



The Humongous Database

Geoffrey Berard / [@geofberard](#)

geoffrey.berard@gmail.com

goo.gl/IQe0pD - pdf

PLAN

- I. MongoDB Presentation
 - Schemaless
 - BSon
 - Document Model
 - Performance
- II. MongoDB Architecture
 - Replica Set
 - Sharding
- III. MongoDB CRUD
 - CREATE
 - READ
 - UPDATE
 - DELETE
- IV. Software with MongoDB
 - Architecture
- V. Practical



Presentation

MONGODB

Présentation

- Document oriented Data Base
- Non relational
- Dynamic : Schemaless
- No query language
- No transactions
- Scalable : Auto Sharding
- Made in C++

MONGODB

Functionnalités

- Text Search
- GeoSearch
- Aggregation
- Map-Reduce

MONGODB

Supported languages



DOCUMENT

JSon

```
{
  _id : 1,
  first_name: 'Paul',
  surname: 'McCartney',
  instruments : ["Guitar", "Bass guitar", "Piano", ...],
  address:{
    street: '20 Forthlin Road',
    city: 'LiverPool',
    zip: 'United Kingdom'
  }
}
```

DOCUMENT

JSON

Available Data Types :

- Array
- Object
- String
- Number
- Boolean
- Null

SCHEMALESS

Exemple RDBMS - Musiciens

| first_name | last_name | birthday |
|------------|-----------|-------------|
| Louis | Armstrong | 4 août 1901 |

SCHEMALESS

Exemple RDBMS - Alter table

| first_name | last_name | title | birthday |
|------------|-----------|-------|--------------|
| Louis | Armstrong | | 4 août 1901 |
| Paul | McCartney | Sir | 18 June 1942 |

SCHEMALESS

Exemple RDBMS - Holed Table

| first_name | last_name | title | nickname | birthday |
|-----------------------|-----------|-------|----------|------------|
| Louis | Armstrong | | | 04/08/1901 |
| Paul | McCartney | Sir | | 18/06/1942 |
| Gordon Matthew Thomas | Sumner | | Sting | 02/10/1951 |

SCHEMALESS

Exemple - MongoDB

```
{
  first_name: 'Louis',
  surname: 'Armstrong',
  birthday: '4 août 1901',
},
{
  first_name: 'Paul',
  surname: 'McCartney',
  title: 'Sir',
  birthday: '18 June 1942',
},
{
  first_name: 'Gordon Matthew Thomas',
  surname: 'Sumner',
  nickname: 'Sting',
  birthday: '2 octobre 1951',
},
```

SCHEMALESS

Be Carefull

```
{
  first_name: 'Louis',
  surname: 'Armstrong',
  birthday: '4 août 1901',
},
{
  first_name: 42,
},
```

SCHEMALESS

Be Carefull



BSON

Binary Representation of JSON - 16MB Maximum

```
{hello: "world"}
```

Gives in BSON

```
\x16\x00\x00\x00\x02hello\x00  
\x06\x00\x00\x00world\x00\x00
```

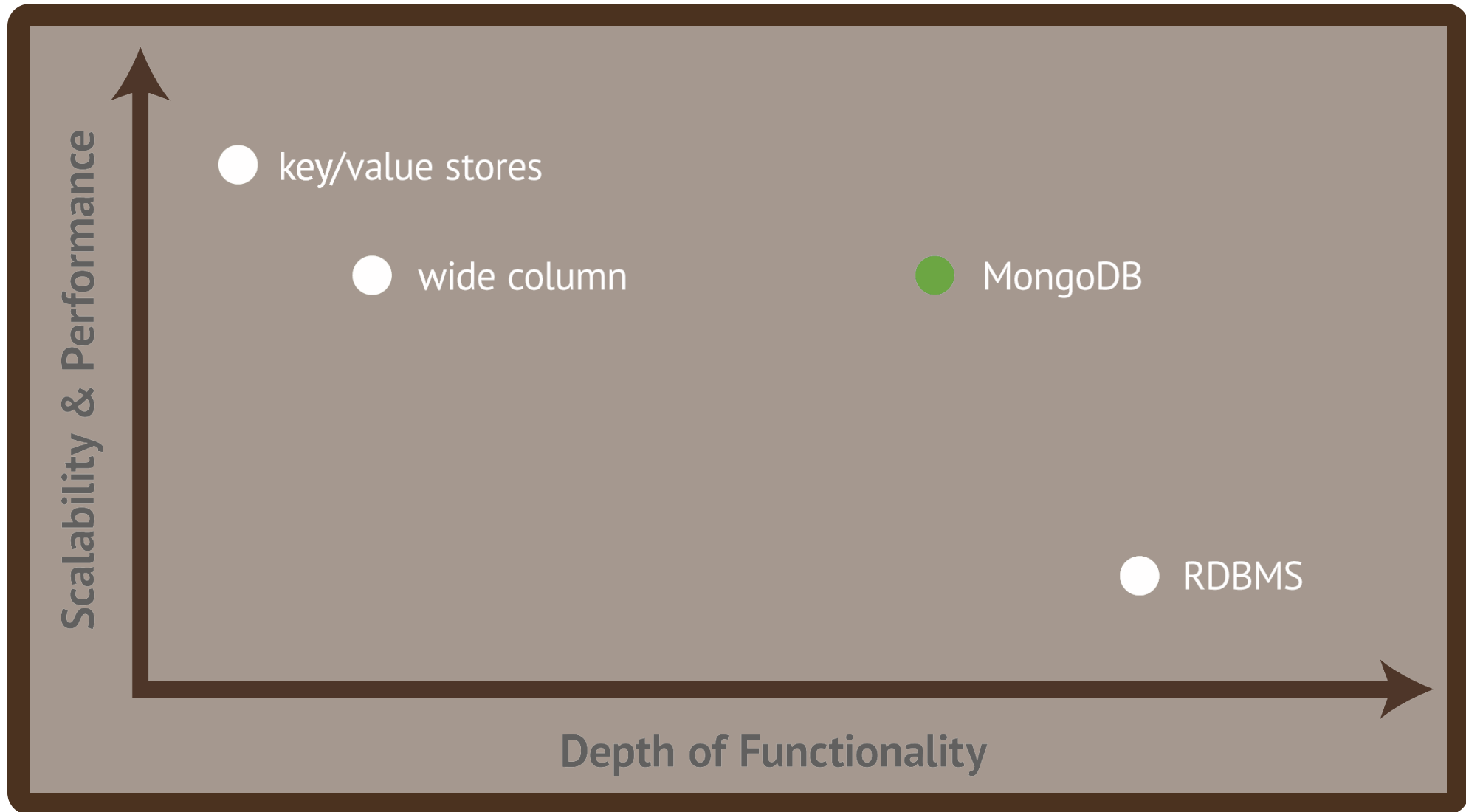
BSON

Enriched Types

| Type | Type |
|-------------|--------------------|
| Double | Regular expression |
| String | JavaScript |
| Object | Symbol |
| Array | 32-bit integer |
| Binary data | Timestamp |
| Object ID | 64-bit integer |
| Boolean | Min key |
| Date | Max key |
| Null | |

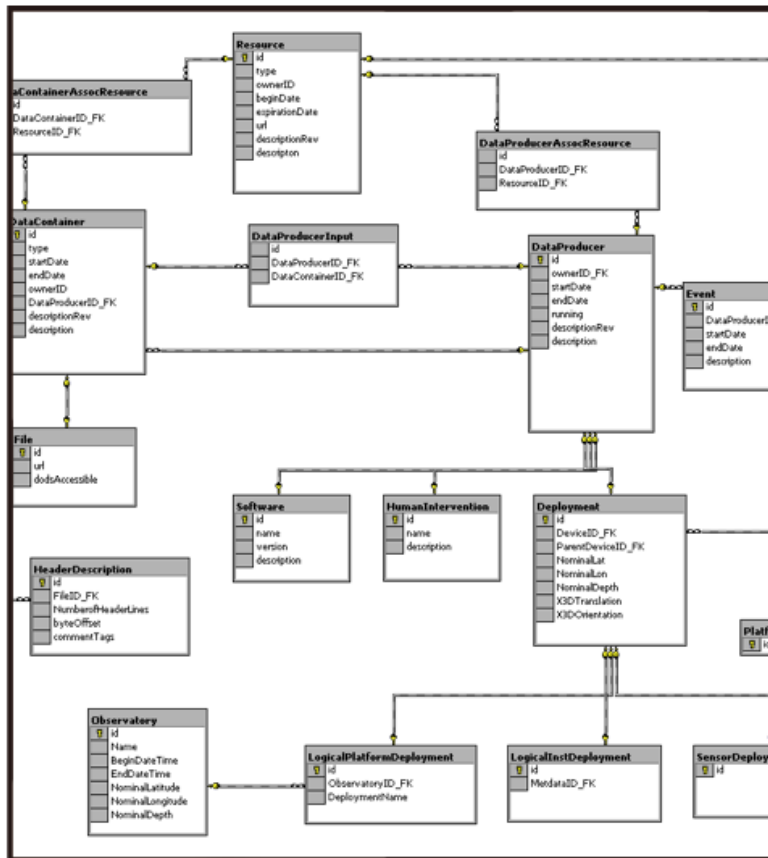
bsonspec.org

POSITION



RELATIONAL VS DOCUMENT

Models Differences



RELATIONAL VS DOCUMENT

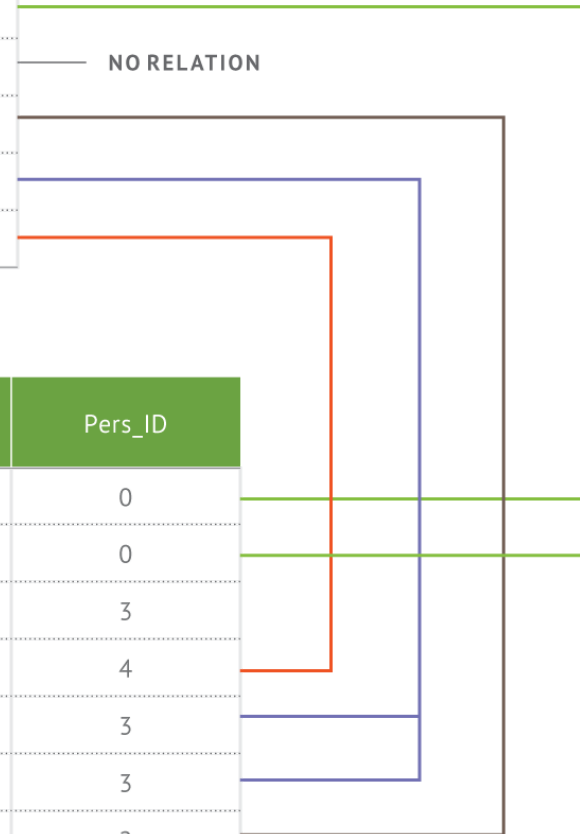
Relational Model

PERSON

| Pers_ID | Surname | First_Name | City |
|---------|-----------|------------|----------|
| 0 | Miller | Paul | London |
| 1 | Ortega | Alvaro | Valencia |
| 2 | Huber | Urs | Zurich |
| 3 | Blanc | Gaston | Paris |
| 4 | Bertolini | Fabrizio | Rome |

CAR

| Car_ID | Model | Year | Value | Pers_ID |
|--------|-------------|------|--------|---------|
| 101 | Bently | 1973 | 100000 | 0 |
| 102 | Rolls Royce | 1965 | 330000 | 0 |
| 103 | Peugeot | 1993 | 500 | 3 |
| 104 | Ferrari | 2005 | 150000 | 4 |
| 105 | Renault | 1998 | 2000 | 3 |
| 106 | Renault | 2001 | 7000 | 3 |
| 107 | Smart | 1999 | 2000 | 2 |



RELATIONAL VS DOCUMENT

MongoDB Model

```
{
  first_name: 'Paul',
  surname: 'Miller',
  city: 'London',
  cars: [
    {
      model: 'Bentley',
      year: 1973,
      value: 100000
    },
    {
      model: 'Rolls Royce',
      year: 1965,
      value: 330000
    }
  ]
}
```

RELATIONAL VS DOCUMENT

Terminology

| RDBMS | Mongo |
|---------------|-------------------|
| Table, View | Collection |
| Row(s) | JSON Document |
| Index | Index |
| Join | Embedded Document |
| Partition | Shard |
| Partition Key | Shard Key |

PERFORMANCES



Relational

vs.

