

JSON stores

Ioana Manolescu, Silviu Maniu

Inria Saclay & Univ. Paris-Sud

M2 Data and Knowledge

Université Paris-Saclay

Motivation

- JSON (JavaScript Object Notation) allows to describe nested, potentially heterogeneous data
 - Very flexible
 - Thus, a good idea for NoSQL!
- Less verbose than XML

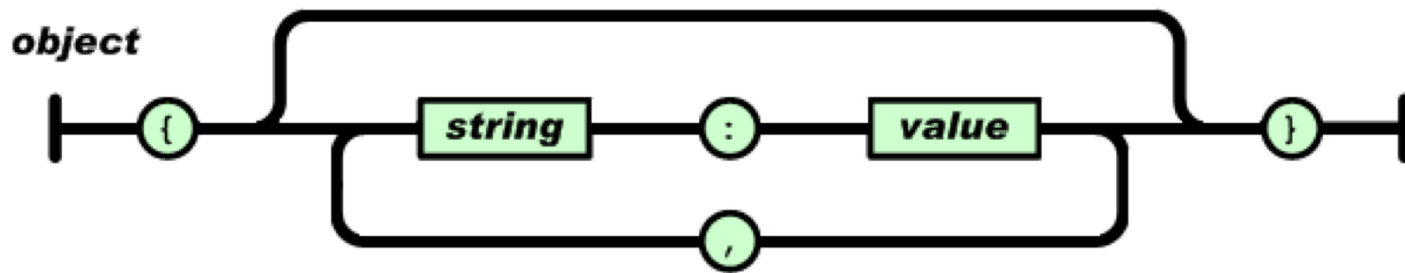
Sample JSON document: Twitter

```
{ "results": [
  { "text": "@twitterapi http://tinyurl.com/ctrefg",
    "to_user_id": 396524,
    "to_user": "TwitterAPI",
    "from_user": "jkoum",
    "metadata": { "result_type": "popular",
    "recent_retweets": 109 },
    "id": 1478555574,
    "from_user_id": 1833773,
    "iso_language_code": "nl",
    "source": "twitter< /a>",

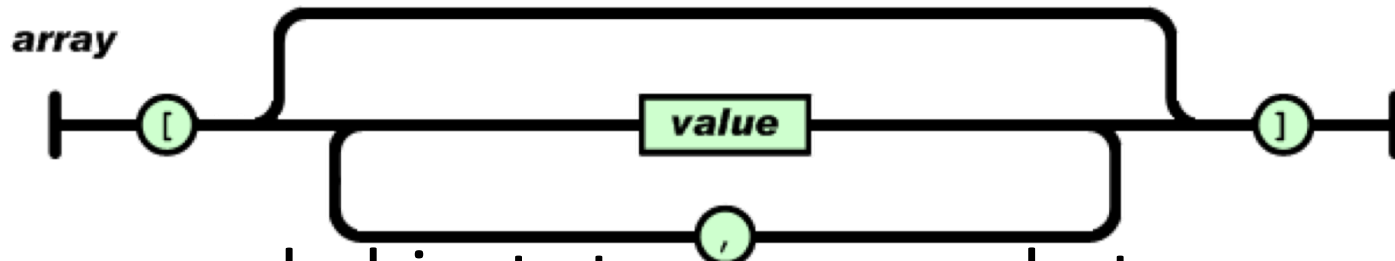
    "profile_image_url": http://s3.amazonaws.com/twitter/a155\_b\_normal.jpg,
    "created_at": "Wed, 08 Apr 2009 19:22:10 +0000"},
    ... truncated ... ],
  "refresh_url": "?since_id=1480307926&q=%40twitterapi",
  "results_per_page": 15,
  "next_page": "?page=2&max_id=1480307926&q=%40twitterapi",
  "completed_in": 0.031704,
  "page": 1,
  "query": "%40twitterapi" }
}
```

