Andreas Jung, www.zopyx.com

PYTHON AND MONGODB
THE PERFECT MATCH

Trainer Andreas Jung

- Python developer since 1993
- Python, Zope & Plone development
- Specialized in Electronic Publishing
- Director of the Zope Foundation
- Author of dozens add-ons for Python, Zope and Plone
- Co-Founder of the German Zope User Group (DZUG)
- Member of the Plone Foundation
- using MongoDB since 2009

Agenda (45 minutes per slot)

1. Introduction to MongoDB
2. Using MongoDB
3. Using MongoDB from Python with PyMongo
4. (PyMongo extensions/ORM-ish layers or Q/A)

Things not covered in this tutorial

- Geospatial indexing
- Map-reduce
- Details on scaling (Sharding, Replica sets)

Part I/4

- Introduction to MongoDB:
  - Concepts of MongoDB
  - Architecture
  - How MongoDB compares with relational databases
  - Scalability

MongoDB is...

- an open-source,
- high-performance,
- schema-less,
- document-oriented

database

Let‘s agree on the following or leave...

- MongoDB is cool
- MongoDB is not the multi-purpose-one-size-fits-all database
- MongoDB is another additional tool for the software developer
- MongoDB is not a replacement for RDBMS in general
- Use the right tool for each task

And.....

- Don‘t ask me about how to do JOINs in MongoDB ☺

Oh, SQL – let‘s have some fun first

A SQL statement walks into a bar and sees two tables. He walks and says: „Hello, may I join you“

A SQL injection walks into a bar and starts to quote something but suddenly stops, drops a table and dashes out.

The history of MongoDB

- 10gen founded in 2007
- Started as cloud-alternative GAE
  - App-engine ed
  - Database p
  - Javascript as implementation language
- 2008: focusing on the database part: MongoDB
- 2009: first MongoDB release
- 2011: MongoDB 1.8:
  - Major deployments
  - A fast growing community
  - Fast adoptation for large projects
  - 10gen growing

Major MongoDB deployments

MongoDB is schema-less

- JSON-style data store
- Each document *can* have its own schema
- Documents inside a collection usually share a common schema by convention

```
{‚name‘ : ‚kate‘, ‚age‘:12, }
{‚name‘ : ‚adam‘, ‚height‘ : 180}
{‚q‘: 1234, ‚x‘ = [‚foo‘, ‚bar‘]}
```

Terminology: RDBMS vs. MongoDB

RDBMS	MongoDB
Database	Database
Tables	Collections
Rows	Documents
Indexes	Indexes
SQL as query language	JSON-style syntax

Characteristics of MongoDB (I)

- High-performance
- Rich query language (similar to SQL)
- Map-Reduce (if you really need it)
- Secondary indexes
- Geospatial indexing
- Replication
- Auto-sharing (partitioning of data)
- Many platforms, drivers for many languages

