    "nscannedObjectsAllPlans" : 1000371,
    "nscannedAllPlans" : 1000372,
    "scanAndOrder" : false,
    "indexOnly" : false,
    "nYields" : 7814,
    "nChunkSkips" : 0,
    "millis" : 2187,
    "indexBounds" : {
        "username" : [
            [
                {
                    "$minElement" : 1
                },
                {
                    "$maxElement" : 1
                }
            ]
        ],
        "age" : [
            [
                {
                    "$minElement" : 1
                },
                {
                    "$maxElement" : 1
                }
            ]
        ]
    },
    "server" : "justin-c-huang.local:27017",
    "filterSet" : false
}
```

```
var explainObj1 = db.users.find({
     "age" : {"$gte" : 21, "$lte" : 50}
    })
    .sort({"username" : 1})
    .limit(1000)
    .hint({age:1,username:1})
    .explain();

print('use age_username index spend:'+explainObj1.millis+'ms');

var explainObj2 = db.users.find({
     "age" : {"$gte" : 21, "$lte" : 50}
    })
    .sort({"username" : 1})
    .limit(1000)
    .hint({username:1,age:1})
    .explain();

print('use username_age index spend:'+explainObj2.millis+'ms');
```

```
use age_username index spend:1639ms
use username_age index spend:9ms
```

Indexing embedded docs

```
{
    "username" : "sid",
    "loc" : {
        "ip" : "1.2.3.4",
        "city" : "Springfield",
        "state" : "NY"
    }
}
```

```
> db.users.ensureIndex({"loc.city" : 1})
```

```
db.users.find({"loc.city" : "Shelbyville"})
```

Ref: MongoDB: The Definitive Guide

Index Cardinality(索引基數)

- *Cardinality* refers to how many distinct values there are for a field in a collection.
- In general, the greater the cardinality of a field, the more helpful an index on that field can be.
- try to create indexes on high-cardinality keys or at least put high- cardinality keys first in compound indexes

Ref: MongoDB: The Definitive Guide

When Not to Index

- Indexes become less and less efficient as you need to get larger percentages of a collection because using an index requires two lookups: one to look at the index entry and one following the index's pointer to the document.

Table 5-1. Properties that affect the effectiveness of indexes

Indexes often work well for	Table scans often work well for
Large collections	Small collections
Large documents	Small documents
Selective queries	Non-selective queries

Ref: MongoDB: The Definitive Guide

When Not to Index

- db.entries.find({"created_at" : {"$lt" : hourAgo}})

- db.entries.find({"created_at" : {"$lt" : hourAgo}}).hint({"$natural" : 1})

Unique Indexes

```
> db.users.ensureIndex({"username" : 1}, {"unique" : true})

> db.users.insert({username: "bob"})
> db.users.insert({username: "bob"})
E11000 duplicate key error index: test.users.$username_1  dup key: { : "bob" }
```

Text Indexes

```
db.reviews.createIndex( { comments: "text" } )
```

```
db.reviews.createIndex(
   {
     subject: "text",
     comments: "text"
   }
 )
```

```
db.collection.createIndex( { "$**": "text" } )
```

$text

```
db.articles.find( { $text: { $search: "coffee" } } )
```

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }
{ "_id" : 7, "subject" : "coffee and cream", "author" : "efg", "views" : 10 }
{ "_id" : 1, "subject" : "coffee", "author" : "xyz", "views" : 50 }
```

```
db.articles.find( { $text: { $search: "bake coffee cake" } } )
```

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }
{ "_id" : 7, "subject" : "coffee and cream", "author" : "efg", "views" : 10 }
{ "_id" : 1, "subject" : "coffee", "author" : "xyz", "views" : 50 }
{ "_id" : 3, "subject" : "Baking a cake", "author" : "abc", "views" : 90 }
{ "_id" : 4, "subject" : "baking", "author" : "xyz", "views" : 100 }
```

$text

```
db.articles.find( { $text: { $search: "\"coffee shop\"" } } )
```

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }
```

```
db.articles.find( { $text: { $search: "coffee -shop" } } )
```

```
{ "_id" : 7, "subject" : "coffee and cream", "author" : "efg", "views" : 10 }
{ "_id" : 1, "subject" : "coffee", "author" : "xyz", "views" : 50 }
```

Partial Index

```
db.restaurants.createIndex(
   { cuisine: 1, name: 1 },
   { partialFilterExpression: { rating: { $gt: 5 } } }
)
```

```
db.restaurants.find( { cuisine: "Italian", rating: { $gte: 8 } } )
```

```
db.contacts.createIndex(
   { name: 1 },
   { partialFilterExpression: { name: { $exists: true } } }
)
```

```
db.contacts.find( { name: "xyz", email: { $regex: /\.org$/ } } )
```

Capped Collections (固定集合)

- created in advance
- fixed-size collections
- Once a capped collection has been created, it cannot be changed
- think carefully about the size of a large collection before creating it.
- capped collections behave like circular queues

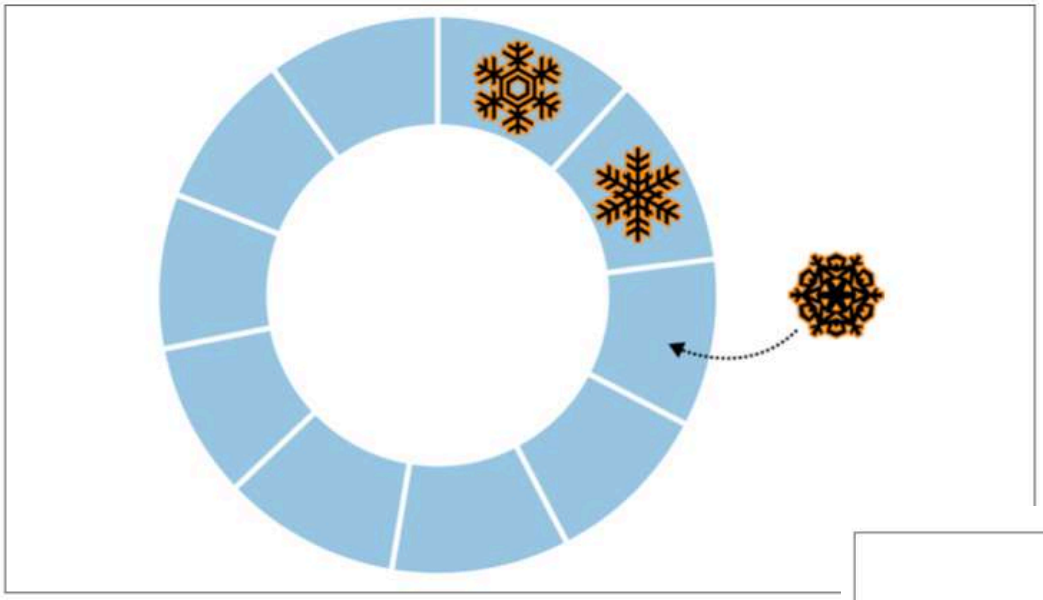


Figure 6-1. New documents are inserted at the end of the queue

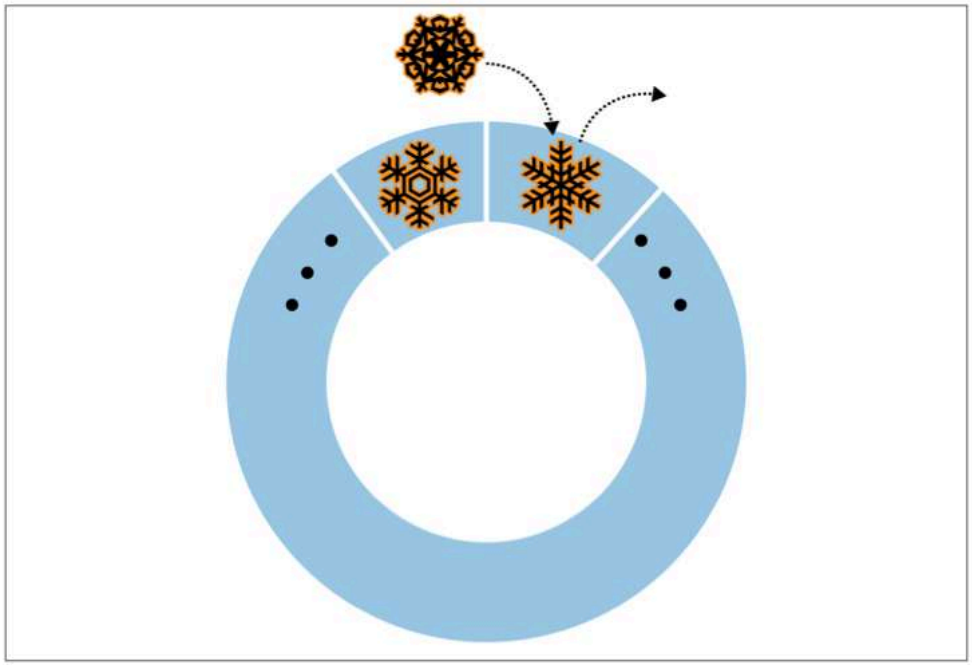


Figure 6-2. When the queue is full, the oldest element will be replaced by the newest

```
db.createCollection('capped_coll',
    {
        capped:true,
        size:10000, //maximum size in bytes for a capped collection.
        max:5 //maximum number of documents
    }
);
for(i = 0; i < 7 ; i++){
    db.capped_coll.insert({x:i})
}

var cursor = db.capped_coll.find();

showCursorItems(cursor);
```

```
{ "_id" : ObjectId("539c48cbdcafed5c0ef22373"), "x" : 2 }
{ "_id" : ObjectId("539c48cbdcafed5c0ef22374"), "x" : 3 }
{ "_id" : ObjectId("539c48cbdcafed5c0ef22375"), "x" : 4 }
{ "_id" : ObjectId("539c48cbdcafed5c0ef22376"), "x" : 5 }
{ "_id" : ObjectId("539c48cbdcafed5c0ef22377"), "x" : 6 }
```

Natural sort

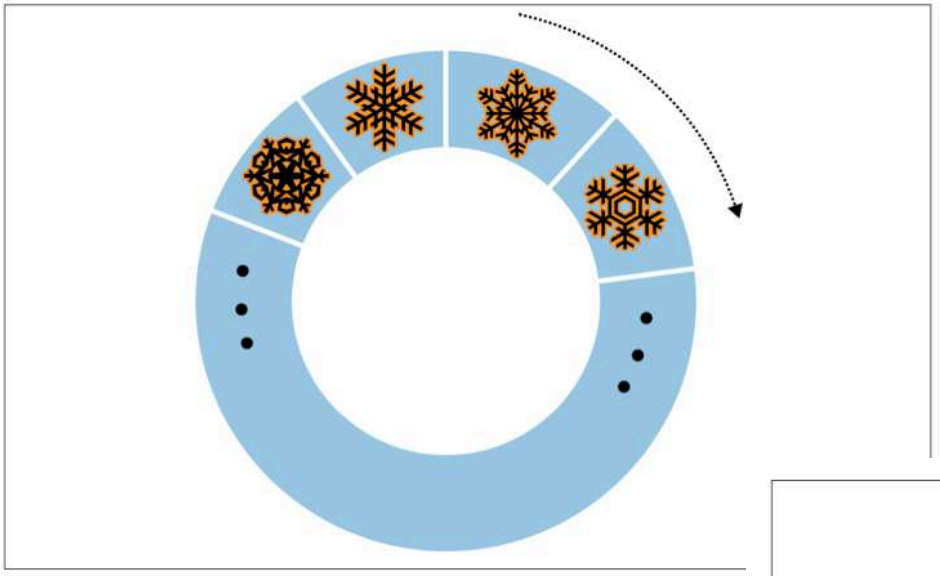


Figure 6-3. Sort by {"$natural" : 1}

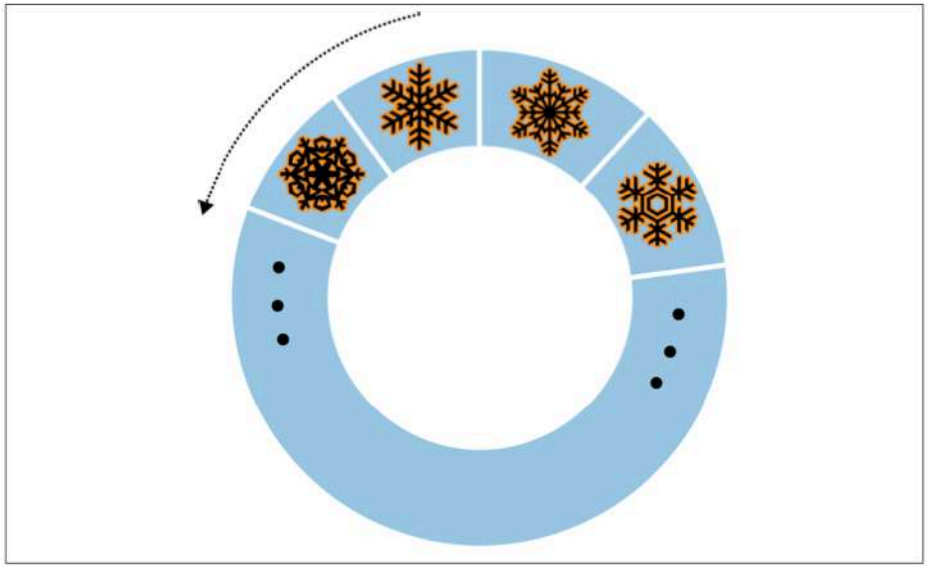


Figure 6-4. Sort by {"$natural" : -1}

Ref: MongoDB: The Definitive Guide

TTL Index

- Automatically remove data after a specified number of seconds.
- For event data, logs, session information.
- MongoDB sweeps the TTL index once per minute

