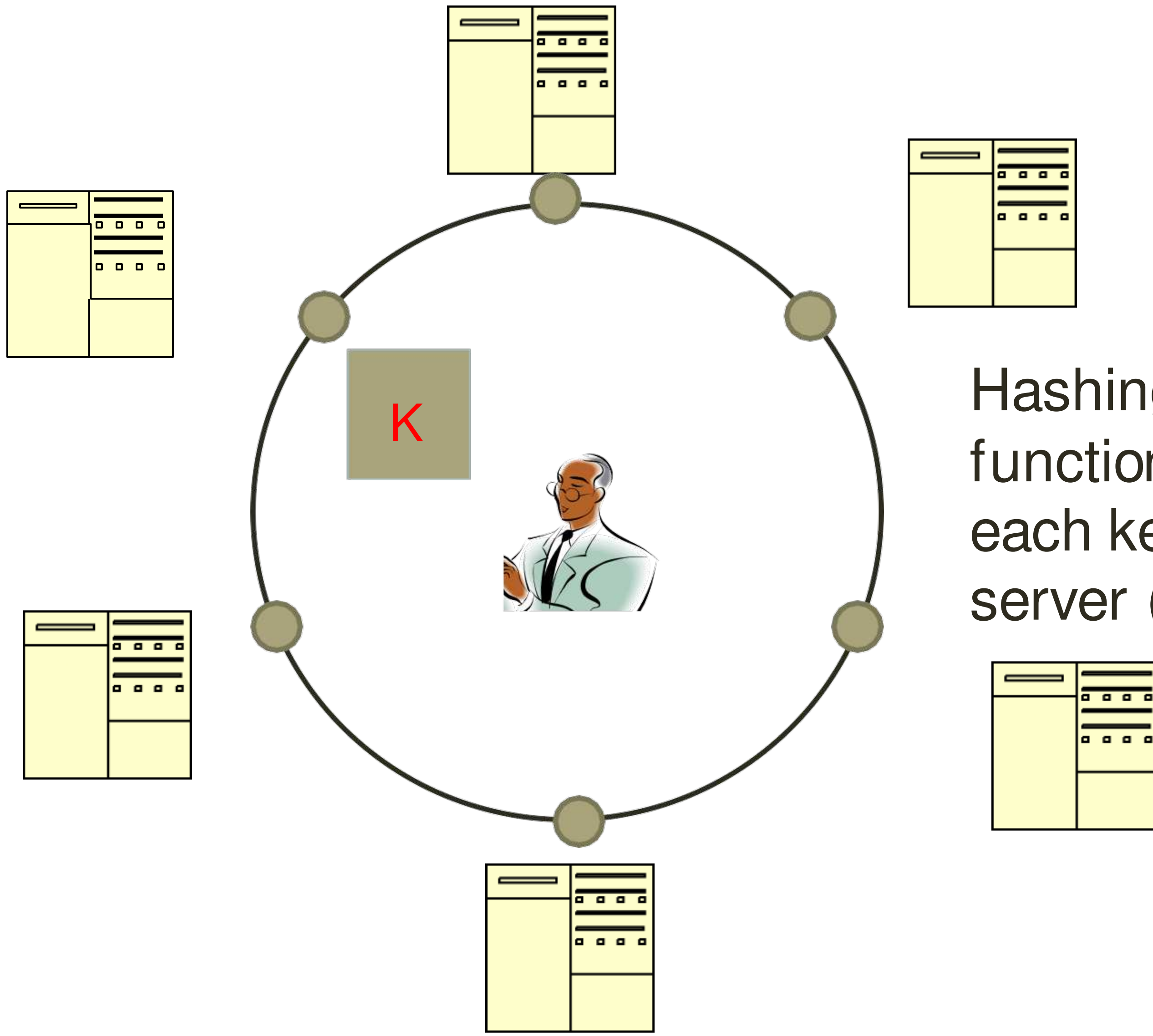
Document stores

Taxonomy of NoSQL

- Key-value
- Graph database
- Document-oriented
- Column family

Typical NoSQL architecture



Hashing function maps each key to a server (node)

Document stores

- Flexible schema
- JSON/BSON documents
  - Embedded documents
  - Referenced documents

- CouchDB: http://couchdb.apache.org/
- MongoDB: http://www.mongodb.org/

We'll use the mongo shell for class, but if you want to use a GUI to interact with MongoDB, you may want to look into Robo 3T (previously robomongo) https://robomongo.org/

What is MongoDB?

- Developed by 10gen
  - Founded in 2007
- A document-oriented, NoSQL database
  - Hash-based, *schema-less database*
    - No Data Definition Language
    - In practice, this means you can store hashes with any keys and values that you choose
      - Keys are a basic data type but in reality stored as strings
      - Document Identifiers (_id) will be created for each document, field name reserved by system
    - Application tracks the schema and mapping
    - Uses BSON format
      - Based on JSON – B stands for Binary
- Written in C++
- Supports APIs (drivers) in many computer languages

MongoDB Features

- Dynamic schema
- Document-Oriented storage
- Full Index Support
- Replication & High Availability
- Auto-Sharding
  - Built-in horizontal scaling via automated range-based partitioning of data
- Querying
- Fast In-Place Updates
- Map/Reduce functionality

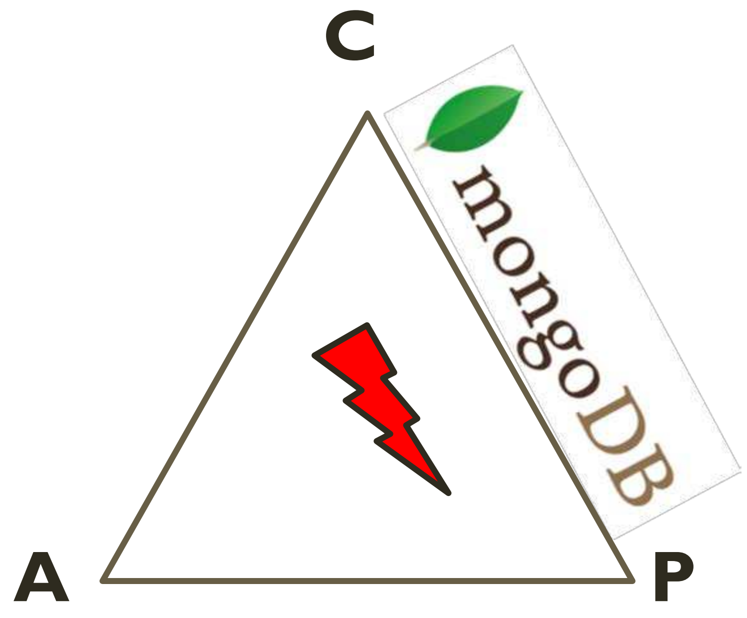
Agile

Scalable

MongoDB: CAP approach

Focus on Consistency and Partition tolerance

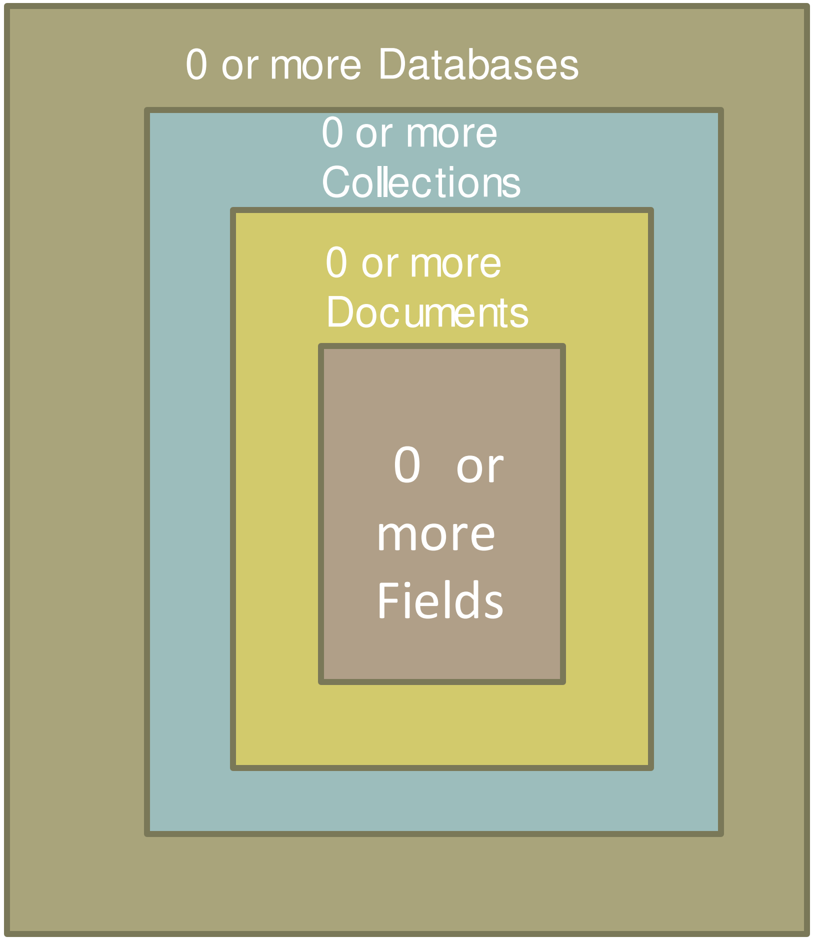
- **C**onsistency
  - all replicas contain the same version of the data
- **A**vailability
  - system remains operational on failing nodes
- **P**artition tolarence
  - multiple entry points
  - system remains operational on system split



CAP Theorem:
satisfying all three at the same time is impossible

MongoDB: Hierarchical Objects

- A MongoDB instance may have zero or more 'databases'
- A database may have zero or more 'collections'.
- A collection may have zero or more 'documents'.
- A document may have one or more 'fields'.
- MongoDB 'Indexes' function much like their RDBMS counterparts.



MongoDB

RDBMS		MongoDB
Database	⇨	Database
Table, View	⇨	Collection
Row	⇨	Document (JSON, BSON)
Column	⇨	Field
Index	⇨	Index
Join	⇨	Embedded Document
Foreign Key	⇨	Reference
Partition	⇨	Shard

Mongo hits a sweet spot between the powerful queryability of a relational database and the distributed nature of other databases

JSON format

- Data is in name / value pairs
- A name/value pair consists of a field name followed by a colon, followed by a value:
  - Example: “name”: “R2-D2”
- Data is separated by commas
  - Example: “name”: “R2-D2”, race : “Droid”
- Curly braces hold objects
  - Example: {“name”: “R2-D2”, race : “Droid”, affiliation: “rebels”}
- An array is stored in brackets []
  - Example [{“name”: “R2-D2”, race : “Droid”, affiliation: “rebels”},
  - {“name”: “Yoda”, affiliation: “rebels”}]

Document store

RDBMS		MongoDB
Database	⇨	Database
Table, View	⇨	Collection
Row	⇨	Document (JSON, B
Column	⇨	Field
Index	⇨	Index
Join	⇨	Embedded Docume
Foreign Key	⇨	Reference
Partition	⇨	Shard

```
> db.user.findOne({age:39})
{
        "_id" : ObjectId("5114e0bd42…"),
        "first" : "John",
        "last" : "Doe",
        "age" : 39,
       "interests" : [
                "Reading",
                "Mountain Biking ]
       "favorites": {
               "color": "Blue",
               "sport": "Soccer"}
}
```

CRUD

Create

    db.collection.insert(<document>)
    db.collection.save(<document>)
    db.collection.update(<query>, <update>, { upsert: true })

Read

    db.collection.find(<query>, <projection>)
    db.collection.findOne(<query>, <projection>)

Update

    db.collection.update(<query>, <update>, <options>)

Delete

    db.collection.remove(<query>, <justOne>)

CRUD example

```
> db.user.insert({
	first: "John",
	last : "Doe",
	age: 39
})
```

```
> db.user.find ()
{
	"_id" : ObjectId("51…"),
	"first" : "John",
	"last" : "Doe",
	"age" : 39
}
```

```
> db.user.update(
	{"_id" : ObjectId("51…")},
	{
		$set: {
			age: 40,
			salary: 7000}
	}
)
```

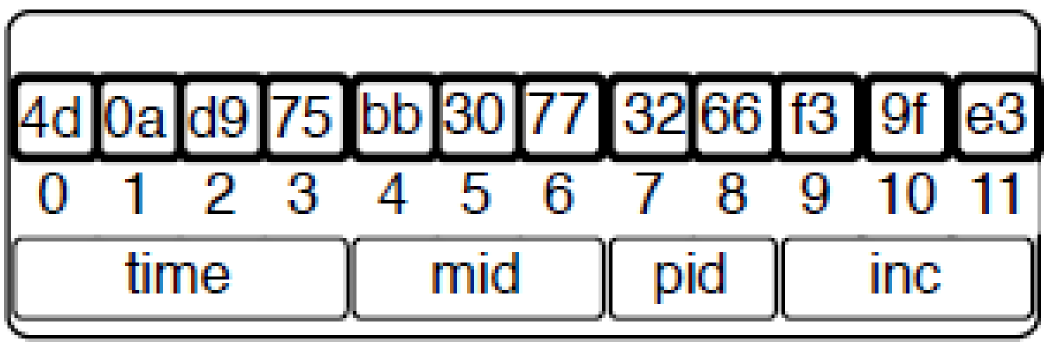
```
> db.user.remove({
	"first": /^J/
})
```

Let’s get started – command-line fun

```
> mongo moderndb

> db.towns.insert({
name: "New York", population:
22200000,
lastCensus: ISODate("2016-07-
01"),
famousFor: [ "the MOMA",
"food", "Derek Jeter" ], mayor
: {
    name : "Bill de Blasio",
    party : "D"
}
})

> show collections

> db.towns.find()
```

ObjectId

- _id field of type ObjectId.
- Akin to SERIAL incrementing a numeric primary key in PostgreSQL.
- The ObjectId is always 12 bytes, composed of a timestamp, client machine ID, client process ID, and a 3-byte incremented counter.



Javascript

- Native tong of MongoDB

- Ask for help
  - ➢ db.help()
  - ➢ db.towns.help()

- Object and functions
  - ➢ typeof db
  - ➢ typeof db.towns
  - ➢ typeof db.towns.insert

- Get source code of a function (call it without parameters)
  - ➢ db.towns.insert

Let’s insert data using our own function

```
function insertCity(name, population, lastCensus, famousFor,
mayorInfo) {
    db.towns.insert({ name: name,
    population: population, lastCensus: ISODate(lastCensus),
    famousFor: famousFor,

    mayor : mayorInfo });
}
```

Now we can call it

```
> insertCity("Punxsutawney", 6200, '2016-01-31', ["Punxsutawney
Phil"], { name : "Richard Alexander" }
)

> insertCity("Portland", 582000, '2016-09-20',
["beer", "food", "Portlandia"], { name : "Ted Wheeler", party :
"D" })
```

Querying data

```
db.towns.find({ "_id" : ObjectId("59094288afbc9350ada6b807") })

db.towns.find({ _id : ObjectId("59094288afbc9350ada6b807") }, {
name : 1 })

db.towns.find({ _id : ObjectId("59094288afbc9350ada6b807") }, {
name : 0 })
```

Perl-compatible regular expression (PCRE)

```
db.towns.find(
{ name : /^P/, population : { $lt : 10000 } },
{ _id: 0, name : 1, population : 1 }
)
```

Can construct operations as you would objects

```
> var population_range = {$lt: 1000000, $gt: 10000}
> db.towns.find( { name : /^P/, population : population_range },{
name: 1 })

> db.towns.find(
{ lastCensus : { $gte : ISODate('2016-06-01') } },
{ _id : 0, name: 1 })
```

Querying nested array data

Matching exact values

```
> db.towns.find(
{ famousFor : 'food' },
{ _id : 0, name : 1, famousFor : 1 }
)
```

Matching partial values:

```
> db.towns.find(
    { famousFor : /moma/ },
    { _id : 0, name : 1, famousFor : 1 })
```

Query by all matching values:

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

Or the lack of matching values:

```
> db.towns.find(
    { famousFor : { $nin : ['food', 'beer'] } },
    { _id : 0, name : 1, famousFor : 1 }
)
```

Querying nested documents

Find towns with mayors from the Democratic party

```
> db.towns.find(
{ 'mayor.party' : 'D' },
{ _id : 0, name : 1, mayor : 1 }
```

Find towns with mayors who don't have a party

```
> db.towns.find(
{ 'mayor.party' : { $exists : false } },
{ _id : 0, name : 1, mayor : 1 }
)
```

New collection for countries

```
> db.countries.insert({
      _id : "us",
      name : "United States",
      exports : {
      foods : [
            { name : "bacon", tasty : true },
            { name : "burgers" }
      ]}
  })
> db.countries.insert({
      _id : "ca",
      name : "Canada",
      exports : {
      foods : [
            { name : "bacon", tasty : false },
            { name : "syrup", tasty : true }
      ]}
  })
> db.countries.insert({
      _id : "mx",
      name : "Mexico",
      exports : {
      foods : [{
            name : "salsa", tasty : true, condiment : true
              }]}
  })
```

elemMatch

Find a country that not only exports bacon but exports tasty bacon

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

Canada?

ElemMatch to the rescue:

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

```
> db.countries.find(
  {
    'exports.foods' : {
        $elemMatch : {
            tasty : true,
            condiment : { $exists : true }
        }
    }
  },
  { _id : 0, name : 1 }
  )
```

Boolean operators

```
> db.countries.find(
      { _id : "mx", name : "United States" },
      { _id : 1 }
)


db.countries.find(
      {
            $or : [
                  { _id : "mx" },
                  { name : "United States" }
            ]
      },
      { _id:1 }
)
```

Some mongodb commands

Command	Description
$regex	Match by any PCRE-compliant regular expression string (or just use the // delimiters as shown earlier)
$ne	Not equal to
$lt	Less than
$lte	Less than or equal to
$gt	Greater than
$gte	Greater than or equal to
$exists	Check for the existence of a field
$all	Match all elements in an array
$in	Match any elements in an array
$nin	Does not match any elements in an array
$elemMatch	Match all fields in an array of nested documents
$or	or
$nor	Not or
$size	Match array of given size
$mod	Modulus
$type	Match if field is a given datatype
$not	Negate the given operator check

- Check the mongo documentation for a complete list
- Cheat sheet on ICON

For today..

