The MongoDB Tutorial
Introduction for MySQL Users

Stephane Combaudon
April 1st, 2014
Agenda

• Introduction
• Install & First Steps
• CRUD
• Aggregation Framework
• Performance Tuning
• Replication and High Availability
• Sharding
Before we start

• No knowledge of MongoDB is required

• Several interesting topics will intentionally be left out

• Any problem, any question? Raise your hand!
Agenda

• Introduction
• Install & First Steps
• CRUD
• Aggregation Framework
• Performance Tuning
• Replication and High Availability
• Sharding
Typical problems with RDBMS

- SQL is difficult to understand
- Changing the schema is difficult with large tables
  - aka “The relational model is not flexible”
- Scaling out and HA are difficult
  - Support for sharding is limited or non-existent
  - Many HA solutions, which one to choose?
“SQL is difficult to understand”

- Everything is a JSON document in MongoDB
- No joins but a powerful way to write complex queries (aggregation framework)
- Queries are easy to read/write

  - **SQL**: SELECT * FROM people
    WHERE name = 'Stephane'

  - **MongoDB**: db.people.find({name:'Stephane'})
“The relational model is not flexible”

- MongoDB is schemaless, no ALTER TABLE!

```javascript
db.people.insert({name: 'Stephane'});
db.people.insert({name: 'Joe', age: 30});
```

• But be careful, this is also allowed

```javascript
db.people.insert({name: 'Stephane'})
db.people.insert({n: 'Joe'})
```
MongoDB solutions (3)

- “Scaling and HA are difficult”

Diagram:
- Shard 1
- Shard 2
- Shard N
- Replica set
- Config servers
- mongos
- mongod
- Application
Summary

Scalability & Performance

- Memcached

Functionality

- MongoDB
- RDBMS
Agenda

• Introduction
• Install & First Steps
• CRUD
• Aggregation Framework
• Performance Tuning
• Replication and High Availability
• Sharding
MongoDB setup

- See mongod_setup.rst
### Useful terminology

<table>
<thead>
<tr>
<th>Relational</th>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Database</td>
</tr>
<tr>
<td>Table</td>
<td>Collection</td>
</tr>
<tr>
<td>Row</td>
<td>(JSON) document</td>
</tr>
</tbody>
</table>
JSON

• Stands for JavaScript Object Notation
  – But not tied to Javascript
• Open standard for human-readable data interchange
• Lightweight alternative to XML
• Internally, docs are stored in BSON
  – Binary JSON
  – See http://bsonspec.org
Invoking the shell

$ mongo (--port=xxx, default 27017)

> show dbs
local  0.03125GB
test   0.0625GB
> use test
switched to db test
> show collections
people
system.indexes
> db.people.findOne()
{ "_id" : ObjectId("523ef7bf8108101415e7d1d1"), "age" : 30, "country" : "XXX", "name" : "Joe" }
{ "_id" : ObjectId("523ef7ac8108101415e7d1d0"), "age" : 33, "country" : "XXX", "name" : "Stephane" }
> db.people.find().pretty()
{    "_id" : ObjectId("523ef7bf8108101415e7d1d1"),    "age" : 30,    "country" : "XXX",    "name" : "Joe" }
{    "_id" : ObjectId("523ef7ac8108101415e7d1d0"),    "age" : 33,    "country" : "XXX",    "name" : "Stephane" }
use test
db.people.insert({name:'Stephane',country:'FR'})

• This inserts a document
  - In the people collection
  - Collection is in the test database

• Collection is created if it did not exist
Inserting data (2)

- The shell embeds a JS interpreter
  - You can also use a variable to build the JSON document step by step

```javascript
> x = {name:'Stephane'}
{ "name" : "Stephane" }
> x.country = 'FR'
FR
> x
{ "name" : "Stephane", "country" : "FR" }
> db.people.insert(x)
> db.people.findOne()
{
    "_id" : ObjectId("526fd69e64ea1df71b82f55b"),
    "name" : "Stephane",
    "country" : "FR"
}
```
• Unique identifier of a doc

db.people.insert({_id:'Stephane',country:'FR'})