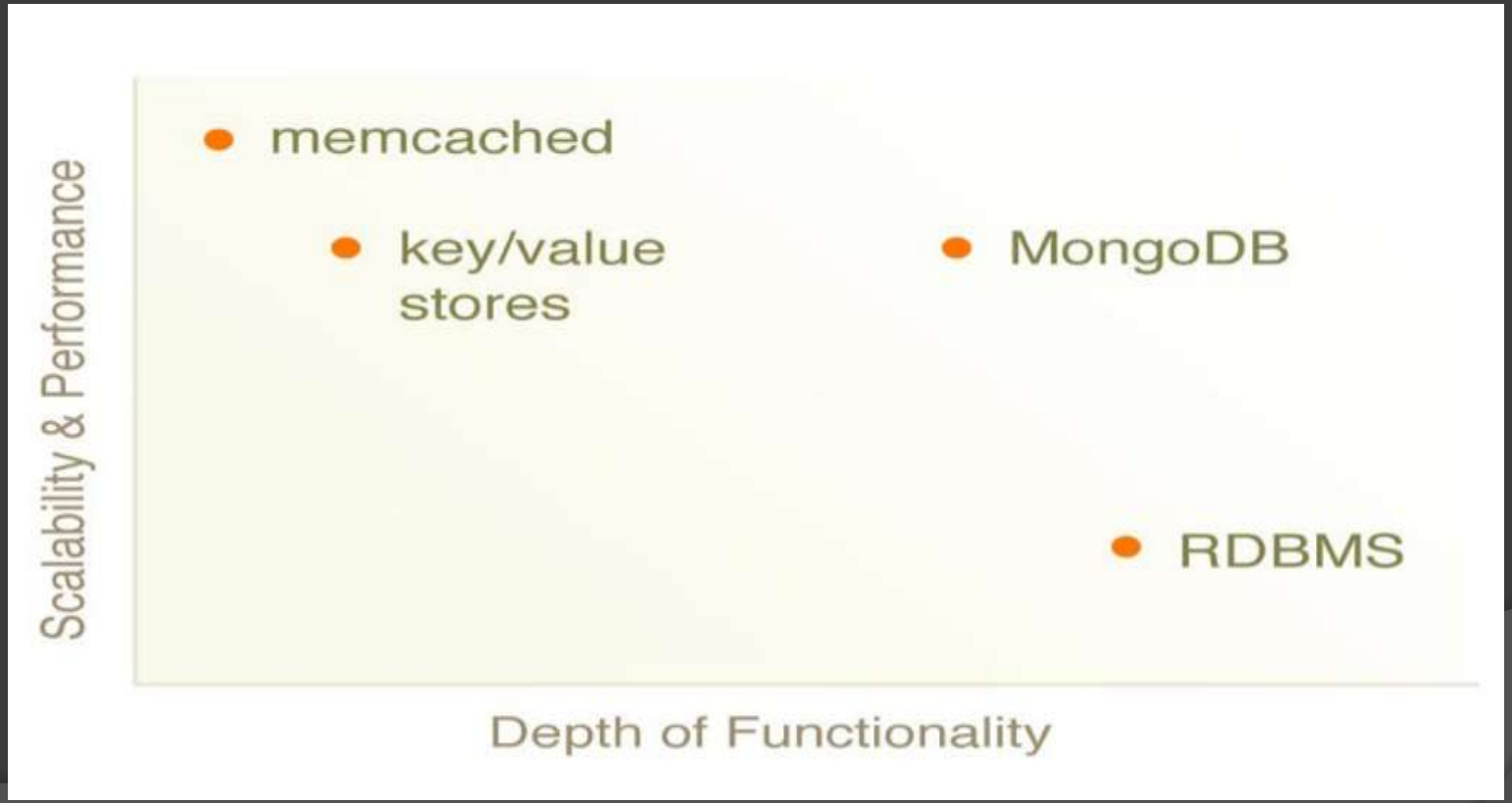
Characteristics of MongoDB (II)

- No transaction support, only atomic operations
- Default: „fire-and-forget“ mode for high throughput
- „Safe-Mode“: wait for server confirmation, checking for errors

Typical performance characteristics

- Decent commoditiy hardware:
  - Up to 100.000 read/writes per second (fire-and-forget)
  - Up to 50.000 reads/writes per second (safe mode)
- Your mileage may vary – depending on
  - RAM
  - Speed IO system
  - CPU
  - Client-side driver & application

Functionality vs. Scability



MongoDB: Pros & Cons

Pros	Cons
Good for the web	Not for highly transactional apps
Caching	Ad-hoc business intelligence (dataware-house apps)
High volumne, low volume apps	Can not replace complex SQL queries
Scalability	
Speed	

Durability

- Default: fire-and-forget (use safe-mode)
- Changes are kept in RAM (!)
- Fsync to disk every 60 seconds (default)
- Deployment options:
  - Standalone installation: use journaling (V 1.8+)
  - Replicated: use replica sets(s)

Differences from Typical RDBMS

- Memory mapped data
  - All data in memory (if it fits), synced to disk periodically
- No joins
  - Reads have greater data locality
  - No joins between servers
- No transactions
  - Improves performance of various operations
  - No transactions between servers

Replica Sets

- Cluster of N servers
- Only one node is ‘primary’ at a time
  - This is equivalent to master
  - The node where writes go
- Primary is elected by concensus
- Automatic failover
- Automatic recovery of failed nodes