- Select a town via a case-insensitive regular expression containing the word *new*.
- Find all towns whose names contain an *e* and are famous for food or beer.
- Summit the two queries to ICON.

CRUD in MongoDB

https://docs.mongodb.com/guides/

CRUD

Create

db.collection.insert(<document>)

db.collection.save(<document>)

db.collection.update(<query>, <update>, { upsert: true })

Read

db.collection.find(<query>, <projection>)

db.collection.findOne(<query>, <projection>)

Update

db.collection.update(<query>, <update>, <options>)

Delete

db.collection.remove(<query>, <justOne>)

Examples

In RDBMS

```
CREATE TABLE users (
    id MEDIUMINT NOT NULL
        AUTO_INCREMENT,
    user_id Varchar(30),
    age Number,
    status char(1),
    PRIMARY KEY (id)
)
```

```
DROP TABLE users
```

In MongoDB

Either insert the 1st docuement

```
db.users.insert( {
    user_id: "abc123",
    age: 55,
    status: "A"
 } )
```

Or create "Users" collection explicitly

```
db.createCollection("users")
```

```
db.users.drop()
```

https://docs.mongodb.com/manual/core/schema-validation/#schema-validation-json

Insert one document

- From the mongo shell

- Switch to the moderndb database
  - use moderndb

```
db.inventory.insertOne(
  { "item" : "canvas",
    "qty" : 100,
    "tags" : ["cotton"],
    "size" : { "h" : 28, "w" : 35.5, "uom" : "cm" }
  }
)
```

Update

`db.users.update(`	collection
`{ age: { $gt: 18 } },`	update criteria
`{ $set: { status: "A" } },`	update action
`{ multi: true }`	update option
`)`	

Otherwise, it will update only the 1st matching document

Equivalent to in SQL:

`UPDATE users`	table
`SET status = 'A'`	update action
`WHERE age > 18`	update criteria

UpdateOne - UpdateMany

```
db.inventory.updateOne(
   { "item" : "paper" }, // specifies the document to update
   {
     $set: {  "size.uom" : "cm",  "status" : "P" },
     $currentDate: { "lastModified": true }
   }
)


db.inventory.updateMany(
   { "qty" : { $lt: 50 } }, // specifies the documents to update
   {
     $set: { "size.uom" : "cm", "status": "P" },
     $currentDate : { "lastModified": true }
   }
)
```

Update (Cont'd)

Two operators

```
db.inventory.update(
    { item: "MNO2" },
    {
      $set: {
        category: "apparel",
        details: { model: "14Q3", manufacturer: "XYZ Company" }
      },
      $currentDate: { lastModified: true }
    }
)
```

For the document with `item` equal to `"MNO2"`, use the `$set` operator to update the `category` field and the `details` field to the specified values and the `$currentDate` operator to update the field `lastModified` with the current date.

Replace a document

Query Condition

New doc

```
db.inventory.update(
    { item: "BE10" },
    {
      item: "BE05",
      stock: [ { size: "S", qty: 20 }, { size: "M", qty: 5 } ],
      category: "apparel"
    }
)
```

For the document having item = “BE10”, replace it with the given document

Insert or Replace

```
db.inventory.update(
   { item: "TBD1" },
   {
     item: "TBD1",
     details: { "model" : "14Q4", "manufacturer" : "ABC Company" },
     stock: [ { "size" : "S", "qty" : 25 } ],
     category: "houseware"
   },
   { upsert: true }
)
```

The ***upsert*** option

If the document having item = “TBD1” is in the DB, it will be replaced
Otherwise, it will be inserted.

Delete

- Deletes the first document that matches the condition

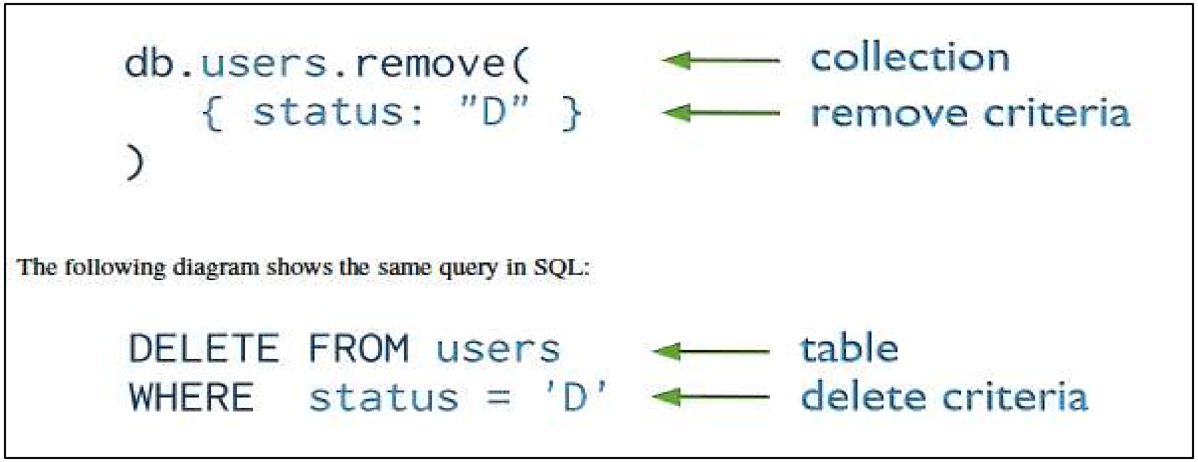
```
db.inventory.deleteOne(
    { "status": "D" } // specifies the document to delete
)
```

- Deletes ALL documents that match the condition

```
db.inventory.deleteMany(
    { "status" : "A" } // specifies the documents to
delete
)
```

Remove (also delete)

You can put condition on any field in the document (even *_id*)



db.users.remove ( ) → Removes all documents from *users* collection

Import json file to MongoDB

https://docs.mongodb.com/guides/server/import/

Download the file:
https://raw.githubusercontent.com/mongodb/docs-assets/primer-dataset/inventory.crud.json

```
mongoimport --db moderndb --collection inventory
        --drop --file ~\downloads\inventory.crud.json
```

Or if you enabled authentication:

```
mongoimport --db moderndb --collection inventory
        --authenticationDatabase admin --username <user>
        --password <password>  --drop
        --file ~\downloads\inventory.crud.json
```

References in Mongo

- Manual references is the practice of including one document's `_id` field in another document. The application can then issue a second query to resolve the referenced fields as needed

- DBRefs are references from one document to another using the value of the first document's _id field, collection name, and, optionally, its database name.

```
db.inventory.update(
{ item : “paper" },
{ $set : { country: { $ref: "countries", $id: "us" } } }
)
```

Retrieving references

```
var paper = db. inventory.findOne({ item : “paper" })
```

Retrieve country, to query the countries collection using the stored $id.

```
db.countries.findOne({ _id: paper.country.$id })
```

Better yet, in JavaScript, you can ask the document the name of the collection stored in the fields reference.

```
var paperCountryRef = paper.country.$ref;
db[paperCountryRef].findOne({ _id: paper.country.$id })
```

The last two queries are equivalent; the second is just a bit more data-driven.

Querying with code

- You can request that MongoDB run a decision function across your documents
- Should be a last resort, this queries cannot be indexed, Mongo do not optimize them

```
db.inventory.find(function() {
    return this.qty > 50 && this.qty < 100;
})
```

- You can also use the $where clause

```
db.inventory.find({$where: "this.qty > 50 && this.qty < 100"})
```

The _id index

- Mongo automatically creates an index by the _id

```
db.inventory.getIndexes()
```

```
db.getCollectionNames().forEach(function(collection) {
    print("Indexes for the " + collection + " collection:");
    printjson(db[collection].getIndexes());
});
```

Let's import the city_inspections.json collection from ICON into the moderndb database, on a new collection called city_inspections

```
db.city_inspections.find({certificate_number:
10003581}).explain("executionStats").executionStats
```

Profiler

- *System profiler* allows to profile queries in a normal test or production environment
  - Level 1 stores only slower queries greater than 100 milliseconds
  - Level 2 stores all queries

```
db.setProfilingLevel(2)
db.city_inspections.find({certificate_number: 10003581})
```

This will create a new object in the system.profile collection, which you can read as any other table to get information about the query, such as a timestamp for when it took place and performance information (such as executionTimeMillis-Estimate as shown). You can fetch documents from that collection like any other:

```
db.system.profile.find()
```

For today..

- Create a new database named blogger with a collection named articles. Insert a new article with an author name and email, creation date, and text.
- Update the article with an array of comments, containing a comment with an author and text.
- Summit the two statements to ICON.

Indexing and aggregation MongoDB

Import json file to MongoDB

https://docs.mongodb.com/guides/server/import/

If you didn’t import the city_inspections last class:

- Download the file city_inspections.json from ICON
- Use the mongoimport utility:

```
mongoimport --db moderndb --collection city_inspections
       --drop --file ~\downloads\city_inspections.json
```

Or if you enabled authentication:

```
mongoimport --db moderndb --collection city_inspections
       --authenticationDatabase admin --username <user>
       --password <password>  --drop
       --file ~\downloads\city_inspections.json
```

Execution stats

Let's get the execution statistics for a query by certificate_number in the city_inspections collection:

```
db.city_inspections.find({certificate_number:
10003581}).explain("executionStats").executionStats
```

- You can also set the profiler to record all or long running queries. To disactivate the profiler (default)

```
db.setProfilingLevel(0)
```

- Recall that Mongo automatically creates an index by the _id

```
db.getCollectionNames().forEach(function(collection) {
    print("Indexes for the " + collection + " collection:");
    printjson(db[collection].getIndexes());
});
```

Create an Index

Let’s create an index on certificate_number
db.city_inspections.createIndex ({“certificate_number”: 1}, {unique: false})

Now let’s see the improvement on executing the same query:

db.city_inspections.find({certificate_number: 10003581}).explain(*"executionStats"*).executionStats

Query by id
db.city_inspections.find({id:”108-2015-UNIT”}).explain(*"executionStats"*).executionStats

Now let’s create a hash index over id
db.city_inspections.createIndex ({“id”: “hashed”})

Create an index on zip code (from address document)

db.city_inspections.findOne()

Ascending order
db.city_inspections.createIndex ({“address.zip”: 1})

Descending order
db.city_inspections.createIndex ({“address.zip”: -1})

db.city_inspections.getIndexes()

Let’s drop one:

db.city_inspections.dropIndex (“address.zip_1”)

More queries

db.city_inspections.find ({“certificate_number”: {“$lt”: 100000}}).sort({“address.zip”: -1})

Single-purpose aggregators:

Count - number of documents in the result
db.city_inspections.find ({“certificate_number”: {“$lt”: 100000}}).count()
db.city_inspections.count ({“certificate_number”: {“$lt”: 100000}})

Distinct – collect the result set into an array of unique values
db.city_inspections.distinct(“address.zip”, {“certificate_number”: {“$gt”: 100000}})

Aggregate – returns document according to the logic you provide

Aggregate (a pipeline-style logic)

https://docs.mongodb.com/manual/aggregation/#aggregation-pipeline

Stages (full list https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/):

$match – filters
$group – group by
$sort – order by
$project – select tags/documents to display in the results set
$limit – limit the number of results

* hint option can be used to force the usage of the specified index

Count the number of passed inspections per city

```
db.city_inspections.aggregate([
{$match: { "result": {$eq: "No Violation Issued"}}},
{$group: {
        _id: "$address.city",
        count: {$sum: 1}
        }
}
])
```

Count the number of passed inspections per city order by count

```
db.city_inspections.aggregate([
{$match: { “result”: {$eq: “No Violation Issued”}}},
{$group: {
        _id: “$address.city”,
        count: {$sum: 1}
        }
},
{$sort: {"count": -1}}
])
```

Cities with over 200 passed inspections order by count

```
db.city_inspections.aggregate([
{$match: { "result": {$eq: "No Violation Issued"}}},
{$group: {
        _id: "$address.city",
        count: {$sum: 1}
        }
},
{$sort: {"count": -1}},
{$match: {"count": {$gt: 200}}}
])
```

Server side commands

- Pre-built Mongo commands execute in the server
- Some of them need to be executed under the admin database

- use moderndb
- db.listCommands() –most of the commands execute on the server, not the client

- To have our own functions executed on the server (similar to stored procedures), add it to collection system.js

```
db.system.js.save ({_id: “getLast”, value: function(collection) {
        return collection.find({}).sort({‘_id’: 1}).limit(1)[0];
        }
})
use moderndb
db.loadServerScripts()
getLast(db.inventory).display
```

For today..

- https://docs.mongodb.com/manual/tutorial/aggregation-zip-code-data-set/
- Write a query to return Largest and Smallest Cities for these Midwest States: **Illinois**, **Indiana**, Iowa, and **Kansas**
- Submit your query to ICON

Map Reduce

Background

- Google deals with very large amounts of data (petabytes)
  - need to process data fairly quickly
  - use very large numbers of commodity machines
  - Cheap nodes fail, especially if you have many
    - Mean time between failures for 1 node = 3 years
    - Mean time between failures for 1000 nodes = 1 day
    - Solution: Build fault-tolerance into the system

- Google developed an infrastructure consisting of
  - the Google distributed file system GFS
  - the MapReduce computational model
- MapReduce
  - functional programming model
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates
- Open source implementation Hadoop from Apache

Map Reduce

- Programming model for indexing and searching large data volumes over computer clusters
- Two Phases, Map and Reduce
  - Map
    - Extract sets of Key-Value pairs from underlying data
    - Potentially in Parallel on multiple machines
  - Reduce
    - Merge and sort sets of Key-Value pairs
    - Results may be useful for other searches

Google MapReduce

- Google's MapReduce is implemented as a C++ library.
- Operates on commodity hardware and standard networking.
- Input data, intermediate results, and final results are stored in GFS.
- A master scheduler process distributes map, reduce tasks to workers.
- Fault tolerance:
  - The master pings workers periodically.
  - Workers that do not respond are marked as failed.
  - Jobs assigned to failed workers are rerun.
  - Master failure aborts the computation.

Data type: key-value *records*

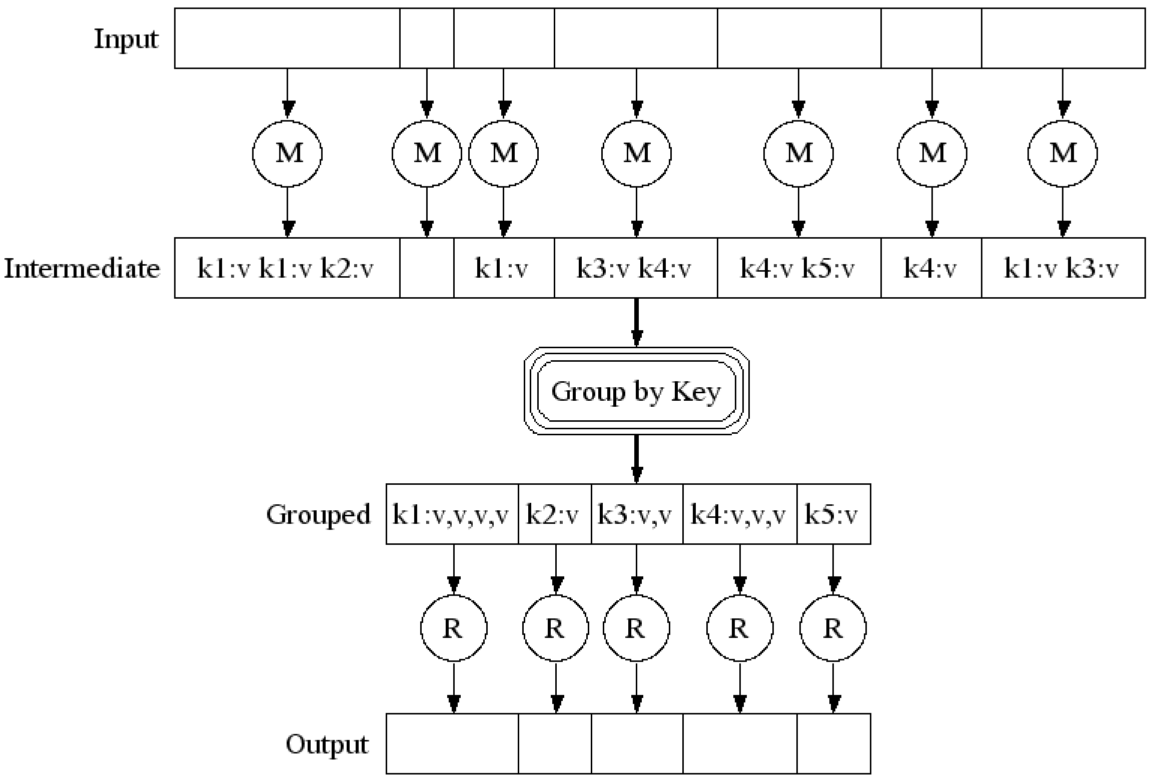
Map function:

$$(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$$

Reduce function:

$$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$$

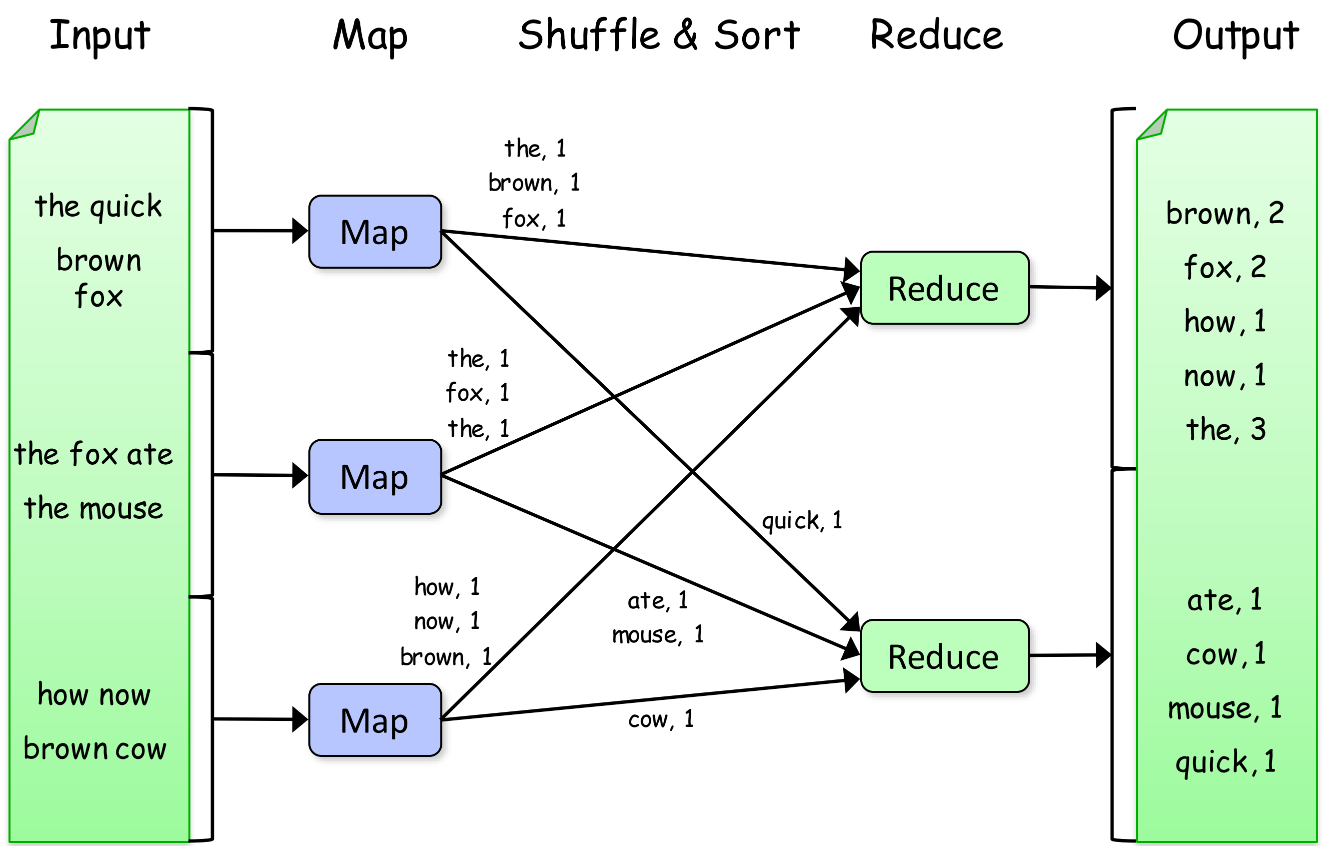
Execution


Input
M
M
M
M
M
M
M
Intermediate
k1:v k1:v k2:v
k1:v
k3:v k4:v
k4:v k5:v
k4:v
k1:v k3:v
Group by Key
Grouped
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
Output