• If you don't, it will be created for you

"_id" : ObjectId("523ef7bf8108101415e7d1d1")

• Monotonically increasing on a single node
  – Think auto_increment in MySQL
Structure of a document

- Set of key/value pairs
  - {'key1':'value1','key2':'value2',...}

- A value can be an array

- A value can be another document
Lab #1

• See lab1.rst
Agenda

• Introduction
• Install & First Steps
• CRUD
• Aggregation Framework
• Performance Tuning
• Replication and High Availability
• Sharding
Inserting a document

- **SQL**
  - `INSERT INTO t (fn, ln) VALUES ('John', 'Doe')`

- **MongoDB**
  - `db.t.insert({fn:'John', ln: 'Doe'})`

- No need to specify non-existing fields
Updating a document (1)

- **SQL**
  - `UPDATE t SET ln='Smith' WHERE ln='Doe'

- **MongoDB**
  - `db.t.update({ln:'Doe'},{$set:{ln:'Smith'}})`

- **Only 1 doc. is updated by default**
  - **Specify** `multi: true` **for multi-doc updates**
  - `db.t.update({ln:'Doe'},{$set:{ln:'Smith'}}, {multi: true})`
• To add a new field
  - No need to run ALTER TABLE!
  - It is a regular update
  - `db.t.update({ln:'Smith'},{$set:{age:30}})`
Removing documents

- **SQL**
  - `DELETE FROM t WHERE fn='John'

- **MongoDB**
  - `db.t.remove({fn:'John'})`
Dropping a collection

• **SQL**
  - `DROP TABLE t`

• **MongoDB**
  - `db.t.drop()`
Selecting all documents

• **SQL**
  - `SELECT * FROM t`

• **MongoDB**
  - `db.t.find()`

• **SQL**
  - `SELECT id, name FROM t`

• **MongoDB**
  - `db.t.find({},{name:1})`
Specifying a “WHERE” clause

- **SQL**
  - SELECT * FROM t WHERE fn='John'

- **MongoDB**
  - db.t.find({ln:'John'})

- **SQL**
  - SELECT id, age FROM t WHERE fn='John' AND ln='Smith'

- **MongoDB**
  - db.t.find({fn:'John',ln:'Smith'}, {age:1})
Sorts, limits

• **SQL**
  
  ```sql
  SELECT country FROM t
  WHERE fn='John'
  ORDER BY age DESC LIMIT 10
  ```

• **MongoDB**
  
  ```javascript
  db.t.find(
    {fn:'John'},
    {country:1, _id:0}
  ).sort({age:1}).limit(10)
  ```
Inequalities

- **SQL**
  - SELECT fn, ln FROM t
    WHERE age > 30

- **MongoDB**
  - db.t.find(
    {age:{ $gt: 30 }},
    {fn: 1, ln: 1, _id: 0}
  )
Conditions on subdocuments

```javascript
{
  _id: ..., 
  fn: 'John', ln: 'Doe', 
  address: {
    country: 'US', 
    state: 'CA', 
  }, 
}

db.t.find( 
  {address.country: 'US'}, 
  {fn: 1, ln: 1, _id: 0}
)
Conditions on arrays

```javascript
{
    _id: ...
    hobbies: ['music', 'MongoDB', 'Python']
}
```

- **Exact match on array**
  ```javascript
db.t.find({hobbies: ['music', 'MongoDB', 'Python']})
```

- **Match on array element**
  ```javascript
db.t.find({hobbies: 'Python'})
```
Counting

• SQL
  - SELECT COUNT(*) FROM t
    WHERE country='US'

• MongoDB
  - db.t.count({country:'US'})
Lab #2

- See lab2.rst file
Agenda

• Introduction
• Install & First Steps
• CRUD
• Aggregation Framework
• Performance Tuning
• Replication and High Availability
• Sharding
Aggregation

- “Means” GROUP BY, SUM(), ...

- Not possible with the operators we saw previously (exception: count)

- 2 ways to aggregate
  - Map-Reduce # Not covered here
  - Aggregation Framework
Aggregation Framework

- Available from MongoDB 2.2
- Easier and faster than Map-Reduce
- Some limitations
  - AF is perfect for simple cases
  - Map-Reduce can be needed for complex situations
10,000 feet overview

- Documents are modified through pipelines

- Similar to Unix pipelines