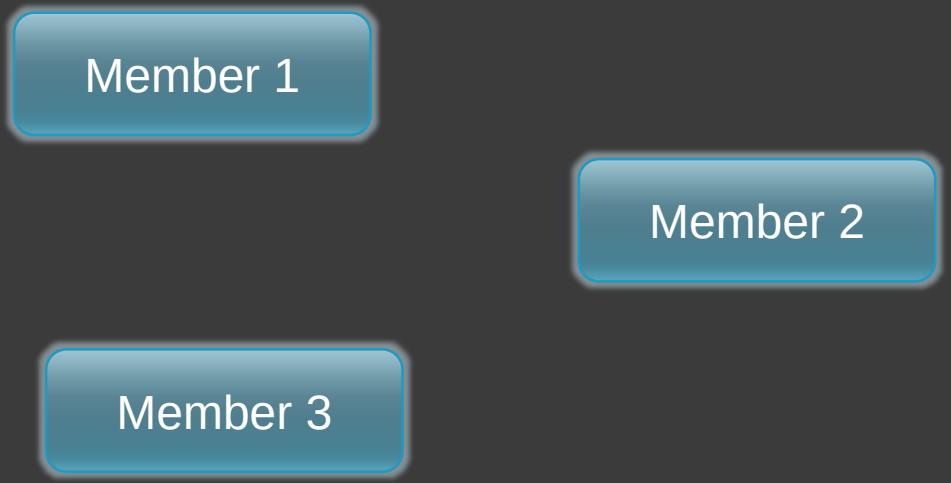
Replica Sets - Writes

- A write is only ‘committed’ once it has been replicated to a majority of nodes in the set
- Before this happens, reads to the set may or may not see the write
- On failover, data which is not ‘committed’ may be dropped (but not necessarily)
  - If dropped, it will be rolled back from all servers which wrote it
- For improved durability, use getLastError/w
  - Other criteria – block writes when nodes go down or slaves get too far behind
- Or, to reduce latency, reduce getLastError/w

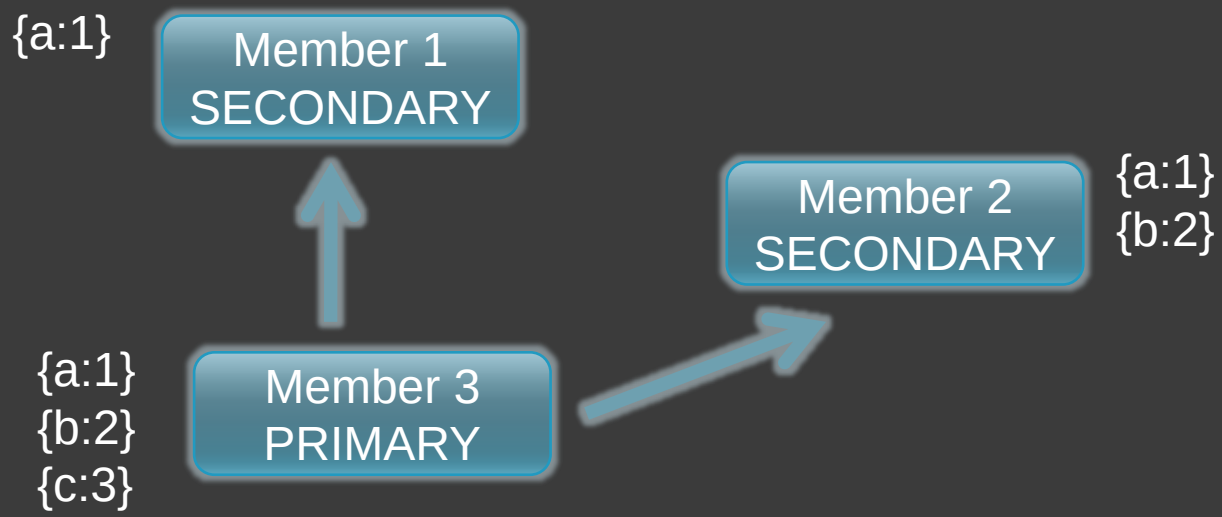
Replica Sets - Nodes

- Nodes monitor each other's heartbeats
  - If primary can't see a majority of nodes, it relinquishes primary status
- If a majority of nodes notice there is no primary, they elect a primary using criteria
  - Node priority
  - Node data's freshness