Example: Word Count

```
def mapper(line):
    foreach word in line.split():
        output(word, 1)

def reducer(key, values):
    output(key, sum(values))
```

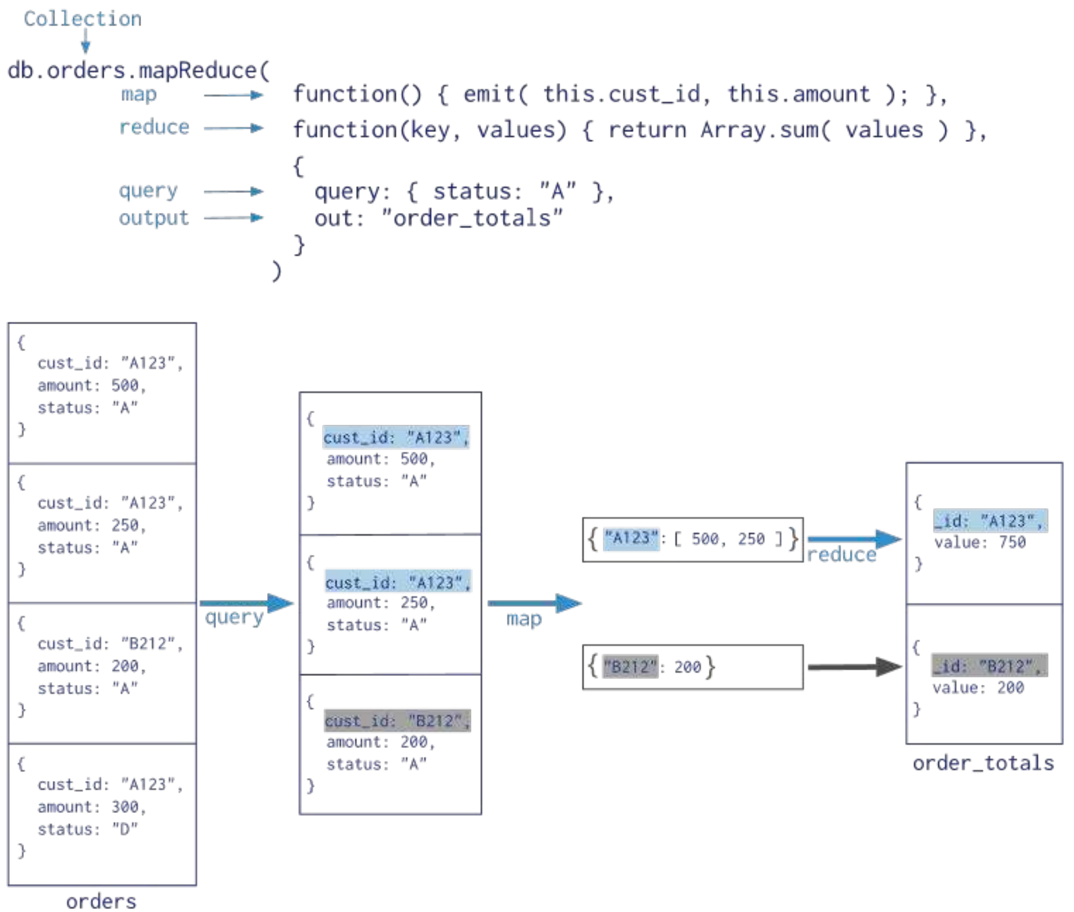
Word Count Execution



MapReduce Execution Details

- Single *master* controls job execution on multiple *slaves*
- Mappers preferentially placed on same node or same rack as their input block
  - Minimizes network usage
- Mappers save outputs to local disk before serving them to reducers
  - Allows recovery if a reducer crashes
  - Allows having more reducers than nodes

Map Reduce in MongoDB



Map/Reduce in Mongo

```
db.collection.mapReduce(
        <mapfunction>,
        <reducefunction>,
        {
                out: <collection>,
                query: <>,
                sort: <>,
                limit: <number>,
                finalize: <function>,
                scope: <>,
                jsMode: <boolean>,
                verbose: <boolean>
        }
    )
```

Example: Tickets

```
{
  "id": 1,
  "day": 20100123,
  "checkout": 100
}
```

```
{
  "id": 2,
  "day": 20100123,
  "checkout": 42
}
```

```
{
  "id": 3,
  "day": 20100123,
  "checkout": 215
}
```

```
{
  "id": 4,
  "day": 20100123,
  "checkout": 73
}
```

Sum(checkout)?

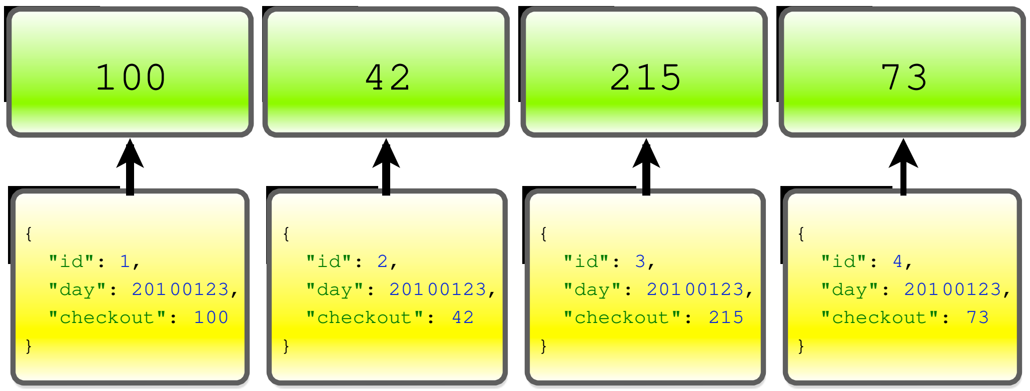

```
{
  "id": 1,
  "day": 20100123,
  "checkout": 100
}
```

```
{
  "id": 2,
  "day": 20100123,
  "checkout": 42
}
```

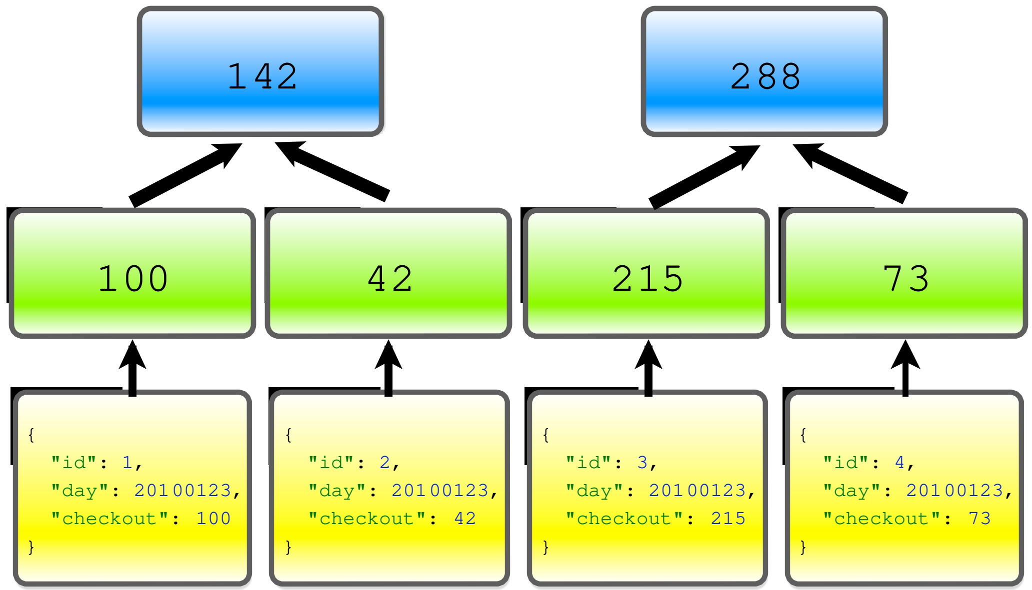
```
{
  "id": 3,
  "day": 20100123,
  "checkout": 215
}
```

```
{
  "id": 4,
  "day": 20100123,
  "checkout": 73
}
```

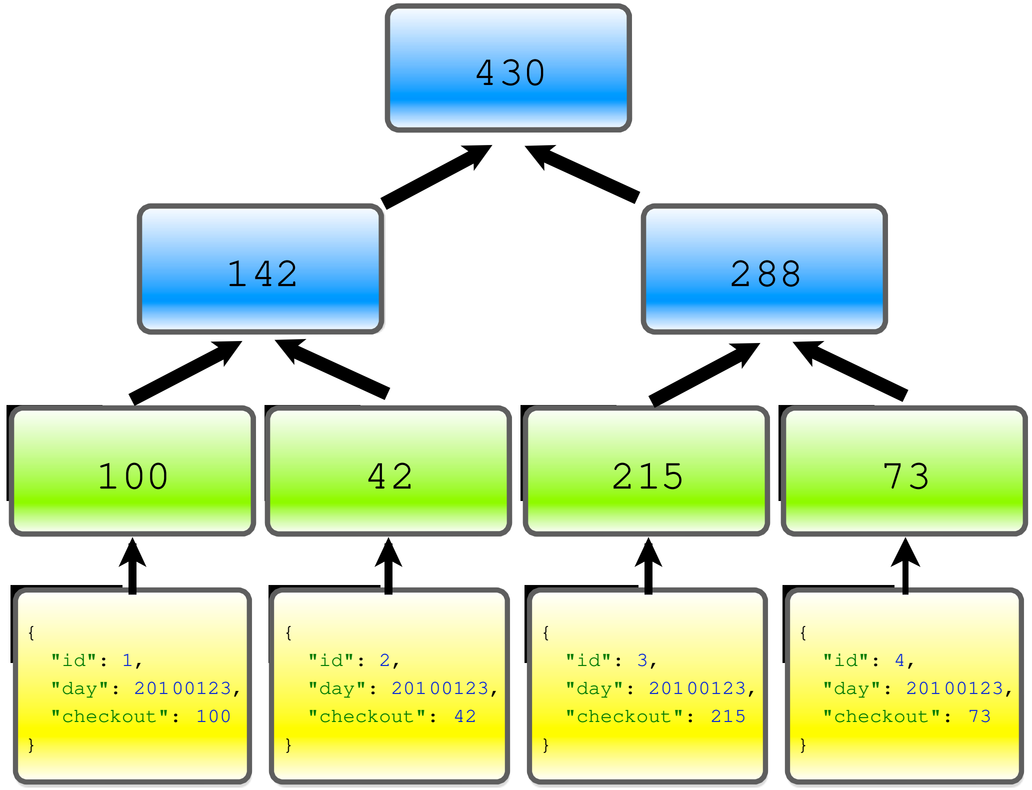
Map: emit(checkout)



Reduce: sum(checkouts)



Reduce: sum(checkouts)



Reduce must be associative

reduce(100 42 215 73) == 430

Must be equal to

reduce(

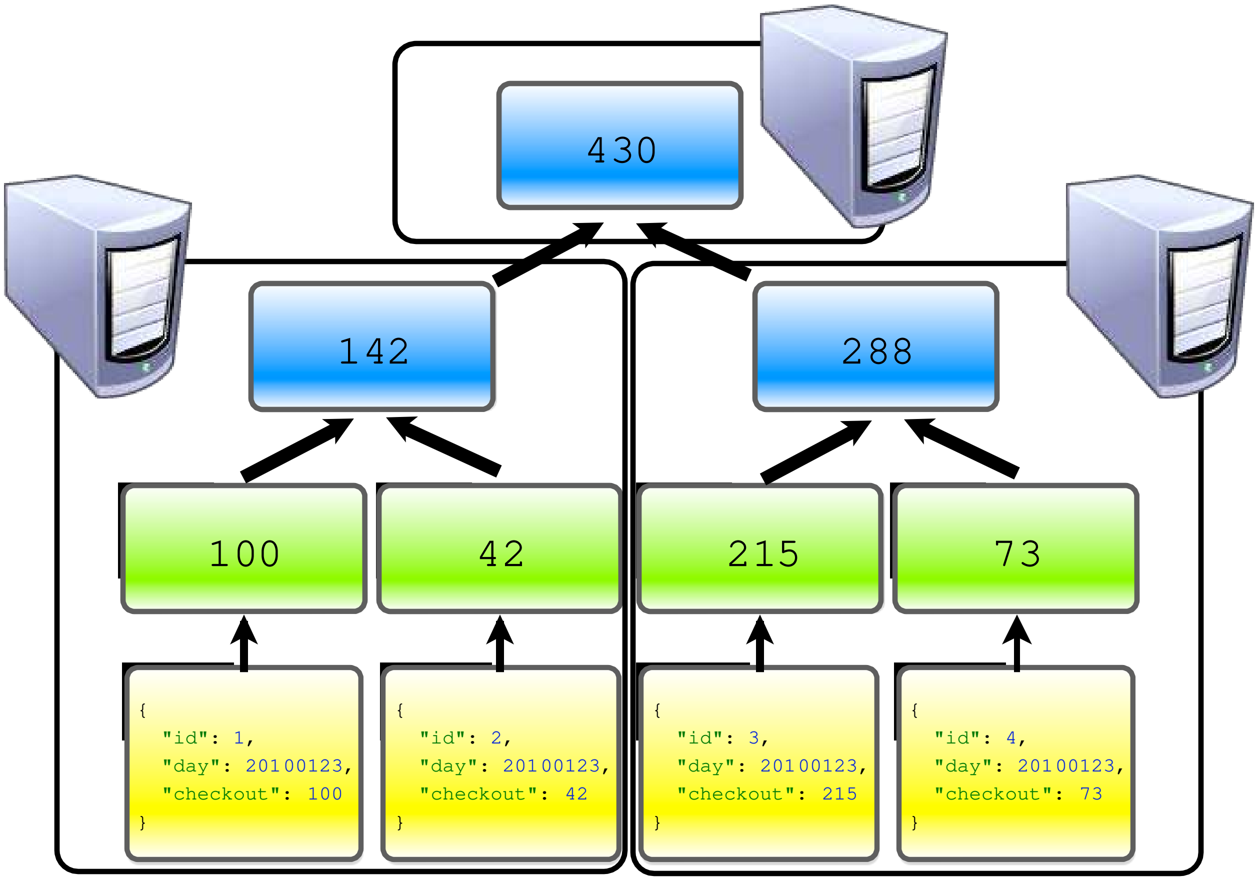
reduce(100 42) == 142

reduce(215 73) == 288

) == 430

SELECT SUM(checkout) FROM ticket

Inherently distributed



Calculate total checkout

```
#Aggregate alternative
db.tickets.aggregate ({
        "$group": {_id: null, "value": {$sum: "$checkout"}}},
        {$out: "sumOfCheckouts_agg"
        })

#Map-reduce alternative
var map = function() {
        emit(null, this.checkout)
}

var reduce = function (key, values) {
   var sum=0
   for (var idx = 0; idx< values.length; idx++)
        sum+=values[idx];
   return sum;
}

db.tickets.mapReduce (map, reduce, {"out":  "sumOfCheckouts"})

db.sumOfCheckouts.findOne().value
```

