MongoDB

About me

- FIT CTU - Software Engineering
- FI MUNI - Artificial Intelligence and NLP
- GoodAI, Neuron Soundware
- Ackee - Web apps
- Smalltalk, Perl, Prolog, PHP, C, C++, C#, Objective-C, Java, Python, Javascript
- MySQL, Neo4j, PostgreSQL, MariaDB, MongoDB

Outline

- Intro to NoSQL
- Intro to MongoDB
- Mongo CLI
- CRUD
- Indexes
- Aggregation
- Replication and Sharding
- C#

Intro to NoSQL

History

- 1970s - Relational DBs
  - Expensive storage
  - Normalized data
  - Storage is abstracted from app

History

- 1980s RDBMSs commercialized
  - Client/Server model
  - SQL becomes standard
  - But the development cycle still the same

History

- 1990s Web is spreading
  - 3-tier model
  - Rise of the internet and the web
  - 1000s and 10000s of users

History

- 2000s Web 2.0
  - Social media
  - Cheaper HW
  - Much more data collected

- Developers
  - Agile, shorter dev cycles
  - Constant evolution of requirements
  - Flexibility at design time
- Relational schema
  - Hard to evolve, migrations
  - Does not reflect business logic

- Need new kind od DBs
  - Horizontal scaling
  - Run anywhere (cloud, commodity model)
  - Flexible data model
  - Faster development
  - Low upfront cost,
  - Low cost of ownership

NoSQL

NoSQL selling points

- Speed and scalability
- Fits OO well
- Agile

What is NoSQL

- Non-relational data stores and databases
- Very different products
- Different data models (non-relational)
- Most are not using SQL for queries
- No predefined schema
- Some allow flexible data structure (like MongoDB)

NoSQL categorization and usage

- Key/value
  - Memcached, Redis
- Column
  - Cassandra, HBase
- Graph
  - Neo4j
- Document
  - CouchDB, MongoDB

CAP theorem

- Consistency
- Availability
- Partition tolerance
- Pick two

NoSQL benefits

- scaling
- training
- data
- flexible
- economics

NoSQL drawback

- support
- maturity
- workforce
- analytics

MongoDB

MongoDB

- Developed in October 2007 by 10gen (now MongoDB Inc.)
- 500+ employees, 2k+ customers
- Goal
  - high-performance, fully consistent, horizontally scalable, general purpose data store
- Open source (AGPL), written in C, C++ and Javascript
- Current version 3.6

What is MongoDB

From official web

*MongoDB is a **document** database with the **scalability** and **flexibility** that you want with the **querying** and **indexing** that you need*

- Does the right thing out of the box
- Few config options
- Easy to deploy and manage

- Support all major platforms
  - Linux, Windows, Solaris, Mac OS X
  - Packages for all popular distros
- Official drivers
  - Java, .NET, PHP, Ruby, Perl, C++, C, JS, Perl, Scala, Python, Node
- Community drivers
  - Erlang, Haskell, R, Clojure, Matlab, Wolfram, F#, Smalltalk...

Terminology

SQL	MongoDB
Database	Database
Table	Collection
Index	Index
Row	Document
Column	Field
Join	Embedding & Linking & `$lookup`
Queries return rows	Queries returns cursors
ACID	BASE

MongoDB install

`mongo`

- MongoDB shell
- JavaScript

Basic commands

- help
- show dbs
- db
- use
- show collections
- db.getCollectionNames()
- db.createCollection('workshop')
- db.copyDatabase('oldname', 'newname')
- db.dropDatabase()

mongorestore

```
git clone https://github.com/vlasy/mongo.git
cd mongo
mongorestore -d workshop ./data/workshop
```

Document-oriented storage

- Documents are just JSON objects that Mongo stores in binary
- Any valid JSON can be easily imported and queried
- Rich data model
- Map to native programming languages data types
- Flexible - schemaless
- Better data locality
- JSON is readable by humans

Data stored in BSON

- BSON is binary serialization of JSON
- Efficient both in storage and scan speed
- Add extra data types to JSON (`date`, `binary`, etc.)
- Size limit 16MB

Data types

Data types

Value of a field may be a simple value, array or another document

Data Type	Number
Double	1
String	2
Object	3
Array	4
Binary data	5
ObjectId	7
Boolean	8
Date	9
Null	10