JSON document structure

- Object: collection of (name, value) pairs



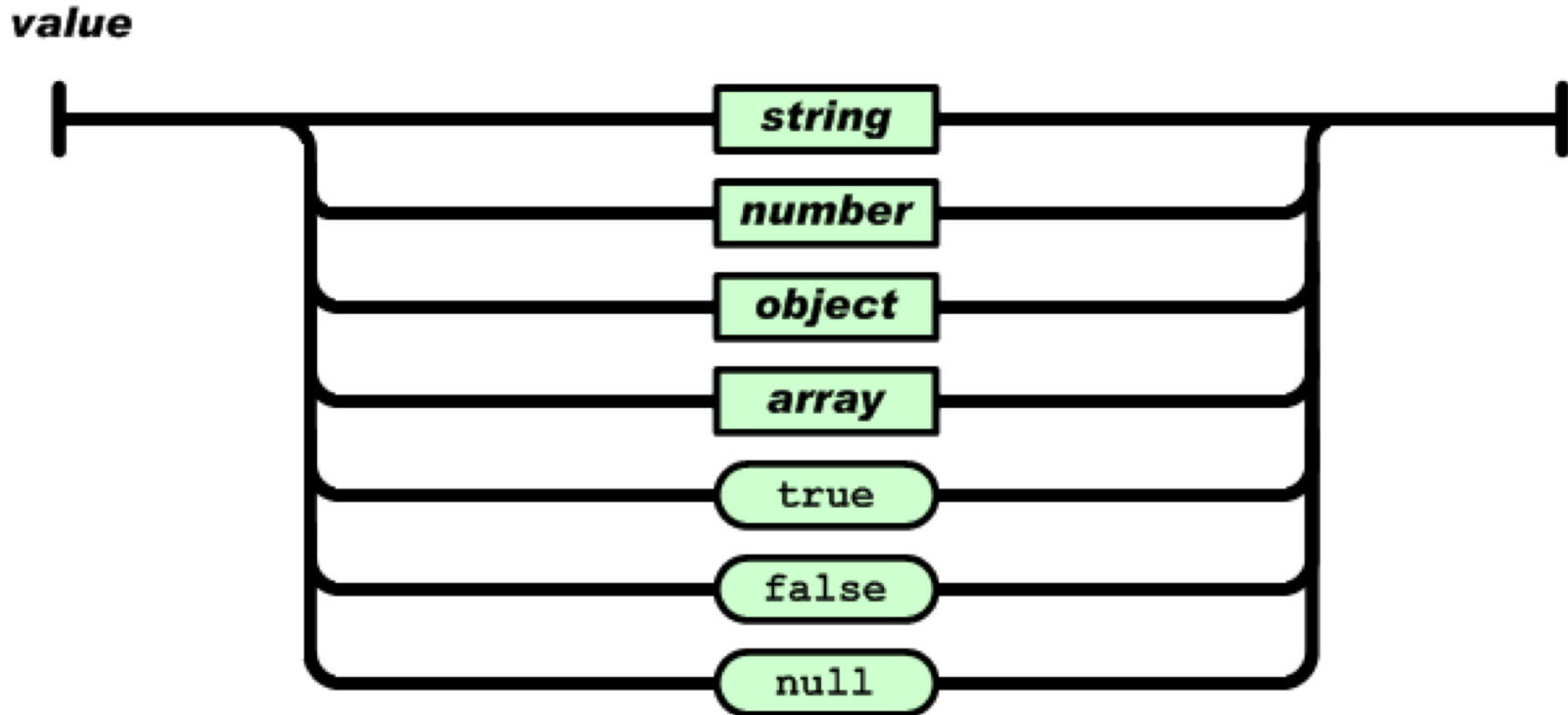
- Array: collection of values



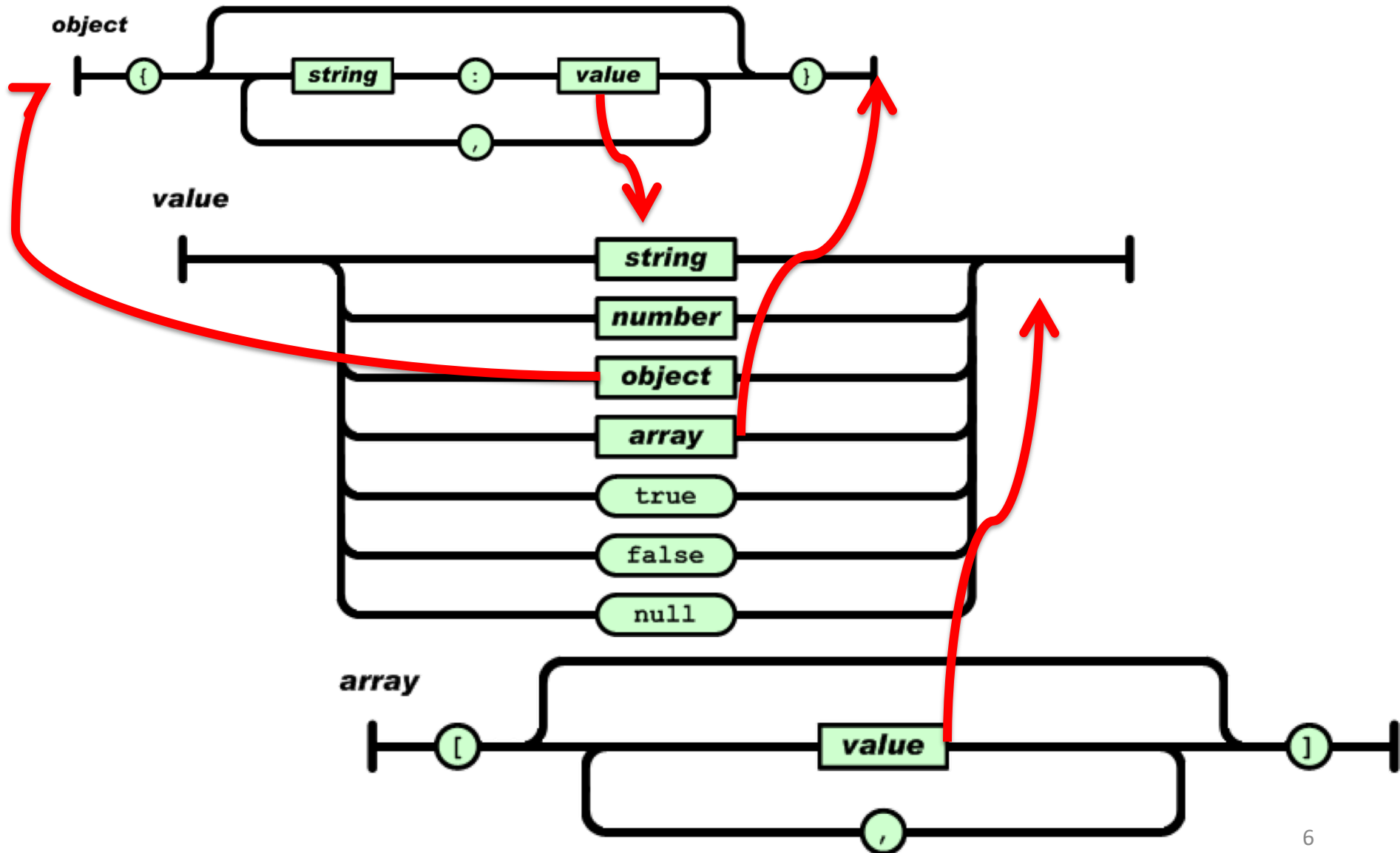
- Arrays and object structure are heterogeneous (no schema)