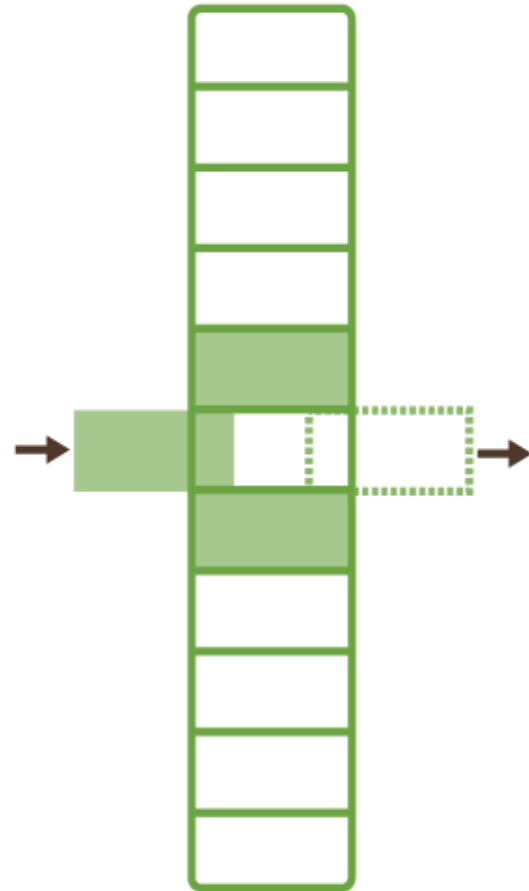


MongoDB

Locality



Caching



Inline Update

DOCUMENT MODEL

Benefits

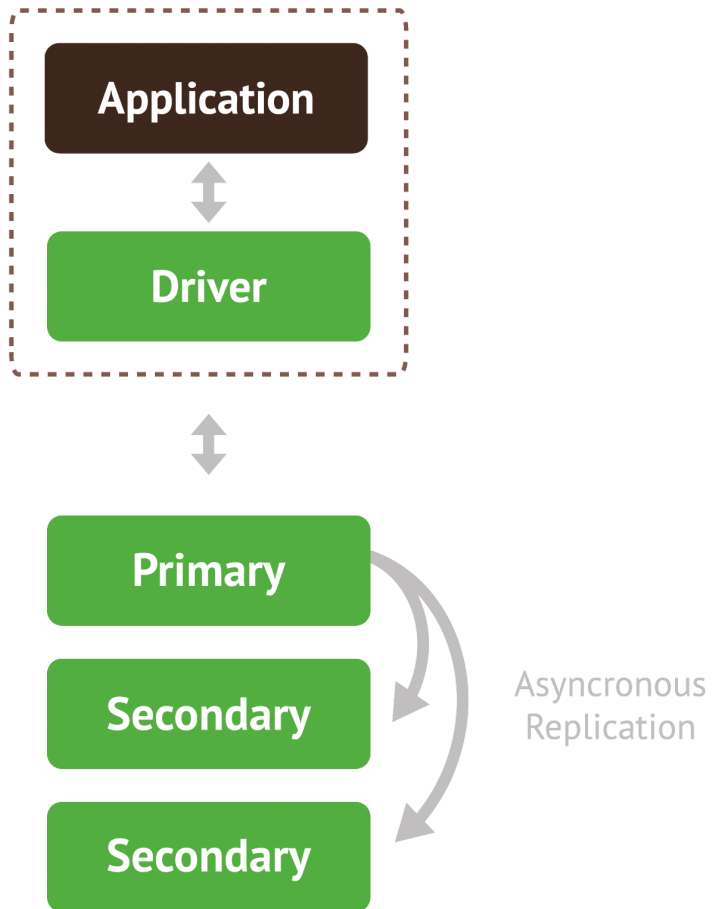
- Efficient
 - Super low latency
 - Scale Easily
- Agility and flexibility
 - Data models can evolve easily
 - Companies can adapt to changes quickly
- Intuitive, natural data representation
 - Developers are more productive
 - Many types of applications are a good fit
- Reduces the need for joins, disk seeks
 - Programming is more simple
 - Performance can be delivered at scale



Architecture

REPLICA SET

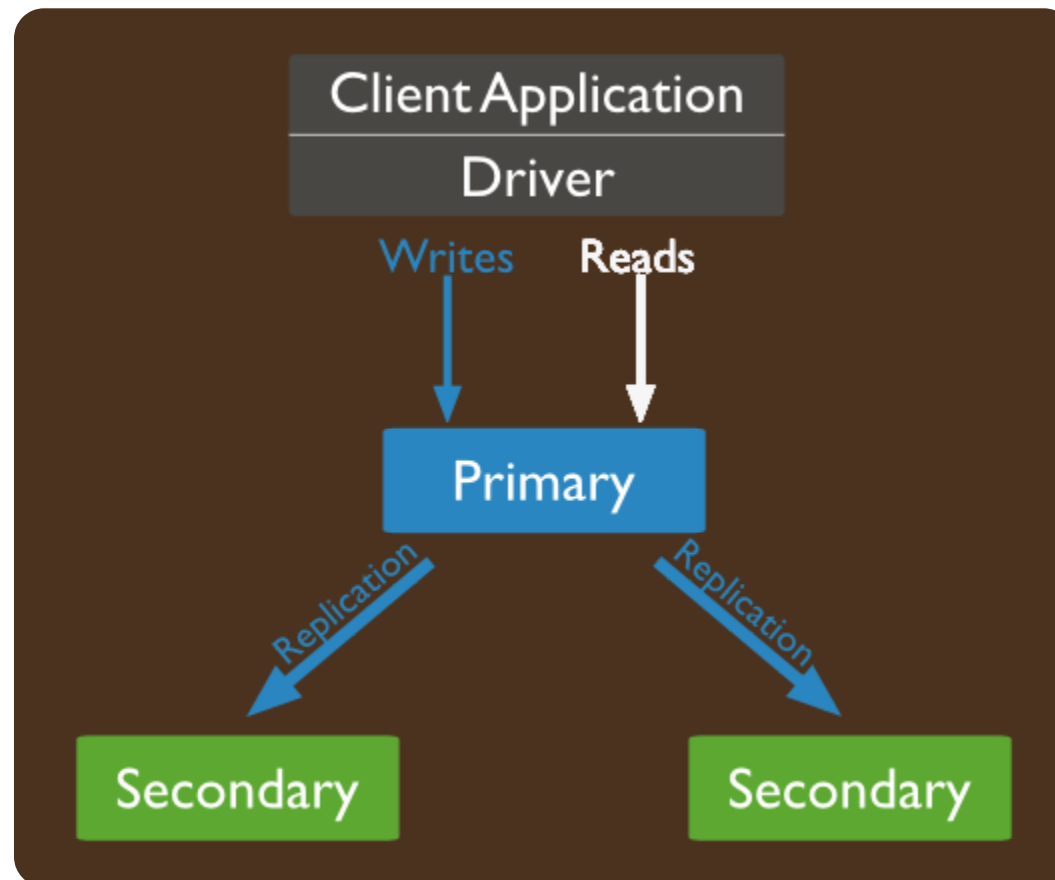
Presentation



- Two copies or more
- Master / Slave
- Automatic Failover
- Purpose :
 - High Availability
 - Data Recovery
 - Maintenance