Data Type	Number
Regular Expression	11
JavaScript	13
JavaScript with scope	15
Integer (32b)	16
Timestamp	17
Long (64b)	18
Min Key	-1
Max Key	127

What is ObjectID

- `_id` field
- String wrapped with object - no autoincrement but it's ordered
- Default primary key for all collections
- Fast to generate
  - 4 bytes timestamp
  - 3 bytes machine identifier
  - 2 bytes process identifier
  - 3 bytes incrementing counter starting at random number

Useful ObjectID methods

- `ObjectId()` - generates new ObjectId
- `ObjectId("507f191e810c19729de860ea")` - generates new ObjectId with custom string
- `oid.str` - get hexadeciaml string
- `oid.getTimestamp()` - get the timestamp portion
- `oid.toString()` - string represenation
- `oid.valueOf()` - same as `oid.str`

A word about MongoDB schemas

- The data is the schema
- There are no joins
- All queries run on a single collection (exceptions later)
- ...but you just NEED to store and retrieve relations
- 3 options
  - Embedded documents
  - Manual references
  - Database references

MongoDB Relations

Embedded Documents

- Store the related data in document structure
- Either as a single field or as an array
- You get all the data in a single query
- Denormalization and possible inconsistencies
- Used for one-to-one and one-to-many relationships

Embedded Documents

```
{
    name: "John Doe",
    address: {
        city: "London",
        street: "Bond street",
        streetNumber: 14
    }
}
```

Manual references

- You save the `ObjectId` of another documents as field's or array's values
- Your application then can query for the related document
- Used for many-to-many relationships
- Extra round-trip
- Is not automatically removed on target document's deletion

Manual references

```
{
    name: "John Doe"
    booksRead: [
        ObjectId("abc..."),
        ObjectId("def..."),
        ...
    ]
}
```

Database references

- Driver support vary (C# supports)
- Useful for specifying links to multiple collections at once
- Official docs recommend using manual references until necessary

Database refereneces

- suppoose we have an `address_home` and `address_work` collections

```
{
    name: "John Doe",
    addresses: [
        {
            $ref: "address_home",
            $id: ObjectId("abc..."),
            $db: "test"
        },
        {
            $ref: "address_work",
            $id: ObjectId("def..."),
            $db:
        }
    ]
}
```

Recommendations

- Use embedded documents - you will leverage the power of document database
- Feel free to precompute values in documents (e.g. data about subdocuments)
- Don't try to make everything nice on document deletion
- Don't worry about round-trip (we will omit it later)
- When no other option, use DBRef

CRUD

Insert

- `INSERT INTO` in SQL
- We can insert single, multiple or embedded documents
- `db.collection.insert(doc)`
- `db.collection.insert(doc1, doc2)`

Find

- `db.collection.find()` - returns a cursor
- `db.collection.find().pretty()`
- `db.collection.find(query, projection)`
  1. selects documents matching criteria
  2. select specified fields according to projection param
  3. returns a cursor to selected and filtered documents

Comparison operators

Operator	Syntax
Equal	`{ <Key>: <Val> }`
Not Equal	`{ <Key>: { $ne: <Val>} }`
Less Than	`{ <Key>: { $lt: <Val> } }`
Greater Than	`{ <Key>: { $gt: <Val> } }`
Less Than Equal	`{ <Key>: { $lte: <Val> } }`
Greater Than Equal	`{ <Key>: { $gte: <Val> } }`

Logical operators

- `$and` - implicit behavior
- `db.collection.find($and: [{key1:value1, key2:value2}] )`
- `$or`
- `db.collection.find({ $or:[{key1:value1}, {key2:value2}] })`
- `$in` and `$nin` - multiple values for single key
- `db.collection.find({ key1:{$in:[value1, value2]} })`

Projection syntax

- `{field1: <boolean>, field2: <boolean>}`
- `_id` is included until explicitly excluded
- projection spec must be just `true`s or `false`s

More stuff you can `find`

- query for array elements `db.posts.find({tags: "sql"})`
- query for embedded documents `db.posts.find({"comments.commentBy": "John"})`
- exact match for embedded document `db.posts.find({comments:{commentBy: 'John', text:'John' }})`