JSON document structure

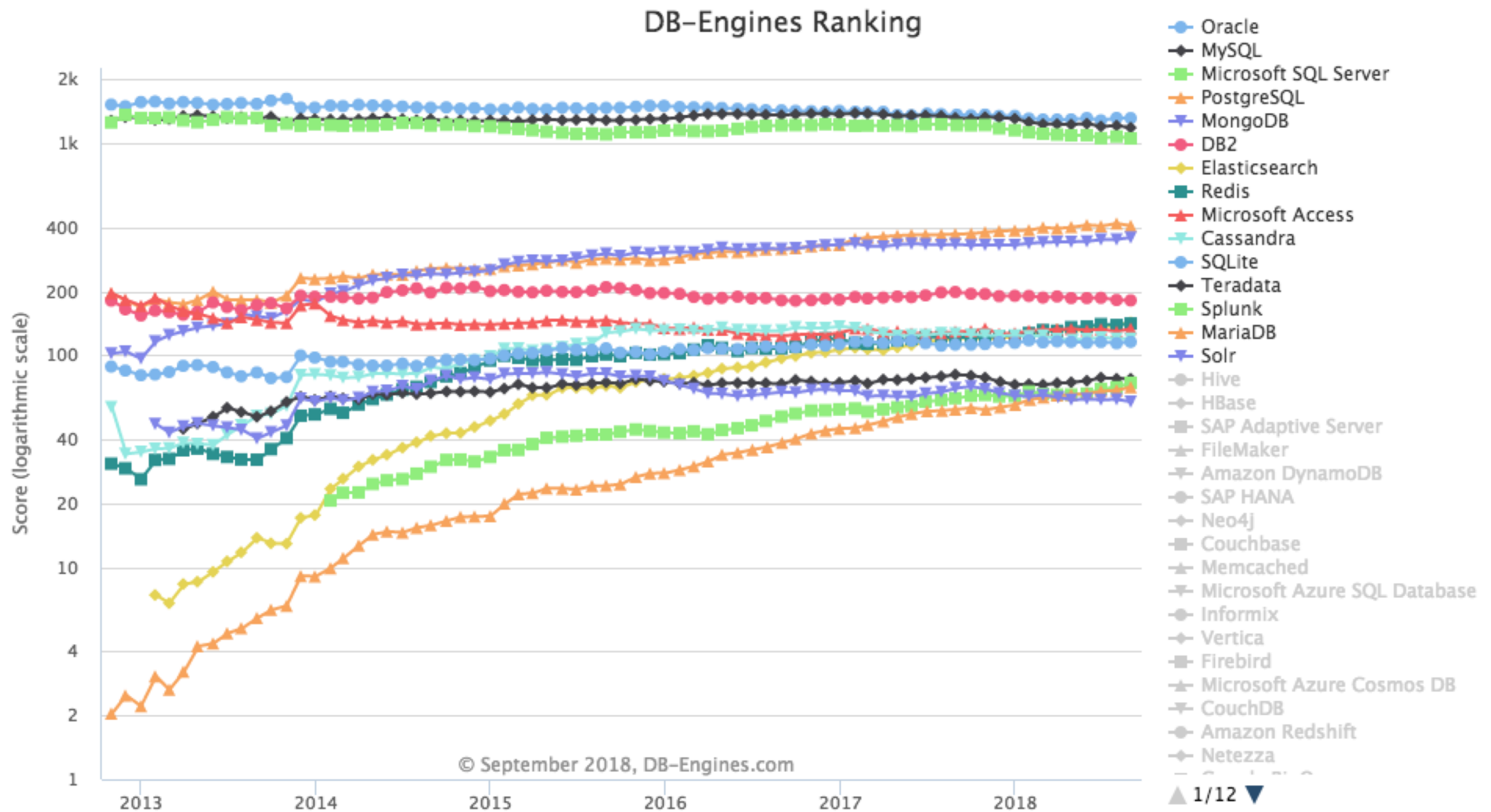
- Values (allow nesting):



JSON document structure



MongoDB: a JSON document store



Computed based on: popularity in search engine results, queries, job offers, social networks, questions on StackOverflow...

MongoDB architecture

- Fundamentally **client-server**
- The server data can be
 - **Replicated**
 - Several identical copies of the same data
 - **Partitioned** (sharded)
 - Distributed across the machines of a cluster in order to take advantage of the storage and processing capacity
- Document processing
 - Selective access
 - Map-Reduce mode

MongoDB storage organization

- **Documents** are stored in **collections** (which may have **indexes**)
 - Collections are part of **databases**
- ```
> mongo myExample // Creates the database myExample
> db.towns.insert({ // Creates the collection towns
 name: "New York", // and inserts a document into it
 population: 22200000,
 last_census: ISODate("2009-07-31"),
 famous_for: ["statue of liberty", "food"],
 mayor : {
 name : "Michael Bloomberg",
 party : "I"
 }
}
```

# MongoDB object IDs

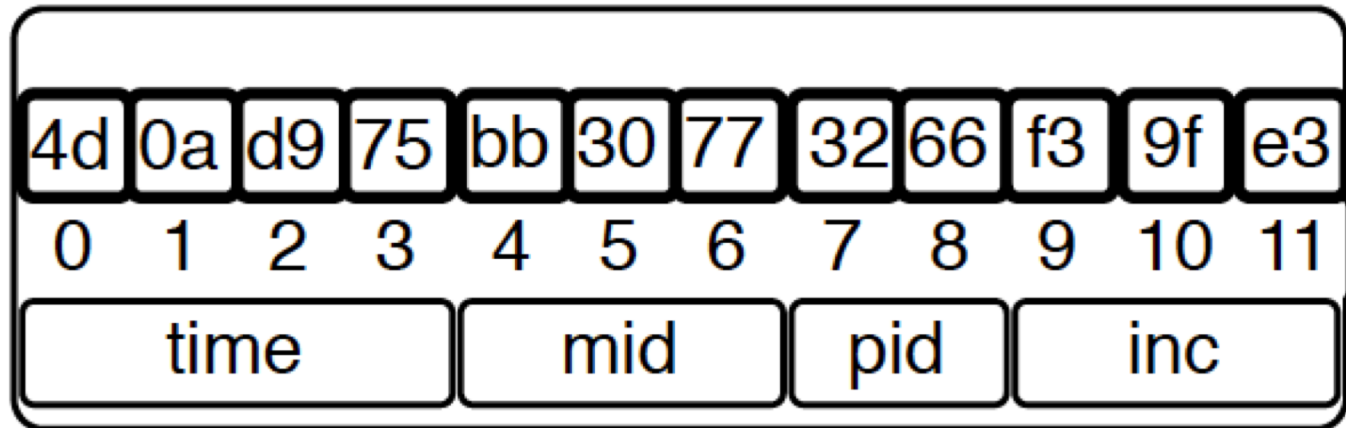
```
> db.towns.find()
{
 "_id" : ObjectId("4d0ad975bb30773266f39fe3"),
 "name" : "New York",
 "population": 22200000,
 "last_census": "Fri Jul 31 2009 00:00:00 GMT-0700
(PDT)",
 "famous_for" : ["statue of liberty", "food"],
 "mayor" : { "name" : "Michael Bloomberg", "party" : "I" }
}
```

\_id is implicitly added by the system

Each object ID is different

# MongoDB object IDs

12 bytes:



Timestamp; client machine ID; process ID;  
incremented counter



IDs are unique across machines and databases

# MongoDB: information about data

It is possible to ask questions about objects, functions etc.

```
> typeof db
object
```

```
> typeof db.towns
object
```

```
> typeof db.towns.insert
function
```

# MongoDB: Operations

- Can find meta-informations about data (e.g., type)
- Allows CRUD operations:
  - Insert / **C**reate
  - Query / **R**ead
  - **U**ppdate
  - **D**eleate

# MongoDB: getting information about data

Calling a function with no parentheses shows the function code

## **db.towns.insert**

```
function (obj, _allow_dot) {
 if (!obj) { throw "no object passed to insert!"; }
 if (!_allow_dot) { this._validateForStorage(obj); }
 if (typeof obj._id == "undefined") {
 var tmp = obj;
 obj = {_id:new ObjectId};
 for (var key in tmp) { obj[key] = tmp[key];}
 }
 this._mongo.insert(this._fullName, obj);
 this._lastID = obj._id;
}
```

# Working with (JavaScript) functions

Typing this into the client shell registers the function:

```
function insertCity(name, population, last_census,
 famous_for, mayor_info) {
 db.towns.insert({name:name, population:population,
 last_census: ISODate(last_census),
 famous_for:famous_for,
 mayor : mayor_info}
);
}
```