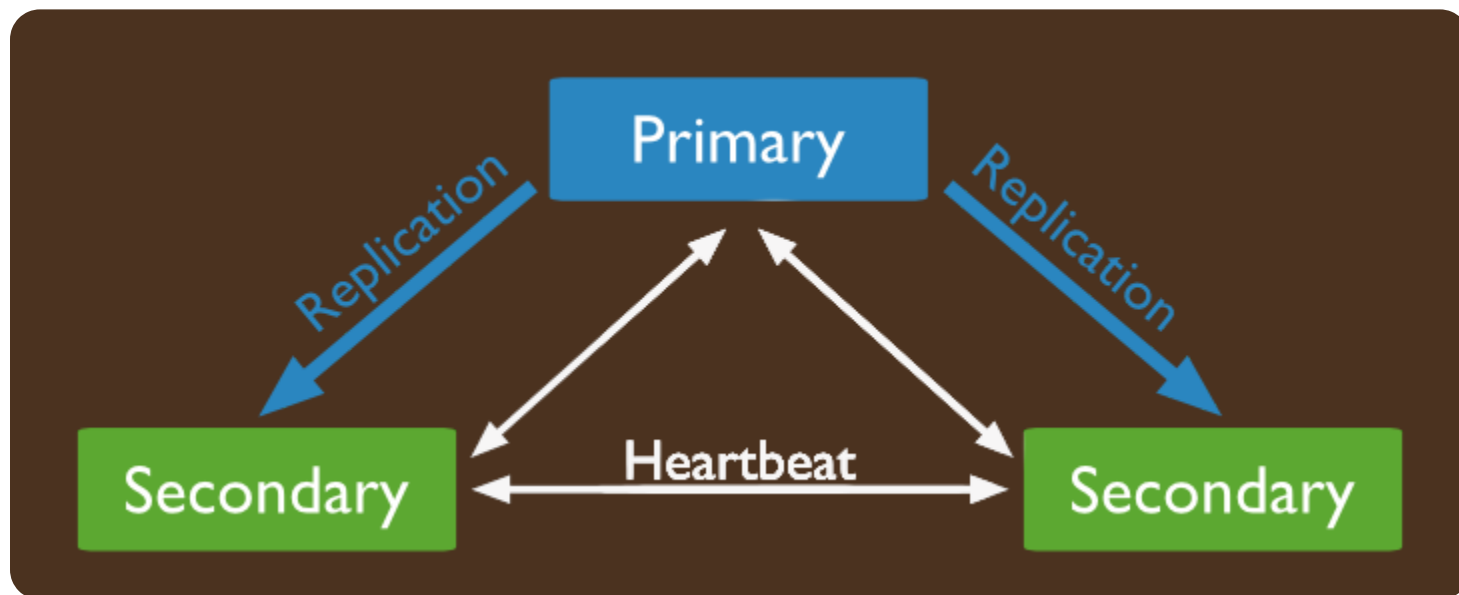
REPLICA SET

Architecture



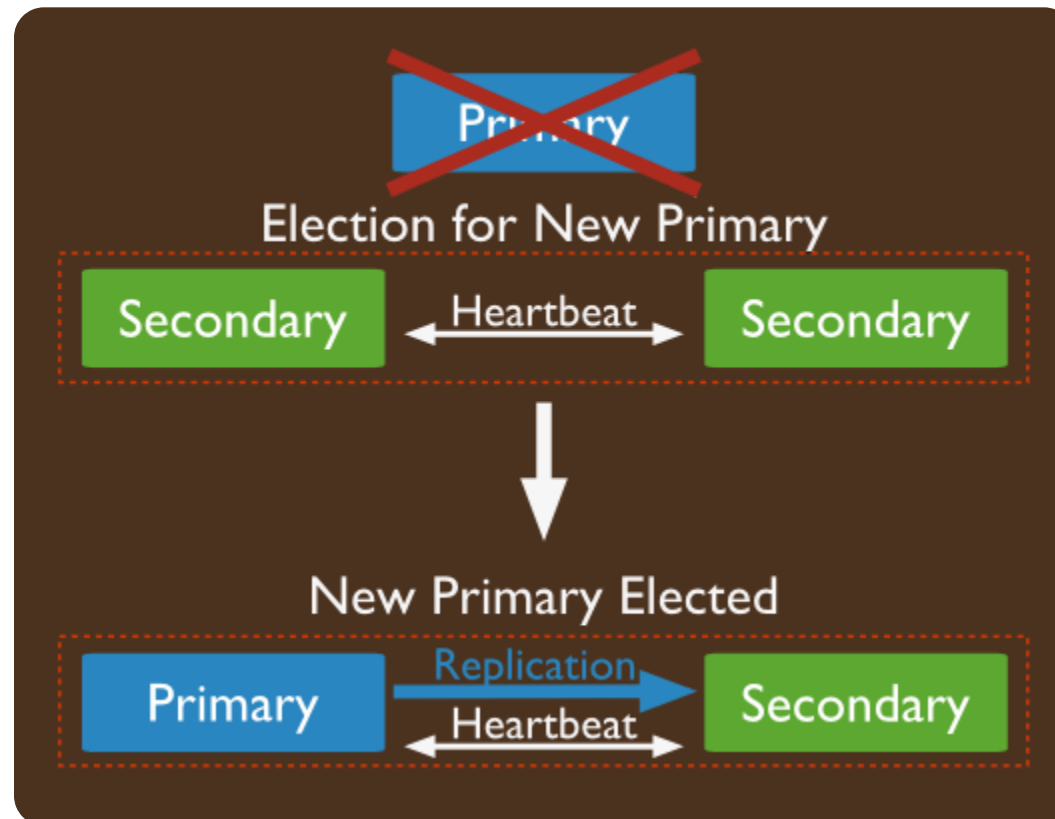
REPLICA SET

Organisation



REPLICA SET

Failover



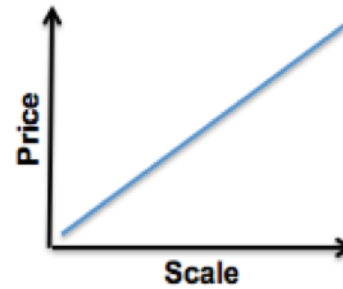
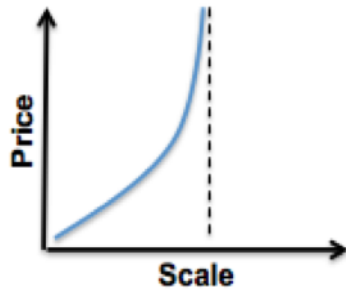
SCALABILITY



Vertical

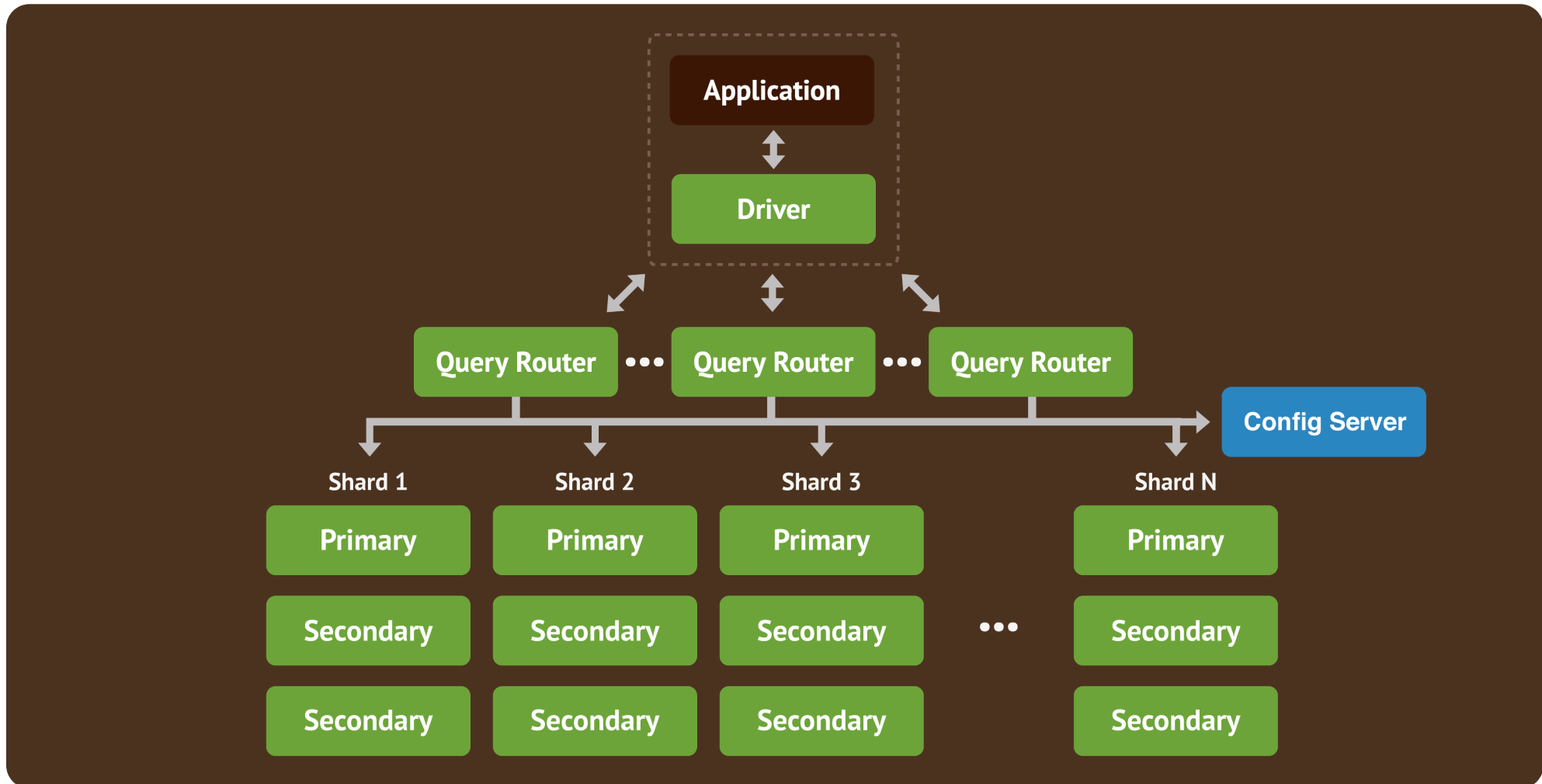


Horizontal



SHARDING

Architecture



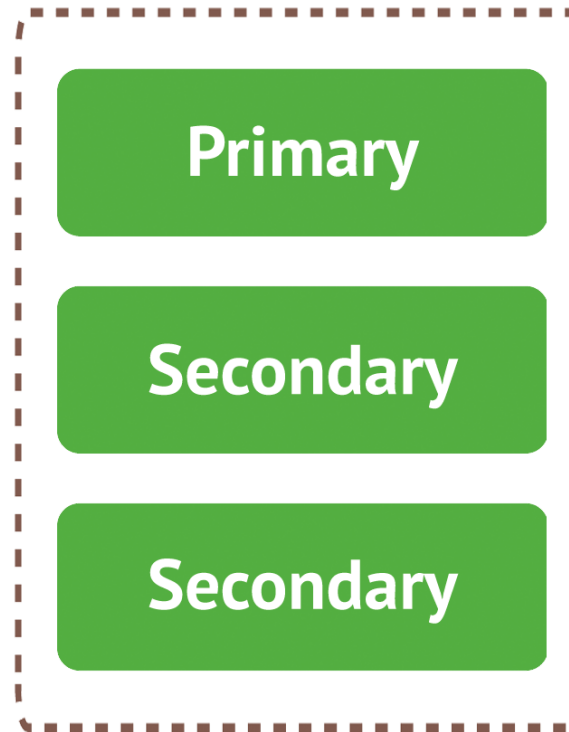
SHARDING

Shard : Cluster Node

Shard



Shard



SHARDING

Config Server

Contains cluster metadatas

- 1 instance in dev, 3 in production
- Contains intervals definitions (chunks)
- Maintenance

SHARDING

Mongos : Query Router

- Behaves identically to mongod
- Query router
- Load Balancer
- Dedicated or shared

SHARDING

Sharding Key

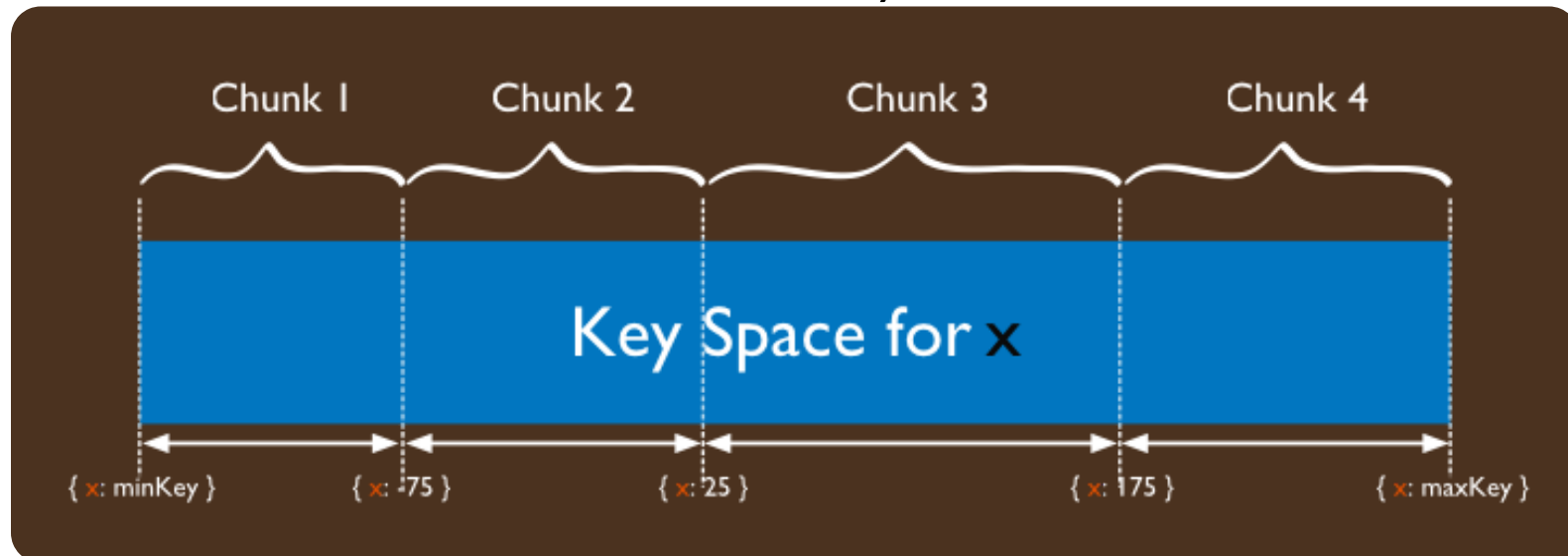
Requirement :

- Immutability (key/value)
- Big cardinality
- Distributed
- Should be indexed
- Limited to 512 octets

SHARDING

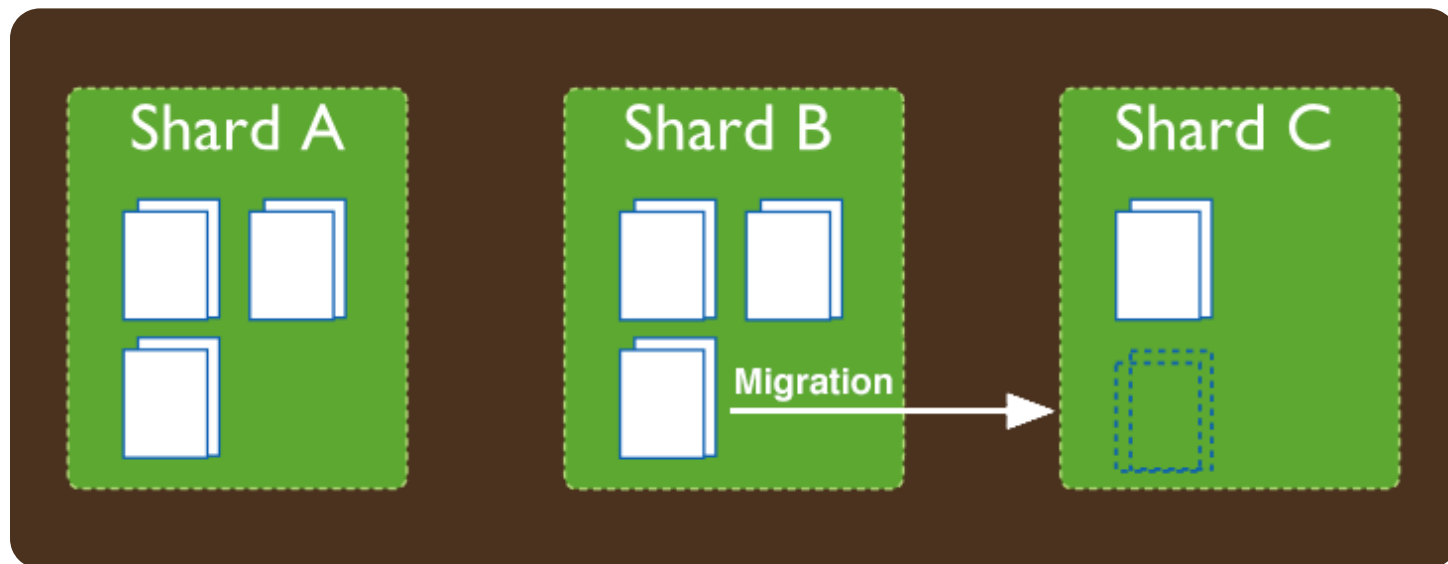
Chunk

- Split if bigger than 64Mo
- Split \neq Moved
- Split only between 2 different values
- Moved Automatically