Expressive queries

- Find all contacts with at least one home phone or hired after 2014-02-02
-
```
SELECT A.id, A.name, A.registerDate, B.type, B.number
FROM authors A
LEFT OUTER JOIN phones B ON (B.author_id = A.id)
WHERE b.type = 'home' OR A.registerDate > DATE '2018-04-06'
```
-
```
db.authors.find({'$or':
  [
      {'phones.type':'home'},
      {'registerDate':
          {'$gt': new ISODate('2014-02-02')}
      }
  ]
});
```

Update

- `db.collection.update({ query, update, { upsert: False, multi: False }})`
- query - same as `find`
- update

```
{
 <operator1>: { <field1>: <value1>, ... },
 <operator2>: { <field2>: <value2>, ... },
 ...
}
```

Update operators

- `$currentDate`
- `$inc`
- `$min` and `$max`
- `$mul`
- `$rename`
- `$set`
- `$unset`
- `$pop`
- `$pull`
- `$push`
- and more...

Update specific fields

- update contains only update operators like `$set`, `$inc`, `$mul`
- to update specific field of embedded document or array, use dot notation

Update return value

- `nMatched` - nr of documents matching query
- `nUpserted` - nr of upserted documents
- `nUpdated` - nr of updated documents

Replace entire documents

- If we pass an `update` document that contains only field and value pairs, in other words we pass an `update` document without update operator, then the `update` document completely replaces the original document except for the `_id` field.

Upsert and Multi

- If the `upsert` option is set to `true` then it creates a new document when no document matches the query criteria otherwise it won't insert a new document.
- If the `multi` option is set to `true`, it updates all matching documents (not just one)

Update embedded document array

Remove

- `db.collection.remove(query, {justOne:false})`
- `db.collection.remove()` - remove all documents
- `db.collection.drop()` - `remove()` + deletes indexes and structure

Collection methods

- `insert`
- `remove`
- `update`
- `find`
- `count`
- `distinct`
- `findOne`
- `save`
- and more...

Note:

CODE

```
>db.posts.count(query)
-- returns number of matched docs
-- equivalent to db.collection.find(query).count().

>db.posts.count()
-- count all docs

>db.posts.distinct(field, query)
-- find distinct values for given field
>db.posts.distinct("tags")
>db.posts.distinct("tags",{likes:{$gt:5}})

>db.posts.findOne(query, projection)
-- same as find but returns only first doc
>db.posts.findOne()
>db.posts.findOne({"likes":{$gt:5}})
```

Find's cursor

- can be saved to a variable
- by default mongo prints first 20 results
- can be changed by setting `DBQuery.shellBatchSize`
- use `it` command to iterate the cursor

Cursor methods - basic

- `count()`
- `pretty()`
- `limit(n)`
- `skip(n)`
- `size()`

Cursor methods - iterating

- `it`
- `hasNext()`
- `next()`
- `forEach()`
- `toArray()`

Cursor methods - sorting

- `sort({field:order})`
- `sort({field:order, field2:order2})`
- `sort().limit()`
- `limit().sort()`

Cursor methods - execution plan

- `explain(verbosity)`
- verbosity
  - `"queryPlanner"` - default
  - `"executionStats"`
  - `"allPlansExecution"`

findAndModify

```
db.runCommand(
   {
     findAndModify: "posts",
     query: { title: "Square", likes: { $gt: 200 } },
     sort: { "author.views": 1 },
     update: { $inc: { likes: 1 } }
   }
)
```

Capped collections

- `db.createCollection("logs", {capped:true,size:654123,max:10})`
- fixed size
- limited operations - cannot remove, updates restricted
- auto-LRU age-out
- super fast
- ideal for logging and caching

MongoDB - day 2

Indexes

- We can create an index for single field, multiple fields, embedded field, embedded document
- Indexes are stored in RAM
- Collection can have up to 64 indexes
- Index name cannot be longer than 125 chars

Basic indexes methods

- db.collection.getIndexes()
- db.collection.dropIndexes()
- db.collection.dropIndex({field})
- db.collection.ensureIndex({field:sort_order})
- db.collection.reIndex()
- db.collection.totalIndexSize()

Index types

- Default Index
- Single Field Index
- Compound Index
- MultiKey Index
- Text Index
- Geospatial Index
- Hashed Index