*Object implicitly created*

# Working with functions

Calling the function previously defined:

```
insertCity("Punxsutawney", 6200, '2008-31-01',
["phil the groundhog"], {name: "Jim Wehrle" })
```

*Array*

*Object*

```
insertCity("Portland", 582000, '2007-20-09',
["beer", "food"],
{ name : "Sam Adams", party : "D" })
```

db.towns.find() returns three objects



# Searching a MongoDB collection

```
db.towns.find({ "_id" :
ObjectId("4d0ada1fbb30773266f39fe4") })
returns the full object
```

```
db.towns.find({"_id" :
ObjectId("4d0ada1fbb30773266f39fe4") },) },
{ name : 1 })
{
 "_id" :
ObjectId("4d0ada1fbb30773266f39fe4"),
 "name" : "Punxsutawney"
}
```

# Searching a MongoDB collection

To exclude an attribute from a result, **set it to 0** in the find parameter

```
db.towns.find({ _id :
ObjectId("4d0ada1fbb30773266f39fe4") }, { name : 0 })
{
 "_id" : ObjectId("4d0ada1fbb30773266f39fe4"),
 "population" : 6200,
 "last_census" : "Thu Jan 31 2008 00:00:00 GMT-0800
(PST)",
 "famous_for" : ["phil the groundhog"]
}
```

# Searching a MongoDB collection

```
db.towns.find(
 { name : /^P/, population : { $lt : 10000 } },
 //selection
 { name : 1, population : 1 })
 // projection
→
{"name" : "Punxsutawney", "population" : 6200 }
```

```
db.towns.find(
 { last_census : { $lte : ISODate('2008-31-01') } },
 { _id : 0, name: 1 })
→
{ "name" : "Punxsutawney" }
{ "name" : "Portland" }
```

# Searching in nested structures

```
db.towns.find(
 { famous_for : "food" },
 { _id : 0, name : 1, famous_for : 1 })
```

→

```
{ "name" : "New York", "famous_for" : ["statue of
liberty", "food"] }
{ "name" : "Portland", "famous_for" : ["beer", "food"]
}
```

\$all matches an array  
containing all the specified  
values

```
db.towns.find(
 { famous_for : { $all : ["food", "beer"] } },
 { _id : 0, name:1, famous_for:1 })
```

→

```
{ "name" : "Portland", "famous_for" : ["beer", "food"
] }
```

# Searching in nested structures

Nodes which *must not* have a match of the search condition:

```
db.towns.find(
 { famous_for : { $nin : ["food", "beer"] } },
 { _id : 0, name : 1, famous_for : 1 })
→
 { "name" : "Punxsutawney", "famous_for" : ["phil the groundhog"
] }
```

Paths in conditions:

```
db.towns.find({ "mayor.party" : "I" }, { _id : 0, name : 1,
mayor : 1 })
→
{ "name" : "New York",
 "mayor" : { "name" : "Michael Bloomberg", "party" : "I" }
}
```

# Searching in nested structures

Countries that export **bacon** and **tasty food**:

```
db.countries.find({ "exports.foods.name" : "bacon",
"exports.foods.tasty" : true }, { _id : 0, name : 1 })
```

Countries that export **tasty bacon**:

```
db.countries.find(
{"exports.foods" : { $elemMatch : { name : "bacon",

tasty : true}
}}, { _id : 0, name : 1 })
```

Matched by:

```
{_id : "us", name : "United States",
exports : { foods : [{ name : "bacon", tasty : true },
{ name : "burgers" }]}
}
```

# More search operators

**\$regex**: matches PCRE-compliant regexes within / /

**\$ne, \$lt, \$lte, \$gt, \$gte**: arithmetics

**\$exists, \$all, \$in, \$nin, \$or, \$nor, \$not**: logical operators

**\$elemMatch**

**\$size**: matches array of given size

**\$type**: matches if field is of a given type

# Updates

```
db.towns.update(
 { _id: ObjectID("4d0ada87bb30773266f39fe5") },
 { $set : { "state" : "OR" } });
 // updates state
```

```
db.towns.update(
 { _id: ObjectID("4d0ada87bb30773266f39fe5") },
 { { "state" : "OR" } });
 // replaces the whole document!
```

```
db.towns.delete(
 { _id: ObjectID("4d0ada87bb30773266f39fe5") })
 // deletes the document
```



# More operators used in updates

**\$set, \$unset** (removes the field)

**\$inc** (increments)

**\$pop, \$push, \$pushall** for arrays

**\$addToSet** like push but avoids duplicates

**\$pull** removes a matching value from an array

**\$pullAll** removes all matching values

# Other Operations: Aggregate and Joins

- **Aggregation** : pipeline consisting of several stages (see full list on the docs)

```
db.collection.aggregate([{ <stage> }, ...])
```

# Other Operations: Aggregate and Joins

- A stage can be **\$lookup**, implements a *left-outer join*, where one can filter document from an input collection based on another collection

```
db.towns.aggregate([
 {
 $lookup: {
 from: "touristDestinations",
 localField: "name", // field in the orders
collection
 foreignField: "town", // field in the items
collection
 as: "fromTowns"
 }
 }
])
```

# Global processing with custom code

One can always define a JS function and run it

```
db.towns.find(
 (function() { return this.population >
60000;});
 // runs the function over all the towns
```

Fails if one town has no population!

Contrast with XML/XQuery

"OK for extra, not OK for missing"

# Indexing MongoDB data

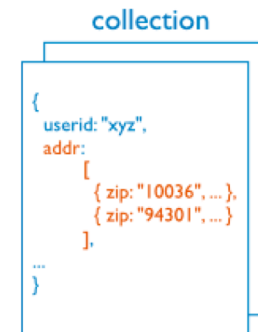
Indexes can be built on a collection calling `collection.createIndex({attr...})`

## 1. B-trees for exact and inequality search

- May be built on a single attribute (simple) or several attributes (compound)
- `Collection.createIndex({"name": 1})` ; *// for ascending order, otherwise use -1*
- B-tree index automatically built on `_id`

## 2. Multikey indexes allow indexing on an array attribute

- Built by default when one requires the indexing of an array attribute



`{ "addr.zip": 1 } Index`

# Indexing MongoDB data

- 3    **Geospatial indexes: 2d** (planar geometry based on x,y ), **2d sphere** (latitude, longitude)
  - Operator: **\$near, \$nearsphere** (coordinates) returns the top k closest documents to the given coordinates
  - Operators: **\$geoWithin, \$geoIntersects** (JSON rectangle)
  