SHARDING

Balancing





CRUD

CRUD

C Create

R Read

U Update

D Delete

CRUD

| | SQL | MongoDB |
|--------|--------|---------|
| Create | Insert | Insert |
| Read | Select | Find |
| Update | Update | Update |
| Delete | Delete | Remove |

QUERYING

~~Query language~~ => methods of objects

IDENTIFIER

```
{  
  _id : 1,  
  first_name: 'Victor',  
  surname: 'Hugo',  
  groups : [ "Writer", "Painter"],  
}
```

- Unique
- Can't be changed

CREATE

insert

```
> db.member.insert({new_document})
```

```
WriteResult({ "nInserted" : 1 })
```

Create the collection if necessary

```
> db.member.insert({first_name: "John", last_name: "Doe"})  
> db.member.insert({first_name: "Jean", last_name: "Dupont", city_of_birth: "Par
```

CREATE INDEX

Syntax (**Index types**)

```
createIndex( { userid: index_type } )
```

```
> db.records.createIndex( { userid: 1 } ) // Ascending index  
> db.records.createIndex( { userid: -1 } ) // Descending index
```

CREATE

mongoimport

Supported file format

- JSon
- CSV
- TSV

```
$ ./mongoimport --db test --collection zips --file ../../../../Downloads/zips.json
```

[Download zips.json](#)

READ

Find

Returning all elements of a collection:

```
> db.member.find()
```

```
{ "_id" : ObjectId("54853dd6dd8fc0fec931fcbc"), "first_name" : "John", "last_name" : "Doe" }
```

Returning only the first element:

```
> db.member.findOne()
```

READ

Find

Formating the result:

```
> db.member.find().pretty()
```

```
{
  "_id" : ObjectId("54853dd6dd8fc0fec931fcbc"),
  "first_name" : "John",
  "last_name" : "Doe"
}
```

READ

Find

Syntax:

```
find({query},{keys_filter})
```

Example

```
> db.zips.find({state:"NY"},{city:true, _id:false})
```

```
{ "city" : "FISHERS ISLAND" }  
{ "city" : "NEW YORK" }  
{ "city" : "NEW YORK" }  
{ "city" : "NEW YORK" }  
{ "city" : "GOVERNORS ISLAND" }
```


READ

Querying :

- Greater than (\$gt, \$gte)

```
{ pop : { $gt : 100000 } }
```

- Lower than (\$lt, \$lte)

```
{ pop : { $lte : 100 } }
```

- Regular Expression (\$regex)

```
{ city : { $regex : "^A" } }
```

READ

Querying

- Value Exists (`$exists`)

```
{ city : { $exists : true } }
```

- Value Type (`$type`) (**Type codes**)

```
{ city : { $type : type_code } }
```

READ

Wrong Query

Overriding query property

```
{ pop: {$gt: 10000}, pop: {$lt: 50000}}
```

is equivalent to

```
{ pop: {$lt: 50000}}
```

READ

Querying in array

- Natural

```
{ groups : "Painter" }
```

- In (\$in)

```
{ groups : { $in : ["Writer", "Sculptor", "Dancer"]}}
```

- All (\$all)

```
{ groups : { $all : ["Painter", "Writer"]}}
```

Can return :

```
{  
  first_name: 'Victor',  
  surname: 'Hugo',  
  groups : [ "Writer", "Painter"]  
}
```

READ

Querying in nested document

```
{ address : {  
  number: 6,  
  street_name: 'Place des Vosges',  
  city: 'Paris',  
  zip: '75004'  
}})
```

Or

```
{ "address.city" : "Paris" }
```

Can return :

```
{  
  first_name: 'Victor',  
  surname: 'Hugo',  
  address:{  
    number: 6,  
    street_name: 'Place des Vosges',  
    city: 'Paris',  
    zip: '75004'  
  }  
}
```

}

}

READ

Wrong queries

- Incomplete object description

```
{ address : { city : "Paris" }})
```

- Document key in the wrong order

```
{ address : {  
  street_name: 'Place des Vosges',  
  number: 6,  
  city: 'Paris',  
  zip: '75004'  
}})
```

READ

Querying combinaison :

- Natural :

```
{ pop : { $gt : 100000, $lt : 2000000 } }
```

- Or (\$or):

```
{ $or : [ {state : "NY"} , {state : "NJ"} ] }
```

- And (\$and):

```
{ $and : [ {state : "NY"} , {pop : { $gt : 50000 } } ] }
```


READ

Querying combinaison combinaison ???

```
{ $and: [ { $and: [ { city: { regex: "^N" } }, { $or: [ { state: "NY" }, { state: "NJ" } ] } ] }, { pop: { $gt: 100000, $lt: 150000 } } ] }
```

```
{  
  $and: [  
    { $and: [  
      { city : { regex: "^N" } },  
      { $or : [  
        { state: "NY" },  
        { state: "NJ" }  
      ] }  
    ] },  
    { pop : { $gt : 100000 , $lt : 150000 } }  
  ]  
}
```

Cities starting with "N" in New York or New Jersey with a population between 100k and 150k inhabitants

READ

Count

```
> db.zips.count({find_query})
```

Or

```
> db.zips.count({state:"NY"})
```

READ

Cursor

```
> cursor = db.zips.find({state:"MA"},{city:true, _id:false}); null;
```

Iterating over results :

```
> cursor.hasNext() // > true
> cursor.next() // > "AGAWAM"
> cursor.next() // > "CUSHMAN"
```

READ

Operation on curser:

```
> cursor.sort({city : -1}) // Sort in reverse alphabetical order
> cursor.limit(5) // Limit the number of results to 5
> cursor.skip(3) // Skip 3 elements before returning the result
```

They can be combined

```
> cursor.sort({city : -1}).limit(5).skip(3)
```

UPDATE

Syntax

```
update( {find_query} , {update_query}, {update_params} )
```

UPDATE

Example

```
update({surname : "Hugo"},
       {surname : "Hugo", groups : [ "Writer", "Painter"]})
```

on

```
{
  first_name: 'Victor',
  surname: 'Hugo',
  address:{
    number: 6,
    street_name: 'Place des Vosges',
    city: 'Paris',
    zip: '75004'
  }
}
```

will give

```
{
  surname: 'Hugo',
  groups : [ "Writer", "Painter"]
}
```

UPDATE

- Ajouter/Modifier des champs (\$set)

```
{ $set : { groups : [ "Writer", "Painter" ] } }
```

- Supprimer des champs (\$unset)

```
{ $unset : { groups : 1 } }
```

- Modify Key (\$rename)

```
{ $rename : { "oldName" : "newName" } }
```

UPDATE

- Modify (number)

```
{$inc : {count : 1}}
```

```
{$mul : {price : NumberDecimal("0.5")}}
```

- Compare to current value (number, date)

```
{$min : {bestPrice : 150}}
```

```
{$max : {highScore : 1275000}}
```


UPDATE

Array manipulation

- Change Value

```
{"groups.2" : "Poet"}
```

- Add element (\$push)

```
{$push : {groups : "Poet"}}  
{$pushAll : {groups : ["Poet","Politician"]}}
```

- Remove element (\$pop)

```
{$pop : {groups : 1}} // remove last element  
{$pop : {groups : -1}} // remove first element
```

UPDATE

Array manipulation

- Remove specific element (\$pull)

```
{ $pull : { groups : "Poet" } }  
{ $pullAll : { groups : [ "Poet", "Politician" ] } }
```

- Take array as a set (\$addToSet)

```
{ $addToSet : { groups : "Poet" } } // Add "Poet"  
{ $addToSet : { groups : "Poet" } } // Do nothing because exists
```

UPDATE

Update or insert

=> Upsert

```
{ upsert : true } // As update third parameter
```

```
> db.member.update({surname : "Washington"},  
  { $set : {groups : [ "Writer", "Painter" ]}},  
  { upsert : true })
```

UPDATE

Multiple update

```
{ multi : true } // As update third parameter
```

```
> db.member.update({},  
  { $set : {title : "Mr"}},  
  { multi : true })
```

```
WriteResult({ "nMatched" : 5, "nUpserted" : 0, "nModified" : 5 })
```

DELETE

Syntax

- Removing lines

```
> db.member.remove({find_query})
```

- Drop a collection

```
> db.member.drop()
```

DEBUGGING

```
> db.zips.explain().find({query});
```

or

```
> cur = db.zips.find({query}); null;  
> cur.explain()
```

DEBUGGING

Example

```
> db.zips.explain().find({state:"MA"},{city:true, _id:false}).sort({city : -1}).
```

```
{
  "queryPlanner" : {
    "plannerVersion" : 1,
    "namespace" : "test.zips",
    "indexFilterSet" : false,
    "parsedQuery" : {
      "state" : {
        "$eq" : "MA"
      }
    }
  },
  "winningPlan" : ...,
  "serverInfo" : {
    "host" : "localhost",
    "port" : 27017,
    "version" : "4.0.4",
    "gitVersion" : "f288a3bdf201007f3693c58e140056adf8b04839"
  },
  "ok" : 1
}
```

DEBUGGING

Execution detail

```
> cur = db.zips.explain("executionStats").find({state:"MA"},{city:true, _id:false})
```

```
{
  "queryPlanner" : {
    ...
    "namespace" : "test.zips",
    ...
  },
  "executionStats" : {
    "executionSuccess" : true,
    "nReturned" : 5,
    "executionTimeMillis" : 20,
    "totalKeysExamined" : 0,
    "totalDocsExamined" : 29353,
    ...
  },
  "serverInfo" : {
    "host" : "it-gbe",
    "port" : 27017,
    "version" : "3.2.11",
    "gitVersion" : "009580ad490190ba33d1c6253ebd8d91808923e4"
```


ADVANCED

Distinct (details)

```
> db.zips.distinct({field} , {search_query})
```

```
> db.zips.distinct("state" , {})  
[  
  "MA" ,  
  "RI" ,  
  "NH" ,  
  ...  
]
```

ADVANCED

Geospacial (details)

```
> db.zips.createIndex( { loc : "2d" } )  
> db.zips.find( { 'loc': {$near : [ -112.416728, 37.781334 ] } } ).limit(5)
```

```
{ "_id" : "84759", "city" : "PANGUITCH", "loc" : [ -112.436886, 37.80777 ], "pop" : 1000 }  
{ "_id" : "84710", "city" : "ALTON", "loc" : [ -112.548389, 37.469905 ], "pop" : 1000 }  
{ "_id" : "84760", "city" : "PARAGONAH", "loc" : [ -112.773972, 37.89172 ], "pop" : 1000 }  
{ "_id" : "84717", "city" : "BRYCE CANYON", "loc" : [ -112.074311, 37.608427 ], "pop" : 1000 }  
{ "_id" : "84761", "city" : "PAROWAN", "loc" : [ -112.832251, 37.844861 ], "pop" : 1000 }
```

ADVANCED

Aggregate (details)

```
> db.zips.aggregate([ { $group: {group} } , { $match: {group} } ])
```