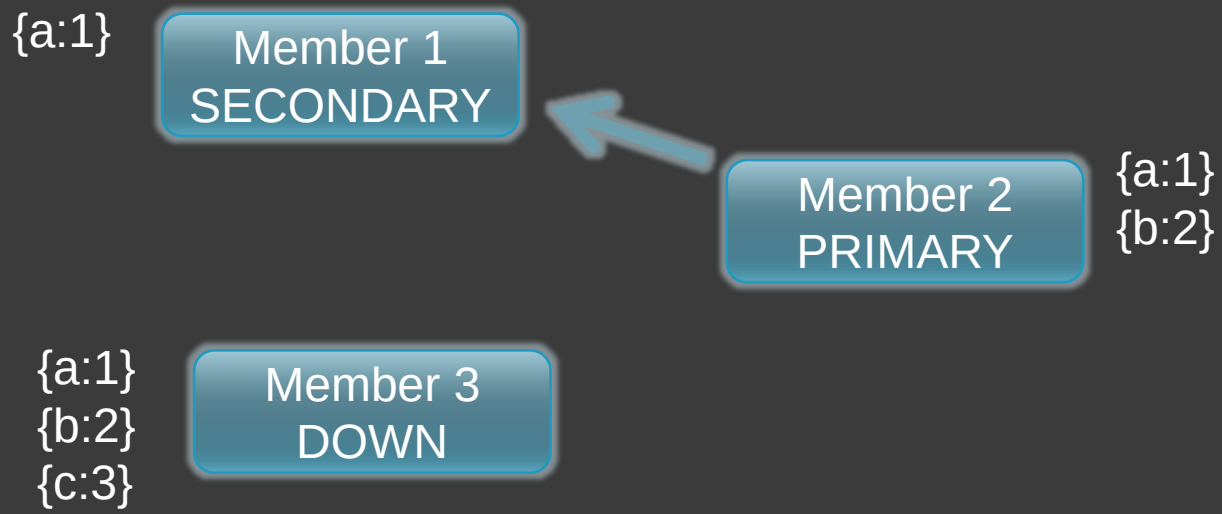
Replica Sets - Nodes



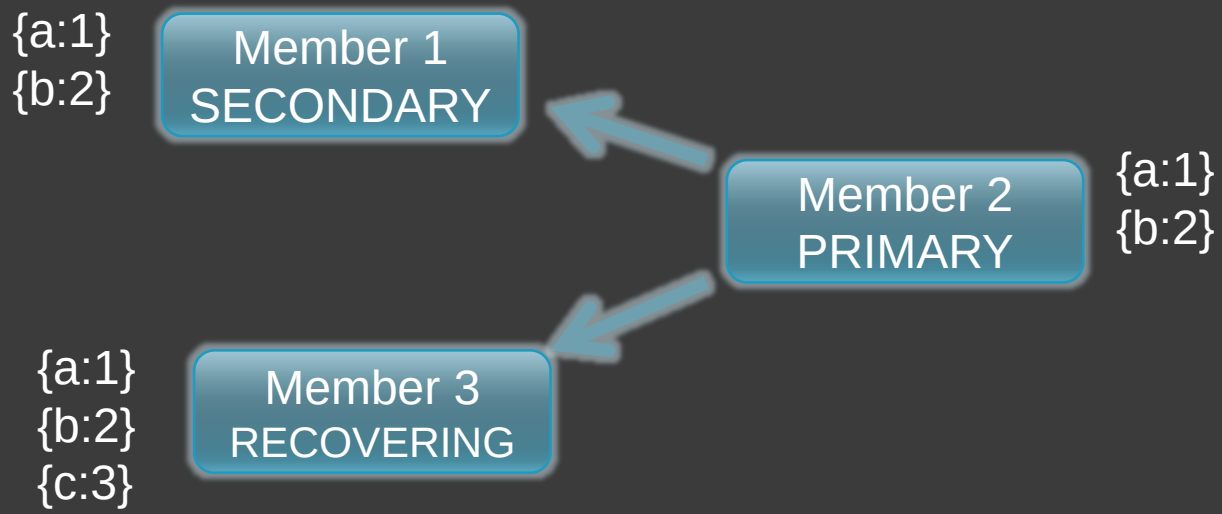
Replica Sets - Nodes



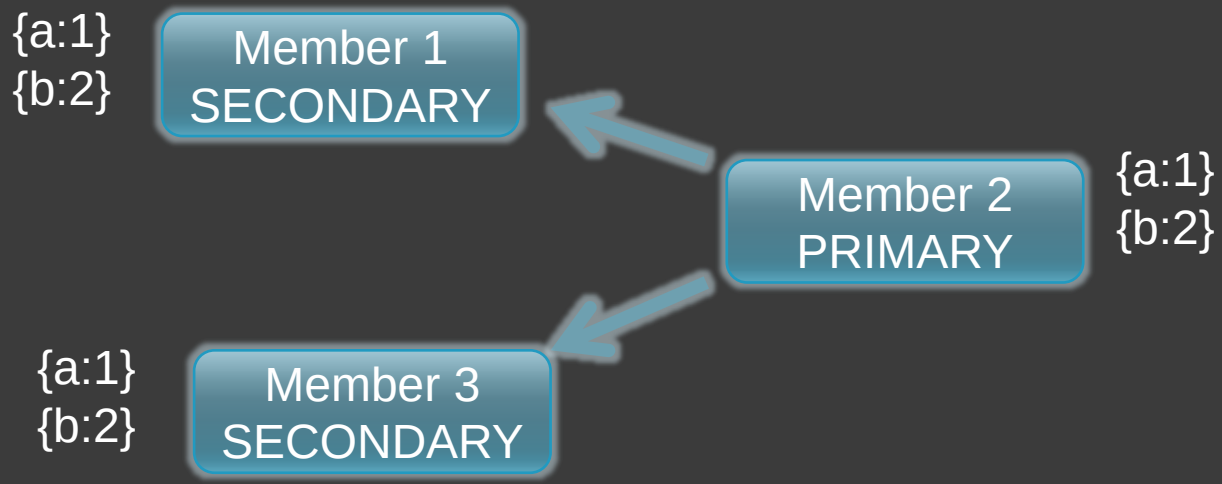
Replica Sets - Nodes



Replica Sets - Nodes



Replica Sets - Nodes



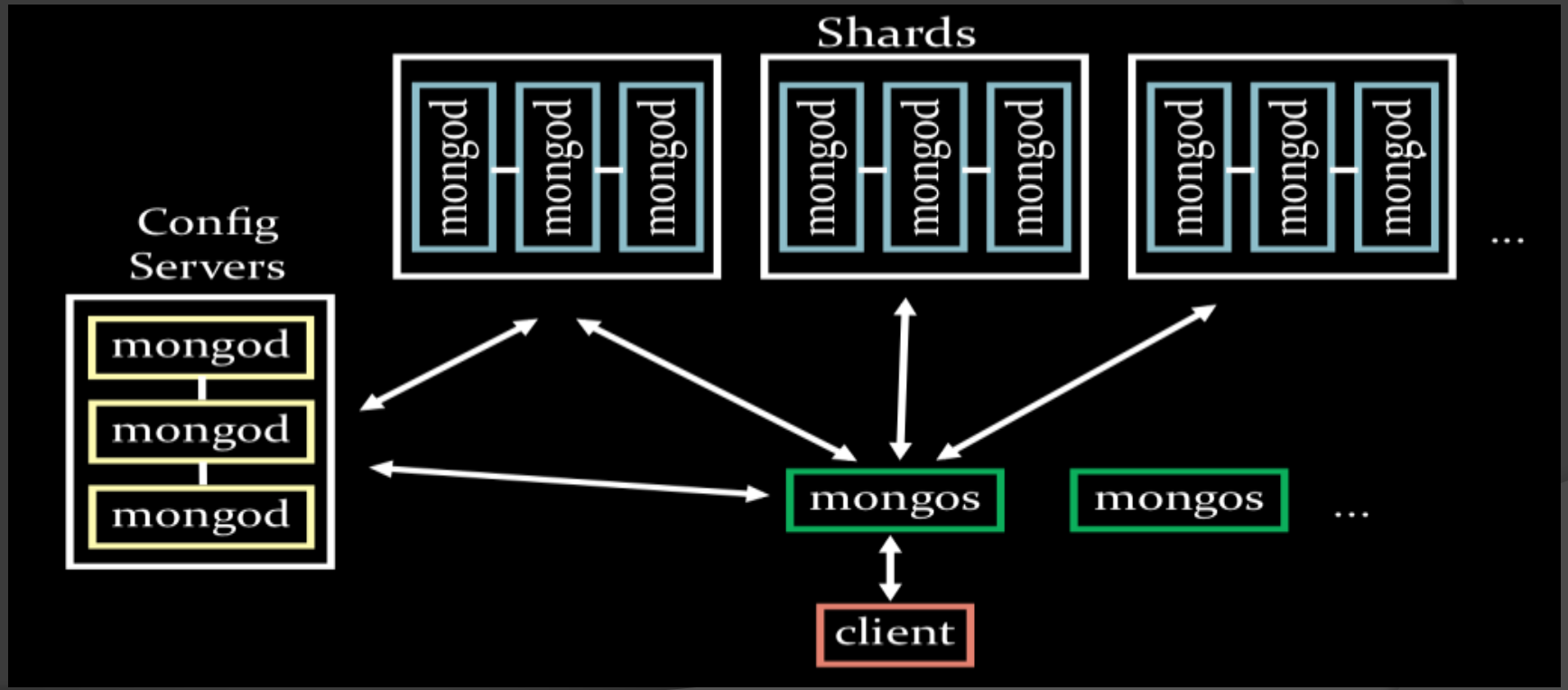
Replica Sets – Node Types

- Standard – can be primary or secondary
- Passive – will be secondary but never primary
- Arbiter – will vote on primary, but won't replicate data

SlaveOk

- db.getMongo().setSlaveOk();
  - Syntax varies by driver
- Writes to master, reads to slave
  - Slave will be picked arbitrarily

Sharding Architecture



Shard

- A replica set
- Manages a well defined range of **shard keys**

Shard

- Distribute data across machines
- Reduce data per machine
  - Better able to fit in RAM
- Distribute write load across shards
- Distribute read load across shards, and across nodes within shards

Shard Key

Collection	Min	Max	location
users	{name:'Miller'}	{name:'Nessman'}	shard 2
users	{name:'Nessman'}	{name:'Ogden'}	Shard 4
…			

- { user_id: 1 }
- { lastname: 1, firstname: 1 }
- { tag: 1, timestamp: -1 }
- { _id: 1 }
  - This is the default