Ref: MongoDB: The Definitive Guide

TTL Index

```
db.ttl_coll.ensureIndex({lastUpdated:1},{expireAfterSeconds:10})

for(i = 0; i < 10 ; i++){
    db.ttl_coll.insert({x:i,lastUpdated:new Date()})
}

findAllAndShow('ttl_coll');
```

Ref: MongoDB: The Definitive Guide

Aggregation(聚合)

```
var findOneAndShow = function(coll_name){
    printjson(db[coll_name].findOne());
}

findOneAndShow('users');

print('------------------------------');

var cursor = db.users.aggregate(
        {$match:{age:15}},
        {$project:{'NAME':'$username',age:1}},
        {$limit:5}
);

showCursorItems(cursor);
```

```
{
        "_id" : ObjectId("539c05971a3a5b2299d862d2"),
        "i" : 0,
        "username" : "user0",
        "age" : 110,
        "created" : ISODate("2014-06-14T08:19:35.356Z")
}
---------------------------
{
        "_id" : ObjectId("539c05c71a3a5b2299d9e9da"),
        "age" : 15,
        "NAME" : "user100104"
}
{
        "_id" : ObjectId("539c05c71a3a5b2299d9e9f3"),
        "age" : 15,
        "NAME" : "user100129"
}
{
        "_id" : ObjectId("539c05c71a3a5b2299d9ea72"),
        "age" : 15,
        "NAME" : "user100256"
}
{
        "_id" : ObjectId("539c05c71a3a5b2299d9eaa7"),
        "age" : 15,
        "NAME" : "user100309"
}
{
        "_id" : ObjectId("539c05c71a3a5b2299d9eb28"),
        "age" : 15,
        "NAME" : "user100438"
}
```

$project

```
var cursor = db.users.aggregate(
        {$match:{age:15}},
        {$project:{
                    'NAME':'$username',
                    'add100Years':{$add:['$age',100]}
                    }
        },
        {$limit:5}
);

showCursorItems(cursor);
```

```
> db.employees.aggregate(
... {
...     "$project" : {
...         "totalPay" : {
...             "$add" : ["$salary", "$bonus"]
...         }
...     }
... })
```

```
> db.employees.aggregate(
... {
...     "$project" : {
...         "totalPay" : {
...             "$subtract" : [{"$add" : ["$salary", "$bonus"]}, "$401k"]
...         }
...     }
... })
```

Ref: MongoDB: The Definitive Guide

```
> db.employees.aggregate(
... {
...     "$project" : {
...         "hiredIn" : {"$month" : "$hireDate"}
...     }
... })
```

```
> db.employees.aggregate(
... {
...     "$project" : {
...         "tenure" : {
...             "$subtract" : [{"$year" : new Date()}, {"$year" : "$hireDate"}]
...         }
...     }
... })
```

Ref: MongoDB: The Definitive Guide

```
> db.employees.aggregate(
... {
...     "$project" : {
...         "email" : {
...             "$concat" : [
...                 {"$substr" : ["$firstName", 0, 1]},
...                 ".",
...                 "$lastName",
...                 "@example.com"
...             ]
...         }
...     }
... })
```

```
> db.students.aggregate(
... {
...     "$project" : {
...         "grade" : {
...             "$cond" : [
...                 "$teachersPet",
...                 100, // if
...                 {    // else
...                     "$add" : [
...                         {"$multiply" : [.1, "$attendanceAvg"]},
...                         {"$multiply" : [.3, "$quizzAvg"]},
...                         {"$multiply" : [.6, "$testAvg"]}
...                     ]
...                 }
...             ]
...         }
...     }
... })
```

group

```
var cursor = db.users.aggregate(
                {
                    $group:{_id:'$age', count : { $sum : 1 }}
                },
                {$sort:{'_id':1}}
)
showCursorItems(cursor);
```

```
{ "_id" : 0, "count" : 8231 }
{ "_id" : 1, "count" : 8599 }
{ "_id" : 2, "count" : 8274 }
{ "_id" : 3, "count" : 8403 }
{ "_id" : 4, "count" : 8180 }
{ "_id" : 5, "count" : 8431 }
{ "_id" : 6, "count" : 8471 }
{ "_id" : 7, "count" : 8406 }
{ "_id" : 8, "count" : 8318 }
{ "_id" : 9, "count" : 8329 }
{ "_id" : 10, "count" : 8311 }
{ "_id" : 11, "count" : 8405 }
```

```
db.article.aggregate(
    { $group : {
        _id : "$author",
        docsPerAuthor : { $sum : 1 },
        viewsPerAuthor : { $sum : "$pageViews" }
    }}
);
```

```
> db.scores.aggregate(
... {
...     "$group" : {
...         "_id" : "$grade",
...         "lowestScore" : {"$min" : "$score"},
...         "highestScore" : {"$max" : "$score"}

...     }
... })
```

Ref: MongoDB: The Definitive Guide

```
> db.scores.aggregate(
... {
...     "$sort" : {"score" : 1}
... },
... {
...     "$group" : {
...         "_id" : "$grade",
...         "lowestScore" : {"$first" : "$score"},
...         "highestScore" : {"$last" : "$score"}
...     }
... })
```

Ref: MongoDB: The Definitive Guide

```
> db.employees.aggregate(
... {
...     "$project" : {
...         "compensation" : {
...             "$add" : ["$salary", "$bonus"]
...         },
...         "name" : 1
...     }
... },
... {
...     "$sort" : {"compensation" : -1, "name" : 1}
... })
```

Ref: MongoDB: The Definitive Guide

MapReduce

- MapReduce uses JavaScript as its “query language” so it can express arbitrarily complex logic.
- MapReduce tends to be fairly slow and should not be used for real-time data analysis.

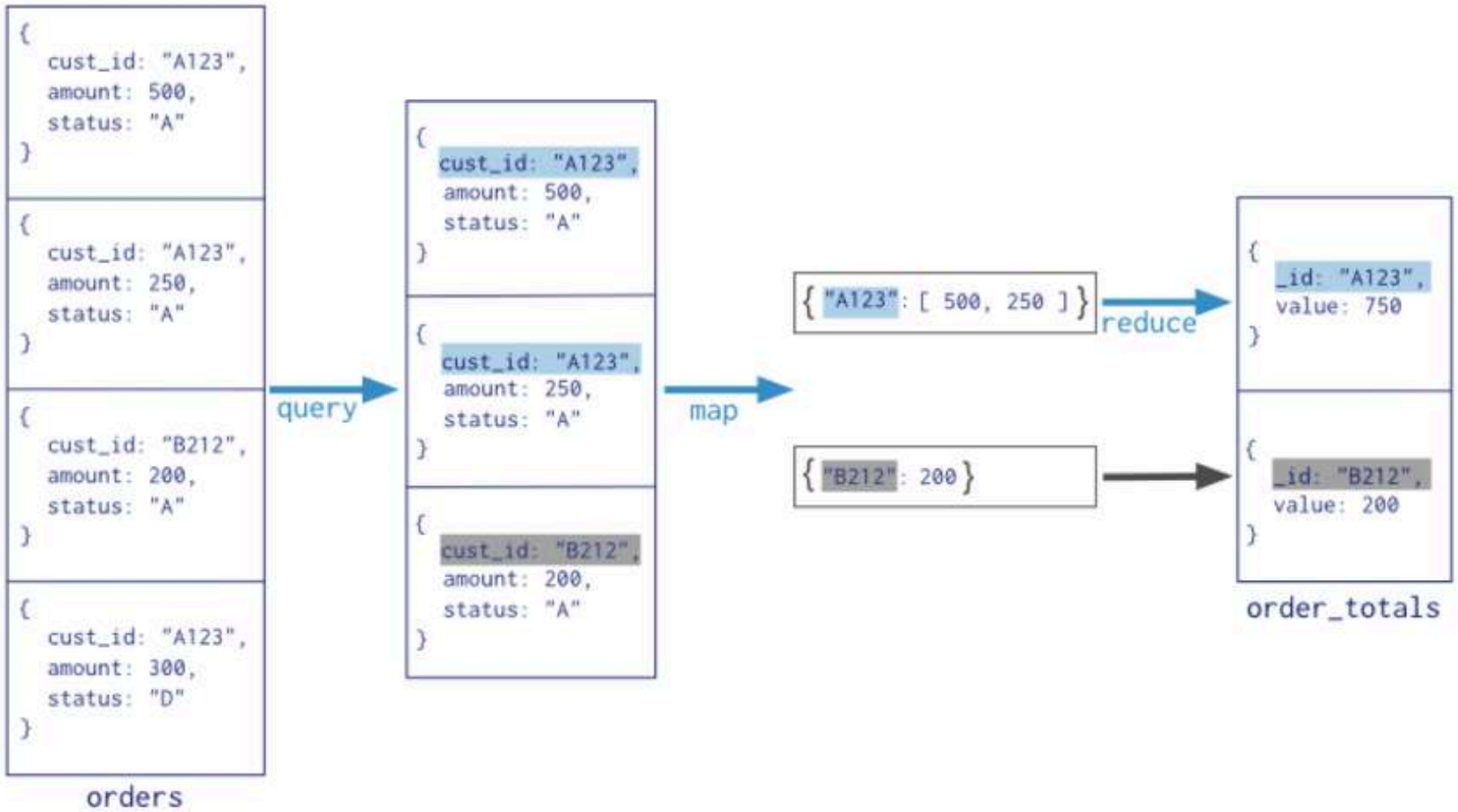
Ref: MongoDB: The Definitive Guide

```
db.mr1.insert(
    [
        {x:10},
        {y:10},
        {z:10},
        {w:10},
        {x:10,w:20}
    ]
);

var map = function(){
    for(var key in this){
        emit(key,{count:1});
    }
}
var reduce = function(key,emits){
    total = 0;
    for(var i in emits){
        total+=emits[i].count;
    }
    return {count:total};
}

var mrResult = db.runCommand({'mapreduce':'mr1','map':map,'reduce':reduce,"out":{inline:1}});
var mrResult = db.mr1.mapReduce(map,reduce,{out:{inline:1}})

printjson(mrResult.results)
```


[
{
"_id" : "_id",
"value" : {
"count" : 5
}
},
{
"_id" : "w",
"value" : {
"count" : 2
}
},
{
"_id" : "x",
"value" : {
"count" : 2
}
},
{
"_id" : "y",
"value" : {
"count" : 1
}
},
{
"_id" : "z",
"value" : {
"count" : 1
}
}
]

Collection

```
db.orders.mapReduce(
        map      →  function() { emit( this.cust_id, this.amount ); },
        reduce   →  function(key, values) { return Array.sum( values ) },
                    {
        query    →     query: { status: "A" },
        output   →     out: "order_totals"
                    }
                )
```



```
map = function() {
    for (var i in this.tags) {
        var recency = 1/(new Date() - this.date);
        var score = recency * this.score;

        emit(this.tags[i], {"urls" : [this.url], "score" : score});
    }
};
```

```
reduce = function(key, emits) {
    var total = {urls : [], score : 0}
    for (var i in emits) {
        emits[i].urls.forEach(function(url) {
            total.urls.push(url);
        }
        total.score += emits[i].score;
    }
    return total;
};
```