Persistent Collection

Sum(checkout) Group By day

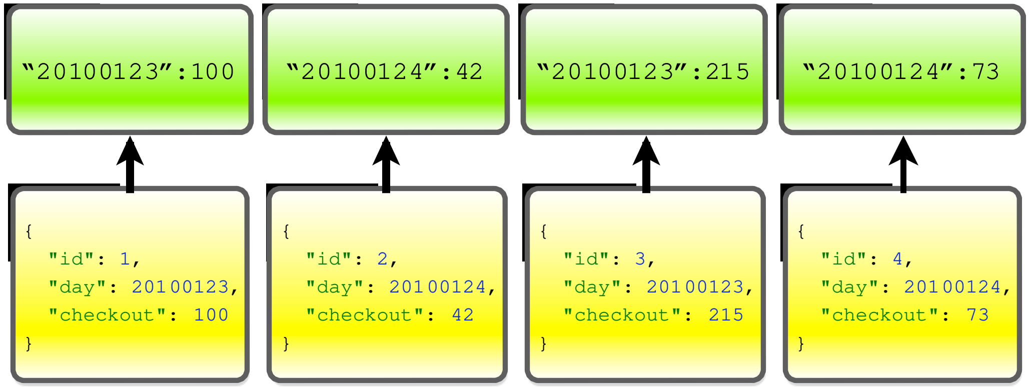

```
{
  "id": 1,
  "day": 20100123,
  "checkout": 100
}
```

```
{
  "id": 2,
  "day": 20100124,
  "checkout": 42
}
```

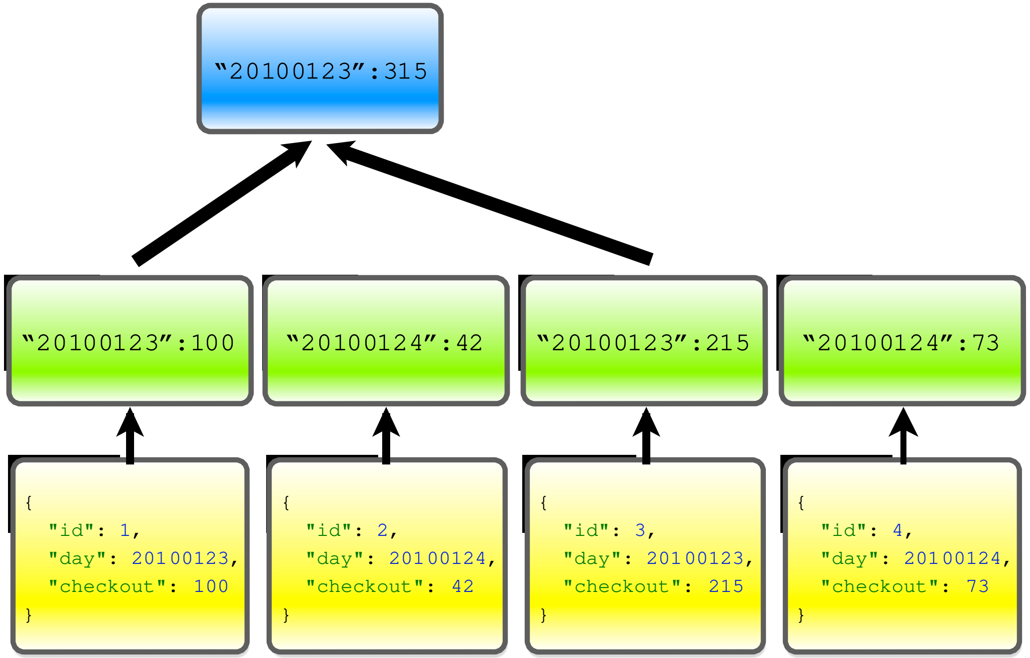
```
{
  "id": 3,
  "day": 20100123,
  "checkout": 215
}
```

```
{
  "id": 4,
  "day": 20100124,
  "checkout": 73
}
```

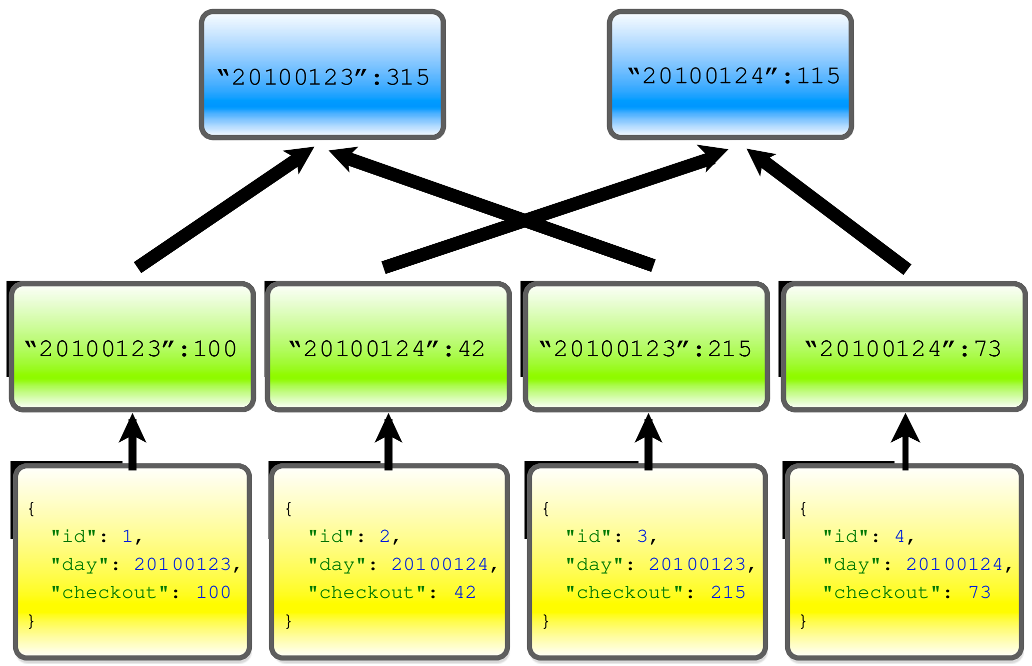
Map: emit(day,checkout)



Reduce: sum(checkouts)



Reduce: sum(checkouts)



Update the tickets to remove checkout

```
> db.tickets.update({ "_id": 1 }, {
... $set: { "products": {
...... "apple": { "qty":      5, "price": 10 },
...... "kiwi": { "qty":      2, "price": 25 }
...... }
... },
... $unset: { "checkout": 1 }
... })

> db.tickets.find()
{ "_id" : 1, "day" : 20190123, "products" : {
  "apple" : { "qty"      : 5, "price" : 10 },
  "kiwi" : { "qty"      : 2, "price" : 25 }
}}
{ "_id" : 2, "day" : 20190123, "checkout" : 42 }
{ "_id" : 3, "day" : 20190123, "checkout" : 215 }
{ "_id" : 4, "day" : 20190123, "checkout" : 73 }
```

Sum(Checkout) by day Calculate Checkout

```
> var map = function() {
... var checkout = 0
... for (var name in this.products) {
...... var product = this.products[name]
...... checkout += product.qty  * product.price
...... }
... emit(this.day, checkout)
}

> var reduce = function(key, values) {
... var sum = 0
... for (var index in values) sum += values[index]
... return sum
}
```

Sum(Checkout) by day Calculate Checkout

```
> db.tickets.mapReduce(map, reduce, { "out": "sumOfCheckouts" })

> db.sumOfCheckouts.find()
{ "_id" : 20190123, "value" : 315 }
{ "_id" : 20190124, "value" : 110 }
```

For today

Follow the map-reduce examples linked below and create the orders collection:
https://docs.mongodb.com/manual/tutorial/map-reduce-examples/

Write a map-reduce aggregation that returns the number of items bought (the number of elements in the items array) per costumer and submit it to ICON.

Replication and Sharding

Replication and Sharding

- So far we have run Mongo as a single server
- But Mongo was designed to run on a cluster of computers
  - Higher availability
  - Enable data replication across servers
  - Shard collections into many pieces
  - Perform queries in parallel

- Replication = duplication
  - Keeps identical copies running
  - Increase fault tolerance against the loss of a single database server
  - Makes sharding more robust

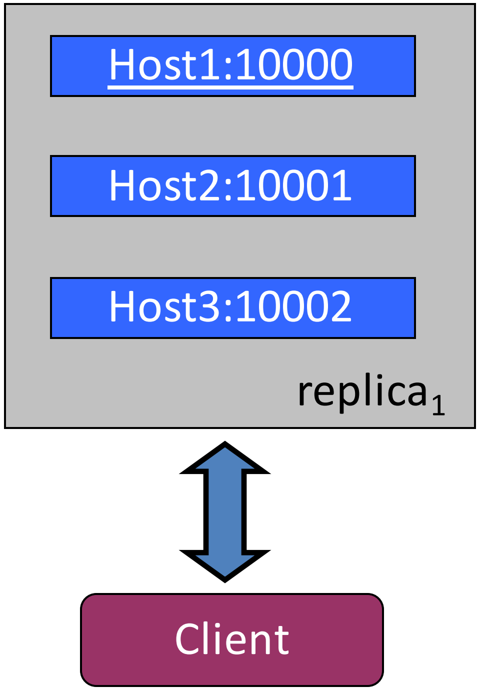
- Sharding distributes data across multiple machines

Replica Sets

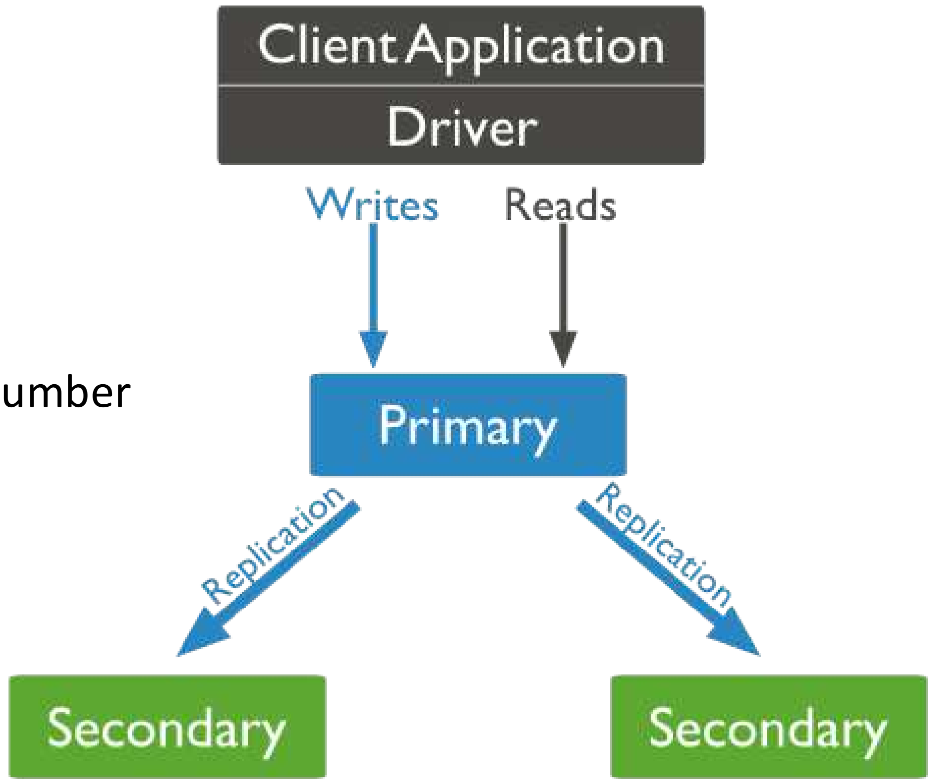
A replica set is a group of mongod instances that maintain the same data set.

Redundancy and Failover
Zero downtime for upgrades and maintaince

Master-slave replication
- Strong Consistency
- Delayed Consistency



Typical replication setup



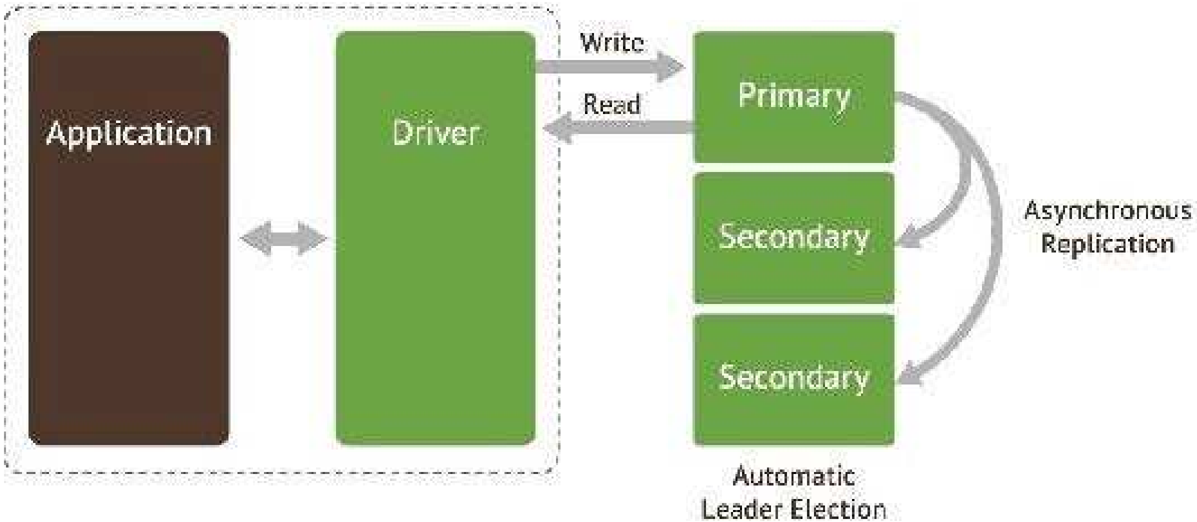
Clients normally talk to the Primary.

Usually an odd number works best!

What's the "majority" here?

Common sense says to put some of these at a different data center!

The replication is asynchronous…



"Eventually up-to-date"

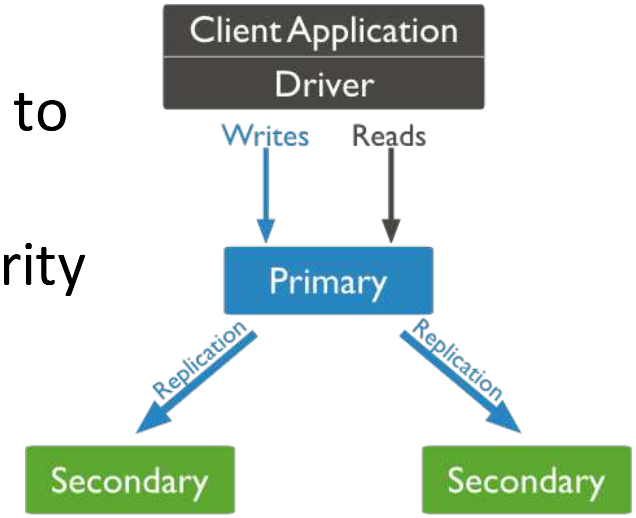
If a secondary goes down…

When it restarts, it will start synching from where it left off in its own oplog. May replay operations already applied – ok.

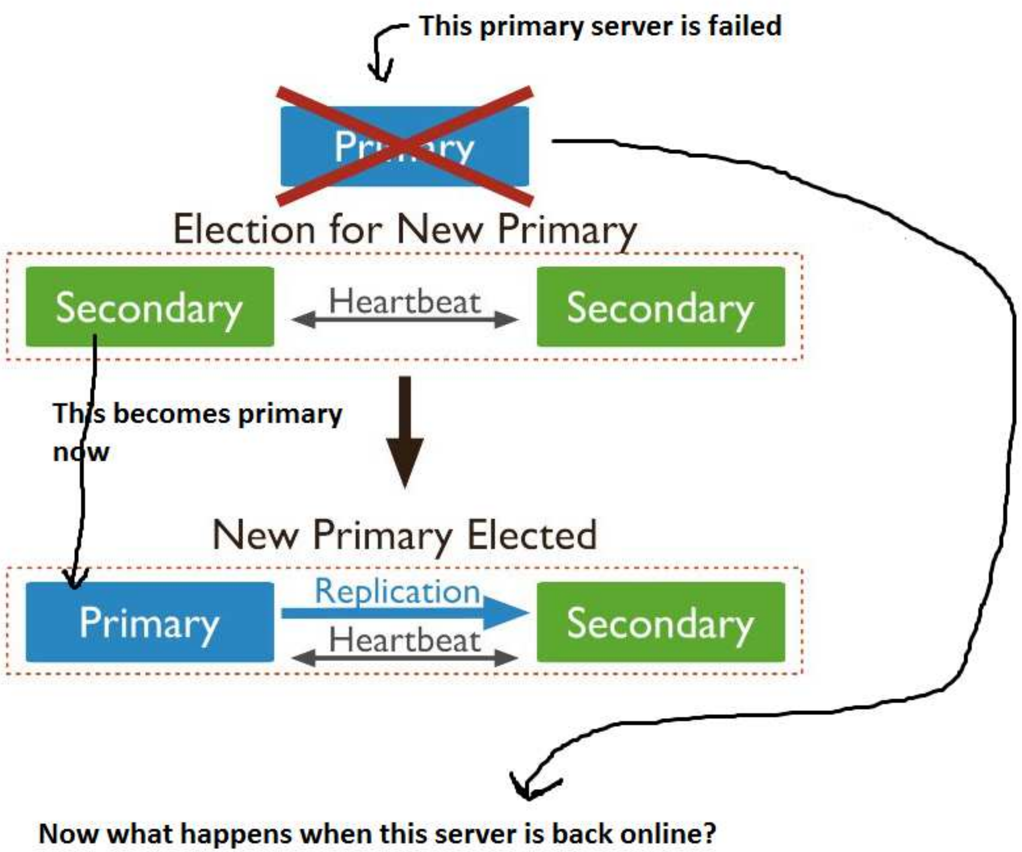
- If it's been down to long for this to work, it is "stale" and will attempt to make a full copy of the data from another member – initial synching.
  - Can also restore from backup, manually.
  - Recoveries can slow down operations.
    - E,g, "working set" in memory gets kicked out.
    - Need to rebuild indexes.

Replica set

- Heartbeats
  - Every two seconds, from every member to every other member.
  - Lets primary know if it can reach a majority of the set.
    - If not, it demotes itself!
  - Members communicate their state.
- Elections
  - If a member can't reach a primary, and is eligible to become a primary, it asks to become primary.
  - Other members go through logic to decide if this is suitable.



Rollbacks



- Typically needed because of network partitions.
- Primary gets disconnected after a write, before replication.
- Can result in conflicting oplogs. Call the administrator!