Mongos

- Routes data to/from shards
- db.users.find({ user_id: 5000 })
- db.users.find({ user_id: { $gt: 4000, $lt: 6000 } })
- db.users.find({ hometown: ‘Seattle’ })
- db.users.find({ hometown: ‘Seattle’ }).sort({ user_id: 1 })

Differences from Typical RDBMS

- Memory mapped data
  - All data in memory (if it fits), synced to disk periodically
- No joins
  - Reads have greater data locality
  - No joins between servers
- No transactions
  - Improves performance of various operations
  - No transactions between servers
- A weak authentication and authorization model

Part 2/4

- Using MongoDB
  - Starting MongoDB
  - Using the interactive Mongo console
  - Basic database operations

Getting started...the server

- wget http://fastdl.mongodb.org/osx/mongodb-osx-x86_64-1.8.1.tgz
- tar xfz mongodb-osx-x86_64-1.8.1.tgz
- cd mongodb-osx-x86_64-1.8.1
- mkdir /tmp/db
- bin/mongod –dbpath /tmp/db

- Pick up your OS-specific package from http://www.mongodb.org/downloads
- Take care of 32 bit bs. 64 bit version

Getting started...the console

- bin/mongod

- mongod listens to port 27017 by default
- HTTP interface on port 28017

- > help
- > db.help()
- > db.some_collection.help()

Datatypes...

- Remember: MongoDB is schema-less
- MongoDB supports JSON + some extra types

JSON	BSON
• string • integer • boolean • double • null • array • object	• date • object id • binary • regular expression • code

A small address database

- Person:
  - firstname
  - lastname
  - birthday
  - city
  - phone

Inserting

- > db.foo.insert(document)
- > db.foo.insert({‚firstname‘ : ‚Ben‘})

- every document has an „_id“ field
- „_id“ inserted automatically if not present

Querying

- > db.foo.find(query_expression)
- > db.foo.find({‚firstname‘ : ‚Ben‘})

- Queries are expressed using JSON notation with JSON/BSON objects
- query expressions combined using AND (by default)
- http://www.mongodb.org/display/DOCS/Querying

Querying with sorting

- > db.foo.find({}).sort({,firstname‘ :1, ,age‘: -1})

- sorting specification in JSON notation
- 1 = ascending, -1 = descending

Advanced querying

- $all
- $exists
- $mod
- $ne
- $in
- $nin
- $nor
- $or
- $size
- $type
- http://www.mongodb.org/display/DOCS/Advanced+Queries

Updating

- > db.foo.update(criteria, obj, multi, upsert)

- update() updates only one document by default (specify multi=1)
- upsert=1: if document does not exist, insert it

Updating – modifier operations

- $inc
- $set
- $unset
- $push
- $pushAll
- $addToSet
- $pop
- $pull
- $pullAll
- $rename
- $bit
- http://www.mongodb.org/display/DOCS/Updating

Updating

- > db.foo.update(criteria, obj, multi, upsert)

- update() updates only one document by default (specify multi=1)
- upsert=1: if document does not exist, insert it

Removing

- db.foo.remove({}) // remove all
- db.foo.remove({‚firstname‘ : ‚Ben‘}) // remove by key
- db.foo.remove({‚_id‘ : ObjectId(...)}) // remove by _id

- Atomic removal (locks the database)
- db.foo.remove({ age: 42, $atomic : true })
- http://www.mongodb.org/display/DOCS/Removing

Indexes

- working similar to index in relational databases
- db.foo.ensureIndex({age: 1}, {background: true})
- one query – one index

- CompoundIndexes
- db.foo.ensureIndex({age: 1, firstname:-1}
- Ordering of query parameters matters
- http://www.mongodb.org/display/DOCS/Indexes

Embedded documents

- MongoDB docs = JSON/BSON-like
- Embeeded documents similar nested dicts in Python
- db.foo.insert({firstname:‘Ben‘, data:{a:1, b:2, c:3})
- db.foo.find({,data.a‘:1})
- Dotted notation for reaching into embedde documents
- Use quotes around dotted names
- Indexes work on embeddes documents

Arrays (1/2)

- Like (nested) lists in Python
- db.foo.insert({colors: [‚green‘, ‚blue‘, ‚red‘]})
- db.foo.find({colors: ‚red‘})
- Use indexes

Arrays (2/2) – matching arrays

- db.bar.insert({users: [
  {name: ‚Hans‘, age:42},
  {name:‘Jim‘, age: 30 },
  ]})

- db.bar.find({users : {,$elemMatch‘: {age : {$gt:42}}}})

Part 3/4

- Using MongoDB from Python
  - PyMongo
  - Installing PyMongo
  - Using PyMongo

Installing and testing PyMongo

- Install pymongo
  - virtualenv –no-site-packages pymongo
  - bin/easy_install pymongo
- Start MongoDB
  - mkdir /tmp/db
  - mongod –dbpath /tmp/db
- Start Python
  - bin/python
  - > import pymongo
  - > conn = pymongo.Connection(‚localhost‘, 27127)

Part 4/4

- ? High-level PyMongo frameworks
  - Mongokit
  - Mongoengine
  - MongoAlchemy
- ? Migration SQL to MongoDB
- ? Q/A
- ? Looking at a real world project done with Pyramid and MongoDB?
- ? Let‘s talk about..