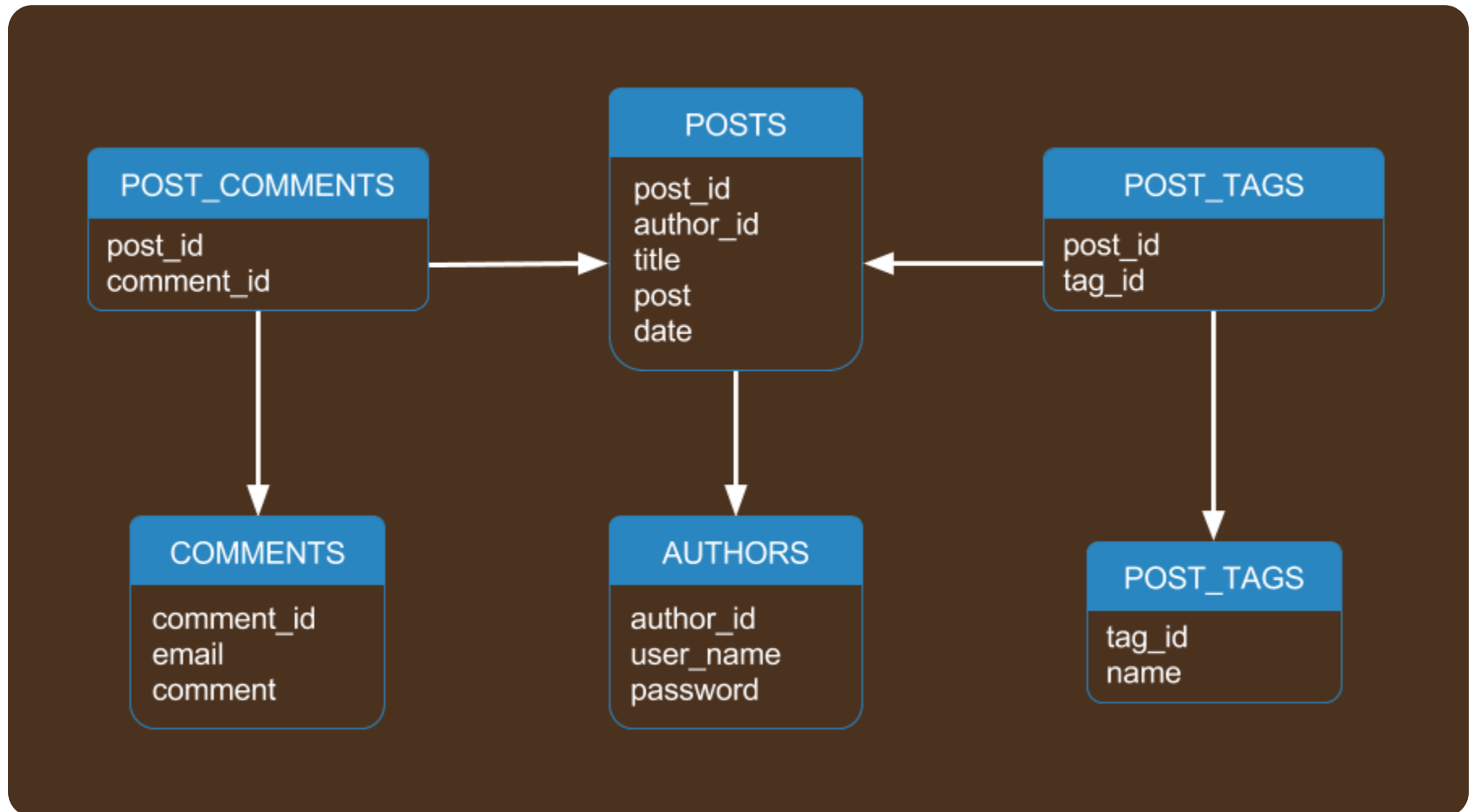
```
> db.zips.aggregate([ { $group: { _id: "$city", totalPop: { $sum: "$pop" } } } ])  
{ "_id" : "CHALKYITSIK", "totalPop" : 99 }  
{ "_id" : "WRANGELL", "totalPop" : 2573 }  
{ "_id" : "SKAGWAY", "totalPop" : 692 }  
{ "_id" : "THORNE BAY", "totalPop" : 744 }  
...
```



Case Study

SIMPLE BLOG

Relational



SIMPLE BLOG

MongoDB

Posts

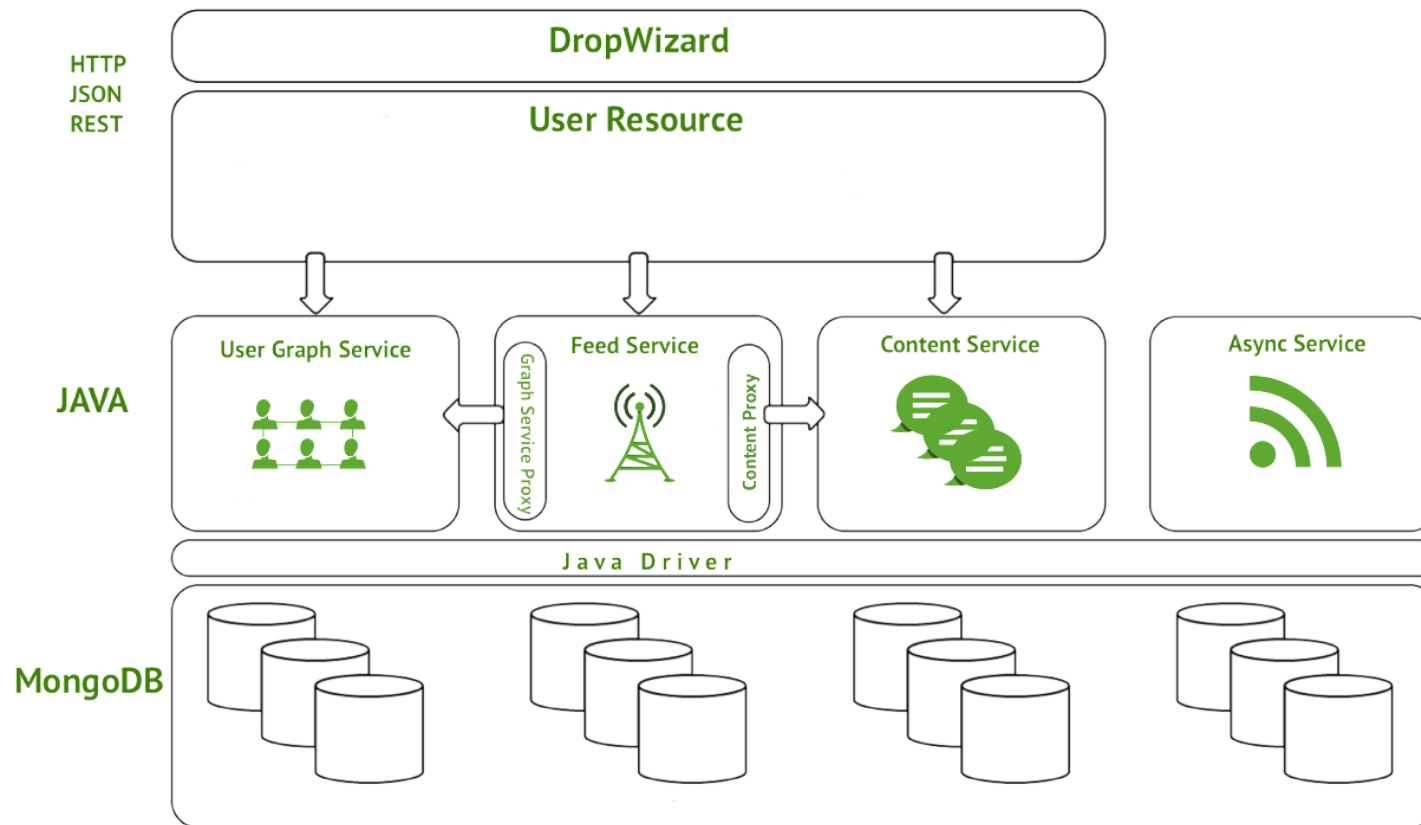
```
{
  _id : ObjectId("54853dd6dd8fc0fec931fcbc"),
  title : "Title",
  body : "...",
  author : "Author",
  date : "Date",
  comments : [
    {
      name: "Observer",
      comment: "Comment",
    }
  ],
  tags: ["Course", "MongoDB"]
}
```

Author

```
{
  _id : "UserName",
  email : "UserEmail"
}
```

SOCIALITE

Architecture



SOCIALITE

Design 1

```
...  
{ _id : "vHugo", email : "victor.hugo@gmail.com", follower:["gWashington"]}   
{ _id : "gWashington", email : "george.washington@gmail.com"}   
...
```


SOCIALITE


Design 1 - Problem



Profile header for Katy Perry. The background is a solid orange color. On the left is a circular profile picture of Katy Perry. To the right of the profile picture are statistics: Tweets (9 321), Abonnements (216), Abonnés (107 M), and J'aime (6 130). The 'Abonnés' value is circled in red. On the far right is a 'Suivre' button and a three-dot menu icon.

| Tweets | Abonnements | Abonnés | J'aime |
|--------|-------------|---------|--------|
| 9 321 | 216 | 107 M | 6 130 |

KATY PERRY ✓
@katyperry



Profile header for 'kat pery'. The background is a solid blue color. On the left is a circular profile picture placeholder. To the right are statistics: Abonnements (2) and Abonnés (8). The 'Abonnés' value is circled in red. On the far right is a 'Suivre' button and a three-dot menu icon.

| Abonnements | Abonnés |
|-------------|---------|
| 2 | 8 |

kat pery
@katpery1

SOCIALITE

Design 2

- User Collection

```
{ _id : "vHugo", email : "victor.hugo@gmail.com"}  
{ _id : "gWashington", email : "george.washington@gmail.com"}
```

- Follower Collection

```
{ _id : 1, _from : "gWashington", _to : "vHugo"}
```

SOCIALITE

Design 3

- User Collection

```
{ _id : "vHugo", email : "victor.hugo@gmail.com" }  
{ _id : "gWashington", email : "george.washington@gmail.com" }
```

- Follower Collection

```
{ _id : 1, _from : "gWashington", _to : "vHugo" }
```

- Following Collection

```
{ _id : 1, _from : "vHugo", _to : "gWashington" }
```



Practical

INSTALLING MONGODB

Linux & MacOS

- Download mongodb zip [here](#)
- Unzip and go in the file

```
$ cd path_to_downloaded_file  
$ tar xvf mongodb-osx-x86_64-4.0.4.tgz
```

- Create the storage directory

```
$ sudo mkdir -p /data/db  
$ sudo chmod 777 /data/db
```

RUNNING MONGODB

Linux & MacOS

- Go to the directory

```
$ cd mongodb-osx-x86_64-4.0.4/bin
```

- Run MongoDB Daemon (server)

```
$ ./mongod
```

- Run MongoShell (client) in an other shell

```
$ ./mongo
```

INSTALLING MONGODB

Windows

- Download mongodb zip [here](#)
- Install it
- Create the storage directory

```
$ cd C:\  
$ md "\\data\db"
```

RUNNING MONGODB

Windows

- Go to the directory

```
$ cd "C:\Program Files\MongoDB\Server\4.0\bin\"
```

- Run MongoDB Daemon (server)

```
$ mongod.exe --dbpath="c:\data\db"
```

- Run MongoShell (client) in an other cmd

```
$ mongo.exe
```


MONGO SHELL

Emails

- I.1) Import the `enron.json` in the collection "emails" [Shell command]
- I.2) What is the total amount of emails ? [Query + Result]
- I.3) What is the amount of emails in inbox ? [Query + Result]
- I.4) List the emails sent from domain yahoo.com [Query]
- I.5) How long took the last request [Request + Time]

MONGO SHELL

Emails

- I.6) Add an index the right field to make the last request run faster [Index Query]
- I.7) How long took the last request with the index [Time]
- I.8) Find only dates and subjects of all messages sent by mike.mcconnell@enron.com [Query]
- I.9) Remove rosalee.fleming@enron.com from all the email recipient [Query]
- I.10) Add rob.bradley@enron.com as recipient to all emails sent by rosalee.fleming@enron.com [Query]