Let’s give it a try

- Today will start a few new servers. Mongo’s default port is 27017, so we’ll use other ports.
- Create three data directories
  - mkdir ./mongo1 ./mongo2 ./mongo3
- Next fire up the Mongo servers using the replSet flag
  - mongod --replSet moderndb --dbpath ./mongo1 --port 27011
  - mongod --replSet moderndb --dbpath ./mongo2 --port 27012
  - mongod --replSet moderndb --dbpath ./mongo2 --port 27013
- Now let’s connect to the first server and initialize our replica set
  - mongo localhost:27011

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

> rs.status().ok
```

Object rs (replica set) instead of db (database)

Insert example

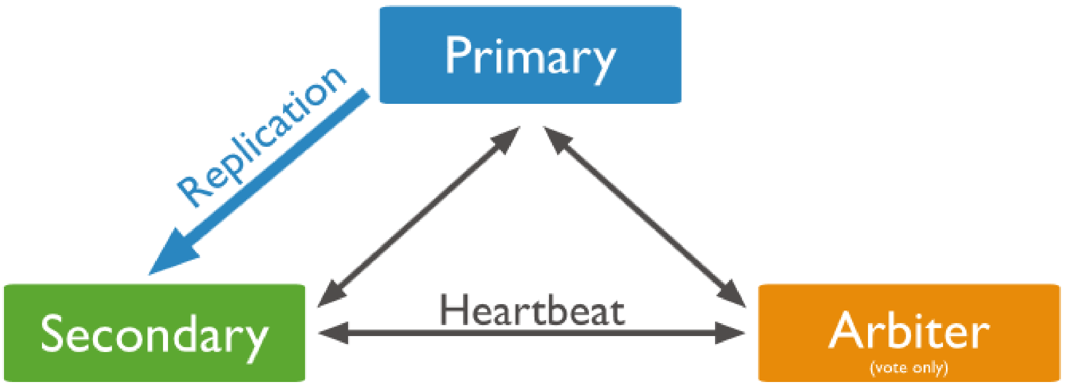
- Insert data in the client connected to the master server (state PRIMARY)
  - ➢ db.echo.insert({ say : *'HELLO!'* })
- Now exit the console and let's stop the PRIMARY server (Ctrl+C)
- Check on the console for the other two servers, one must have been promoted to master
- Connect to that server and search for the inserted value
  - ➢ db.echo.find()

- Now open a shell to the remaining secondary server and run the isMaster() function
  - ➢ db.isMaster().ismaster
  - ➢ db.isMaster().primary
- Let's try to insert another value
  - ➢ db.echo.insert({ say : *'can I insert here?'* })

- Now let's kill the current master
- And insert again, then restart the two servers

The problem with even number of nodes

- If network partition occurs, the majority of nodes that can still communicate would constitute the network
- The network without majority of nodes becomes no functional (the primary demotes itself to secondary)
- An arbiter is a voting but nonreplicating server in the replica set.



Sharding

id	company	customer	article	currency	price
4250250	020	073000	5994537812	00	142,50
4250251	020	073000	5994537852	00	141,12
4250252	020	073000	5994537854	00	105,99
4250253	020	073000	5994537856	00	109,52
4250254	020	073000	5994537862	00	131,49
4250255	020	073000	5994567308	00	29,86
4250256	020	073000	5994567422	00	57,13
4250257	020	073000	5994567428	00	68,59
4250258	020	073000	5994605089	00	51,09
4250259	020	073000	5994607975	00	93,93
4250260	020	073000	5994701005	00	74,22

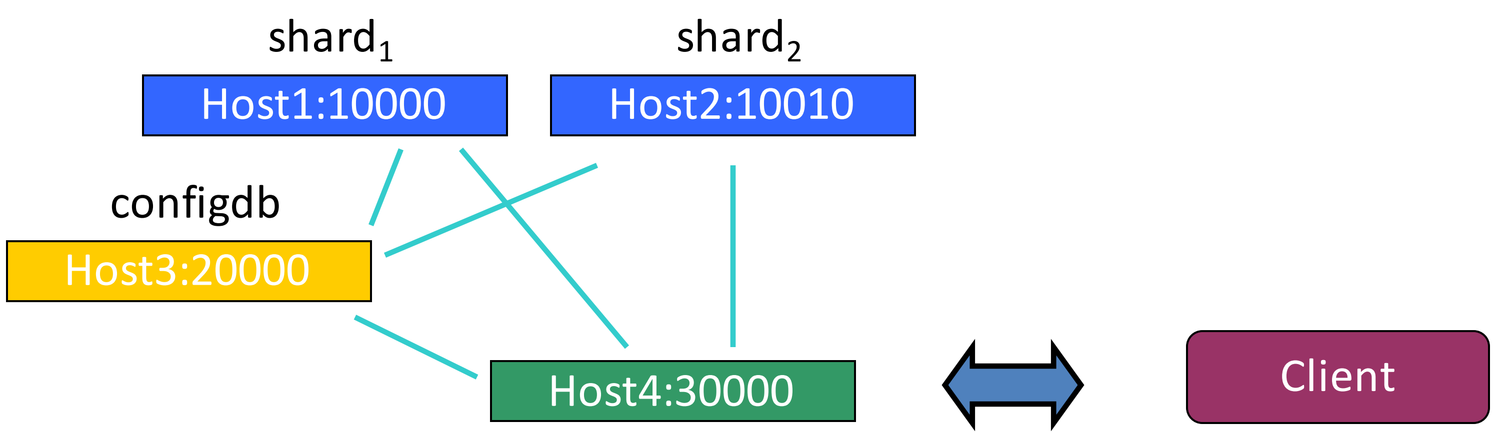
Partition your data

Scale write throughput

Increase capacity

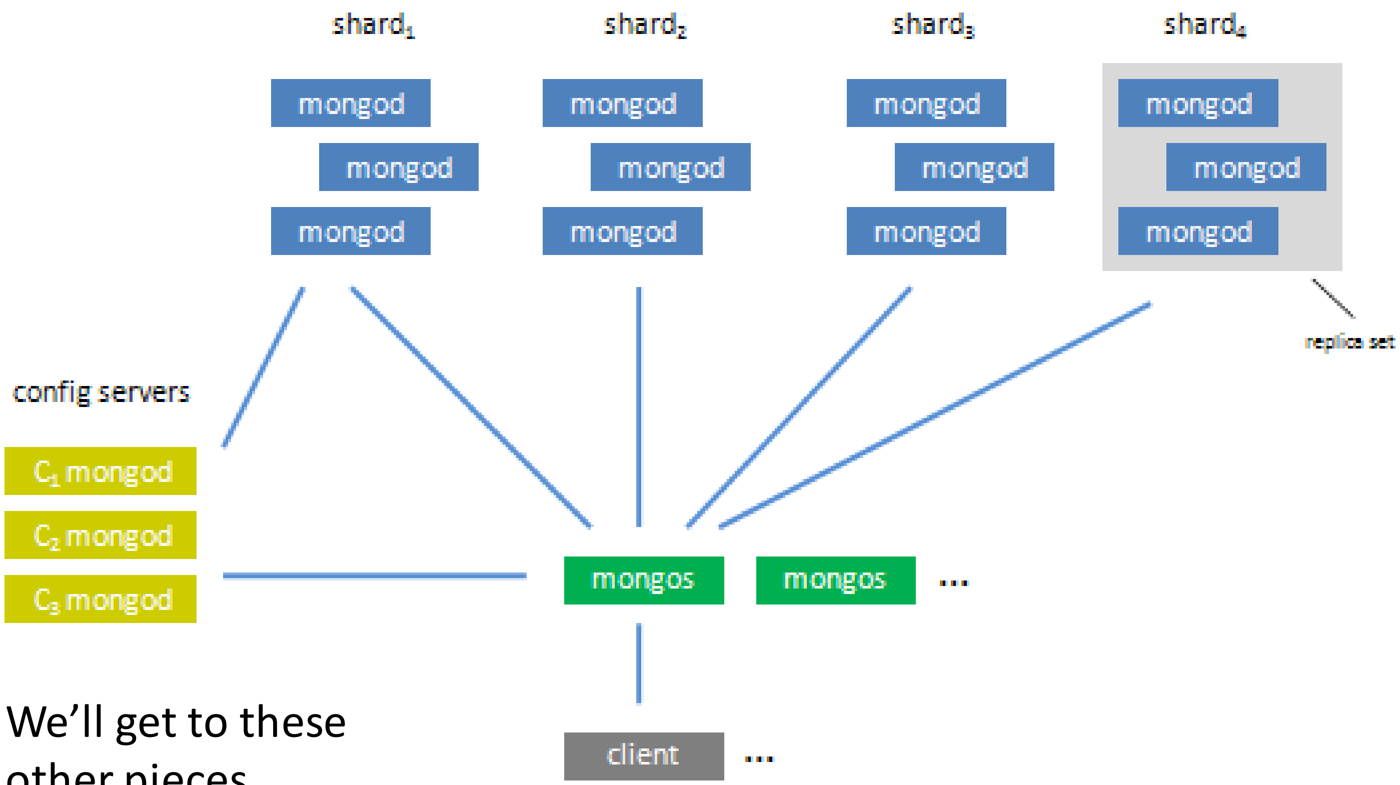
Auto-balancing

MongoDB's built-in system abstracts the architecture and simplifies the admin.



Typically used *along with* replication

Don't confuse with replication!

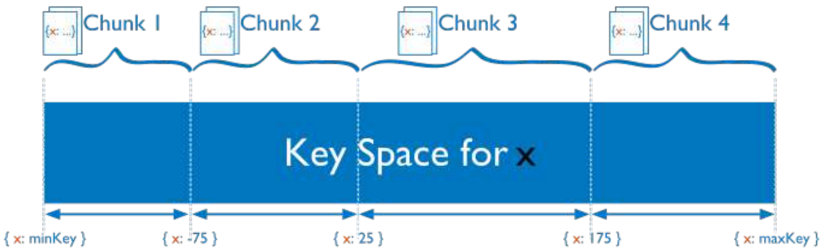


We'll get to these other pieces.

Shard key

A field or two used to breakup the data.

- Like “username”.
- Must be indexed.
- MongoDB divides the data into “chunks” based on this key.
  - Each chunk is for a range of the keys.



The chunks are then evenly distributed across “shards.” (The separate servers being used.)

Client-side queries work normally. Routing done by “mongos”.

Can use “explain” to see what really happens.

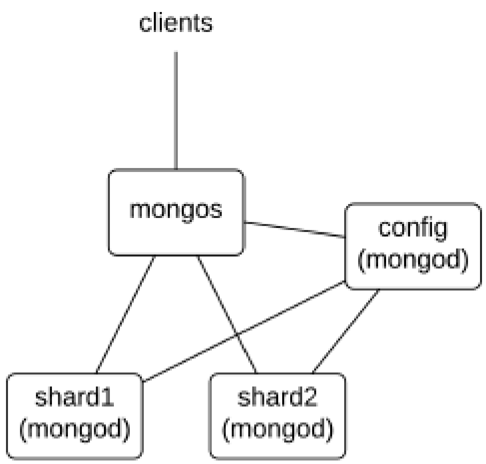
You can still do big operations on sharded datasets. e.g., mongos does sorting with a merge-sort across the shards.

Configuring Sharding

- When to shard
  - 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.
- Monitor to decide when sharding becomes necessary.
- Starting the servers
  - Need to set up the mongos and the shards.
  - Need to set up “config servers.” Usually 3!
    - These are “the brains of your cluster.” Used by mongos. “Table of contents.”
    - Set up first. Started first.
    - Each on a separate machine, geographically distributed.

Let’s give it a try

- Let’s launch a couple of (nonreplicating) mongod server. Parameter --shardsvr just means the server is capable of sharding
  $ mkdir ./mongo4 ./mongo5
  $ mongod --shardsvr --dbpath ./mongo4 --port 27014
  $ mongod --shardsvr --dbpath ./mongo5 --port 27015
- Now we need a config server to keep track of the keys
  $ mkdir ./mongoconfig
  $ mongod --configsvr --replSet configSet --dbpath ./mongoconfig --port 27016
- Let’s configure the configSet replica set with only this server
  rs.initiate({ _id: *'configSet’*, configsvr: **true**, members: [{ _id: 0, host: *'localhost:27016’* }] })
- Now the mongos server will be entry point for our clients
  $ mongos --configdb configSet/localhost:27016 --port 27020
- Now let’s connect to the mongos server admin database
  $ mongo localhost:27020/admin
- And configure some sharding
  - sh.addShard(*'localhost:27014’*)
  - sh.addShard(*'localhost:27015'*)



Sharding example, cont.

- Now we have to give it the database and collection to shard and the field to shard by
  - ➢ db.runCommand({ enablesharding : *"test"* })
  - ➢ db.runCommand({ shardcollection : *"test.zips"*, key : {city : 1} })
- Now let's import some data

  $ mongoimport \
  --host localhost:27020 \
  --db test \
  --collection zips \
  --type json \
  ~\download\zips.json

- Now you can query the zips collection

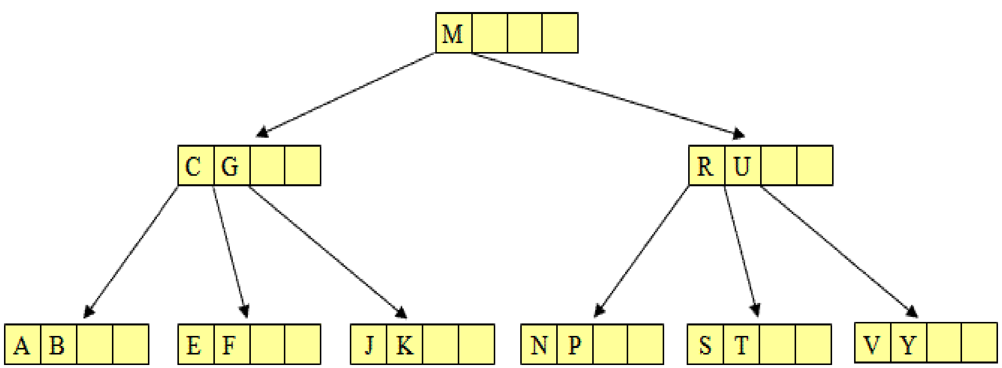
  mongo localhost:27020/test
  - ➢ db.zips.find({state: "IA"}).explain()

- Submit a screenshot of the result of this explain to ICON

Spatial and text indexes

Indexes

- MongoDB uses B-Tree indexes
- Can build the index on any field of the document
- Skips documents that do not have the indexed field (Sparse index)



Examples

```
{"_id": ObjectId(...),
 "name": "John Doe",
 "address": {
         "street": "Main",
         "zipcode": "53511",
         "state": "WI"
         }
}
```

Field Level

db.people.createIndex(“name”:1)

Sub-Field Level

db.people.createIndex(“address.zipcode”:1)

db.people.createIndex(“address”:1)

Embedded document Level
(equality search only)

Examples

```
{"_id": ObjectId(...),
 "name": "John Doe",
 "address": {
         "street": "Main",
         "zipcode": "53511",
         "state": "WI"
         }
}
```

db.people.createIndex({“name”: 1, “_id”: -1})

Compound-Field Index

db.people.find(“_id”: 1000})

Index cannot answer this query (must have a predicate on “name”)

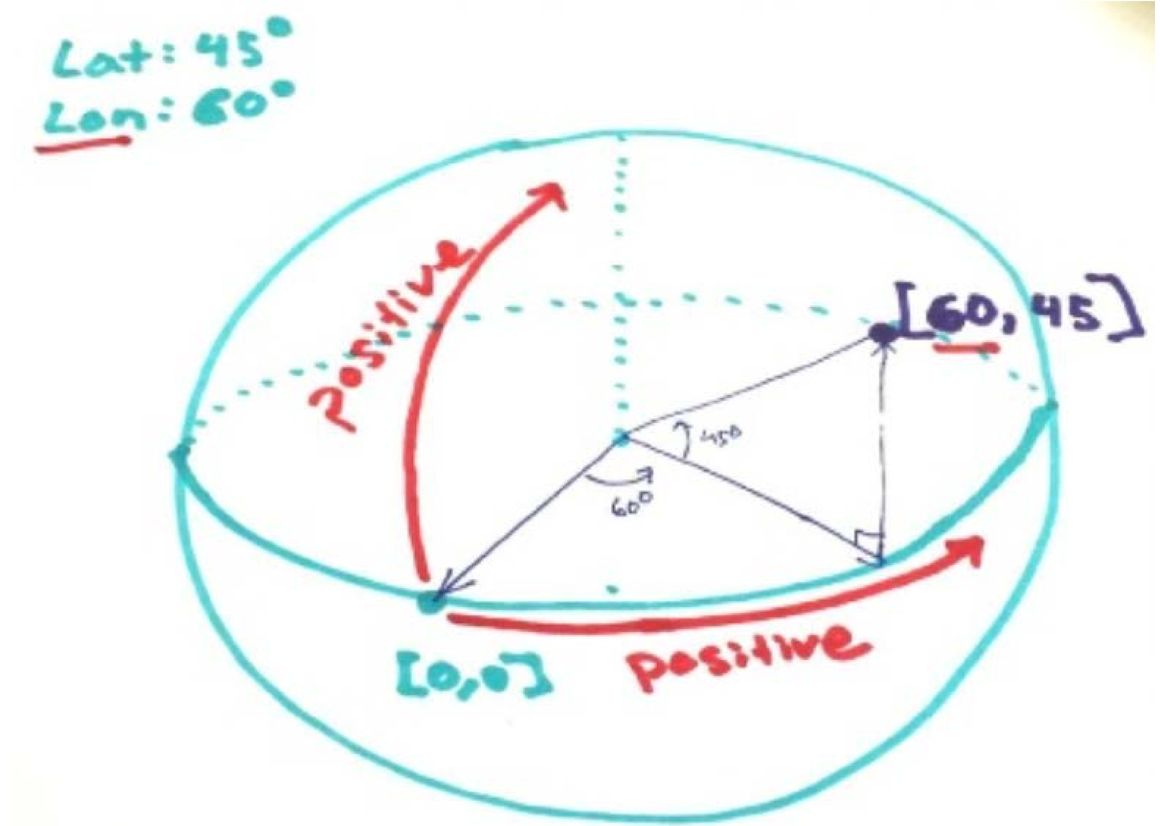
Index Creation Options