- 4    **Text indexing** for full-text search
  - db.reviews.createIndex( { comments: "text" } )
  - db.collection.createIndex( { "\$\*\*": "text" } )

Inspect indexes: `db.system.indexes.find()`

# Searching with and w/o an index

## Without an index:

```
db.phones.find({display: "+1
800-5650001"}).explain()
```

```
{ "cursor" : "BasicCursor",
 "nscanned" : 109999,
 "nscannedObjects" : 109999,
 "n" : 1,
 "millis" : 52,
 "indexBounds" : { }
}
```

## With an index on display:

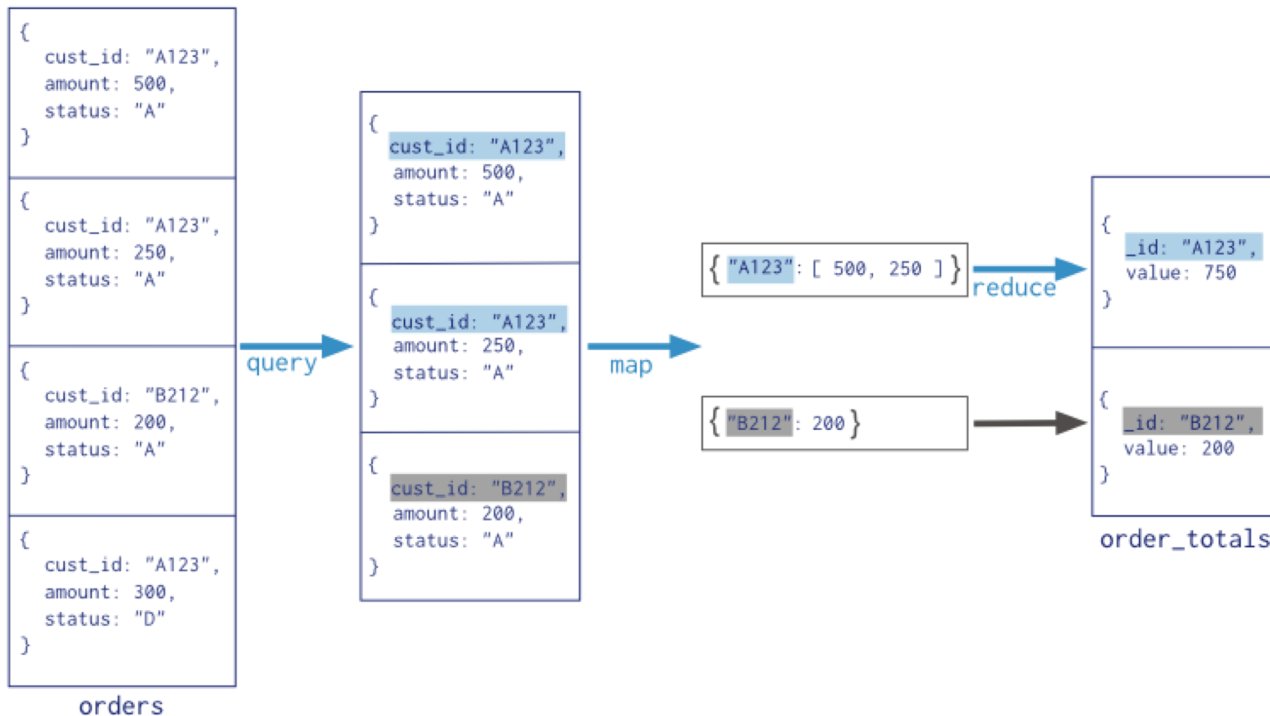
```
db.phones.find({ display: "+1
800-5650001" }).explain()
```

```
{ "cursor" : "BtreeCursor
display_1",
 "nscanned" : 1,
 "nscannedObjects" : 1,
 "n" : 1,
 "millis" : 0,
 "indexBounds" : {
 "display" : [["+1 800-
5650001", "+1 800-5650001"]] }
}
```

# MapReduce processing

Collection  
↓  
db.orders.mapReduce(  
  map     → function() { emit( this.cust\_id, this.amount ); },  
  reduce  → function(key, values) { return Array.sum( values ) },  
  query   → {  
    query: { status: "A" },  
    out: "order\_totals"  
  }  
)

- Execution order:
1. Query
  2. Map
  3. Reduce
  4. [finalize] to wrap up reducer results, e.g. to take the max among all
  5. [output]





# Replica sets

**Duplication (replication) to prevent against server failure and data loss**

Example (three servers): mkdir ./mongo1 ./mongo2 ./mongo3

Create replication set:

```
mongod --replSet book --dbpath ./mongo1 --port 27011 --rest
```

```
mongod --replSet book --dbpath ./mongo2 --port 27012 --rest
```

```
mongod --replSet book --dbpath ./mongo3 --port 27013 --rest
```

Then in one of the servers initialize replication set:

```
mongo localhost:27011
```

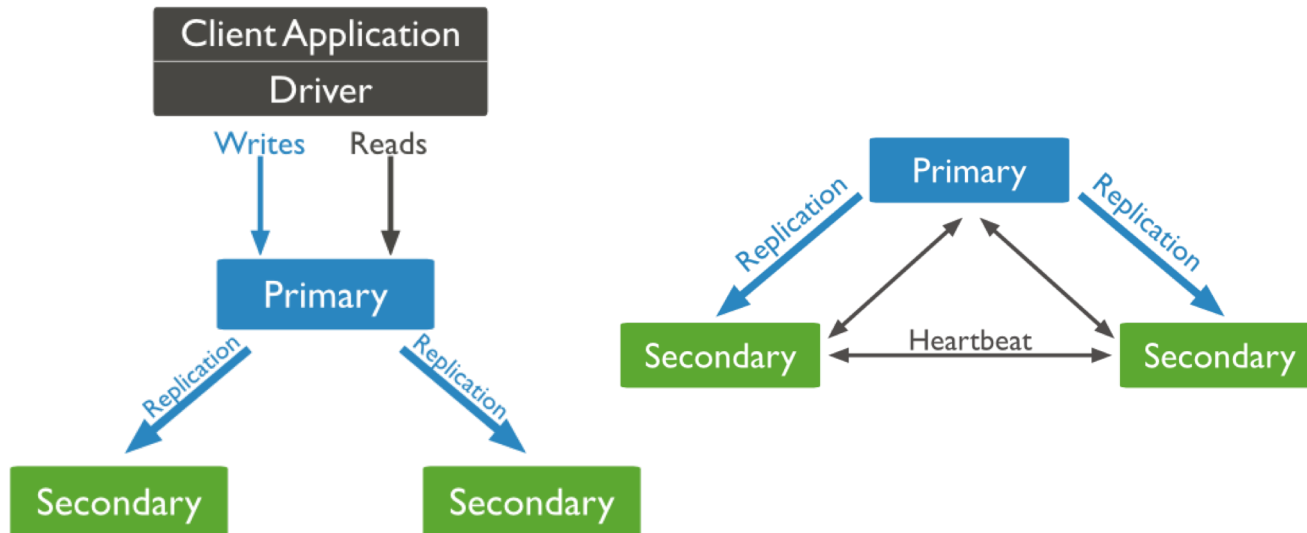
```
> rs.initiate({_id: 'book',
 members: [{_id: 1, host: 'localhost:27011'},
 {_id: 2, host: 'localhost:27012'},
 {_id: 3, host: 'localhost:27013'}] })
```