```
grep oo /etc/passwd | sort -rn | awk -F ':' '{print $1,$3,$4}'
```

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>Stage 2</th>
<th>Stage 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
<td>Output</td>
</tr>
</tbody>
</table>
First example

- **SQL**
  
  ```sql
  SELECT fn, count(*) AS total FROM people WHERE fn >= 'M%' GROUP BY fn
  ```

- **MongoDB**
  
  ```javascript
  db.people.aggregate([{ $match: { fn: { $gte: 'M' } } }, { $group: { _id: "$fn", total: { $sum: 1 } } }])
  ```

- You probably want some explanation!
First example, explained

```javascript
db.people.aggregate(
  {$match:XXX}, # Pipeline 1, filtering criteria
  {$group:XXX}  # Pipeline 2, group by
)
```

- Filtering condition like with `find()`
  ```javascript
  $match:{fn:{$gte:'M'}}
  ```

- Specifying the grouping field
  ```javascript
  $group:{_id:"$fn",...}
  ```

- Specifying the aggregated fields
  ```javascript
  $group:{..., total:{$sum:1}}
  ```
Pipeline order matters

db.people.aggregate(
    {
        $match:...
    },
    {
        $group:...
    }
)

vs

db.people.aggregate(
    {
        $group:...
    },
    {
        $match:...
    }
)
SQL analogy

SELECT ... FROM people
WHERE ...
GROUP BY ...

vs

SELECT ... FROM people
GROUP BY ...
HAVING ...
The projecting operator allows renaming keys/values

db.people.aggregate(
  {$match:...},
  {$group:...},
  {$project:{_id:0, name:{$toUpper:"$_id"}, total:1}}
)
Final query

- **SQL**

```sql
SELECT UPPER(fn) AS name, count(*) AS total
FROM people
WHERE fn >= 'M%
GROUP BY fn
```

- **MongoDB**

```javascript
db.people.aggregate(
  {$match:{fn:{$gte:'M'}}},
  {$group:{_id:"$fn",total:{$sum:1}}},
  {$project:{_id:0, name:{$toUpper:"$_id"}, total:1}}
)
```
Other operators

- **$sort**
  - `{sort: {total: -1}}`

- **$limit**
  - `{limit: 10}`

- **$skip**
  - `{skip: 100}`

- **$unwind**
  - Splits a document with an array into multiple documents
Lab #3

• See lab3.rst
Agenda

• Introduction
• Install & First Steps
• CRUD
• Aggregation Framework
• Performance Tuning
• Replication and High Availability
• Sharding
Indexes

• MongoDB supports indexes
  – At the collection level
  – Similar to indexes on RDBMS

• Can be used for
  – More efficient filtering
  – More efficient sorting
  – Index-only queries (covering index)
Main types of indexes

- **Single-field/multiple field index**
  - `db.t.ensureIndex({fn:1})`
  - `db.t.ensureIndex({fn:1, age:-1})`

- **Unique index**
  - All collections have an index on _id
    - `db.t.ensureIndex({username:1}, {unique:true})`

- **Indexes on arrays, embedded fields, subdocuments** are also supported
Compound indexes (1)

- **Sort order**
  - **Traversal in either direction**

  \[
  \text{db.t.find().sort({country:1,age:-1})}
  \]

- **Good indexes**
  - \{country:1, age:-1\}
  - \{country:-1, age:1\}

- **Bad index**
  - \{country:1, age:1\}
Compound indexes (2)

• Prefixes
  – Leftmost prefixes are supported

\[ \text{db.t.ensureIndex(\{age:1,country:1\})} \]

– Good for
  • \text{db.t.find(\{age:30\})}
  • \text{db.t.find(\{age:30,country:'US'\})}

– Not good for
  • \text{db.t.find(\{country:'US'\})}
Lab #4

• See lab4.rst
Agenda

• Introduction
• Install & First Steps
• CRUD
• Aggregation Framework
• Performance Tuning
• Replication and High Availability
• Sharding
Overview

• Replication is asynchronous
• All writes go to the master
• Secondary can accept reads
• A new master is elected if the current master fails
• An arbiter (no data) can be set up for the election process
Primary

Writes

Heartbeat

Secondary

Secondary
Setup of a 3 node replica set

- Start 3 instances with `--replSet rsname`
- Then on the master