```
{"_id": ObjectId(...),
 "name": "John Doe",
 "address": {
        "street": "Main",
        "zipcode": "53511",
        "state": "WI"
        }
}
```

db.people.createIndex({“name”: 1, “_id”: -1},
{“background: True”, “Sparse”: True,
“unique”: True})

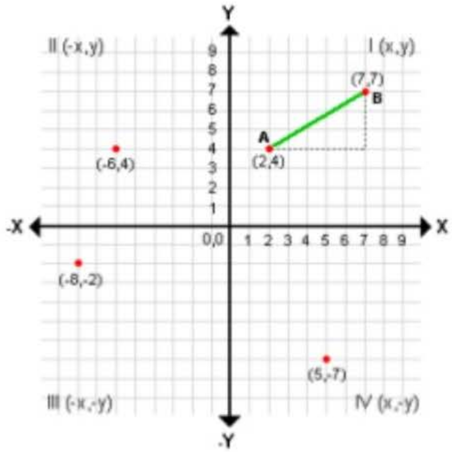
Geospatial indexes

[Longitude, Latitude]



Surface type

Flat



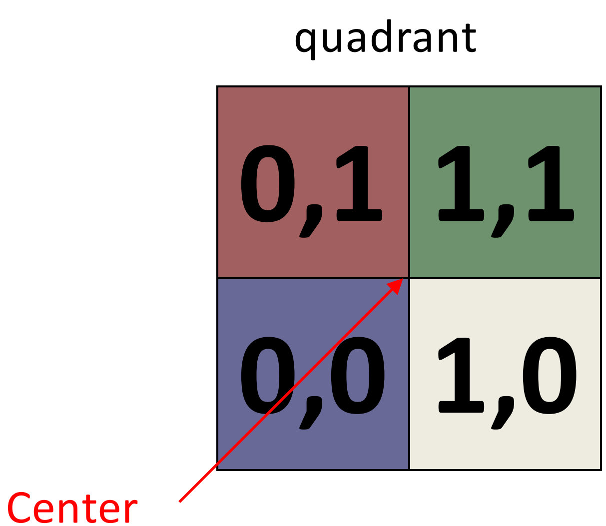
2d Indexes

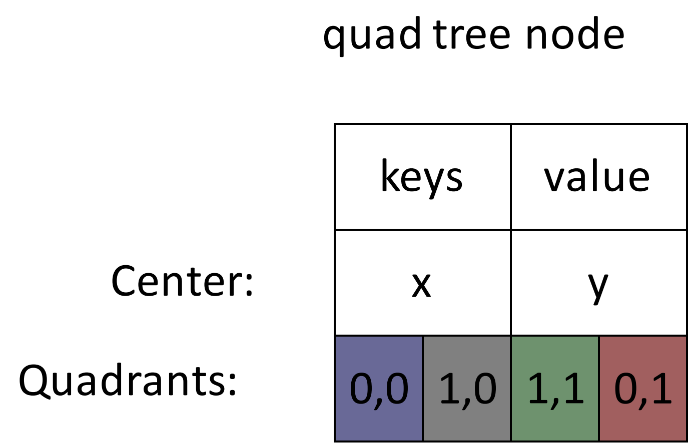
Spherical

2dsphere Indexes

Quad Trees

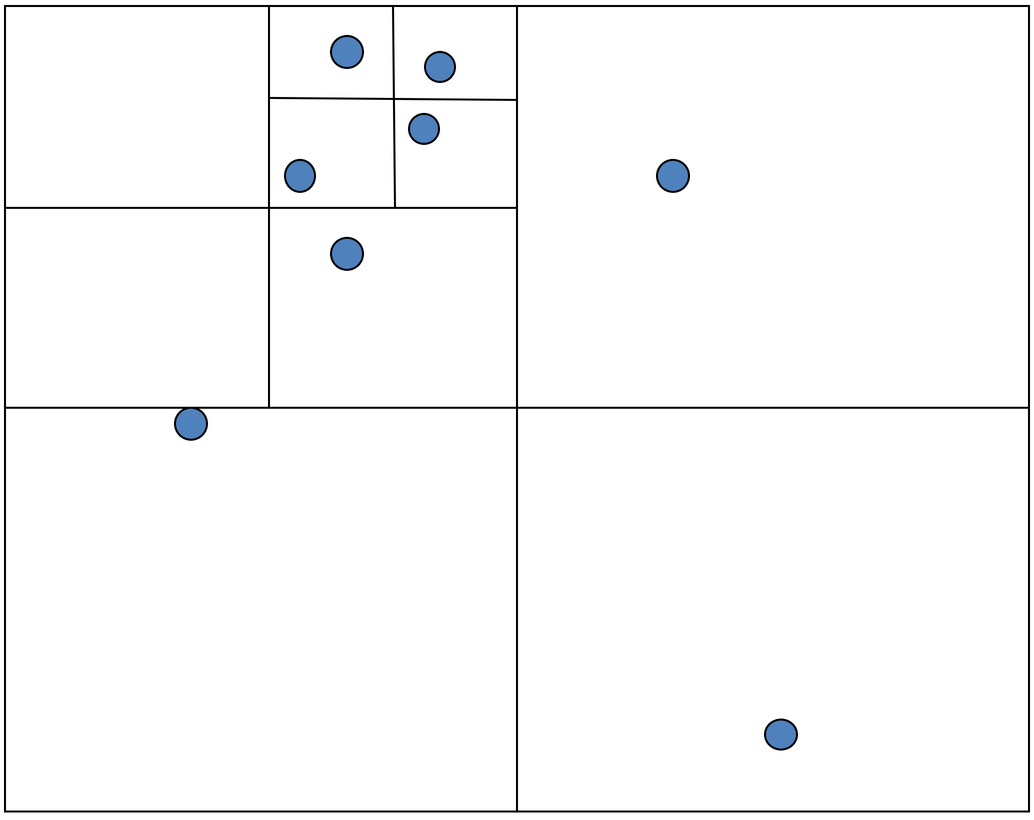
- Split on *all* (two) dimensions at each level
- Split key space into equal size partitions (quadrants)
- Add a new node by adding to a leaf, and, if the leaf is already occupied, split until only one node per leaf





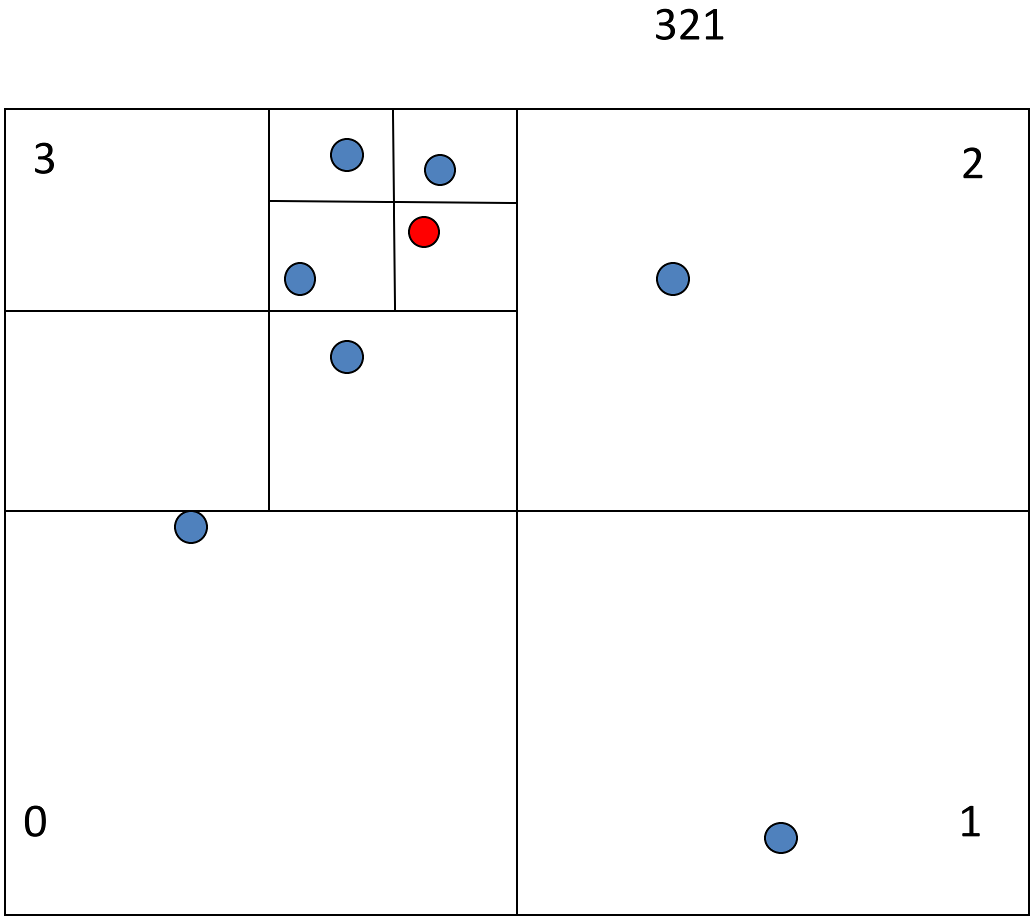
Quadtree

Simplest spatial structure on Earth !



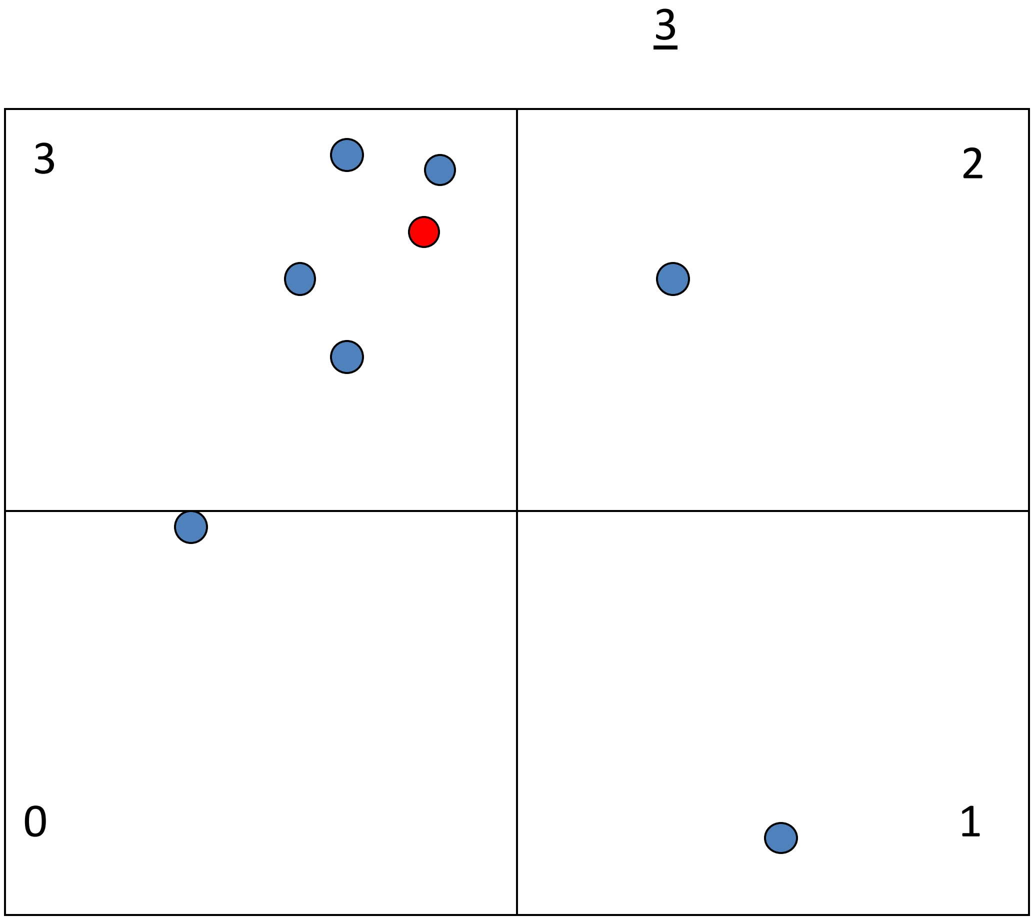
Quadtree

Simplest spatial structure on Earth !



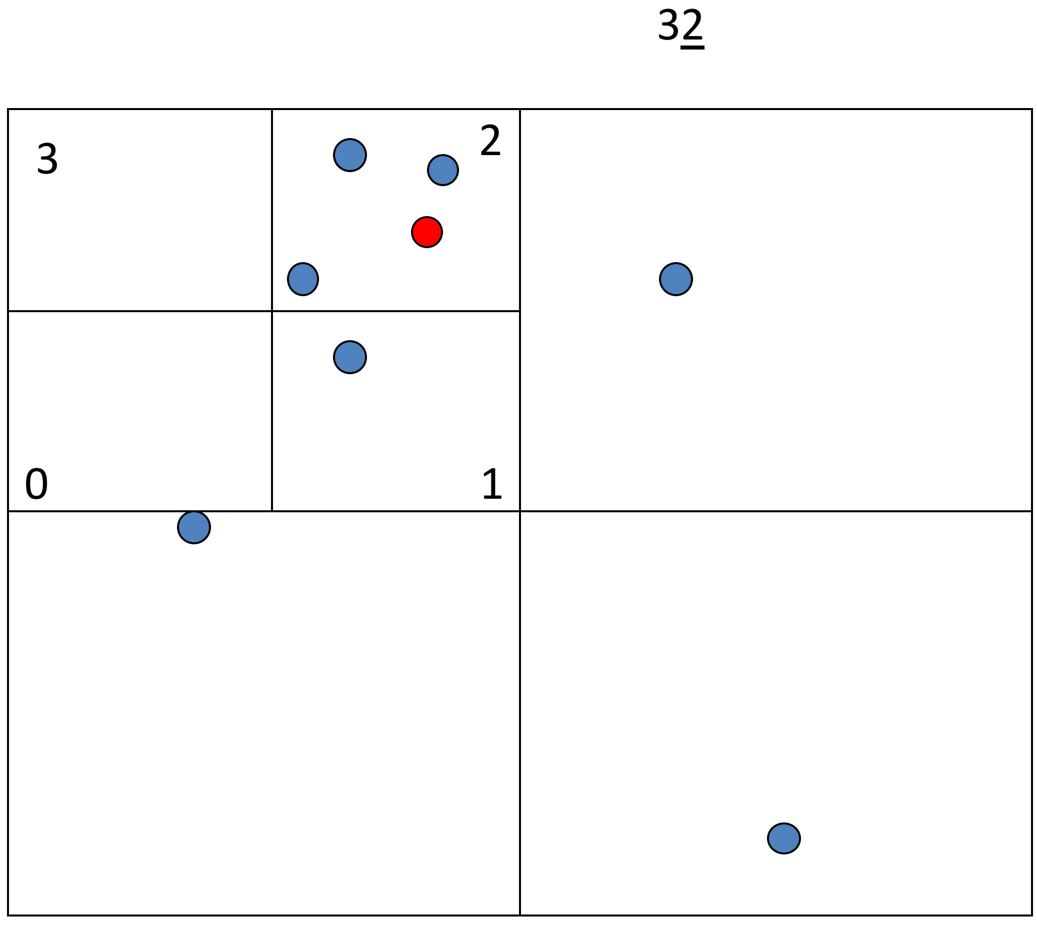
Quadtree

Simplest spatial structure on Earth !



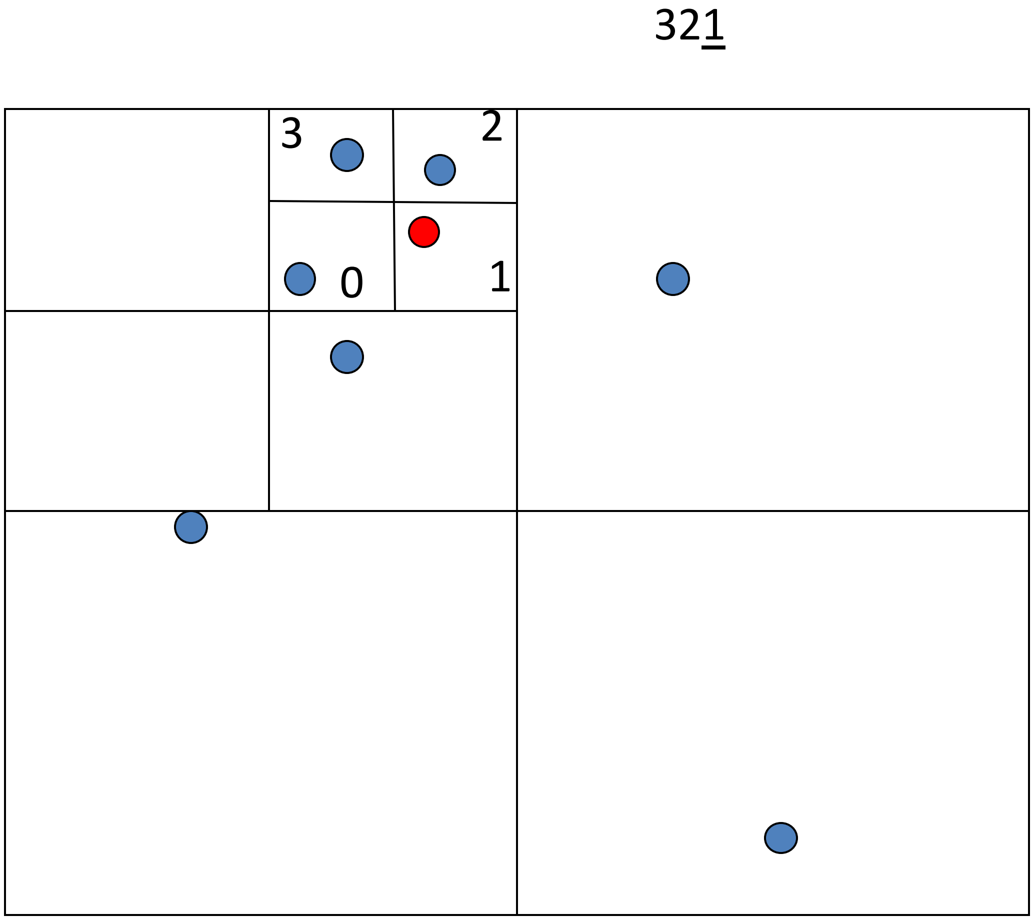
Quadtree

Simplest spatial structure on Earth !

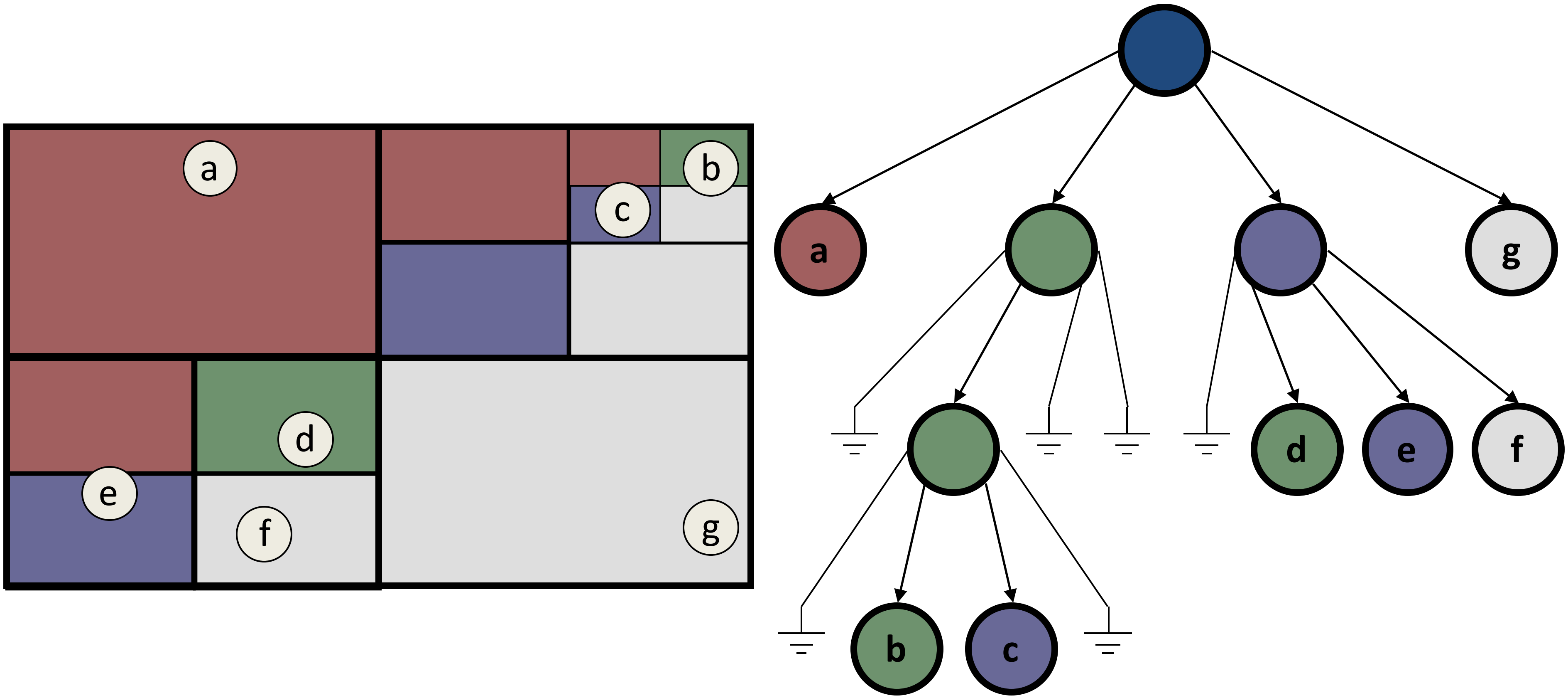


Quadtree

Simplest spatial structure on Earth !

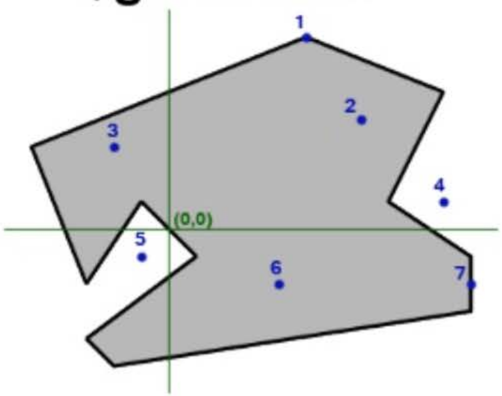


Quadtree Example



Geospatial operators

$geoWithin

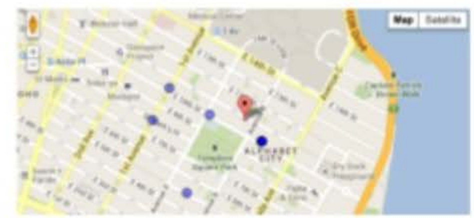


No index, 2d, 2dsphere

$geoIntersects

Index required - 2dsphere

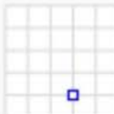
$near/$nearSphere

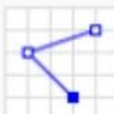


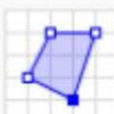
2d, 2dsphere

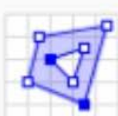
GeoJSON

Type	Examples	

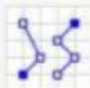
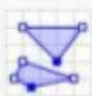
Point		`{ "type": "Point",`
		`"coordinates": [30, 10]`
		`}`

LineString		`{ "type": "LineString",`
		`"coordinates": [`
		`[30, 10], [10, 30], [40, 40]`
		`]`
		`}`

Polygon		`{ "type": "Polygon",`
		`"coordinates": [`
		`[[30, 10], [40, 40], [20, 40], [10, 20], [30, 10]]`
		`]`
		`}`

		`{ "type": "Polygon",`
		`"coordinates": [`
		`[[35, 10], [45, 45], [15, 40], [10, 20], [35, 10]],`
		`[[20, 30], [35, 35], [30, 20], [20, 30]]`
		`]`
		`}`

GeoJSON

Type	Examples	
MultiPoint		{ "type": "MultiPoint", "coordinates": [[10, 40], [40, 30], [20, 20], [30, 10]] }
MultiLineString		{ "type": "MultiLineString", "coordinates": [[[10, 10], [20, 20], [10, 40]], [[40, 40], [30, 30], [40, 20], [30, 10]]] }
MultiPolygon		{ "type": "MultiPolygon", "coordinates": [[[[30, 20], [45, 40], [10, 40], [30, 20]]], [[[15, 5], [40, 10], [10, 20], [5, 10], [15, 5]]]] }
		{ "type": "MultiPolygon", "coordinates": [[[[40, 40], [20, 45], [45, 30], [40, 40]]], [[[20, 35], [10, 30], [10, 10], [30, 5], [45, 20], [20, 35]], [[30, 20], [20, 15], [20, 25], [30, 20]]]] }

2d Query