Then one server will output `[rs Manager] replSet PRIMARY`  
while two will output `[rs sync] replSet SECONDARY`

# Replica sets

The servers held a vote to determine who is the master (primary); the two others are replicas ("secondary")

By default, applications read/write through the primary, who pushes updates to secondary servers

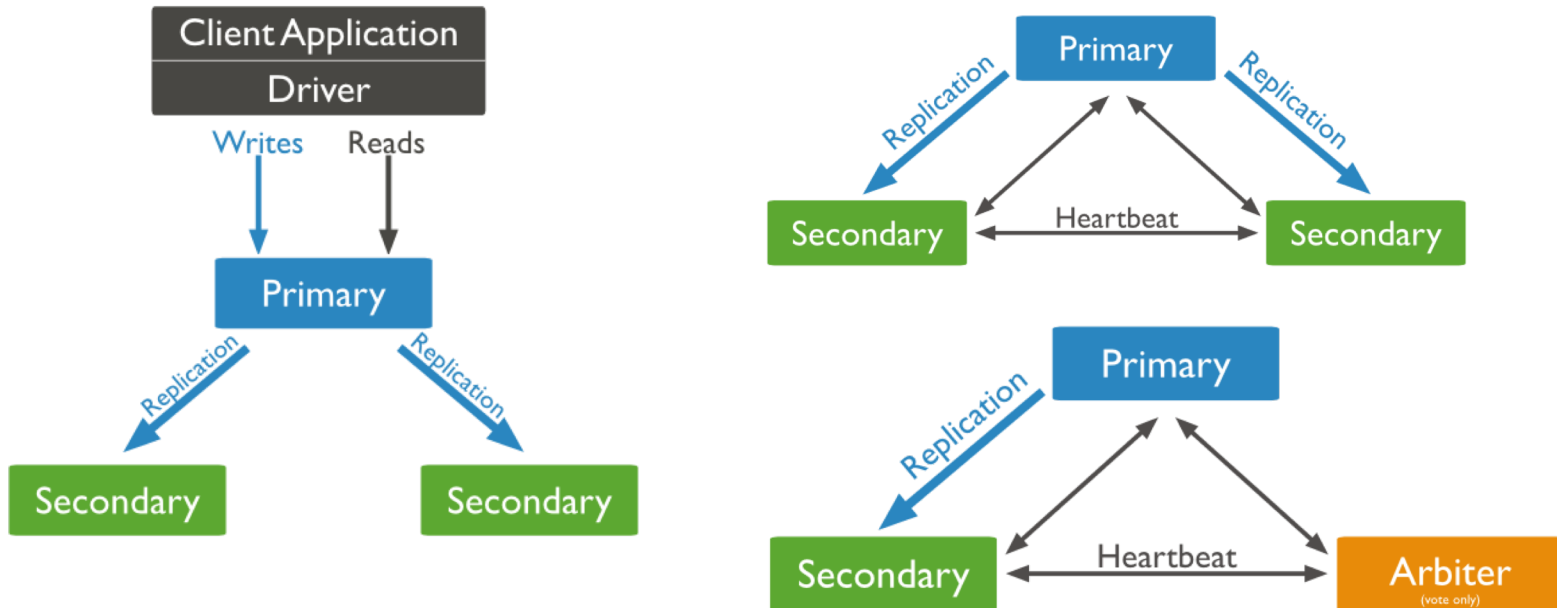


If the master is considered dead, there are new elections

- *Only succeed if more than half of the original replication set votes*
- Operations attempted on a demoted (dead) master are lost
- A write is considered successful only if  $>$  half of the replicas "saw" it

# Replica sets

By default, write to the primary, who pushes updates to secondary servers



MongoDB recommends an *odd number of server in a replica set*, to allow a majority in case of network failure. **One arbiter** may be added to the replica set