Ref: MongoDB: The Definitive Guide

```
> db.runCommand({"mapreduce" : "analytics", "map" : map, "reduce" : reduce,
                 "query" : {"date" : {"$gt" : week_ago}}})
```

```
> db.runCommand({"mapreduce" : "analytics", "map" : map, "reduce" : reduce,
                 "limit" : 10000, "sort" : {"date" : -1}})
```

Ref: MongoDB: The Definitive Guide

count

```
> db.foo.count()
0
> db.foo.insert({"x" : 1})
> db.foo.count()
1


> db.foo.insert({"x" : 2})
> db.foo.count()
2
> db.foo.count({"x" : 1})
1
```

Ref: MongoDB: The Definitive Guide

distinct

```
> db.runCommand({"distinct" : "people", "key" : "age"})
```

For example, suppose we had the following documents in our collection:

```
{"name" : "Ada", "age" : 20}
{"name" : "Fred", "age" : 35}
{"name" : "Susan", "age" : 60}
{"name" : "Andy", "age" : 35}
```

If you call distinct on the "age" key, you will get back all of the distinct ages:

```
> db.runCommand({"distinct" : "people", "key" : "age"})
{"values" : [20, 35, 60], "ok" : 1}
```

Ref: MongoDB: The Definitive Guide

Application Design

- MongoDB has **no joining facilities**, so gathering documents from multiple collections will require multiple queries.
- ***Normalization*** is dividing up data into multiple collections with references between collections. (**require multiple queries**)
- ***Denormalization*** is the opposite of normalization: embedding all of the data in a single document. (**single query**)

Ref: MongoDB: The Definitive Guide

```
> db.studentClasses.findOne({"studentId" : id})
{
    "_id" : ObjectId("512512c1d86041c7dca81915"),
    "studentId" : ObjectId("512512a5d86041c7dca81914"),
    "classes" : [
        ObjectId("512512ced86041c7dca81916"),
        ObjectId("512512dcd86041c7dca81917"),
        ObjectId("512512e6d86041c7dca81918"),
        ObjectId("512512f0d86041c7dca81919")
    ]
}
```

```
{
    "_id" : ObjectId("512512a5d86041c7dca81914"),
    "name" : "John Doe",
    "classes" : [
        ObjectId("512512ced86041c7dca81916"),
        ObjectId("512512dcd86041c7dca81917"),
        ObjectId("512512e6d86041c7dca81918"),
        ObjectId("512512f0d86041c7dca81919")
    ]
}
```

Ref: MongoDB: The Definitive Guide

```
{
    "_id" : ObjectId("512512a5d86041c7dca81914"),
    "name" : "John Doe",
    "classes" : [
        {
            "class" : "Trigonometry",
            "credits" : 3,
            "room" : "204"
        },
        {
            "class" : "Physics",
            "credits" : 3,
            "room" : "159"
        },
        {
            "class" : "Women in Literature",
            "credits" : 3,
            "room" : "14b"
        },
        {
            "class" : "AP European History",
            "credits" : 4,
            "room" : "321"
        }
    ]
}
```

```
{
    "_id" : ObjectId("512512a5d86041c7dca81914"),
    "name" : "John Doe",
    "classes" : [
        {
            "_id" : ObjectId("512512ced86041c7dca81916"),
            "class" : "Trigonometry"
        },
        {
            "_id" : ObjectId("512512dcd86041c7dca81917"),
            "class" : "Physics"
        },
        {
            "_id" : ObjectId("512512e6d86041c7dca81918"),
            "class" : "Women in Literature"
        },
        {
            "_id" : ObjectId("512512f0d86041c7dca81919"),
            "class" : "AP European History"
        }
    ]
}
```

Embedding or Reference

Embedding is better for...	References are better for...
Small subdocuments	Large subdocuments
Data that does not change regularly	Volatile data
When eventual consistency is acceptable	When immediate consistency is necessary
Documents that grow by a small amount	Documents that grow a large amount
Data that you'll often need to perform a second query to fetch	Data that you'll often exclude from the results
Fast reads	Fast writes

whether or not they should be embedded

Account preferences

They are only relevant to this user document, and will probably be exposed with other user information in this document. Account preferences should generally be embedded.

Recent activity

This one depends on how much recent activity grows and changes. If it is a fixed-size field (last 10 things), it might be useful to embed.

Friends

Generally this should not be embedded, or at least not fully. See the section below on advice on social networking.

All of the content this user has produced

No.

Friends, Followers, and Other Inconveniences

```
{
    "_id" : ObjectId("51250a5cd86041c7dca8190f"),
    "username" : "batman",
    "email" : "batman@waynetech.com"
    "following" : [
        ObjectId("51250a72d86041c7dca81910"),
        ObjectId("51250a7ed86041c7dca81936")
    ]
}
```

```
{
    "_id" : ObjectId("51250a7ed86041c7dca81936"),
    "username" : "joker",
    "email" : "joker@mailinator.com"
    "followers" : [
        ObjectId("512510e8d86041c7dca81912"),
        ObjectId("51250a5cd86041c7dca8190f"),
        ObjectId("512510ffd86041c7dca81910")
    ]
}
```

```
{
    "_id" : ObjectId("51250a7ed86041c7dca81936"), // followee's "_id"
    "followers" : [
        ObjectId("512510e8d86041c7dca81912"),
        ObjectId("51250a5cd86041c7dca8190f"),
        ObjectId("512510ffd86041c7dca81910")
    ]
}
```

“continuation” document

```
> db.users.find({"username" : "wil"})
{
    "_id" : ObjectId("51252871d86041c7dca8191a"),
    "username" : "wil",
    "email" : "wil@example.com",
    "tbc" : [
        ObjectId("512528ced86041c7dca8191e"),
        ObjectId("5126510dd86041c7dca81924")
    ]
    "followers" : [
        ObjectId("512528a0d86041c7dca8191b"),
        ObjectId("512528a2d86041c7dca8191c"),
        ObjectId("512528a3d86041c7dca8191d"),
        ...
    ]
}
{
    "_id" : ObjectId("512528ced86041c7dca8191e"),
    "followers" : [
        ObjectId("512528f1d86041c7dca8191f"),
        ObjectId("512528f6d86041c7dca81920"),
        ObjectId("512528f8d86041c7dca81921"),
        ...
    ]
}
{
    "_id" : ObjectId("5126510dd86041c7dca81924"),
    "followers" : [
        ObjectId("512673e1d86041c7dca81925"),
        ObjectId("512650efd86041c7dca81922"),
        ObjectId("512650fdd86041c7dca81923"),
        ...
    ]
}
```

Basics: Modeling One-to-Few

An example of "one-to-few" might be the addresses for a person. This is a good use case for embedding – you'd put the addresses in an array inside of your Person object:

```
> db.person.findOne()
{
  name: 'Kate Monster',
  ssn: '123-456-7890',
  addresses : [
     { street: '123 Sesame St', city: 'Anytown', cc: 'USA' },
     { street: '123 Avenue Q', city: 'New York', cc: 'USA' }
  ]
}
```

Basics: One-to-Many

```
> db.parts.findOne()
{
    _id : ObjectID('AAAA'),
    partno : '123-aff-456',
    name : '#4 grommet',
    qty: 94,
    cost: 0.94,
    price: 3.99
}
```

```
> db.products.findOne()
{
    name : 'left-handed smoke shifter',
    manufacturer : 'Acme Corp',
    catalog_number: 1234,
    parts : [     // array of references to Part documents
        ObjectID('AAAA'),    // reference to the #4 grommet above
        ObjectID('F17C'),    // reference to a different Part
        ObjectID('D2AA'),
        // etc
    ]
```

```
> db.hosts.findOne()
{
    _id : ObjectID('AAAB'),
    name : 'goofy.example.com',
    ipaddr : '127.66.66.66'
}

>db.logmsg.findOne()
{
    time : ISODate("2014-03-28T09:42:41.382Z"),
    message : 'cpu is on fire!',
    host: ObjectID('AAAB')       // Reference to the Host document
}
```

Intermediate: Two-Way Referencing

```
db.person.findOne()
{
    _id: ObjectID("AAF1"),
    name: "Kate Monster",
    tasks [     // array of references to Task documents
        ObjectID("ADF9"),
        ObjectID("AE02"),
        ObjectID("AE73")
        // etc
    ]
}
```

one hosted with ❤ by GitHub view raw

```
db.tasks.findOne()
{
    _id: ObjectID("ADF9"),
    description: "Write lesson plan",
    due_date:  ISODate("2014-04-01"),
    owner: ObjectID("AAF1")     // Reference to Person document
}
```

```
> db.products.findOne()
{
    name : 'left-handed smoke shifter',
    manufacturer : 'Acme Corp',
    catalog_number: 1234,
    parts : [
        { id : ObjectID('AAAA'), name : '#4 grommet' },
        { id: ObjectID('F17C'), name : 'fan blade assembly' },
        { id: ObjectID('D2AA'), name : 'power switch' },
        // etc
    ]
}
```

```
> db.parts.findOne()
{
    _id : ObjectID('AAAA'),
    partno : '123-aff-456',
    name : '#4 grommet',
    product_name : 'left-handed smoke shifter',
    product_catalog_number: 1234,
    qty: 94,
    cost: 0.94,
    price: 3.99
}
```

MongoDB Limits and Thresholds

- The maximum BSON document size is 16 megabytes.
- MongoDB supports no more than 100 levels of nesting for BSON documents.
- A single collection can have no more than 64 indexes.

Ref: MongoDB: The Definitive Guide

Optimizations

- Optimizing reads: correct indexes and returning as much of the information as possible in a single document.
- Optimizing writes :minimizing the number of indexes and making updates as efficient as possible.

Ref: MongoDB: The Definitive Guide

```
{
    "_id" : ObjectId(),
    "restaurant" : "Le Cirque",
    "review" : "Hamburgers were overpriced."
    "userId" : ObjectId(),
    "tags" : []
}
```

The "tags" field will grow as users add tags, so the application will often have to perform an update like this:

```
> db.reviews.update({"_id" : id},
... {"$push" : {"tags" : {"$each" : ["French", "fine dining", "hamburgers"]}}}})
```

Ref: MongoDB: The Definitive Guide

```
{
    "_id" : ObjectId(),
    "restaurant" : "Le Cirque",
    "review" : "Hamburgers were overpriced."
    "userId" : ObjectId(),
    "tags" : [],
    "garbage" : "..................................................."+
        "..................................................."+
        "..................................................."
}
```

When you update the document, always "$unset" the "garbage" field:

```
> db.reviews.update({"_id" : id},
... {"$push" : {"tags" : {"$each" : ["French", "fine dining", "hamburgers"]}}},
...  "$unset" : {"garbage" : true}})
```

The "$unset" will remove the "garbage" field if it exists and be a no-op if it does not.

Ref: MongoDB: The Definitive Guide

Removing Old Data

- Use a capped collection
- TTL collections
- Use multiple collections: for example, one collection per month.