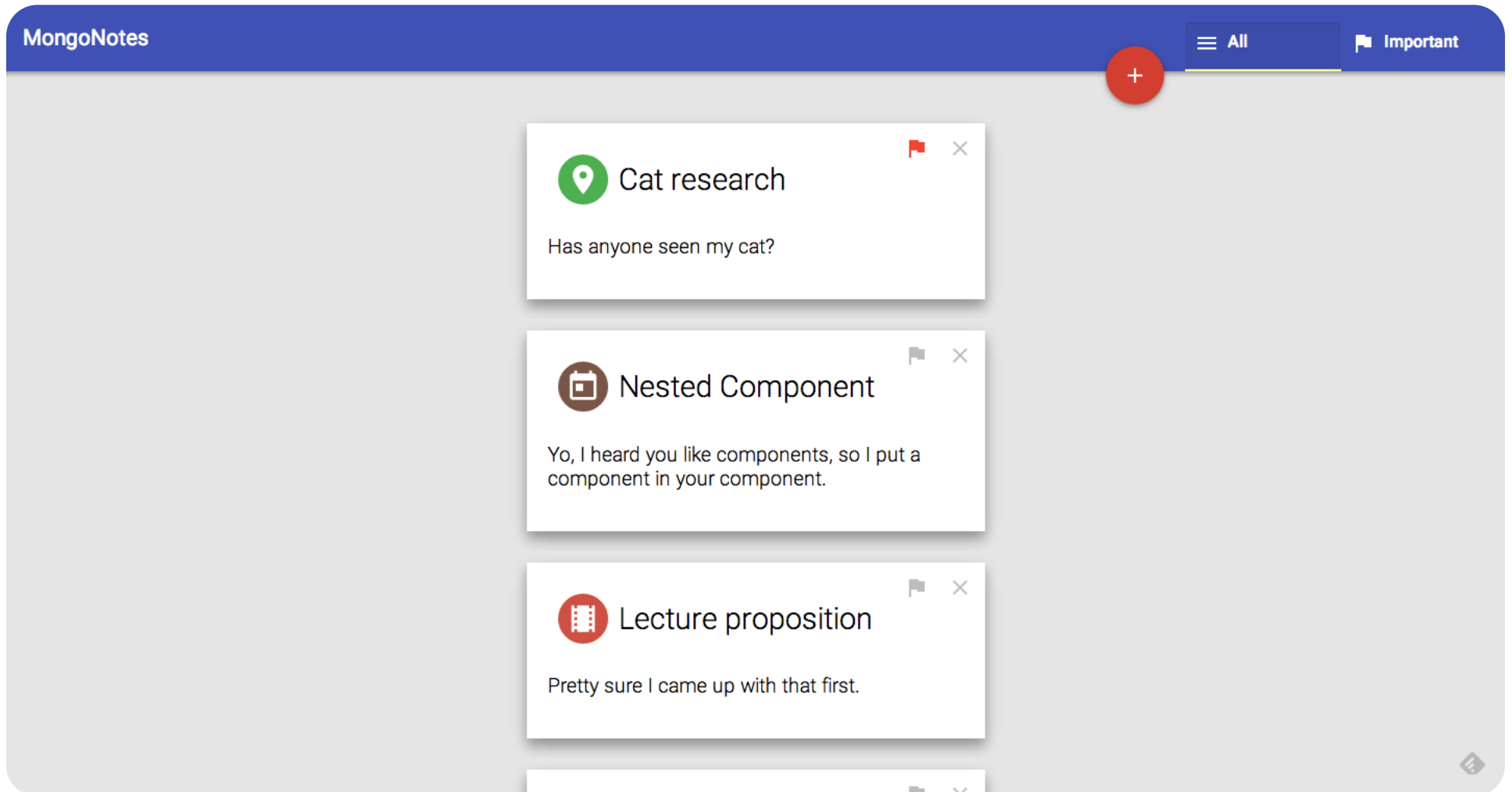
MONGO SHELL

ZIP Codes

- Import the `zips.json` in the collection "zips" [Shell command]
- II.1. List the 10 most populated zones in California and Louisiana [Query]
- II.2. Then most populated zones in California and Louisiana ranked 10 to 20 [Query]
- II.3. Add a field country with the value USA to all the zips [Query]
- II.4. List all zones with more than 100 000 inhabitant located on the east side of meridian 110. [Query]
- II.5. What is the closest zones to coordinates -73.996705, 40.74838 [Query + Answer]
- II.6. The cities that are less than 5km away from -73.996705, 40.74838: [Query + Answer]
- II.7. The cities that have more than 500 000 inhabitants. [Query + Answer]

MongoNotes

MongoDB with JAVA



MongoNotes Project

Github : goo.gl/xszvRW



MongoNotes

Bootstrap

Startup project

```
$ git clone https://github.com/geofberard/MongoNotes.git
$ cd MongoNotes
$ mvn clean install
$ git checkout step-0
```

MongoNotes

Git Detail

Each step of the practical is saved in a special branch.

If you are stuck, you can checkout the next branch to go to the next step with :

```
$ git reset HEAD --hard  
$ git checkout branch_name
```

(all your current modification will be lost)

The branches are :

- Step-0 : Empty implementation
- Step-1 : Service bootstrap + find() implementation
- Step-2 : findAll() implementation
- Step-3 : create() implementation
- Step-4 : delete() implementation
- Step-final : full implementation

MongoNotes

Step 0

```
$ git checkout step-0
```

- NoteService.java : The file you need to work in
- Note.java : Modelisation for a note:

```
String id;           //String extracted from MongoDB ObjectID
String title;
String text;
String type;         //Can be description/room/today/theaters
boolean important;  //Say if the document is flagger or not
```

- NotesServer.java : Web Server, run the with main() to start server
- JJsonTransformet.java : format data to/from client
- Resources : web content

MongoNotes

Step 1

```
$ git checkout step-1
```

You need to get all the notes from the collection

There are some hints :

- Use find() of DBCollection to get a cursor
- Use hasNext() and next() to iterate through elements
- Cast DBObject to BasicDBObject to get more methods

MongoNotes

Step 2

```
$ git checkout step-2
```

You need to add a new document in the collection

There are some hints :

- Use use insert() of DBCollection
- Use new Gson().fromJson(body, Note.class) to get a note object

MongoNotes

Step 3

```
$ git checkout step-3
```

You need to delete a document from the collection

There are some hints :

- You must specify a query on `_id` : `{_id:...}`
- Use `delete()` of `DBCollection`
- You can create a query with `BasicDBObject` or with `QueryBuilder`
- You need to transform the `String` `id` to an `ObjectId` (`new ObjectId(uid)`)

MongoNotes

Step 4

```
$ git checkout step-4
```

You need to update document in the collection

There are some hints :

- You must specify a query on `_id` : `{_id:...}`
- You must specify a modifier on `_id` : `{$set:...}`
- Use `update()` of `DBCollection`
- You can create a query with `BasicDBObject` or with `QueryBuilder`
- You need to transform the `String` `id` to an `ObjectId` (`new ObjectId(uid)`)

MongoNotes

Step Final

```
$ git checkout step-final
```

You can add a new feature



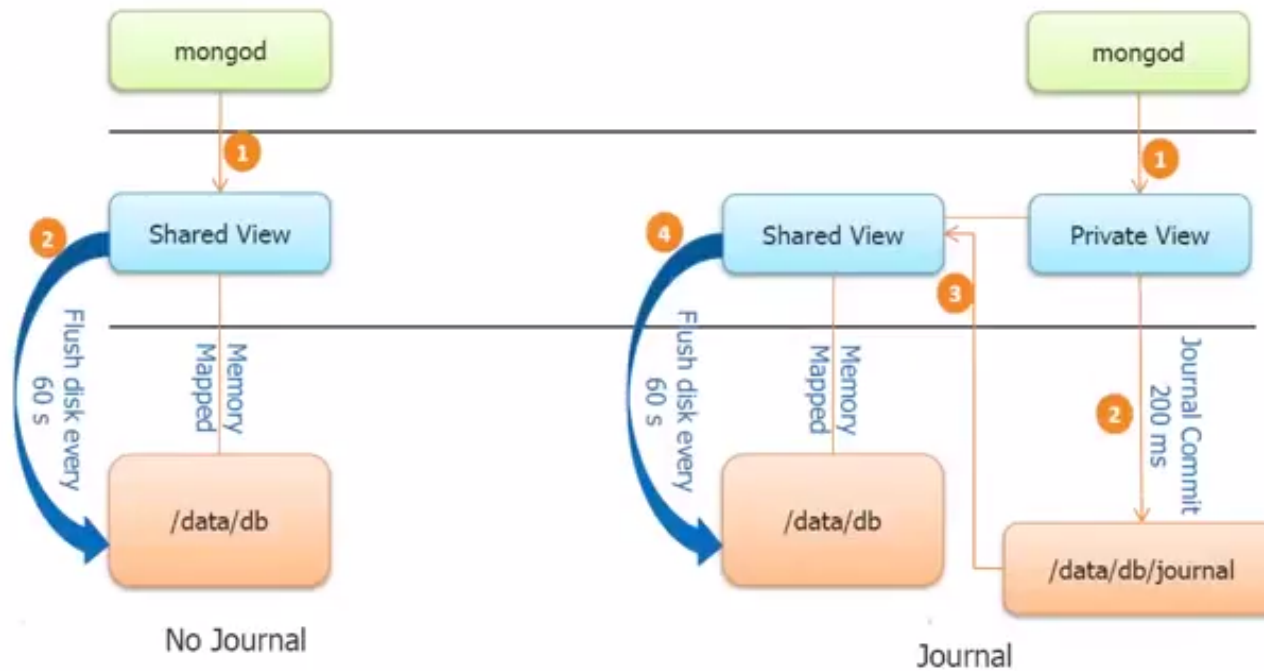
Replication

MongoDB Replication

1. MongoDB write path
2. Replication principles
3. Replica set Read and Write Semantics
4. Replica set in practice

MongoDB write path

Journaling Mechanics



MongoDB Journal vs Oplog

- **journal**
 - low level log of an operation for crash recovery (can be turned off)
- **oplog**
 - similar to RDBMS binlog
 - stores (idempotent) high-level transactions that modify the database
 - kept on the master and used for replication

<https://docs.mongodb.org/manual/core/read-isolation-consistency-recency/>

MongoDB Replication

1. MongoDB write path
2. **Replication principles**
3. Replica set Read and Write Semantics
4. Replica set in practice

Replica set

- **Replica set** = a group of *mongod processes* that provide **redundancy** and **high availability**
- **Writes:** write to single node replicated to the others members of the replica set
- **Read:** read from a single member of the replica set

Disclaimer:

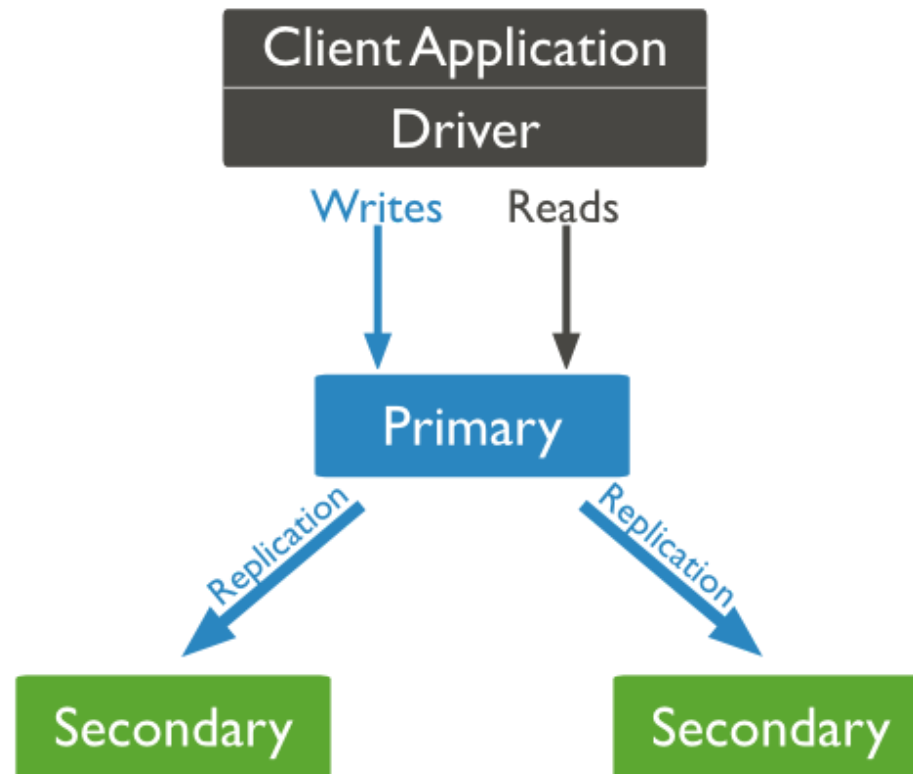
- we only consider replica sets **without sharding** (for now)
- we not include proposed MongoDB 3.2 replication modifications (readConcern...)