- Strong consistency on read
- Resistance to some partitioning

# MongoDB sharding

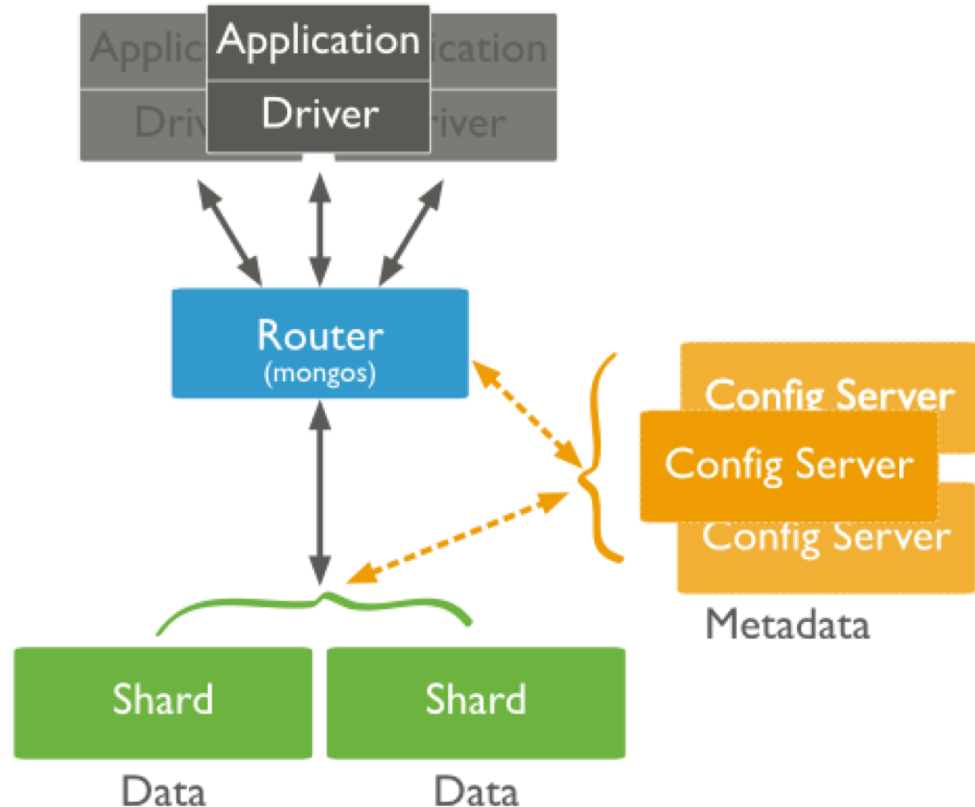
*Sharding* = partitioning

1 shard = 1 fragment

- To distribute a very large collection across several servers

Sharding is logically *on top of replication*

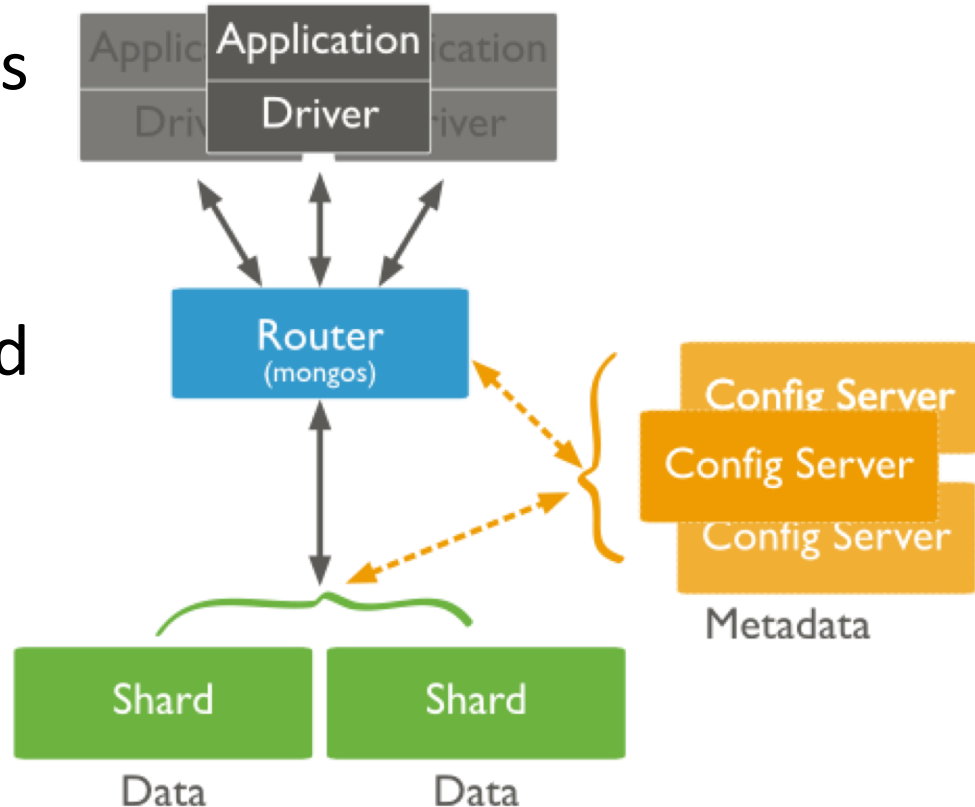
- Each shard server may participate to a replica set



# MongoDB sharding

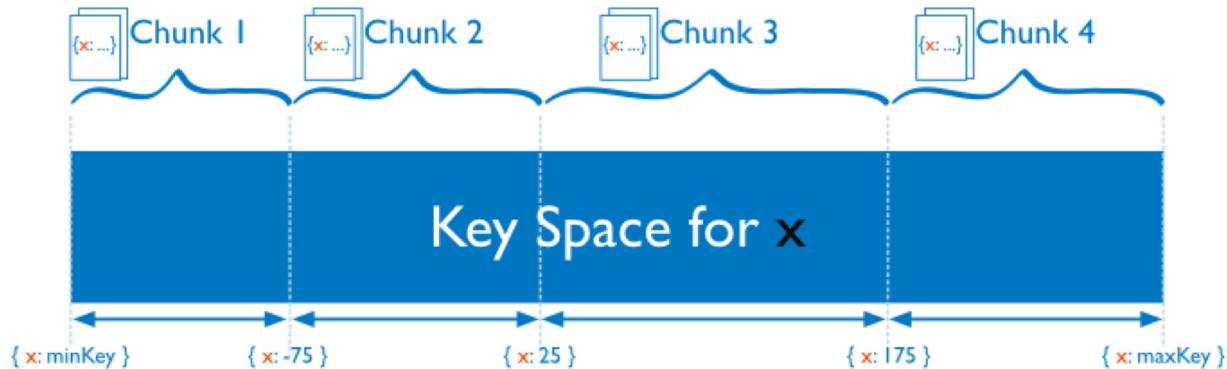
## Roles:

- **Shard** (shard server): stores a collection fragment
- **Config server(s)**: store(s) information on which shard has what
- **Single point of entry:** mongos



# MongoDB sharding

- A data collection is partitioned into **chunks** based on the value of a **shard key**
- Each chunk covers a key range



- 1 shard = a set of chunks
- Mongos routes writes to the appropriate chunk based on the shard key value
- Chunks are split when they grow beyond a fixed chunk size (64 MB default, can be split)
- MongoDB migrates chunks across shards for load balancing

# MongoDB sharding

*// starting the shard servers:*

```
mkdir ./mongo4 ./mongo5
```

```
mongod --shardsvr --dbpath ./mongo4 --port 27014
```

```
mongod --shardsvr --dbpath ./mongo5 --port 27015
```

*// starting the config server:*

```
mkdir ./mongoconfig
```

```
mongod --configsvr --dbpath ./mongoconfig --port 27016
```

*// starting mongos connected to the config*

```
mongos --configdb localhost:27016 --chunkSize 1 --port 27020
```

*// talking to mongos to configure sharding:*

```
mongo localhost:27020/admin
```

```
> db.runCommand({ addshard : "localhost:27014" }) -> { "shardAdded" :
"shard0000", "ok" : 1 }
```

```
> db.runCommand({ addshard : "localhost:27015" }) -> { "shardAdded" :
"shard0001", "ok" : 1 }
```

```
> db.runCommand({ enablesharding: "test"}) -> { "ok" : 1 }
```

```
> db.runCommand({ shardcollection : "test.cities", key : {name : 1} }) // {
"collectionsharded" : "test.cities", "ok" : 1 }
```