```javascript
rsconf = {
    _id: "RS",
    members: [{
        _id: 0,
        host: "localhost:30001"
    }]
}
rs.initiate(rsconf)
rs.add("localhost:30002")
rs.add("localhost:30003")
```
If the master crashes

• If a heartbeat does not return within 10s, the master is considered unavailable

• Election of a new master starts then

• You can influence the choice by setting a priority (between 0 and 100)
Setting priorities

cfg = rs.conf()
cfg.members[1].priority = 0
cfg.members[2].priority = 10
rs.reconfig(cfg)

• A node set to have higher priority than the master will be elected the new master
Write concerns (aka w option)

• How many nodes should ack a write?
  - \( w=1 \)
  - Primary only (default setting)
  - \( w=2 \)
  - Primary and one secondary
  - \( w=\text{majority} \)
  - Majority of the nodes
Setting a write concern

cfg = rs.conf()
cfg.settings.getLastErrorDefaults = {
  w: "majority"
}  
rs.reconfig(cfg)
Read preferences

• Can be any of
  - primary
  - primaryPreferred
  - secondary
  - secondaryPreferred
  - nearest
  - Or you can use a custom tag
Lab #5

- See lab5.rst
Agenda

• Introduction
• Install & First Steps
• CRUD
• Aggregation Framework
• Performance Tuning
• Replication and High Availability
• Sharding
When to shard

- When the dataset no longer fits in a single server
- When write activity exceeds the capacity of a single node
- When the working set no longer fits in memory
Architecture of a sharded cluster

Shard 1
- mongos
- mongod
- Config servers
  - Application

Shard 2
- mongos
- mongod
  - Application

Shard N
- mongos
- mongod
  - Application

Replica set
Setup (1)

- **Start the config servers**
  - `mongod --configsvr --dbpath <path> --port 40001`
  - `mongod --configsvr --dbpath <path> --port 40002`
  - `mongod --configsvr --dbpath <path> --port 40003`

- **Start a router**
  - `mongos --configdb localhost:40001, localhost:40002, localhost:40003 --port 31000`

- **Connect to mongos**
  - `mongo --host localhost --port 31000`
Setup (2)

• Add shards
  - `sh.addShard('RS/localhost:30001')`

• Enable sharding for a database
  - `sh.enableSharding('test')`

• Enable sharding for a collection
  - `sh.shardCollection('test.people',{'fn':1})`
Balancing (1)

• Data is stored in chunks

```
sh.addShard(shard1)
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>45</td>
</tr>
<tr>
<td>46</td>
<td>82</td>
</tr>
<tr>
<td>83</td>
<td>+∞</td>
</tr>
</tbody>
</table>

Shard 1
Balancing (2)

```
sh.addShard(shard2)
```

**Shard 1**
- $-\infty$ 32
- 46 82

**Shard 2**
- 33 45
- 83 $+\infty$
Balancing (3)

```python
sh.addShard(shard3)
```

<table>
<thead>
<tr>
<th>Shard 1</th>
<th>Shard 2</th>
<th>Shard 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-∞</td>
<td>33</td>
<td>46</td>
</tr>
<tr>
<td>32</td>
<td>45</td>
<td>82</td>
</tr>
<tr>
<td>83</td>
<td>+∞</td>
<td></td>
</tr>
</tbody>
</table>
Lab #6

- See lab6.rst
Thank you for your attention!

stephane.combaudon@percona.com