```
db.<collection>.find( { <location field> :
                  { $geoWithin :
                    { $box|$polygon|$center : <coordinates>
                }}})

db.<collection>.find( { <location field> :
                  { $near : [ <x> , <y> ]
                }})


db.<collection>.find( { loc: [ <x> , <y> ] } )
```

Geospatial indexing zips collection

- db.zips.createIndex({loc: “2d”})
- db.zips.find ({loc: { $geoWithin: {$box: [[-73,42.5], [-72, 43]] } } })
- db.zips.find ({loc: { $geoWithin: {$center: [[-73,42.5], 10] } } })
- db.zips.find ({loc: { $near: [-73,42.5] } })

Text Indexes

- Over fields that are strings or array of strings
- Index is used when using ***$text*** search operator
- Only one index on the collection
  - But it can include multiple fields

One field

```
db.collection.createIndex({content:"text”});
```

Two fields

```
db.collection.createIndex({subject:"text”,content:"text”});
```

All text fields

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

$Text

Text search in mongoDB (Exact match)
Uses a text index and searches the indexed fields

```
{ $text: { $search: <string>, $language: <string> } }
```

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

Search for “coffee” in the indexed field(s)

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

Apply “OR” semantics

$Text

Text search in mongoDB
Uses a text index and searches the indexed fields

```
{ $text: { $search: <string>, $language: <string> } }
```

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

Treated as one sentence

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

“bake” or “coffee” but not “cake”

$Text Score

$Text returns a score for each matching document
Score can be used in your query

```
db.articles.find(
   { $text: { $search: "cake" } },
   { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } ).limit(3)
```

For regular expression match use $regex operator

City_inspections examples

- db.city_inspections.createIndex({”$**": "text”})

- db.city_inspections.find({ $text: { $search: "food deli" } })

- db.city_inspections.find({ $text: { $search: "\"food deli\"" } })

- db.city_inspections.find({ $text: { $search: "grocery -cigarette" } })

- db.city_inspections.find(
            { $text: { $search: “passed" } },
            { score: { $meta: "textScore" } }
).sort({ score: { $meta: "textScore" } }).limit(3)

Collection Modeling

Collection Modeling

Modeling multiple collections that reference each other

In Relational DBs ➔ FK-PK Relationships

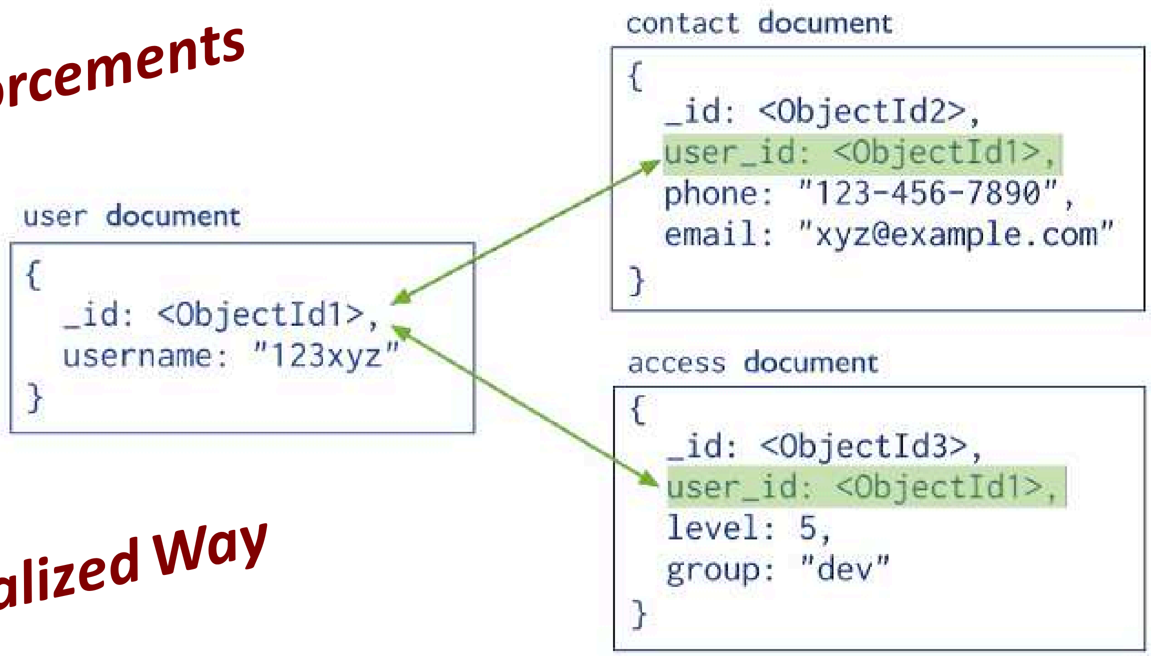
In MongoDB, two options:

- ***Referencing*** between two collections
  - Use Id of one and put in the other
  - Very similar to FK-PK in Relational DBs
  - **Does not come with enforcement mechanism**

- ***Embedding*** between two collections
  - Put the document from one collection inside the other one

Referencing

No Enforcements



Normalized Way

- Have three collections in the DB: “User”, “Contact”, “Access”
- Link them by _id (or any other field(s))

Embedding

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document

Embedded sub-document

De-Normalized Way

Have one collection in DB: “User”
The others are embedded inside each user’s document

Examples (1)

“Patron” & “Addresses”

```
{
   _id: "joe",
   name: "Joe Bookreader"
}
```

```
{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA".
   zip: "12345"
}
```

Referencing

- If it is 1-1 relationship
- If usually read the address with the name
- If address document usually does not expand

If most of these hold
➔ better use Embedding

Examples (2)

"Patron" & "Addresses"

```
{
   _id: "joe",
   name: "Joe Bookreader",
   address: {
              street: "123 Fake Street",
              city: "Faketon",
              state: "MA",
              zip: "12345"
            }
}
```

Embedding

- When you read, you get the entire document at once
- In Referencing → Need to issue multiple queries

Examples (3)

What if a “Patron” can have many “Addresses”

```
{
    _id: "joe",
    name: "Joe Bookreader"
}
```

```
{
    patron_id: "joe",
    street: "123 Fake Street",
    city: "Faketon",
    state: "MA",
    zip: "12345"
}
```

Referencing

- Do you read them together ➔ Go for Embedding
- Are addresses dynamic (e.g., add new ones frequently)

➔ Go for Referencing

Examples (4)

What if a “Patron” can have many “Addresses”

```
{
   _id: "joe",
   name: "Joe Bookreader",
   addresses: [
                {
                  street: "123 Fake Street",
                  city: "Faketon",
                  state: "MA",
                  zip: "12345"
                },
                {
                  street: "1 Some Other Street",
                  city: "Boston",
                  state: "MA",
                  zip: "12345"
                }
              ]
 }
```

Embedding

Use array of addresses

Examples (5)

If addresses are added frequently …

```
{
    _id: "joe",
    name: "Joe Bookreader",
    addresses: [
                 {
                   street: "123 Fake Street",
                   city: "Faketon",
                   state: "MA",
                   zip: "12345"
                 },
                 {
                   street: "1 Some Other Street",
                   city: "Boston",
                   state: "MA",
                   zip: "12345"
                 }
               ]
 }
```

This array will expand frequently

Size of "Patron" document increases frequently

May trigger re-locating the document each time *(Bad)*

Document Size and Storage

Each document needs to be contiguous on disk

If doc size increases ➔ Document location must change

If doc location changes ➔ Indexes must be updates

➔ leads to more expensive updates

```
{
  _id: "joe",
  name: "Joe Bookreader",
  addresses: [
               {
                 street: "123 Fake Street",
                 city: "Faketon",
                 state: "MA",
                 zip: "12345"
               },
               {
                 street: "1 Some Other Street",
                 city: "Boston",
                 state: "MA",
                 zip: "12345"
               }
             ]
}
```

- Each document is allocated a ***power-of-2 bytes*** (the smallest above its size)
- Meaning, the system keeps some space empty for possible expansion

Examples (6)

One-to-Many "Book", "Publisher"

A book has one publisher

A publisher publishes many books

Referencing is better in this case

If embed "Publisher" inside "Book"

Repeating publisher info inside each of its books

Very hard to update publisher's info

If embed "Book" inside "Publisher"

Book becomes an array (many)

Frequently update and increases in size

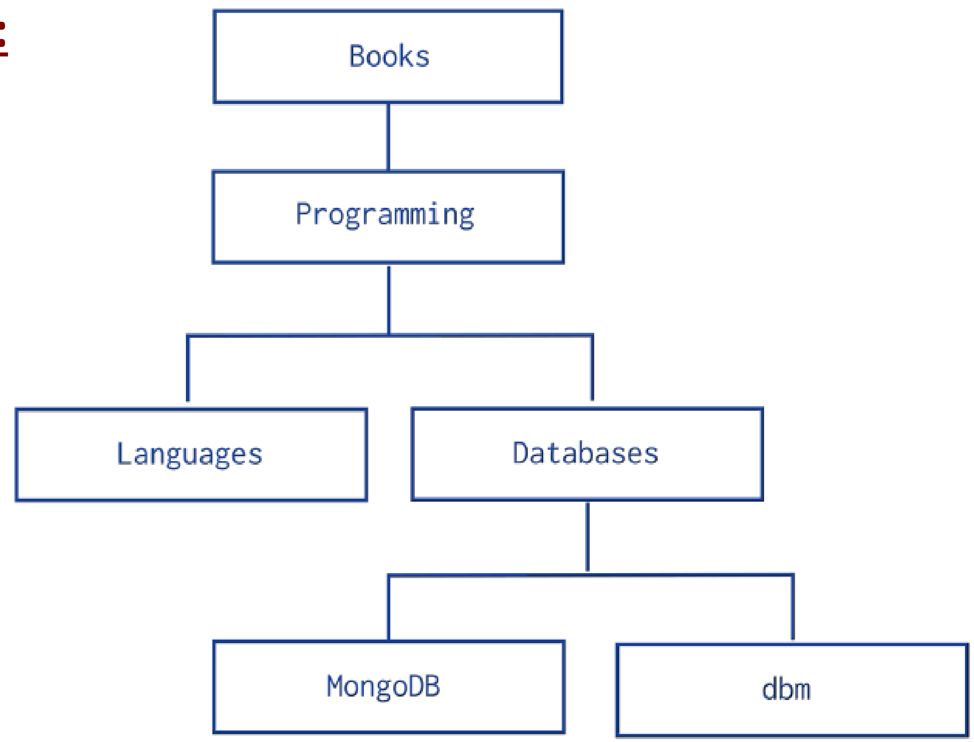
Modeling Tree Structure

Collections with Tree-Like Relationships

- Insert these records while maintaining this tree-like relationship

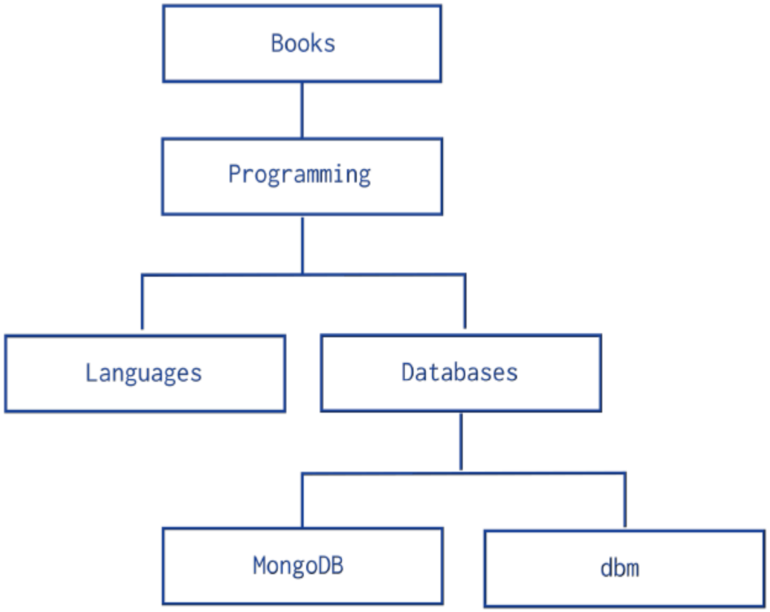
Given one node, answer queries:

- Report the parent node
- Report the children nodes
- Report the ancestors
- Report the descendants
- Report the siblings



Method 1: Parent References

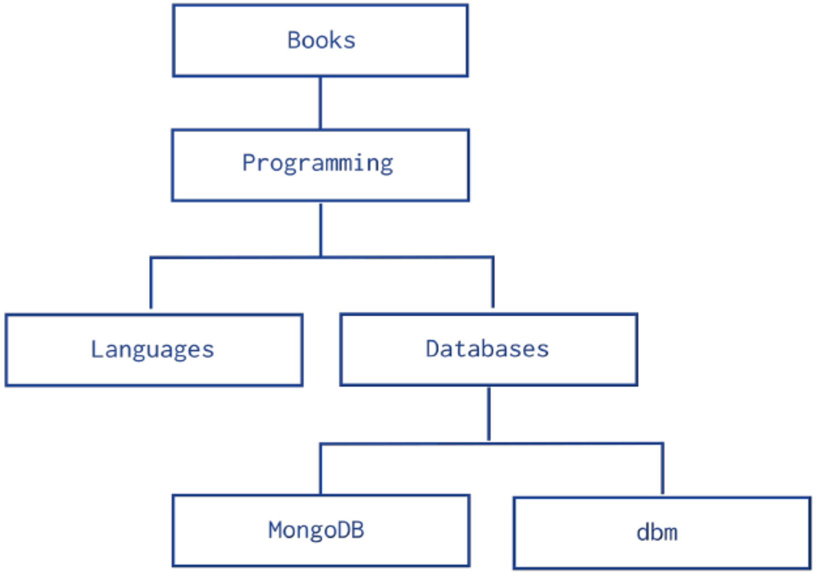
Each document has a field “parent”
Order does not matter