# Another JSON store: CouchDB

<http://db-engines.com/en/system/CouchDB%3BMongoDB>

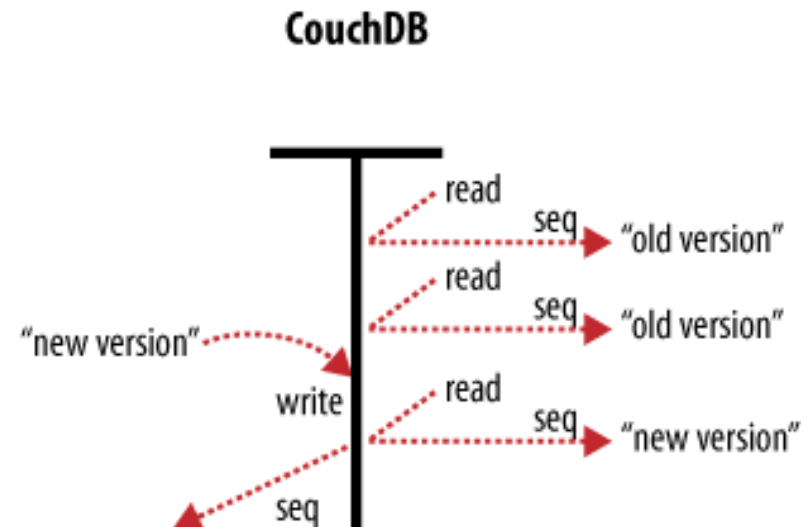
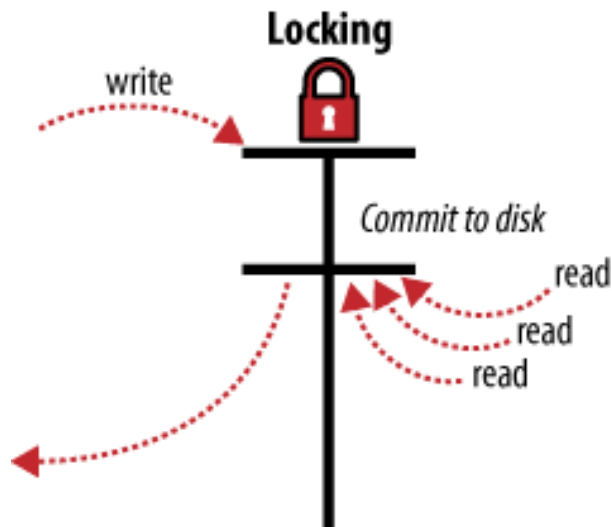
| Feature                       | CouchDB                                               | MongoDB                                       |
|-------------------------------|-------------------------------------------------------|-----------------------------------------------|
| Since                         | 2005                                                  | 2009                                          |
| Ranking                       | #23 overall, #2 document store                        | #4 overall, #1 document store                 |
| From                          | Apache Software                                       | MongoDB Inc                                   |
| APIs and other access methods | RESTful HTTP/JSON API                                 | proprietary protocol using JSON               |
| Replication methods           | Master-master replication<br>Master-slave replication | Master-slave replication                      |
| MapReduce                     | yes                                                   | yes                                           |
| Consistency concepts          | Eventual Consistency                                  | Eventual Consistency<br>Immediate Consistency |
| Foreign keys                  | no                                                    | no                                            |

More document stores: [http://db-engines.com/en/ranking\\_trend/document+store](http://db-engines.com/en/ranking_trend/document+store)



# Concurrency control in CouchDB

- Update granularity = document
- To change a document's attribute, rewrite the document!
- Multi-Version Concurrency Control: some requests may return "old" versions but they each return a version that was valid at some point



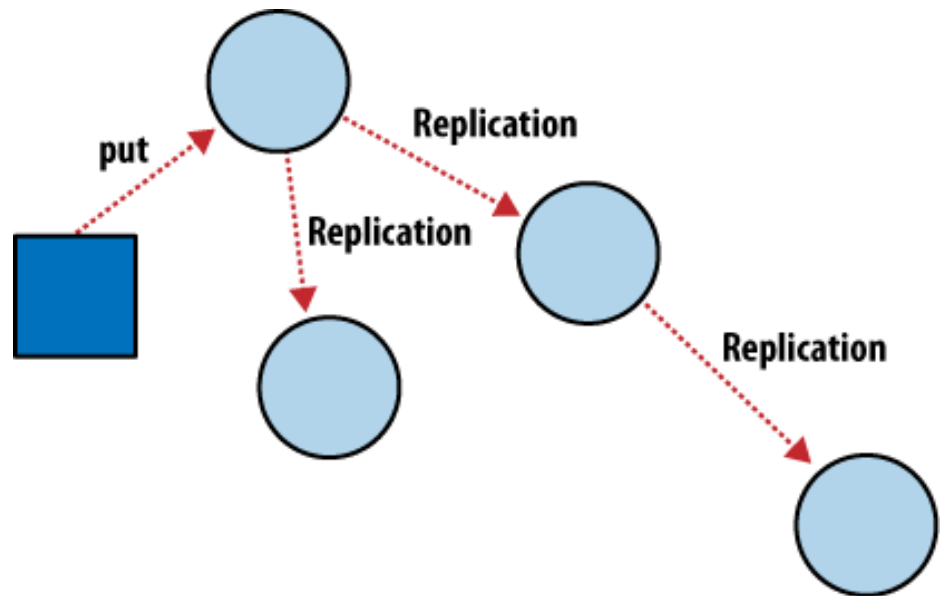
# Synchronization in CouchDB

Each server can work independently

**Incremental replication** can be set to run in the background

**Synchronization** is on demand between any pair of servers

Diverging changes are flagged as **conflicts**; conflict resolution policy must be specified. One document version wins, the other is considered older.



# Combining information from several documents

MongoDB does not provide all types of joins!

- In most cases, data that should be used together is stored in the same document(s)

To combine data from several documents, two options:

1. Store the ID of a document within another:

```
original_id = ObjectId()
db.places.insert({ "_id": original_id,
 "name": "Broadway Center",
 "url": "bc.example.net" })
db.people.insert({
 "name": "Erin",
 "places_id": original_id,
 "url": "bc.example.net/Erin" })
```

Collection and database  
are not specified!  
Either consider the same  
or extra logic is needed.

# Combining information from several documents

To combine data from several documents, two options:

2. Use a **DBRef** instance, which is recognized as a pointer to another document:

```
{ "$ref" : <value>, "$id" : <value>, "$db" : <value> }
```

"\$ref" points to the collection; "\$id" points to the document; "\$db" points to the database

```
{ "_id" : ObjectId("5126bbf64aed4daf9e2ab771"), // ..
 "creator" : { "$ref" : "creators", "$id" :
 ObjectId("5126bc054aed4daf9e2ab772"), "$db" : "users" } }
```

Recognized in some language drivers, not in all

E.g. Java, Python, Perl, PHP OK

E.g. C, C++, Scala: not supported (yet)