Mongokit (1/3)

- schema validation (wich use simple python type for the declaration)
- doted notation
- nested and complex schema declaration
- untyped field support
- required fields validation
- default values
- custom validators
- cross database document reference
- random query support (which returns a random document from the database)
- inheritance and polymorphisme support
- versionized document support (in beta stage)
- partial auth support (it brings a simple User model)
- operator for validation (currently : OR, NOT and IS)
- simple web framework integration
- import/export to json
- i18n support
- GridFS support
- document migration support

Mongokit (2/3)

```
class BlogPost(Document):
    structure = {
        'title': unicode,
        'body': unicode,
        'author': pymongo.objectid.ObjectId,
        'created_at': datetime.datetime,
        'tags': [unicode],
    }
    required_fields = ['title','author', 'date_creation']
```

```
blog_post = BlogPost()
blog_post['title'] = 'my blog post'
blog_post['created_at'] = datetime.datetime.utcnow()
blog_post.save()
```

Mongokit (3/3)

- Speed and performance impact
- Mongokit is always behind the most current pymongo versions
- one-man developer show
- http://namlook.github.com/mongokit/

Mongoengine (1/2)

- MongoEngine is a Document-Object Mapper (think ORM, but for document databases) for working with MongoDB from Python. It uses a simple declarative API, similar to the Django ORM.
- http://mongoengine.org/

Mongokit (2/2)

```
class BlogPost(Document):
    title = StringField(required=True)
    body = StringField()
    author = ReferenceField(User)
    created_at = DateTimeField(required=True)
    tags = ListField(StringField())
```

```
blog_post = BlogPost(title='my blog
post', created_at=datetime.datetime.utcnow())
blog_post.save()
```

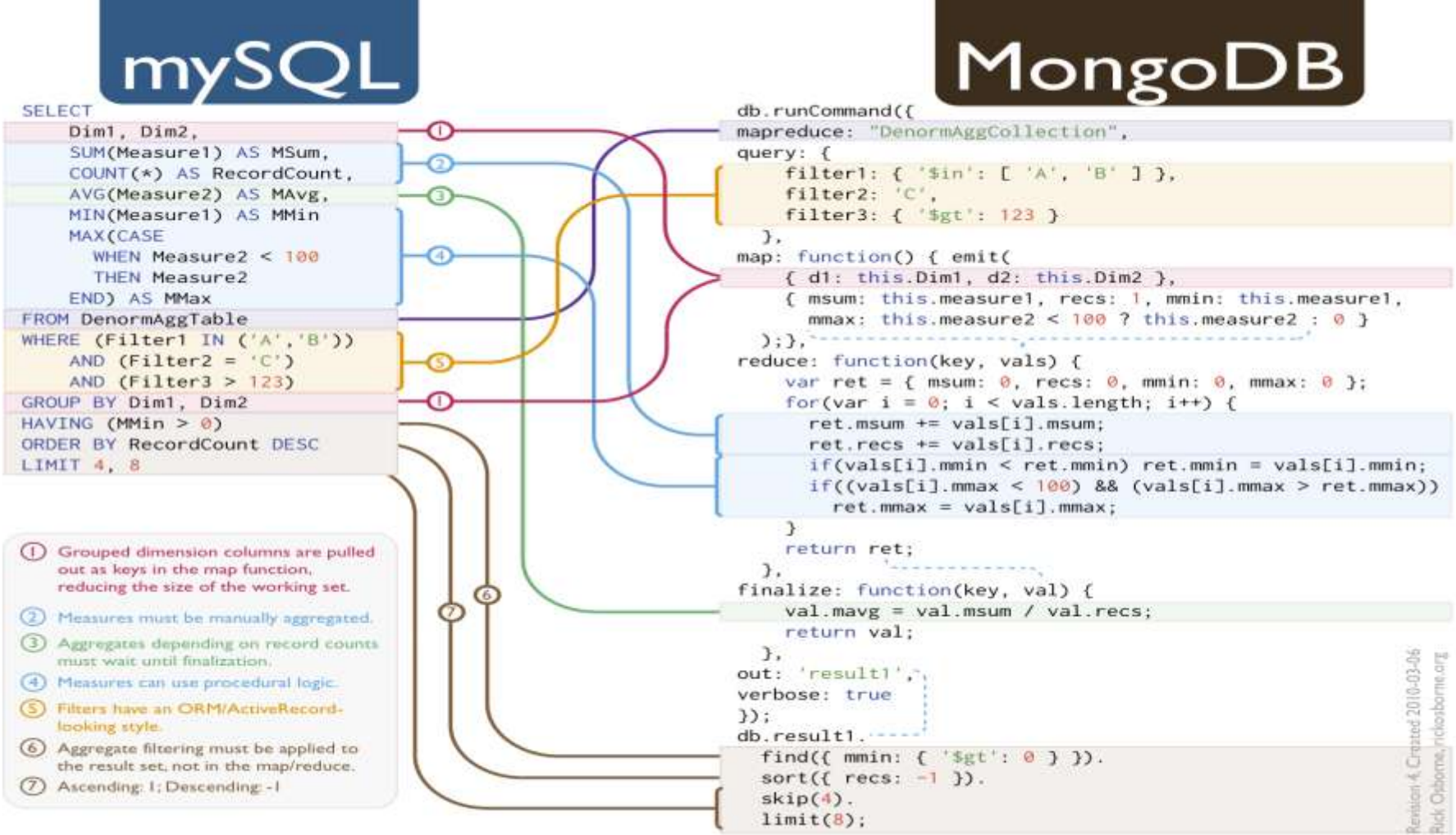
MongoAlchemy (1/2)