Consistency

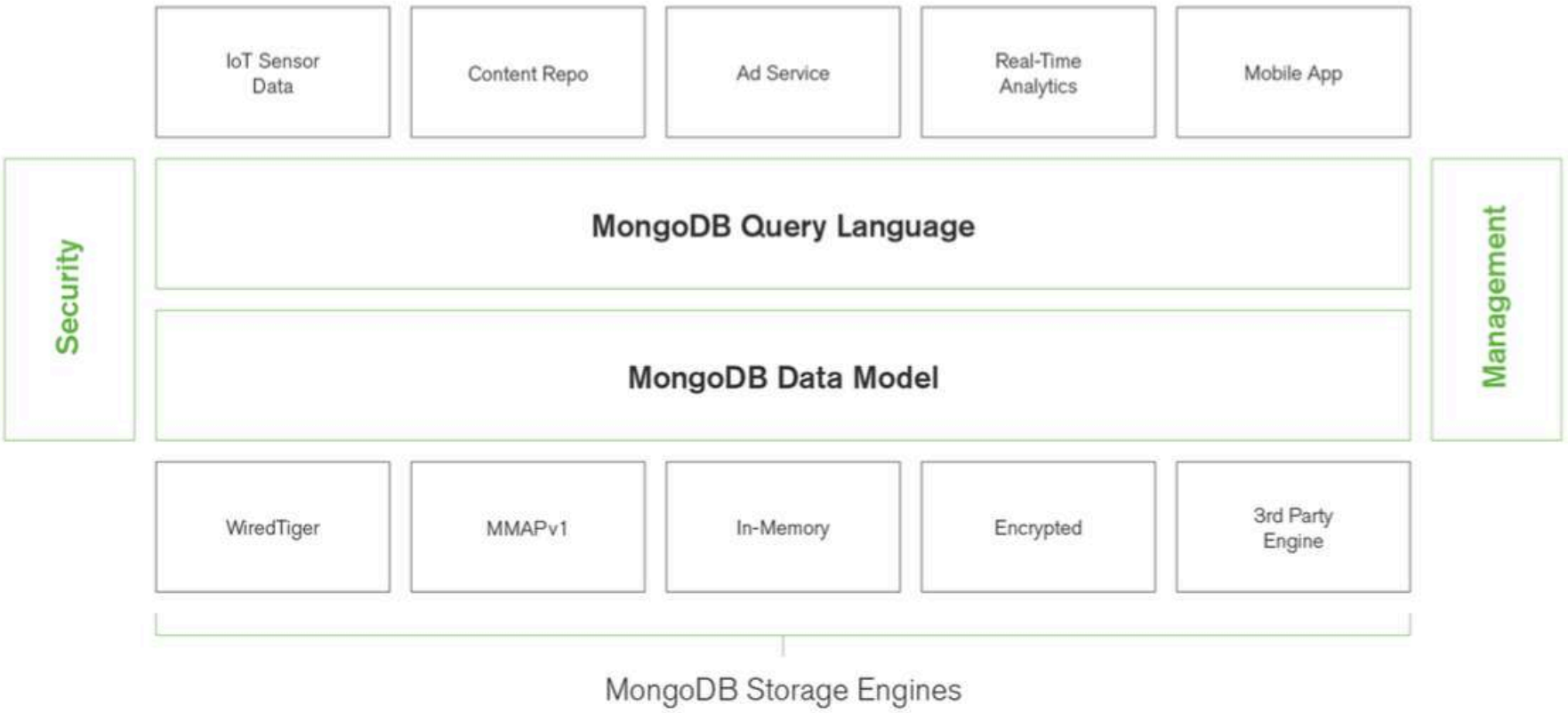
- The server keeps a queue of requests for each connection.
- A single connection has a consistent view of the database and can always read its own writes.
- use connection pool
- reads to a replica set

Migrating Schemas

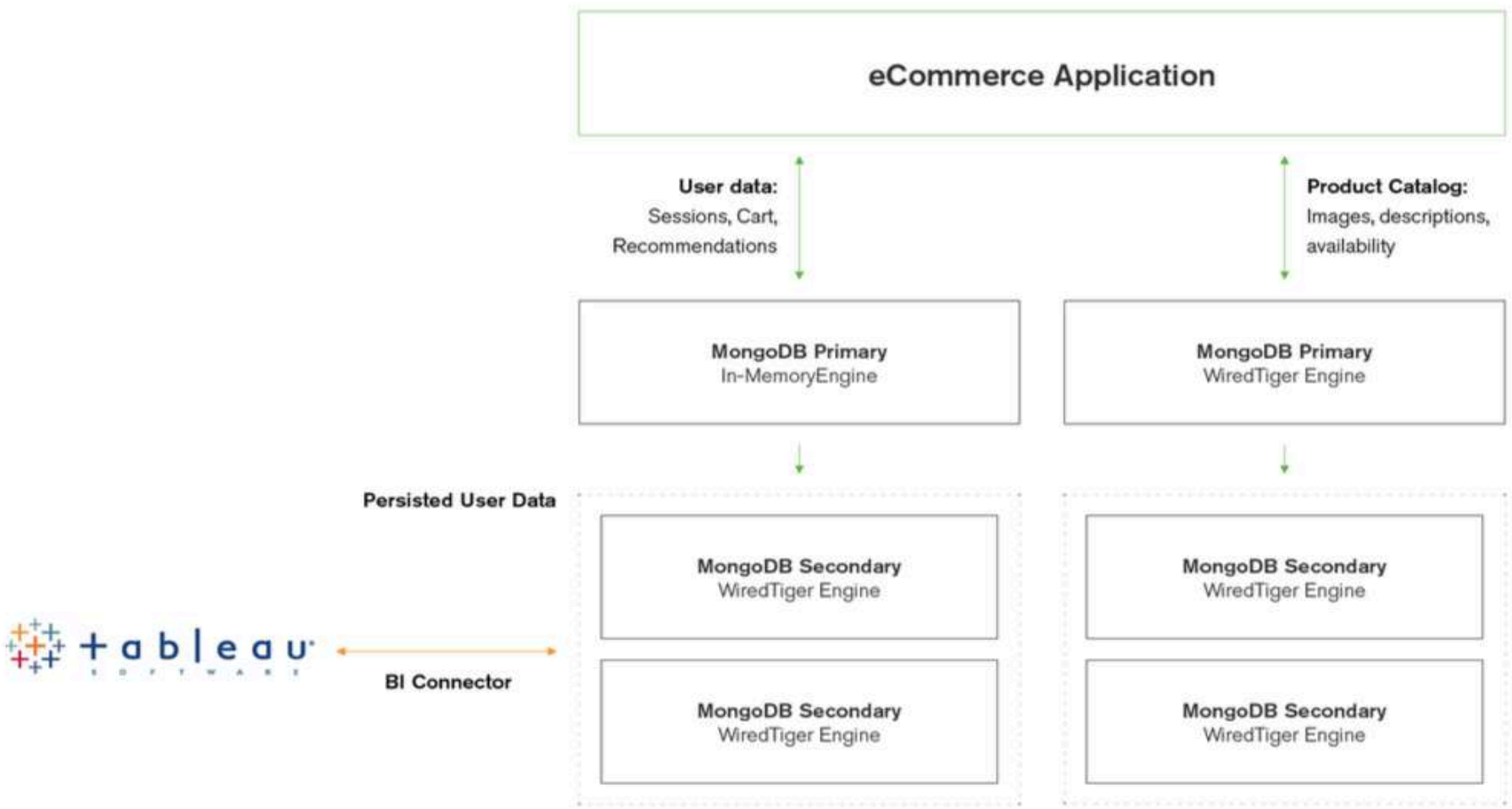
- Make sure that your application supports all old versions of the schema.

- Include a "version" field (or just "v") in each document and use that to determine what your application will accept for document structure.

MongoDB 3.2


IoT Sensor Data
Content Repo
Ad Service
Real-Time Analytics
Mobile App
Security
MongoDB Query Language
MongoDB Data Model
Management
WiredTiger
MMAPv1
In-Memory
Encrypted
3rd Party Engine
MongoDB Storage Engines

MongoDB 3.2


eCommerce Application
User data:
Sessions, Cart,
Recommendations
Product Catalog:
Images, descriptions,
availability
MongoDB Primary
In-MemoryEngine
MongoDB Primary
WiredTiger Engine
Persisted User Data
MongoDB Secondary
WiredTiger Engine
MongoDB Secondary
WiredTiger Engine
MongoDB Secondary
WiredTiger Engine
MongoDB Secondary
WiredTiger Engine
tableau
BI Connector

MongoDB 3.2

```
db.createCollection( "contacts",
   { validator: { $or:
      [
         { phone: { $type: "string" } },
         { email: { $regex: /@mongodb\.com$/ } },
         { status: { $in: [ "Unknown", "Incomplete" ] } }
      ]
   }
} )
```

MongoDB 3.2

```
db.runCommand({
    collMod: "contacts",
    validator: {
        $and: [
            {year_of_birth: {$lte: 1994}},
            {$or: [
                {phone: { $type: "string" }},
                {email: { $type: "string" }}
    ]}]
})
```

$lookup (aggregation)

```
{
   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection>,
       as: <output array field>
     }
}
```

Create View ¶

To create or define a view, MongoDB 3.4 introduces:

- the **viewOn** and **pipeline** options to the existing `create` command (and `db.createCollection` helper):

```
db.runCommand( { create: <view>, viewOn: <source>, pipeline: <pipeline> } )
```

- a new `mongo` shell helper `db.createView()`:

```
db.createView(<view>, <source>, <pipeline>, <collation> )
```

Replication

- *Replication* is a way of keeping identical copies of your data on multiple servers and is recommended for **all production deployments**.
- A replica set is a group of servers with one *primary*, the server taking client requests, and **multiple *secondaries***, servers that keep copies of the primary's data.

```
PRIMARY> db.isMaster()
{
        "setName" : "rs-1",
        "setVersion" : 3,
        "ismaster" : true,
        "secondary" : false,
        "hosts" : [
                "justin-c-huang.local:27095",
                "justin-c-huang.local:27097",
                "justin-c-huang.local:27096"
        ],
        "primary" : "justin-c-huang.local:27095",
        "me" : "justin-c-huang.local:27095",
        "maxBsonObjectSize" : 16777216,
        "maxMessageSizeBytes" : 48000000,
        "maxWriteBatchSize" : 1000,
        "localTime" : ISODate("2014-06-15T06:27:19.878Z"),
        "maxWireVersion" : 2,
        "minWireVersion" : 0,
        "ok" : 1
}
```

```
rs-1:SECONDARY> db.isMaster()
{
        "setName" : "rs-1",
        "setVersion" : 3,
        "ismaster" : false,
        "secondary" : true,
        "hosts" : [
                "justin-c-huang.local:27096",
                "justin-c-huang.local:27097",
                "justin-c-huang.local:27095"
        ],
        "primary" : "justin-c-huang.local:27095",
        "me" : "justin-c-huang.local:27096",
        "maxBsonObjectSize" : 16777216,
        "maxMessageSizeBytes" : 48000000,
        "maxWriteBatchSize" : 1000,
        "localTime" : ISODate("2014-06-15T06:29:13.791Z"),
        "maxWireVersion" : 2,
        "minWireVersion" : 0,
        "ok" : 1
}
```

```
apple@justin-c-huang:bin$ ./mongo 127.0.0.1:27095
MongoDB shell version: 2.6.1
connecting to: 127.0.0.1:27095/test
Server has startup warnings:
2014-06-15T14:24:41.301+0800 [initandlisten]
2014-06-15T14:24:41.301+0800 [initandlisten] ** WARNING: soft rlimits too low. Number of files is 256, should
rs-1:PRIMARY>
rs-1:PRIMARY> db
test
rs-1:PRIMARY> for(i=0;i<100;i++){db.coll.insert({count:i})}
WriteResult({ "nInserted" : 1 })
rs-1:PRIMARY> db.coll.count()
100
rs-1:PRIMARY> exit
bye
apple@justin-c-huang:bin$ ./mongo 127.0.0.1:27096
MongoDB shell version: 2.6.1
connecting to: 127.0.0.1:27096/test
Server has startup warnings:
2014-06-15T14:25:32.436+0800 [initandlisten]
2014-06-15T14:25:32.436+0800 [initandlisten] ** WARNING: soft rlimits too low. Number of files is 256, should
rs-1:SECONDARY> db.coll.count()
2014-06-15T14:32:43.851+0800 count failed: { "note" : "from execCommand", "ok" : 0, "errmsg" : "not master" }
rs-1:SECONDARY> rs.slaveOk()
rs-1:SECONDARY> db.coll.count()
100
rs-1:SECONDARY>
```

```
rs-1:PRIMARY> db.adminCommand({'shutdown':1})
2014-06-15T14:39:08.728+0800 DBClientCursor::init call() failed
2014-06-15T14:39:08.730+0800 Error: error doing query: failed at src/mongo/shell/query.js:8
2014-06-15T14:39:08.732+0800 trying reconnect to 127.0.0.1:27095 (127.0.0.1) failed
2014-06-15T14:39:08.732+0800 warning: Failed to connect to 127.0.0.1:27095, reason: errno:6
2014-06-15T14:39:08.732+0800 reconnect 127.0.0.1:27095 (127.0.0.1) failed failed couldn't c
ction attempt failed
2014-06-15T14:39:08.735+0800 trying reconnect to 127.0.0.1:27095 (127.0.0.1) failed
2014-06-15T14:39:08.735+0800 warning: Failed to connect to 127.0.0.1:27095, reason: errno:6
2014-06-15T14:39:08.735+0800 reconnect 127.0.0.1:27095 (127.0.0.1) failed failed couldn't c
ction attempt failed
> exit
bye
apple@justin-c-huang:bin$ ./mongo 127.0.0.1:27095
MongoDB shell version: 2.6.1
connecting to: 127.0.0.1:27095/test
2014-06-15T14:39:29.307+0800 warning: Failed to connect to 127.0.0.1:27095, reason: errno:6
2014-06-15T14:39:29.308+0800 Error: couldn't connect to server 127.0.0.1:27095 (127.0.0.1),
.js:148
exception: connect failed
apple@justin-c-huang:bin$ ./mongo 127.0.0.1:27096
MongoDB shell version: 2.6.1
connecting to: 127.0.0.1:27096/test
Server has startup warnings:
2014-06-15T14:25:32.436+0800 [initandlisten]
2014-06-15T14:25:32.436+0800 [initandlisten] ** WARNING: soft rlimits too low. Number of fi
rs-1:SECONDARY> exit
bye
apple@justin-c-huang:bin$ ./mongo 127.0.0.1:27097
MongoDB shell version: 2.6.1
connecting to: 127.0.0.1:27097/test
Server has startup warnings:
2014-06-15T14:26:16.757+0800 [initandlisten]
2014-06-15T14:26:16.757+0800 [initandlisten] ** WARNING: soft rlimits too low. Number of fi
rs-1:PRIMARY>
```

```
rs-1:SECONDARY> rs.add('justin-c-huang.local:27098')
{
        "ok" : 0,
        "errmsg" : "replSetReconfig command must be sent to the current replica set pr
}
rs-1:SECONDARY> exit
bye
apple@justin-c-huang:bin$ ./mongo 127.0.0.1:27097
MongoDB shell version: 2.6.1
connecting to: 127.0.0.1:27097/test
Server has startup warnings:
2014-06-15T14:26:16.757+0800 [initandlisten]
2014-06-15T14:26:16.757+0800 [initandlisten] ** WARNING: soft rlimits too low. Number
rs-1:PRIMARY> rs.add('justin-c-huang.local:27098')
{ "ok" : 1 }
rs-1:PRIMARY> rs.config()
{
        "_id" : "rs-1",
        "version" : 4,
        "members" : [
                {
                        "_id" : 0,
                        "host" : "justin-c-huang.local:27095"
                },
                {
                        "_id" : 1,
                        "host" : "justin-c-huang.local:27096"
                },
                {
                        "_id" : 2,
                        "host" : "justin-c-huang.local:27097"
                },
                {
                        "_id" : 3,
                        "host" : "justin-c-huang.local:27098"
                }
        ]
}
```