Replica set members

- **Primary**
 - accepts all **writes** and reads
 - 1 primary per replica set
- **Secondaries** replicates data (and can serve **reads** ⇒ reads preferences)
 - Priority 0 ⇒ Hidden members ⇒ Delayed
- **Arbiters** (usually at most one) : break ties

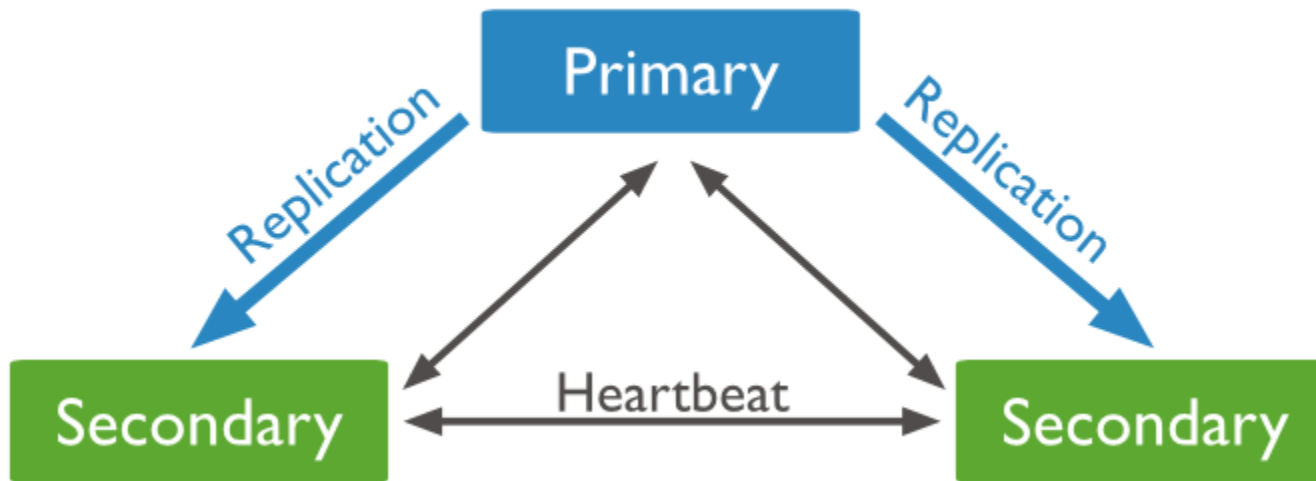
Primary and secondary members

- **Primary** accepts all **writes** + reads + records them in oplog
- **Secondary** replicates primary oplogs (also accept reads)

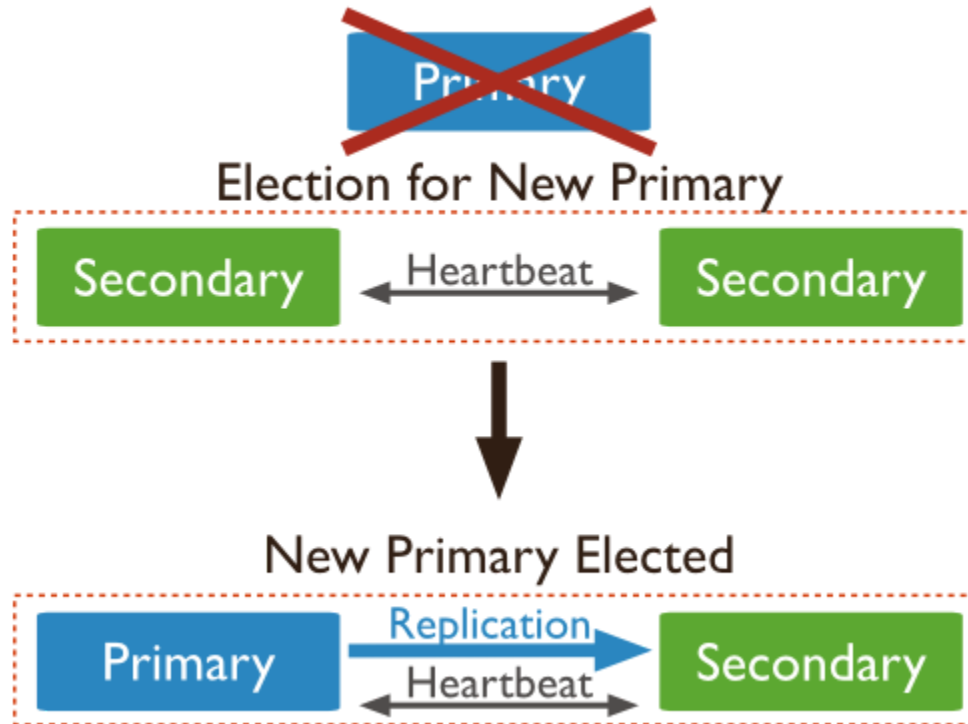


Replication data flow

- asynchronous **oplog** replication
- heartbeat for monitoring status



Automatic failover via new primary election

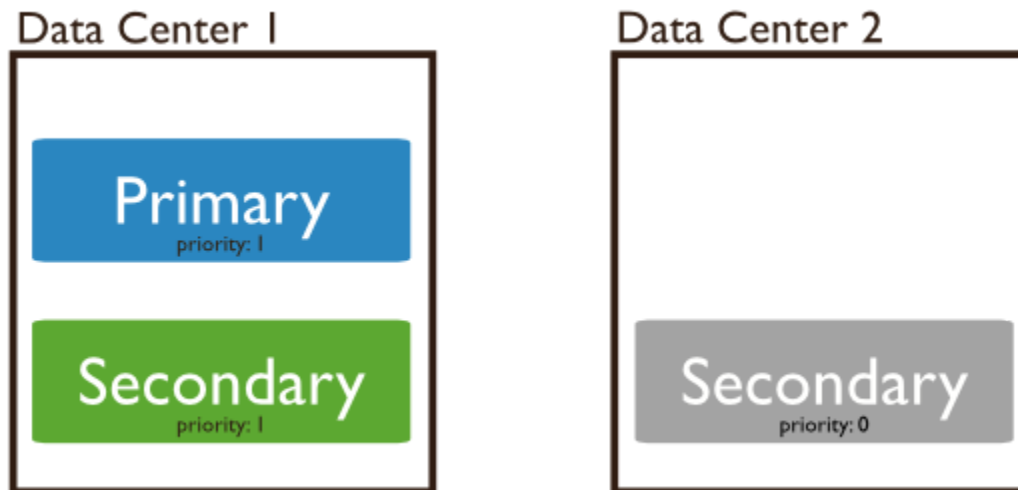


Strategy for election

- member's priority
- latest optime in the oplog
- uptime
- break the tie rules

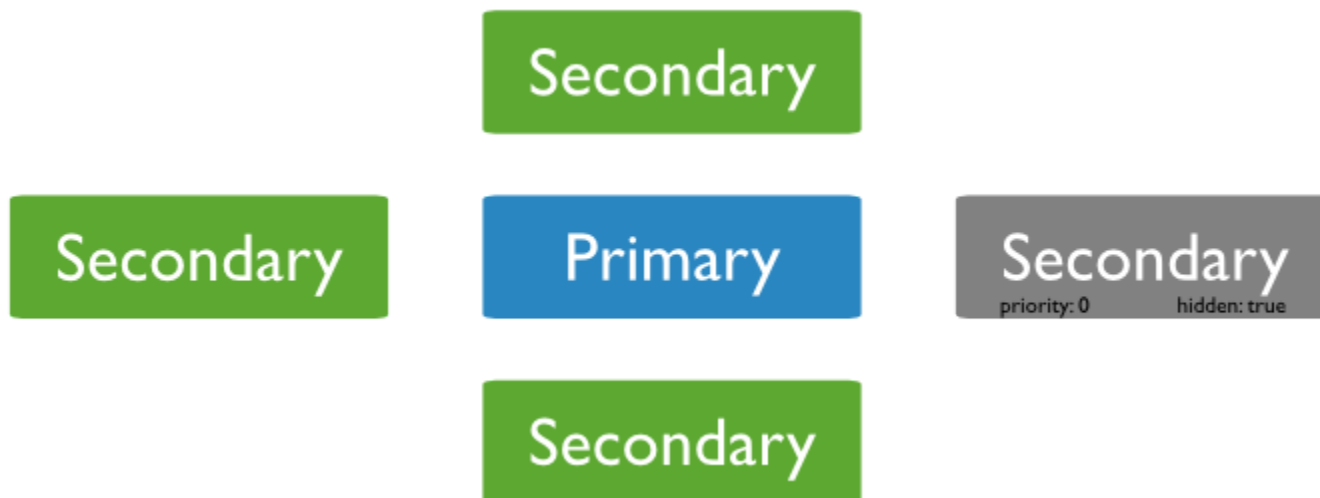
Secondary members: Priority 0

- cannot become primary
- cannot trigger elections
- can vote in elections
- copy of data + accepts reads



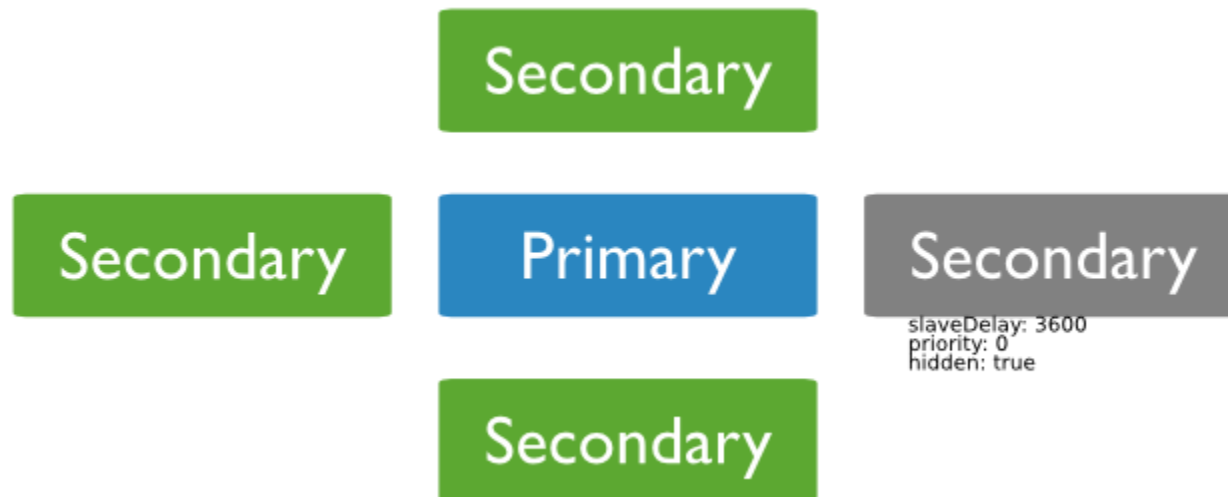
Secondary members: Hidden replica set member