- MongoAlchemy is a layer on top of the Python MongoDB driver which adds client-side schema definitions, an easier to work with and programmatic query language, and a Document-Object mapper which allows python objects to be saved and loaded into the database in a type-safe way.
- An explicit goal of this project is to be able to perform as many operations as possible without having to perform a load/save cycle since doing so is both significantly slower and more likely to cause data loss.
- http://mongoalchemy.org/

MongoAlchemy(2/2)

```
from mongoalchemy.document import Document, DocumentField
from mongoalchemy.fields import *
from datetime import datetime
from pprint import pprint
class Event(Document):
    name = StringField()
    children = ListField(DocumentField('Event'))
    begin = DateTimeField()
    end = DateTimeField()

    def __init__(self, name, parent=None):
        Document.__init__(self, name=name)
        self.children = []
        if parent != None:
            parent.children.append(self)
```


mySQL
MongoDB
SELECT
Dim1, Dim2,
SUM(Measure1) AS MSum,
COUNT(*) AS RecordCount,
AVG(Measure2) AS MAvg,
MIN(Measure1) AS MMin
MAX(CASE
WHEN Measure2 < 100
THEN Measure2
END) AS MMax
FROM DenormAggTable
WHERE (Filter1 IN ('A','B'))
AND (Filter2 = 'C')
AND (Filter3 > 123)
GROUP BY Dim1, Dim2
HAVING (MMin > 0)
ORDER BY RecordCount DESC
LIMIT 4, 8
db.runCommand({
mapreduce: "DenormAggCollection",
query: {
filter1: { '$in': ['A', 'B'] },
filter2: 'C',
filter3: { '$gt': 123 }
},
map: function() { emit(
{ d1: this.Dim1, d2: this.Dim2 },
{ msum: this.measure1, recs: 1, mmin: this.measure1,
mmax: this.measure2 < 100 ? this.measure2 : 0 }
);},
reduce: function(key, vals) {
var ret = { msum: 0, recs: 0, mmin: 0, mmax: 0 };
for(var i = 0; i < vals.length; i++) {
ret.msum += vals[i].msum;
ret.recs += vals[i].recs;
if(vals[i].mmin < ret.mmin) ret.mmin = vals[i].mmin;
if((vals[i].mmax < 100) && (vals[i].mmax > ret.mmax))
ret.mmax = vals[i].mmax;
}
return ret;
},
finalize: function(key, val) {
val.mavg = val.msum / val.recs;
return val;
},
out: 'result1',
verbose: true
});
db.result1.
find({ mmin: { '$gt': 0 } }).
sort({ recs: -1 }).
skip(4).
limit(8);
1 Grouped dimension columns are pulled out as keys in the map function, reducing the size of the working set.
2 Measures must be manually aggregated.
3 Aggregates depending on record counts must wait until finalization.
4 Measures can use procedural logic.
5 Filters have an ORM/ActiveRecord-looking style.
6 Aggregate filtering must be applied to the result set, not in the map/reduce.
7 Ascending: 1; Descending: -1
Revision 4, Created 2010-03-06
Rick Osborne, rickosborne.org

The CAP theorem

- Consistency
- Availablity
- Tolerance to network Partitions

Pick two...

ACID versus Base

- Atomicity
- Consistency
- Isolation
- Durability

- Basically Available
- Soft state
- Eventually consistent