Majority

Number of members in the set	Majority of the set
1	1
2	2
3	2
4	3
5	3
6	4
7	4

Ref: MongoDB: The Definitive Guide

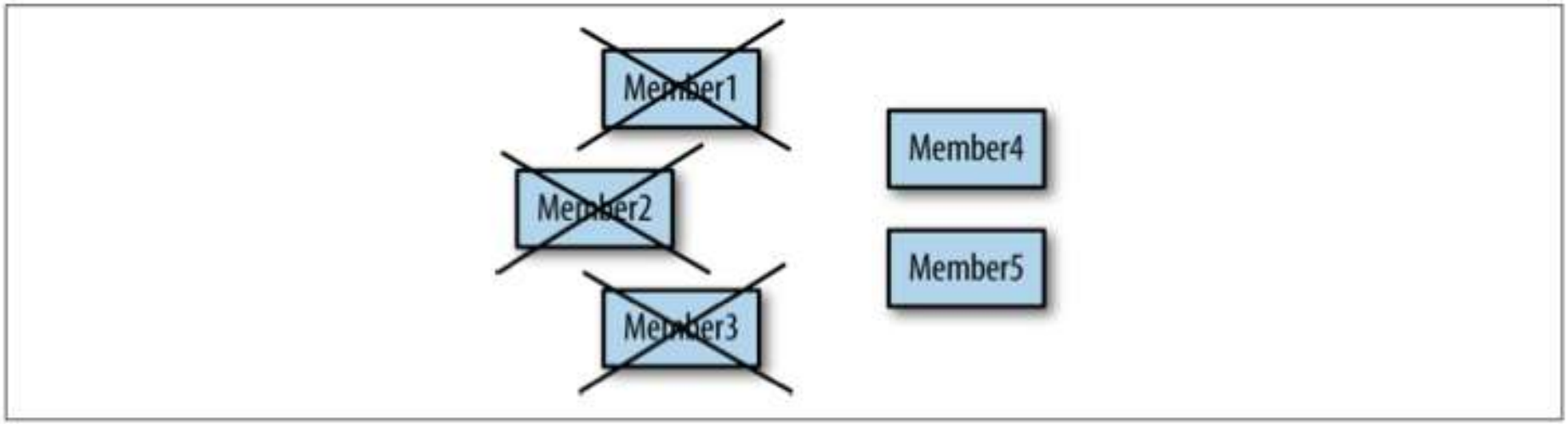


Figure 9-1. With a minority of the set available, all members will be secondaries

Ref: MongoDB: The Definitive Guide

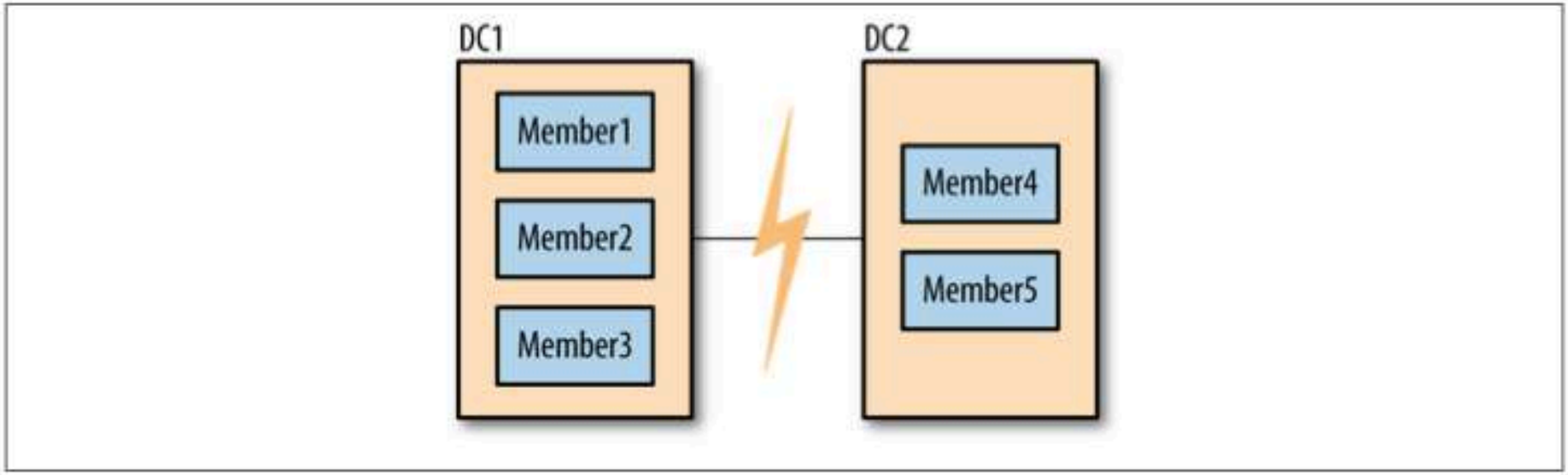


Figure 9-2. For the members, a network partition looks identical to servers on the other side of the partition going down

Ref: MongoDB: The Definitive Guide

How Elections Work

- When a secondary cannot reach a primary, it will contact all the other members and request that it be elected primary.
- These other members do several sanity checks:

1. Secondary nodes可否連到primary node.
2. 要選舉的那個nodes資料是不是最新的
3. 有沒有其他更高優先權的nodes

Election Arbiters

- Arbiter only purpose is to **participate in elections**. Arbiters **hold no data**.
- arbiter as a lightweight process

```
> rs.addArb("server-5:27017")
```

Equivalently, you can specify the `arbiterOnly` option in the member configuration:

```
> rs.add({"_id" : 4, "host" : "server-5:27017", "arbiterOnly" : true})
```

Ref: MongoDB: The Definitive Guide

Priority

The highest-priority member will always be elected primary (so long as they can reach a majority of the set and have the most up-to-date data). For example, suppose you add a member with priority of 1.5 to the set, like so:

```
> rs.add({"_id" : 4, "host" : "server-4:27017", "priority" : 1.5})
```

Ref: MongoDB: The Definitive Guide

Hidden node and slaveDelay node

```
cfg = rs.conf()
cfg.members[0].priority = 0
cfg.members[0].hidden = true
cfg.members[0].slaveDelay = 3600
rs.reconfig(cfg)
```

Building Indexes

- Sometimes a secondary does not need to have the same (or any) indexes that exist on the primary.
- If you are using a secondary only for backup data or offline batch jobs, you might want to specify "buildIndexes" : false
- This is a permanent setting

Ref: MongoDB: The Definitive Guide

oplog

- MongoDB accomplishes this is by keeping a log of operations, or *oplog*, **containing every write that a primary performs.**
- This is a capped collection that lives in the *local* database on the primary. The secondaries query this collection for operations to replicate.
- Each secondary maintains its own oplog
- This allows any member to be used as a sync source for any other member

Ref: MongoDB: The Definitive Guide

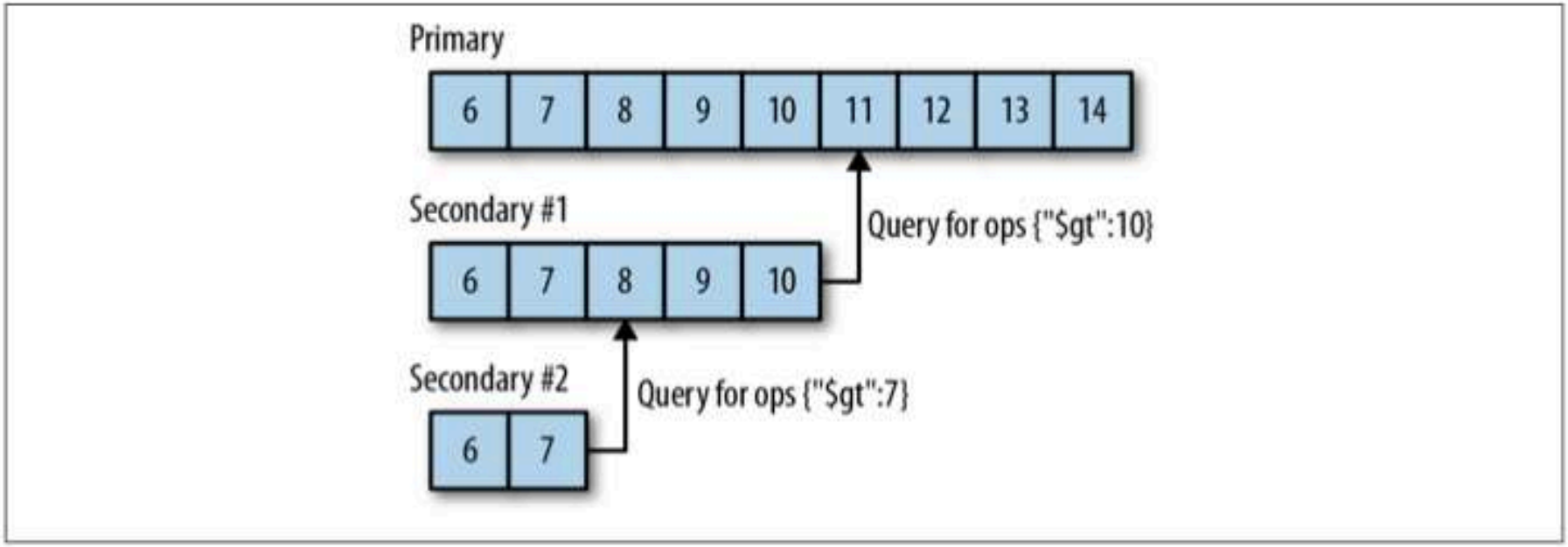


Figure 10-1. Oplog keep an ordered list of write operations that have occurred. Each member has its own copy of the oplog, which should be identical to the primary's (modulo some lag).