Default Index

Each collection contains builtin ascending index on `_id` field

Single Field Index

```
db.Collection.createIndex({field:sort_order})
```

Index on embedded field

```
db.post.createIndex({"comment.commentBy":1})
```

Index on embedded document

```
db.post.createIndex({"comment":1})
```

Compound index - index on multiple fields

- `db.post.createIndex({"Comment":1,Title:1})`
  - {Comment:1, Title:1} OK
  - {Title:1, Comment:1} NOT
- `db.post.createIndex({"Comment":1,Title:-1})`
  - {Comment:1, Title:-1} OK
  - {Comment:-1, Title:1} OK
  - {Comment:-1, Title:-1} NOT
  - {Title:1, Comment:1} NOT

Index prefixes

- `db.post.createIndex({Comment:1,Title:-1,Pos`
  - Comment and Title field
  - Comment and Title and PostBY field
  - Comment and PostBy field
- Will not be used for
  - Title field
  - PostBy field
  - Title and PostBy field

MultiKey indexes

- Index on a field containing array creates index key for all array elements
- You cannot create compound multikey index if more than one of the fields are arrays

Text indexes

- `db.collection.createIndex({comments:"text"})`
- can contain any fields that contain string or array of strings
- collection can have at most one text index
- but it can have multiple fields

Wildcard text index

- Sometimes it is hard to predict which fields will contain strin
  => you can create text index for all the fields containing strin
  or text data
- `db.post.createIndex( { "$**": "text" } )`
- Wildcard index can be compound
- `db.post.createIndex({"Title":1,"$**":"text"`

Geospatial index

- you can store GeoJSON data in MongoDB

```
location: {
    type: "Point",
    coordinates: [-73.856077, 40.848447]
}
```

- Point, Line, Polygon, etc.
- `2dsphere` or `2d`
- `$geoIntersects`, `$geoWithin`, `$near`, `$nearSphere`

Hashed index

- `db.collection.createIndex({field:"hashed"})`
- hash of all indexed fields is computed
- cannot be unique (discussed later)
- cannot be used for range queries
- cannot be used in compound indexes
- does not support arrays
- does not support range queries
- but you can create normal index and hashed index on the same field
- useful for sharding (discussed later)

Index properties

- Unique index
- TTL index
- Sparse index

Unique index

```
db.collection.createIndex({field:sort_order},
{unique:true})
```

- restricts document field uniqueness (default is False)

TTL Index - Time to live

- `db.collection.createIndex({field:sort_order { expireAfterSeconds:time})`
- only applicable on date field or array containing fields
- documents are automatically deleted after some time
- useful for machine-generated data, logs, sessions, temporary data

Partial index

- `db.collection.createIndex({field:sort_order`
  `{ partialFilterExpression: filter })`
- filter can contain equality match, `$exists`, `$gt`, `$gte`, `$lt`
  `$lte`
- index is only on documents matching query
- why would you want that?
  - less space (indexes must fit into RAM)
  - reduced performace cost (for creation and maintenance)

Conclusion

- know the performance hit
- know when they are used
- review indexes
- it is common to have multiple indexes on a collection
- it is common to have multiple indexes on the same field set in different orders
- => so that you have all your app queries "covered"

Covered query

- can be satisfied entirely with index
- all the fields in query are part of index
- all the fields returned are part of the same index

Index performance

Aggregation Queries

- Performs operation on a group of documents and return result
- Types of aggregate functions
  - Single purpose aggregate methods and commands
  - Pipeline
  - Map-Reduce

Single-purpose aggregate methods and commands

- Count
  - `db.collection.count()` = cursor method
  - `db.runCommand({count:'collection',query:{likes:{$gt:200}}})` = command

Single-purpose aggregate methods and commands

- Distinct
  - `db.collection.distinct("Salary")` = cursor method
  - `db.runCommand({distinct:'collection', key:"Salary"})` = command

Aggregate pipeline

- `db.collection.aggregate({Pipeline expression})`
- Like classic pipeline = output from one command is input to next one
- Pipeline expression consist of "stages"

Common operators in aggregate pipeline

- `$sum`
- `$avg`
- `$first`
- `$dateToString`
- `$arrayElemAt`

Common stages in aggregate pipeline

`$project`	`$count`
`$match`	`$lookup`
`$group`	`$sample`
`$sort`	`$unwind`
`$skip`	`$out`
`$limit`	and more

$project

```
{ $project: { <specification(s)> } }
```

- `<field>: <1 or true>` - include
- `_id: <0 or false>` - exclude
- `<field>: <expression>` - adds a field or set its value