- Priority 0 members that don't accept reads



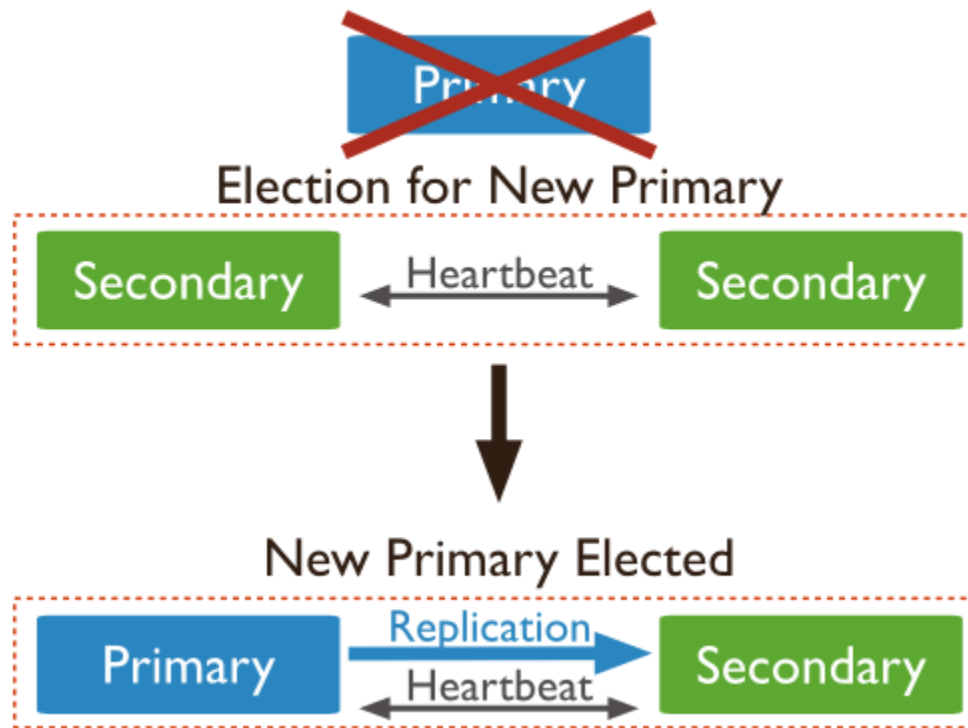
Secondary members: Delayed replica set members

- reflect an delayed state of the set
 - **must be priority 0** ⇒ prevent them to become primary
 - **should be hidden** ⇒ prevent application to query stale data



Elections on odd number of nodes

- a replica cannot become primary with only 1 vote
- majority with even numbers of members ?



- use **Arbitrers** to break ties
 - does not hold data
 - cannot become a primary

Arbiters



Fault tolerance

- **No primary** \Rightarrow writes no longer possible, reads still accepted
- **Fault tolerance** : number of members that can become unavailable and still be able to elect a primary

| Number of members | Majority required to elect a primary | Fault tolerance |
|-------------------|--------------------------------------|-----------------|
| 3 | 2 | 1 |
| 4 | 3 | 1 |
| 5 | 3 | 2 |
| 6 | 4 | 2 |

<https://docs.mongodb.org/manual/core/replica-set-architectures/>

Rollbacks during replica set failover

- a rollback reverts write operations on a former primary when the member rejoins its replica set after a failover
 - the primary accepted a write that was not successfully replicated to secondaries !

Cause of the problem ?

default write semantics { w:1 } ⇒ the primary acknowledge the write after the local write (local Journal!)

How to handle rollbacks

- manually apply/discard rollbacks (**rollback/** folder)
- *avoid* rollbacks use { **w:majority** }
 - READ UNCOMMITTED SEMANTICS
 - ! Regardless of write concern, other clients can see the result of the write operations before the write operation is acknowledged to the issuing client.
 - ! Clients can read data which may be subsequently rolled back.
<https://docs.mongodb.org/manual/core/replica-set-rollbacks/>
<https://docs.mongodb.org/manual/core/read-isolation-consistency-recency/>

MongoDB Replication

1. MongoDB write path
2. Replication principles
3. **Replica set Read and Write Semantics**
 1. Write concerns
 2. Read preferences
4. Replica set in practice

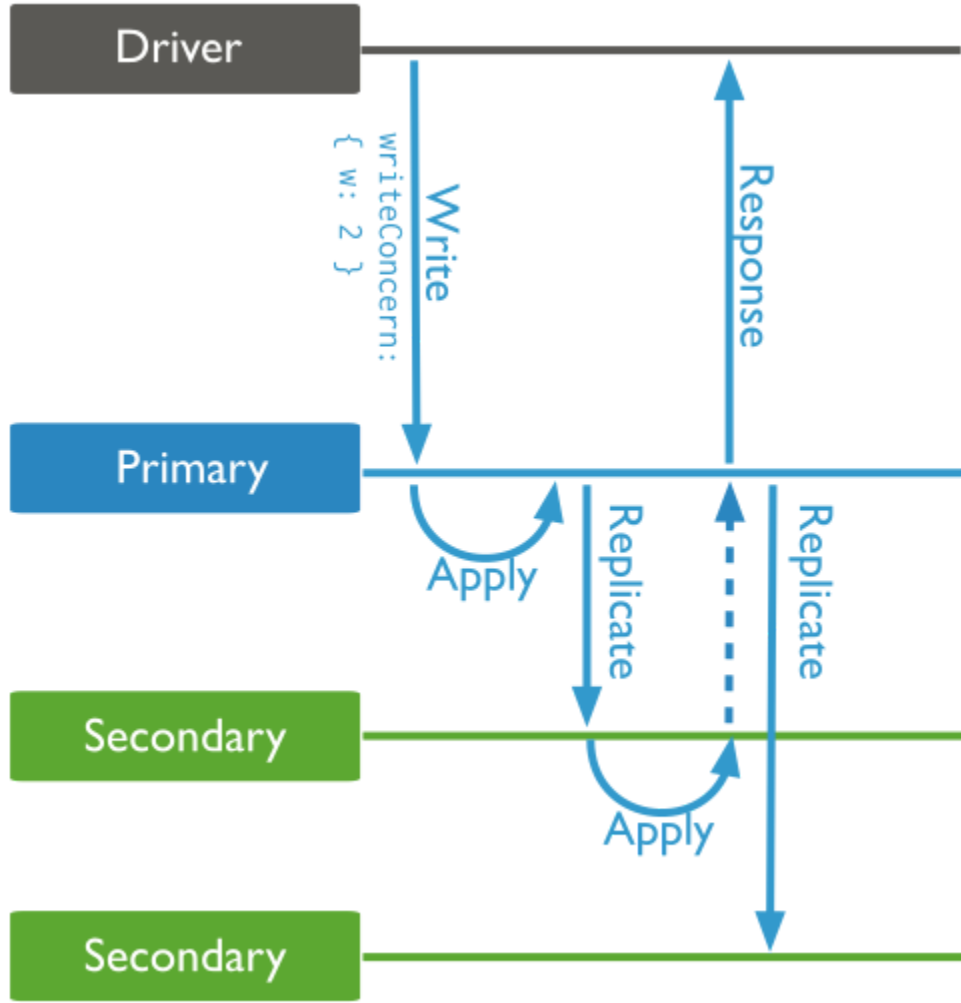
Replica set Read and Write Semantics

- parameters that change the default read/write semantics (**move the CAP cursor**)
 - **write concern**
 - is the guarantee an application requires from MongoDB to consider a write operation successful
 - **read preference**
 - applications specify read preference to control how drivers **direct read operations** to members of the replica set

Write semantics

- **w:1** (*default*)
 - the primary acknowledge the write after the local write
- other options:
 - **w:N**
 - ack the write after the ack of N members
 - **x:majority**
 - ack the write after the ack of the majority of the members
- optional parameter **wtimeout**
 - prevents write operations from blocking indefinitely if the write concern is unachievable

W:2 write semantics



Changing the write semantics

- at the query level

```
db.products.insert(  
    { item: "envelopes", qty : 100, type: "Clasp" },  
    { writeConcern: { w: 2, wtimeout: 5000 } }  
)
```

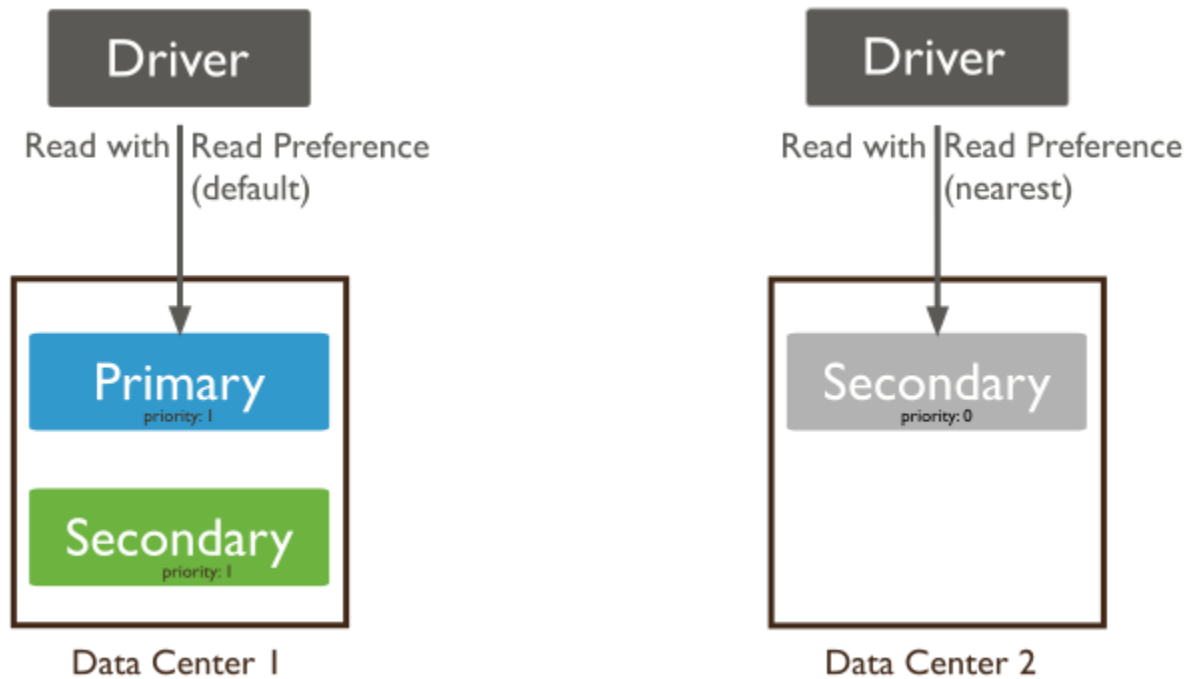
- change the default write concern:

```
cfg = rs.conf()  
  
cfg.settings = {}  
cfg.settings.getLastErrorDefaults = { w: "majority", wtimeout:  
rs.reconfig(cfg)
```

Read preference

- **primary** (*default*)
 - read from the current replica set primary.
 - **primaryPreferred**
 - read from primary (or secondary iff no primary)
 - **secondary**
 - read from secondary members
 - **secondaryPreferred**
 - read from secondary (or primary iff no primary)
 - **nearest**
 - read from the member with the least network latency
- Async replication ⇒ stale data if read from replica**

Read preferences example



Read preferences use cases

- Maximize Consistency \Rightarrow primary read preference
- Maximize Availability \Rightarrow primaryPreferred read preference
- Minimize Latency \Rightarrow nearest read preference

MongoDB Replication

1. MongoDB write path
2. Replication principles
3. Replica set Read and Write Semantics
4. **Replica set in practice**

MongoDB consistency in real world

Read the documentation for the systems you depend on thoroughly—then verify their claims for yourself. You may discover surprising results!

— Kyle Kingsbury(Aphyr)

<https://aphyr.com/posts/322-jepsen-mongodb-stale-reads>

Learn more:

- read the MongoDB documentation and the Jespen blog entry:
 - [MongoDB Documentation](#)
 - [Jepsen MongoDB Stale reads on](#)
- do the replica set tutorial in the MongoDB documentation:
 - <https://docs.mongodb.org/manual/administration/replica-set-deployment/>

BIBLIOGRAPHY

MongoDB Manual
MongoDB University
JSON Specification
BSON Specification
Socialite