Ref: MongoDB: The Definitive Guide

Heartbeats

- A heartbeat request is a short message that checks everyone's state.

- One of the most important functions of heartbeats is to **let the primary know if it can reach a majority of the set**. If a primary can no longer reach a majority of the servers, it will demote itself and become a secondary.

Ref: MongoDB: The Definitive Guide

Rollbacks

- if a primary does a write and goes down before the secondaries have a chance to replicate it, the next primary elected may not have the write.

Ref: MongoDB: The Definitive Guide

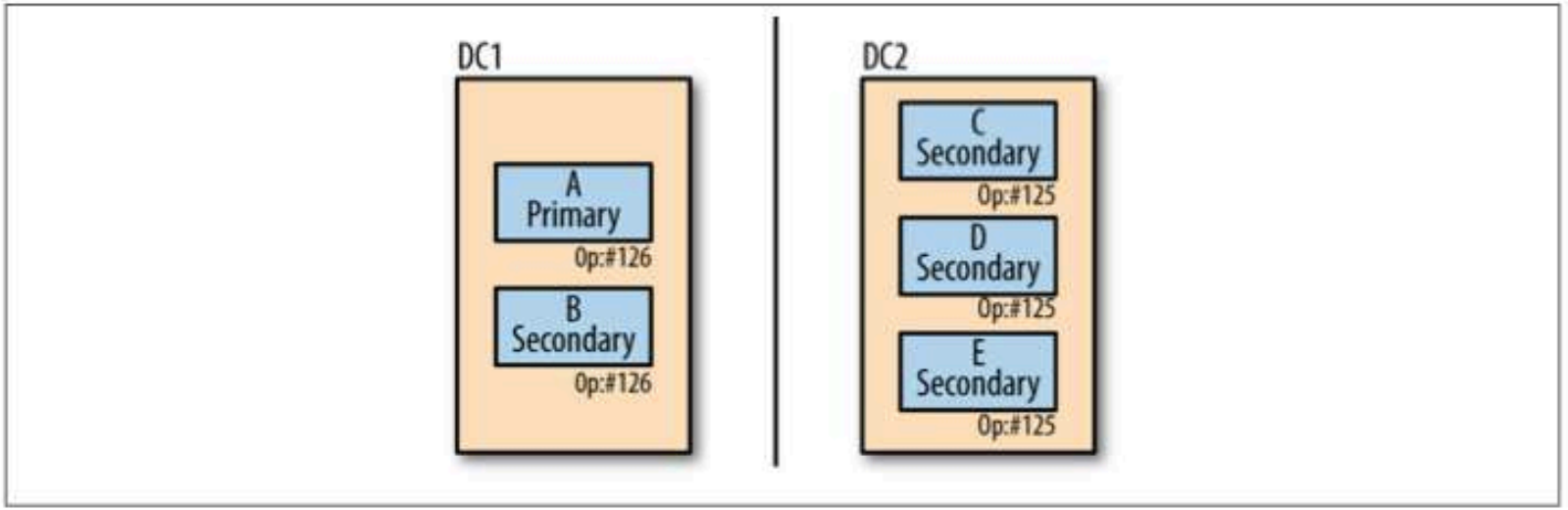


Figure 10-3. Replication across data centers can be slower than within a single data center

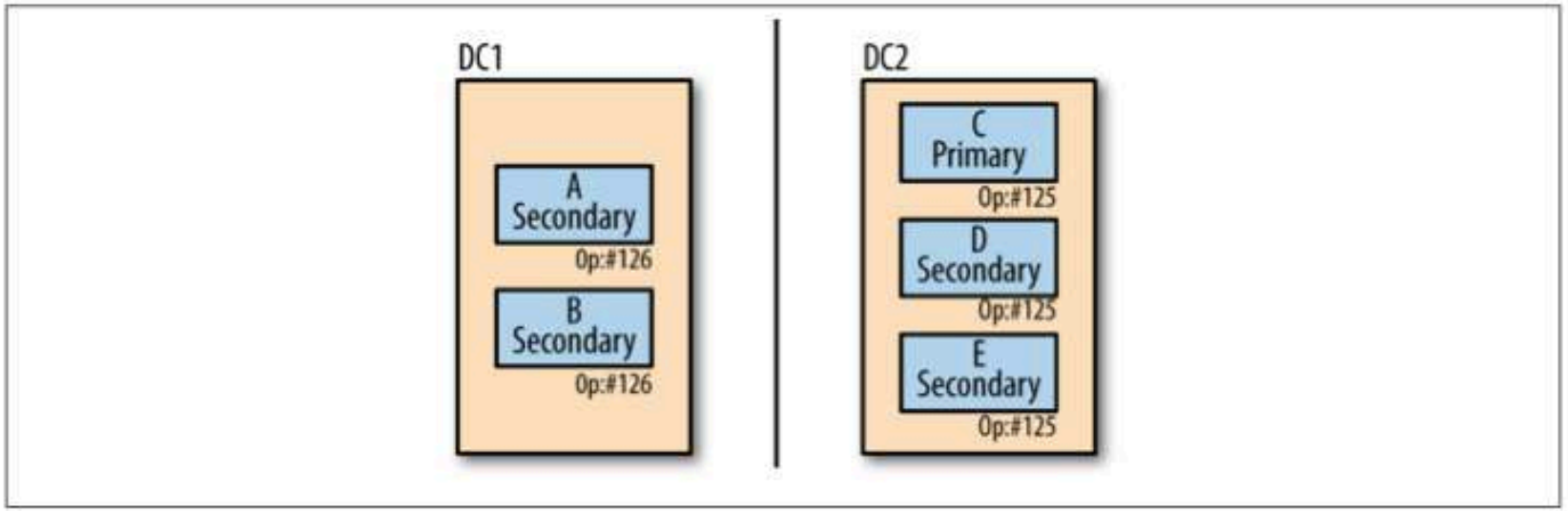


Figure 10-4. Unreplicated writes won't match writes on the other side of a network partition

Ref: MongoDB: The Definitive Guide

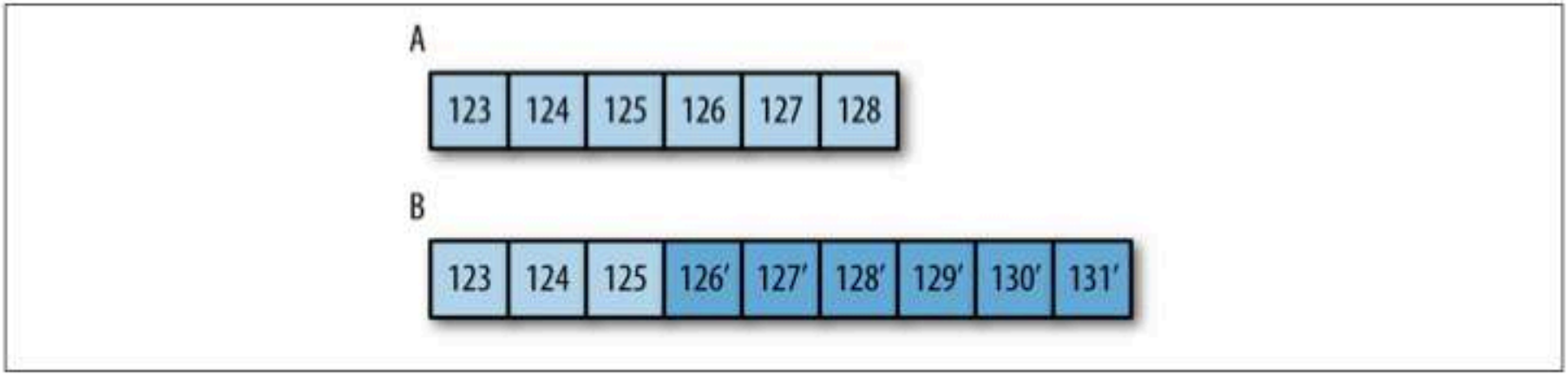


Figure 10-5. Two members with conflicting oplogs: A apparently crashed before replicating ops 126–128, so these operations are not present on B, which has more recent operations. A will have to rollback these three operations before resuming syncing.

Ref: MongoDB: The Definitive Guide

Sending Reads to Secondaries

- By default, drivers will route all requests to the primary.
- Applications that require strongly consistent reads should not read from secondaries.
- Load Considerations
- Application to still be able to perform reads if the primary goes down
- Primary preferred vs. secondary preferred

Manipulating Member State

You can demote a primary to a secondary using the `stepDown` function:

```
> rs.stepDown()
```

```
> rs.freeze(10000)
```

Again, this takes a number of seconds to remain secondary.

Visualizing the Replication Graph

```
> server1.adminCommand({replSetGetStatus: 1})['syncingTo']
server0:27017
> server2.adminCommand({replSetGetStatus: 1})['syncingTo']
server1:27017
> server3.adminCommand({replSetGetStatus: 1})['syncingTo']
server1:27017
> server4.adminCommand({replSetGetStatus: 1})['syncingTo']
server2:27017
```

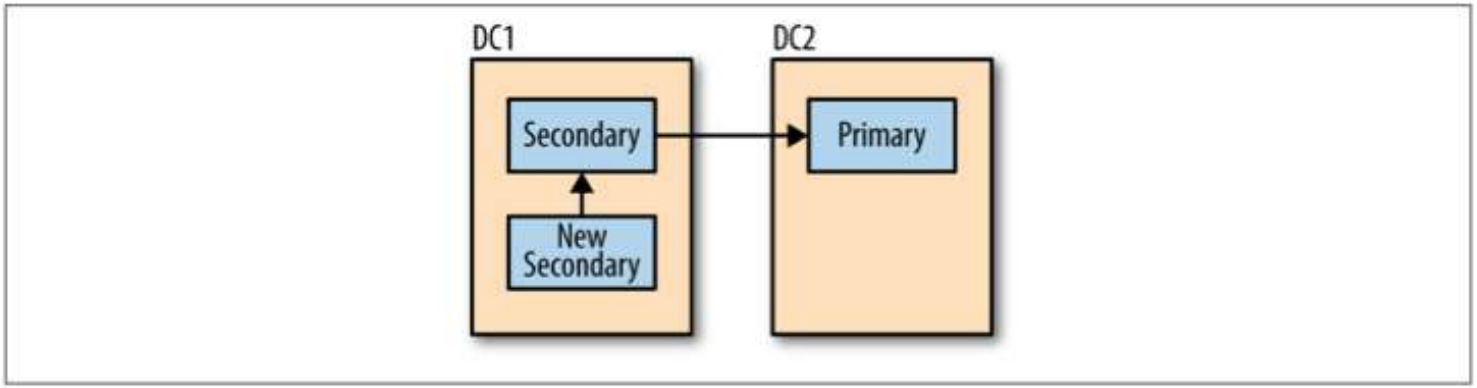


Figure 12-1. New secondaries will generally choose to sync from a member in the same data center

Ref: MongoDB: The Definitive Guide

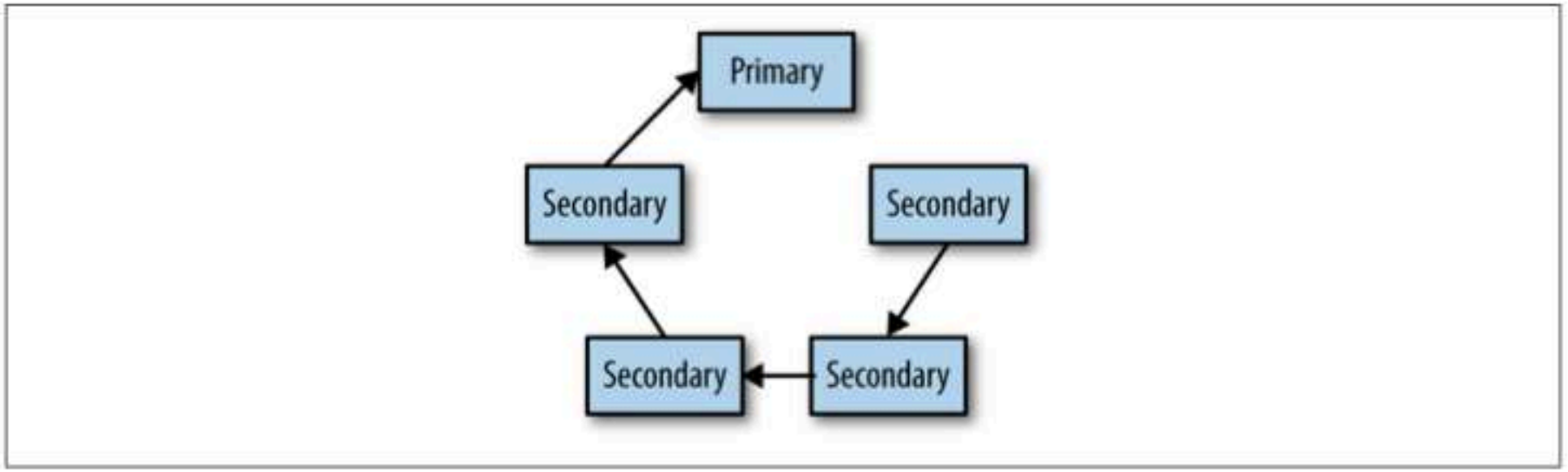


Figure 12-2. As replication chains get longer, it takes longer for all members to get a copy of the data

```
> secondary.adminCommand({"replSetSyncFrom" : "server0:27017"})
```