$match

```
{ $match: { <query> } }
```

- best to place early in pipeline

$group

```
{ $group: { _id: <exp>, <field1>: {
<accumulator1> : <exp1> }, ... } }
```

- `_id` is mandatory but can be null
- `$sum`, `$avg`, `$first`, `$last`, `$min`, `$max`, `$push`

$sort

```
{ $sort: { <field1>: <sort order>,
  <field2>: <sort order> ... } }
```

$skip

```
{ $skip: <positive integer> }
```

$limit

```
{ $limit: <positive integer> }
```

$count

```
{ $count: <string> }
```

$lookup

```
{
   $lookup:
     {
       from: <collection to join>,
       localField: <field from the input documents>,
       foreignField: <field from the documents of the "from" collection
       as: <output field>
     }
}
```

- performs left outer join

$sample

```
{ $sample: { size: <positive integer> } }
```

$unwind

```
{ $unwind: <field path> }
```

- outputs document for each item in array

Examples

```
git clone https://github.com/vlasy/mongo.git
cd mongo
mongorestore -d js ./data/js
```

Map + Reduce

- `Map` - processes each document and retrieves one or more objects for each input document
- `Reduce` - combines results of `Map` operations
- optional `finalize` function - result

Profiling

- mongod log - logs slow queries
- profiler - `db.setProfilingLevel`
  - 0 - default - off
  - 1 - log slow queries
  - 2 - log all queries

High Availability

Replication and Sharding

Replication

- Replication in mongo is done by replica set
- A replica set is a group of MongoDB processes (3 at minimum) that maintain the same data.
- All writes go to `primary` and to `oplog`
- Asynchronous replication
- Reads go to `primary` or `secondary`
- Automatic failover
- Maintenance with no downtime

Automatic failover

- If primary does not communicate with `secondary` instance for more than 10 seconds, replica set attempts to pick one of `secondary` instances as a new `primary`
- First `secondary` that gets majority of votes is the new `primary`

Sharding

- Vertical scaling / scaling up
  - Buy better/larger/faster HW
  - Increase load on single machine, increas chance of system fail
- Horizontal scaling / scaling out / sharding
  - Buy more of the same cheap HW
- Increase capacity with no downtime

Sharding in MongoDB

- done using a shared cluster
  - shards - data
  - config servers - metadata
  - query routers - routes queries from clients to shards using config server

Data partitioning

- Data are distribtued on collection level - distribution is done by a shard key
- shard key - determines the distribution of collection on shards
  - may be an indexed field or indexed compound field present in all documents
- Sharding models
  - Range based partitioning
  - Hash based partitioning

Range-based partitioning

- non-overlapping ranges

Hash-based sharding

- Mongo computes hash of given field
- provides more random data distribution
- avoid "hot" shard

Balanced data distribution

- MongoDB always tries to balance the data distribution, for this MongoDB uses the following two approaches.
  - Splitting
  - Balancing

Recap

- Sharding distributes the data over multiple shards so it reduces the number of operations for each shard.
- Removes the dependency from a single server.
- Protects against system failover.
- Increases capacity and throughput.

Monitoring

- MongoDB has three methods for monitoring
  - Utilities
  - Database Commands
  - Monitoring Tools

Utilities

- `mongostat`
- `mongotop`
- Web interface

DB Commands for monitoring

- `db.serverStatus()`
- `db.stats()`
- `db.collection.stats()`

Backup and Restore

- Backup options
  - Backup by copying underlying data files.
  - Backup using `mongodump` tool.

mongodump

- `mongodump` - backup of `mongod` instance on `localhost:27017` to `dump` folder
- `mongodump --host hostname:port`
- `mongodump --out directory_path`
- `mongodump --db cookbook`
- `mongodump --db cookbook --collection recipes`

mongorestore

- `mongorestore` - restores all data from `dump` directory to `localhost:27017`
- `mongorestore --host hostname:port`
- `mongorestore --drop` - drop all collections already present in target
- `mongorestore --db cookbook dump/cookbook`
- `mongorestore --db coolbook --collection recipes dump/cookbook/recipes.bson`

Other data manipulation commands

- `mongoexport`
- `mongoimport`

What's new in 3.6

- field names can contain dots and dollars, however only nested field names may start with a dollar
- still avoid using `.` and $ in field names, since querying on such fields is not yet functional
- `$lookup` with more join options

What will be in 4.0

Multi-document transactions!

Thank you for attention

michal.vlasak@ackee.cz