```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "dbm", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", parent: "Books" } )
db.categories.insert( { _id: "Books", parent: null } )
```

Method 2: Child References

Each document has an array of immediate children

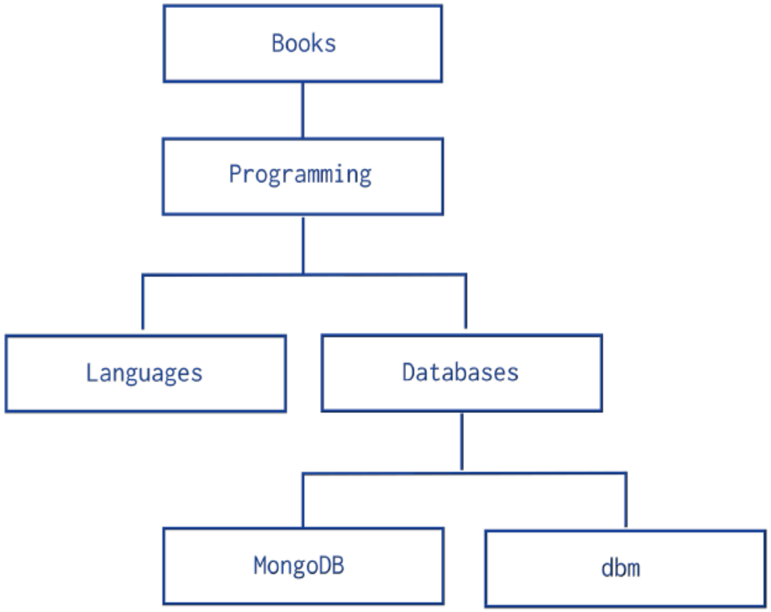



```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )

db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

Method 1: Parent References

Q1: Parent of "Programming"

Q2: Siblings of "Databases"

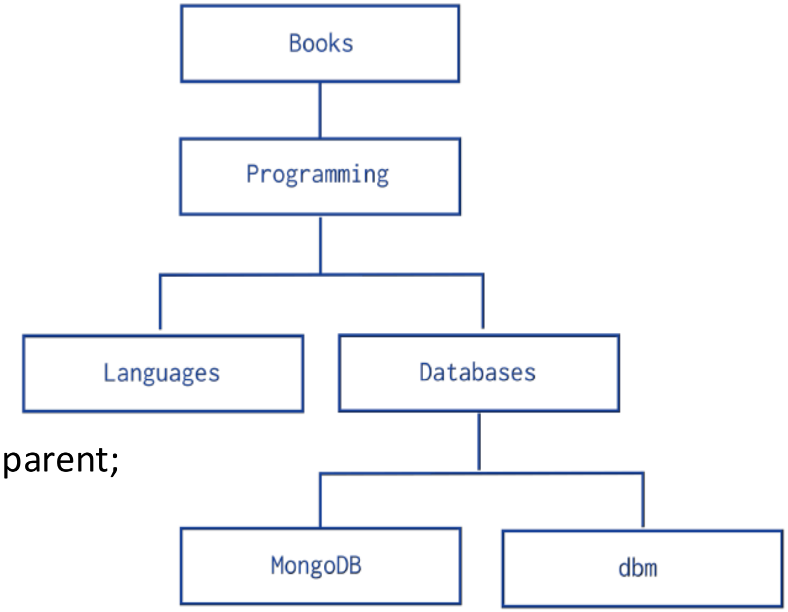



```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "dbm", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", parent: "Books" } )
db.categories.insert( { _id: "Books", parent: null } )
```

Method 1: Parent References

Q1: Parent of “Programming”

```
db.categories.find( {_id: "Programming"}, {parent: 1, _id: 0});
```



Q2: Siblings of “Databases”

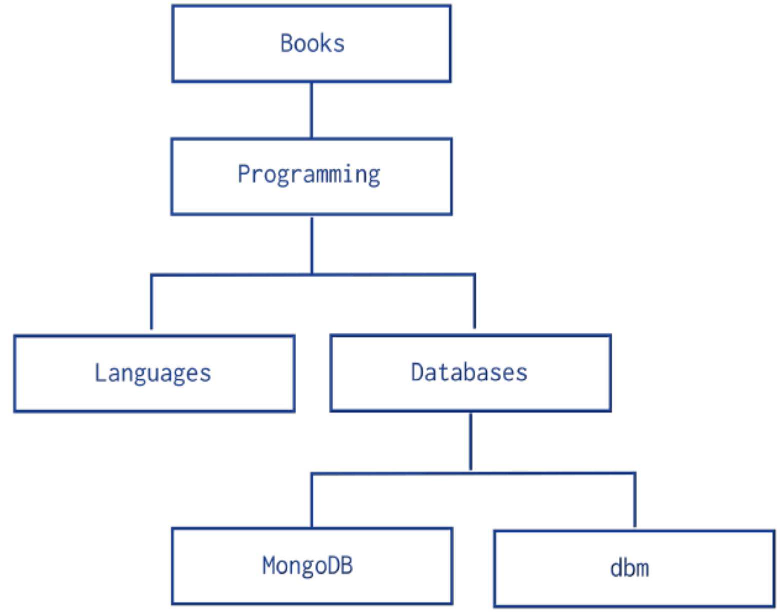
```
var  parentDoc = db.categories.findOne( {_id: "Databases"}).parent;

db.categories.find( {parent:  parentDoc,
                      _id: { $ne :"Databases"}    });
```

```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "dbm", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", parent: "Books" } )
db.categories.insert( { _id: "Books", parent: null } )
```

Method 1: Parent References

Q3: Descendants of "Programming"

Complex…Requires recursive calls

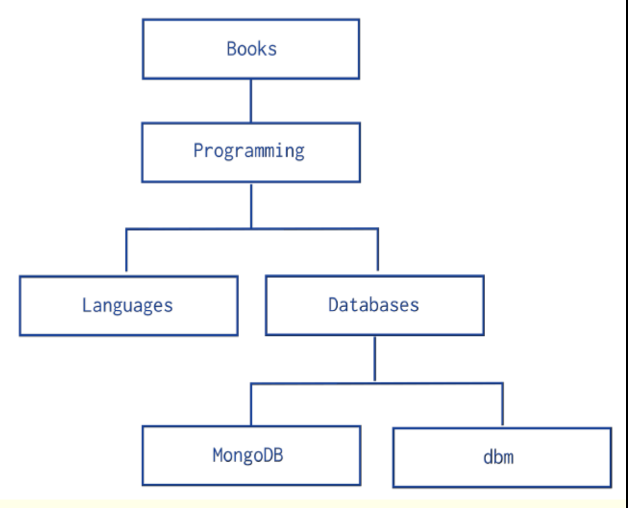



```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "dbm", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", parent: "Books" } )
db.categories.insert( { _id: "Books", parent: null } )
```

Method 1: Parent References

Q3: Descendants of "Programming"

```
var descendants = [];
var stack = [];
var item = db.categories.findOne({_id: "Programming"});
stack.push(item);
while (stack.length > 0) {
        var current = stack.pop();
       var children =  db.categories.find( {parent: current._id});
        while (children.hasNext() == true) {
                var child = children.next();
                descendants.push(child._id);
                stack.push(child);
        }
}

descendants;
```



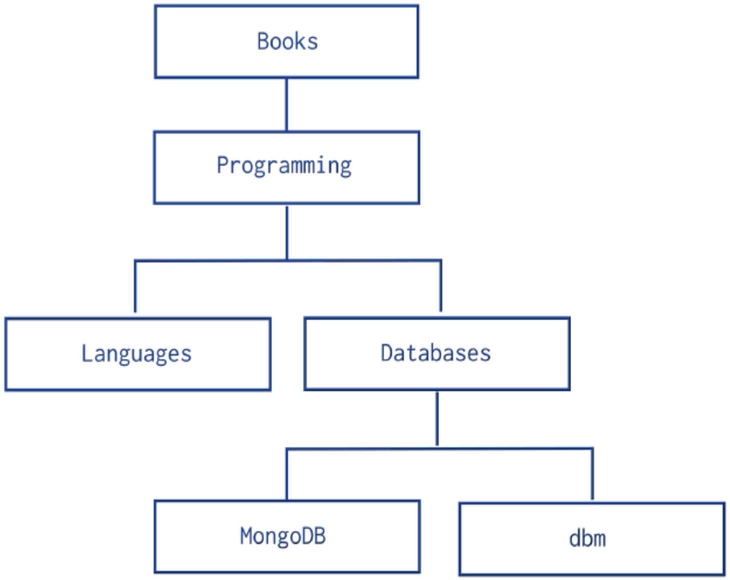
Method 1: Parent References

Q4: Ancestors of "MongoDB"

Try it yourself….

Should be:

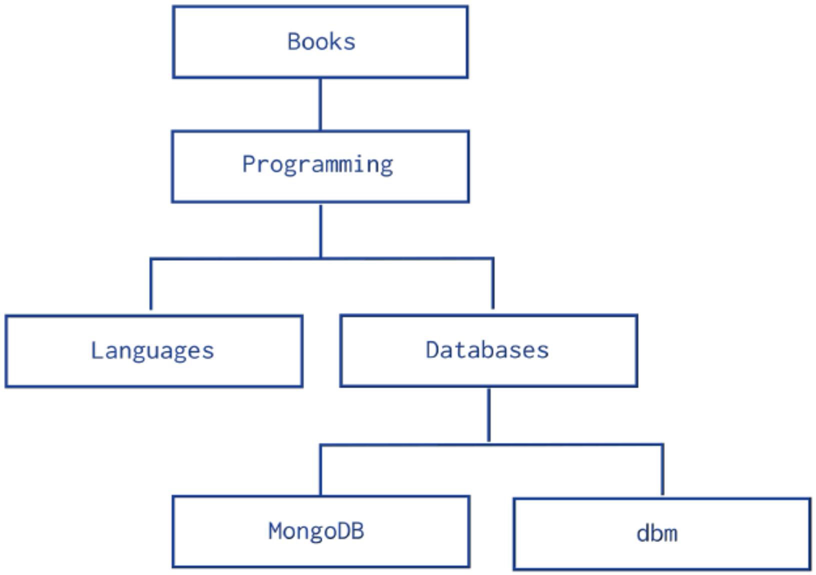
"Databases", "Programming", "Books"




```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "dbm", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", parent: "Books" } )
db.categories.insert( { _id: "Books", parent: null } )
```

Method 2: Child References

Q1: Get children documents of "Programming"



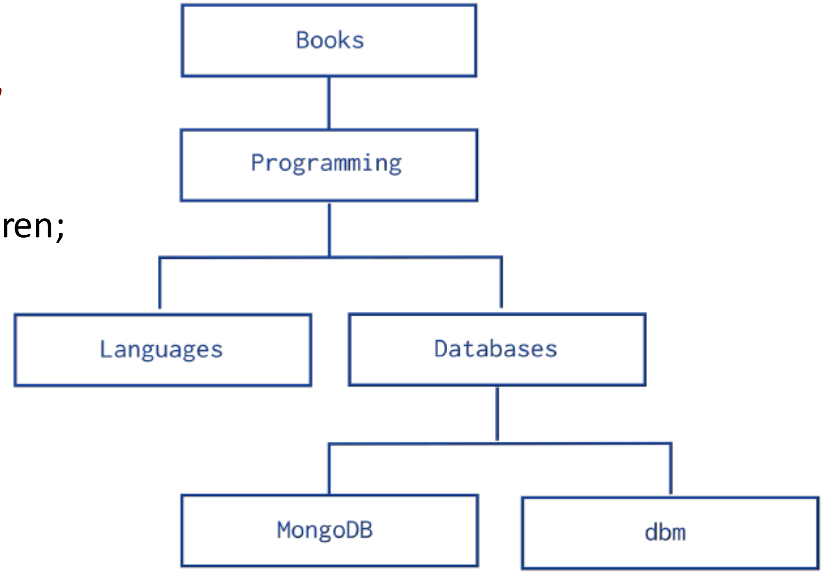

```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )

db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

Method 2: Child References

Q1: Get children documents of “Programming”

var x = db.categories.findOne({_id: "Programming"}).children;

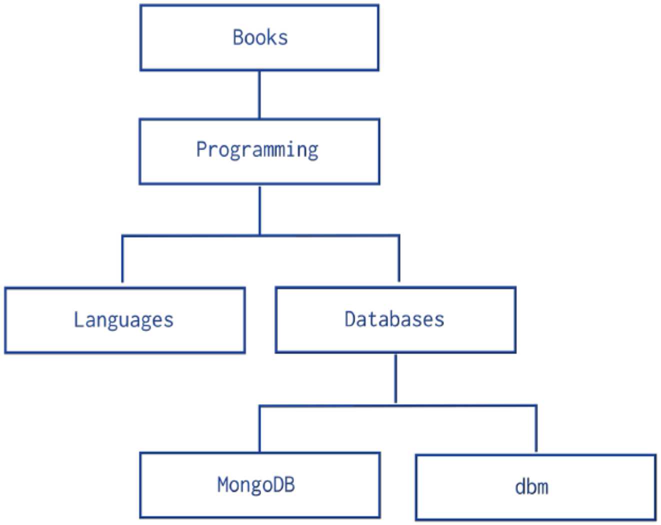
db.categories.find({_id: {$in: x}});




```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )

db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

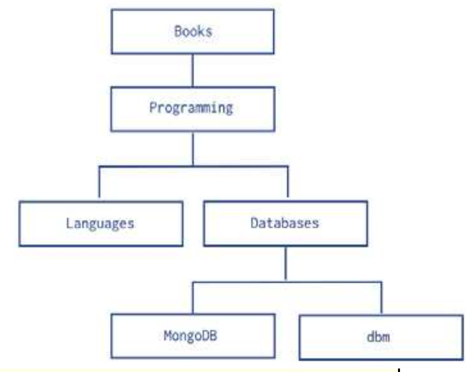
Method 2: Child References

Q2: Ancestors of "MongoDB"




```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )

db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

Method 2: Child References

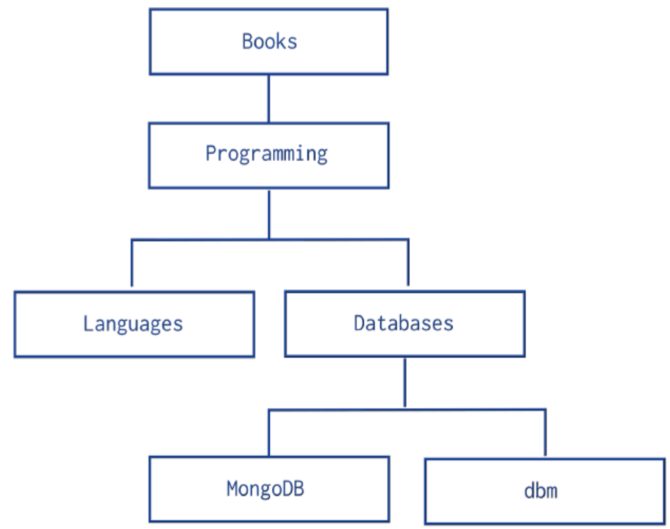


Q2: Ancestors of "MongoDB"

```
var results=[];
var parent = db.categories.findOne({children: "MongoDB"});
while(parent){
	print({Message: "Going up one level…"});
	results.push(parent._id);
	parent = db.categories.findOne({children: parent._id});
}

results;
```

Method 2: Child References

Q3: descendants of "Books"



Try it yourself….

Should be all nodes

```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )

db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

For today

- Download and install Neo4j community edition
- https://neo4j.com/download-center/#community

If you don’t already have it, you will also need to install Oracle JDK or Open JDK - Java Development Kit Standard Edition.

Visit http://localhost:7474 in your web browser
Default username and password: ‘neo4j’

Submit a screenshot to ICON