Ref: MongoDB: The Definitive Guide

Sharding

- *Sharding* refers to the process of splitting data up across machines
- By putting a subset of data on each machine, it becomes possible to store more data and handle more load without requiring larger or more powerful machines

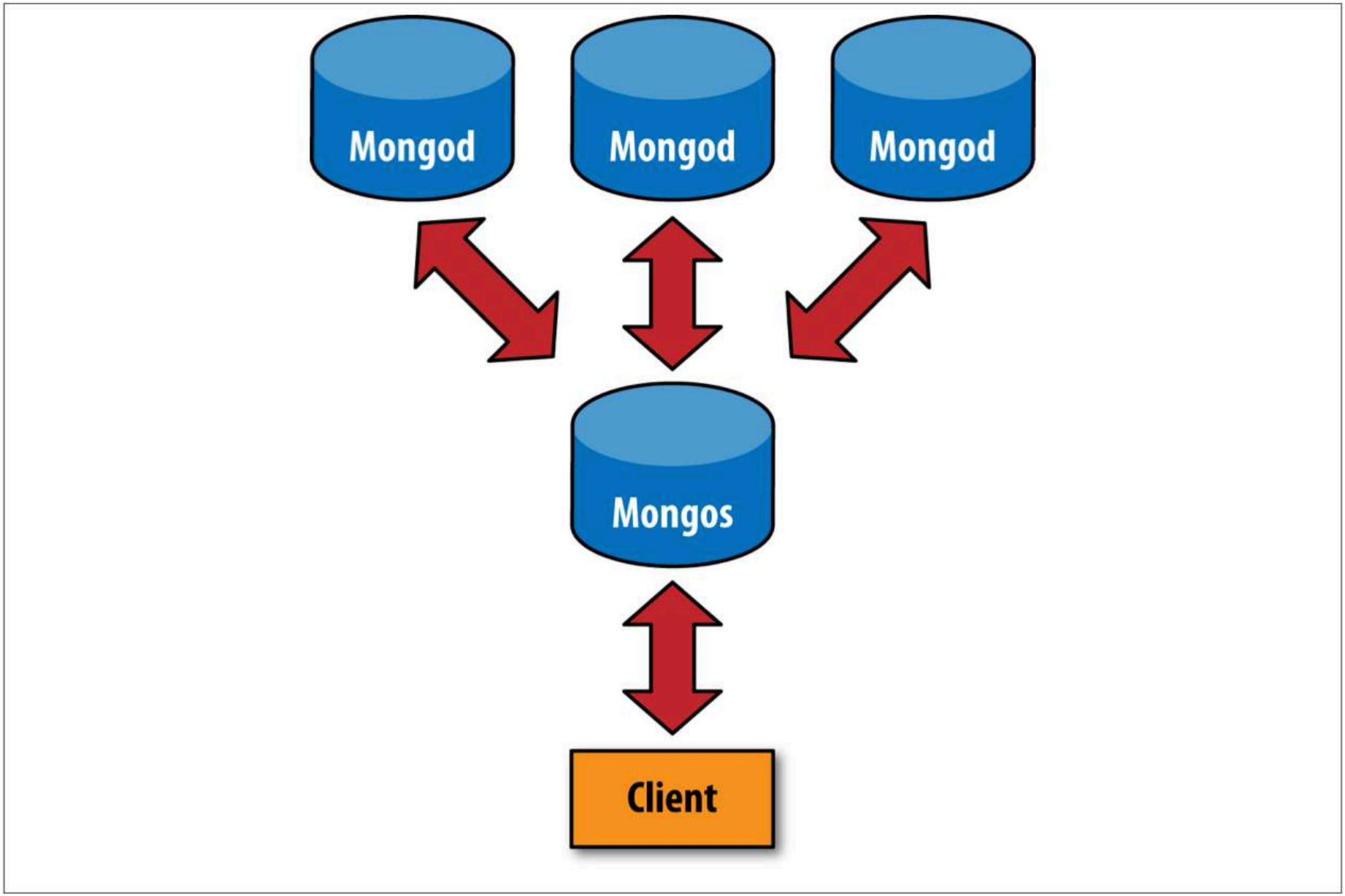


Figure 13-1. Sharded client connection

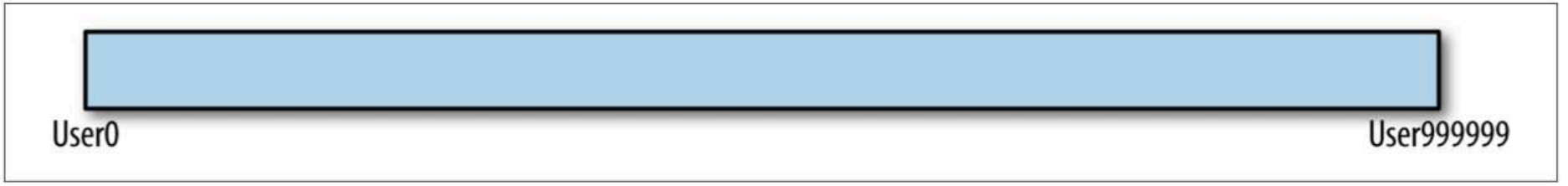


Figure 13-3. Before a collection is sharded, it can be thought of as a single chunk from the smallest value of the shard key to the largest

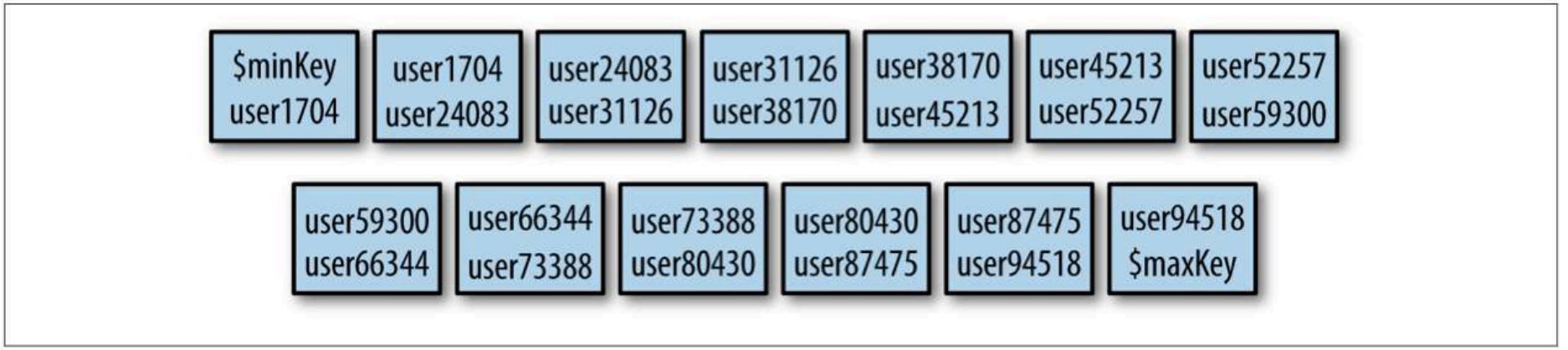


Figure 13-4. Sharding splits the collection into many chunks based on shard key ranges

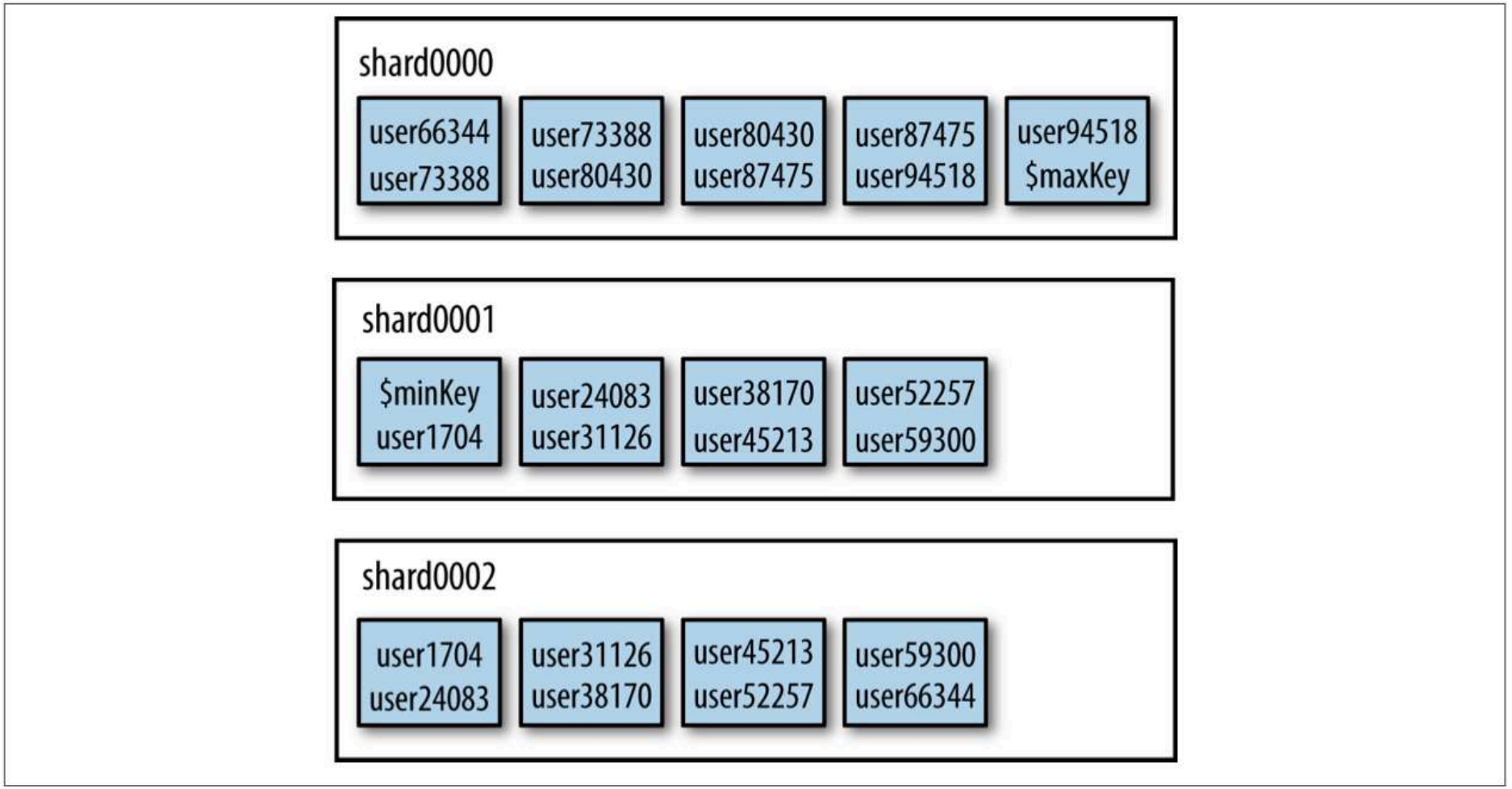


Figure 13-5. Chunks are evenly distributed across the available shards

```
> db.users.find({username: "user12345"})
{
    "_id" : ObjectId("50b0451951d30ac5782499e6"),
    "username" : "user12345",
    "created_at" : ISODate("2012-11-24T03:55:05.636Z")
}
```

```
> db.users.find({username: "user12345"}}).explain()
{
  "clusteredType" : "ParallelSort",
  "shards" : {
    "localhost:30001" : [
      {
        "cursor" : "BtreeCursor username_1",
        "nscanned" : 1,
        "n" : 1,
        "millis" : 0,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "isMultiKey" : false,
        "indexOnly" : false,
        "indexBounds" : {
          "username" : [
            [
              "user12345",
              "user12345"
            ]
          ]
        }
      }
    ]
  },
  "n" : 1,
  "nChunkSkips" : 0,
  "nYields" : 0,
  "nscanned" : 1,
  "nscannedObjects" : 1,
  "millisTotal" : 0,
  "millisAvg" : 0,
  "numQueries" : 1,
  "numShards" : 1
}
```

When to Shard

- In general, sharding is used to:
  - Increase available RAM
  - Increase available disk space
  - Reduce load on a server
  - Read or write data with greater throughput than a single mongod can handle

Config Servers

- Config servers are the brains of your cluster: they hold all of the metadata about which servers hold what data.
- Config servers are standalone *mongod* processes

```
mongod --configsvr --dbpath <path> --port 27017
```

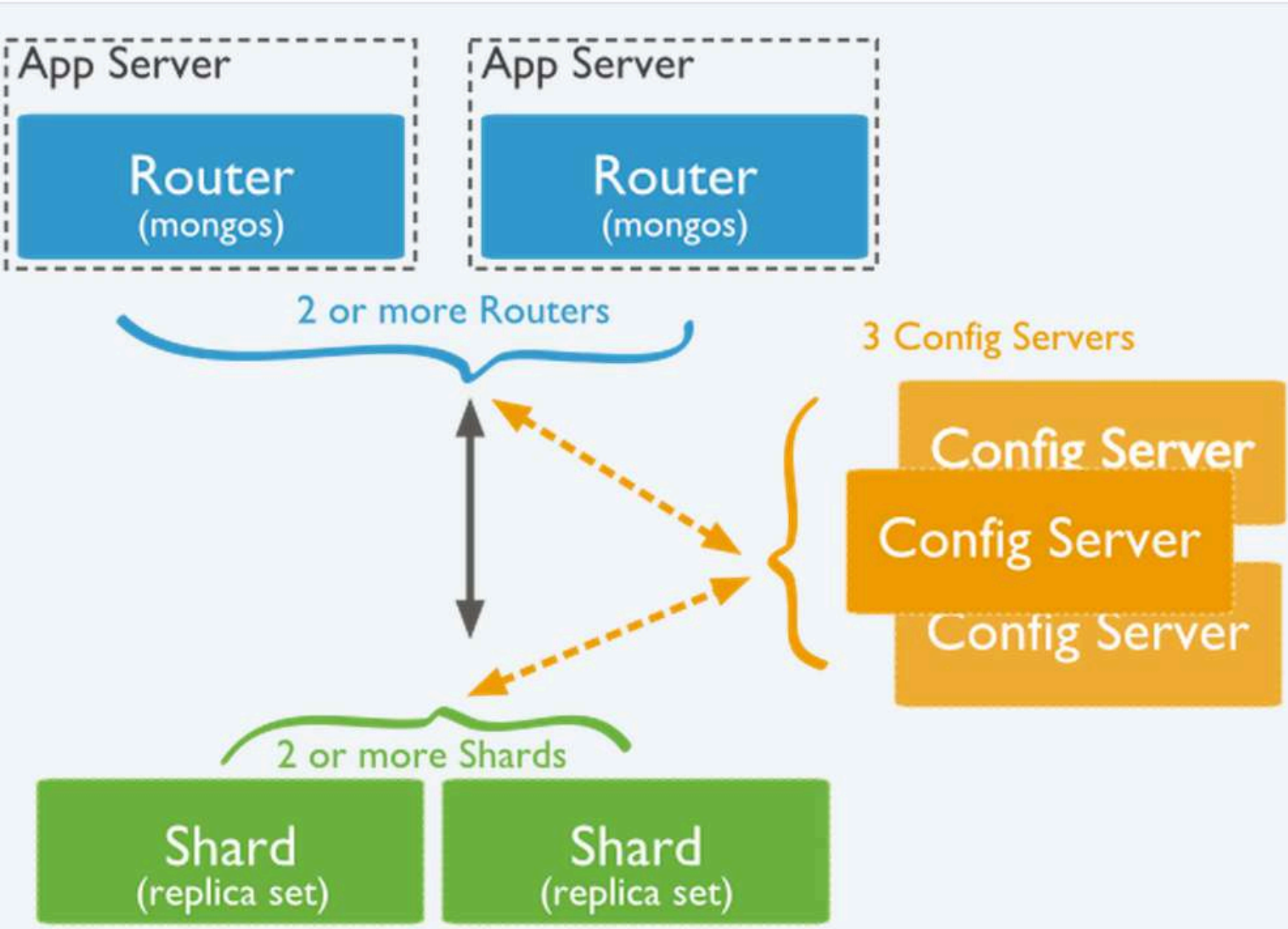
The mongos Processes

- *mongos* processes need to know where the config servers are, so you must always start *mongos* with the --configdb option:

```
$ mongos --configdb config-1:27019,config-2:27019,config-3:27019
```

```
> sh.addShard("some-server:27017")
```

```
> sh.addShard("spock/server-1:27017,server-2:27017,server-4:27017")
{
    "added" : "spock/server-1:27017,server-2:27017,server-4:27017",
    "ok" : true
}
```


App Server
Router
(mongos)
App Server
Router
(mongos)
2 or more Routers
3 Config Servers
Config Server
Config Server
Config Server
2 or more Shards
Shard
(replica set)
Shard
(replica set)

Sharding Data

- MongoDB won't distribute your data automatically until you tell it how to do so.

```
> db.enableSharding("music")
```

```
> sh.shardCollection("music.artists", {"name" : 1})
```

Table 1-1. Phone book v1

id	name	phone_number	zip_code
1	Rick	555-111-1234	30062
2	Mike	555-222-2345	30062
3	Jenny	555-333-3456	01209

Table 1-2. Phone book v2

id	name	phone_numbers	zip_code
1	Rick	555-111-1234	30062
2	Mike	555-222-2345;555-212-2322	30062
3	Jenny	555-333-3456;555-334-3411	01209

```
SELECT name FROM contacts WHERE phone_numbers LIKE '%555-222-2345%';
```

Table 1-3. Phone book v2.1 (multiple columns)

id	name	phone_number0	phone_number1	zip_code
1	Rick	555-111-1234	NULL	30062
2	Mike	555-222-2345	555-212-2322	30062
3	Jenny	555-333-3456	555-334-3411	01209

In this case, our caller ID query becomes quite verbose:

```
SELECT name FROM contacts
    WHERE phone_number0='555-222-2345'
        OR phone_number1='555-222-2345';
```

Table 1-4. Phone book v3

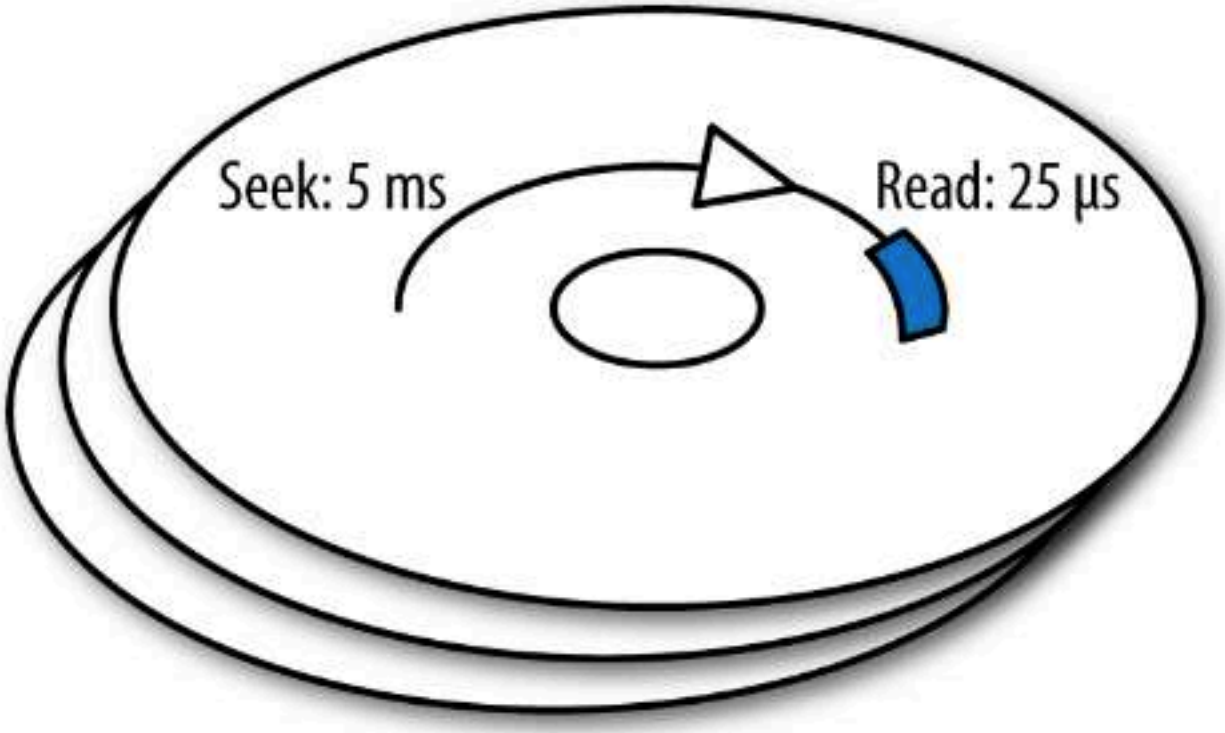
id	name	phone_number	zip_code
1	Rick	555-111-1234	30062
2	Mike	555-222-2345	30062
2	Mike	555-212-2322	30062
2	Jenny	555-333-3456	01209
2	Jenny	555-334-3411	01209

Table 1-5. Phone book v4 (contacts)

contact_id	name	zip_code
1	Rick	30062
2	Mike	30062
3	Jenny	01209

Table 1-6. Phone book v4 (numbers)

contact_id	phone_number
1	555-111-1234
2	555-222-2345
2	555-212-2322
3	555-333-3456
3	555-334-3411


Seek: 5 ms
Read: 25 µs

```
{
  "_id": 3,
  "name": "Jenny",
  "zip_code": "01209",
  "numbers": [ "555-333-3456", "555-334-3411" ]
}
```

```
// Contact document:
{
  "_id": 3,
  "name": "Jenny",
  "zip_code": "01209"
}

// Number documents:
{ "contact_id": 3, "number": "555-333-3456" }
{ "contact_id": 3, "number": "555-334-3411" }
```

```
contact_info = db.contacts.find_one({'_id': 3})
number_info = list(db.numbers.find({'contact_id': 3})
```

```
BEGIN TRANSACTION;
DELETE FROM contacts WHERE contact_id=3;
DELETE FROM numbers WHERE contact_id=3;
COMMIT;
```

```
db.contacts.remove({'_id': 3})
db.numbers.remove({'contact_id': 3})
```

```
{
  "_id": "First Post",
  "author": "Rick",
  "text": "This is my first post",
  "comments": [
     { "author": "Stuart", "text": "Nice post!" },
     ...
     ]
}
```

```
db.posts.find(
    {'comments.author': 'Stuart'},
    {'comments': 1})
```

The result of this query, then, would be documents of the following form:

```
{ "_id": "First Post",
  "comments": [
    { "author": "Stuart", "text": "Nice post!" },
    { "author": "Mark", "text": "Dislike!" } ] },
{ "_id": "Second Post",
  "comments": [
    { "author": "Danielle", "text": "I am intrigued" },
    { "author": "Stuart", "text": "I would like to subscribe" } ] }
```

```
// db.posts schema
{
  "_id": "First Post",
  "author": "Rick",
  "text": "This is my first post"
}

// db.comments schema
{
  "_id": ObjectId(...),
  "post_id": "First Post",
  "author": "Stuart",
  "text": "Nice post!"
}
```

```
db.comments.find({"author": "Stuart"})
```