mongoDB

{name: "mongo", type:"DB"}

document-oriented database

Janani Chathapuram
Alejandra Rincon
Padmini Vankayla

Outline

- Introduction to MongoDB
- Programming in MongoDB
- Use cases
- Demo

SQL vs MongoDB

SQL	**MongoDB**
database	database
table	collection
row	document

MongoDB uses JSON style documents but data is stored internally in BSON format

Features of MongoDB

- Scalability
- Document based queries
- Map-Reduce
- GridFS
- Indexing

Data Modeling

- Data has flexible schema
- Collections do not enforce document structure.
- Data modeling decisions involve determining how to structure the documents to model the data effectively.
  - Embedding: To de-normalize data, store two related pieces of data in a single document.
  - Referencing: To normalize data, store references between two documents to indicate a relationship between the data represented in each document.

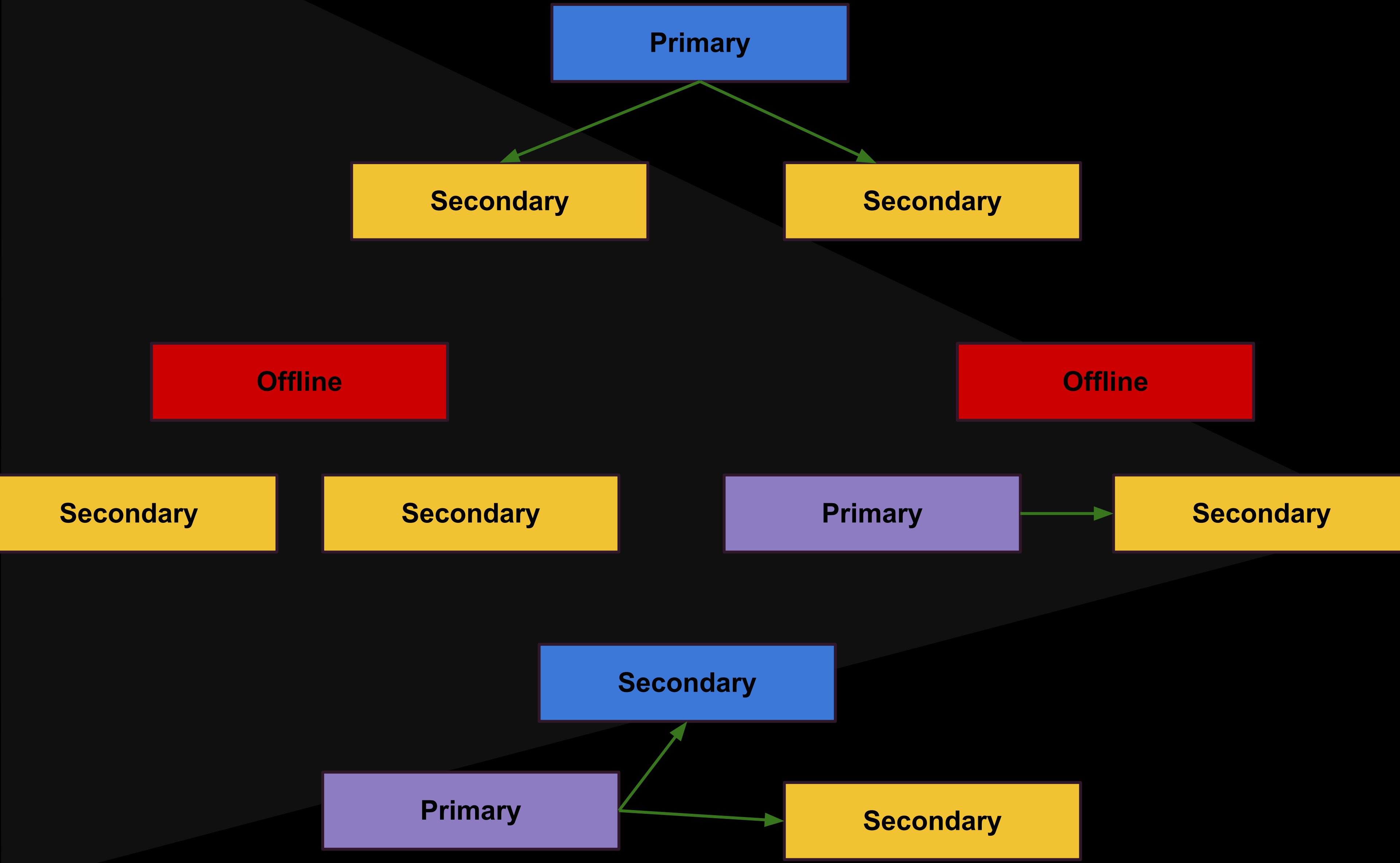
GridFS

- Specification for chunking files.
- creates 2 collections to store files: chunk and file.
- File metadata is tightly coupled with files.
- Chunked for performing range operations.

Replication

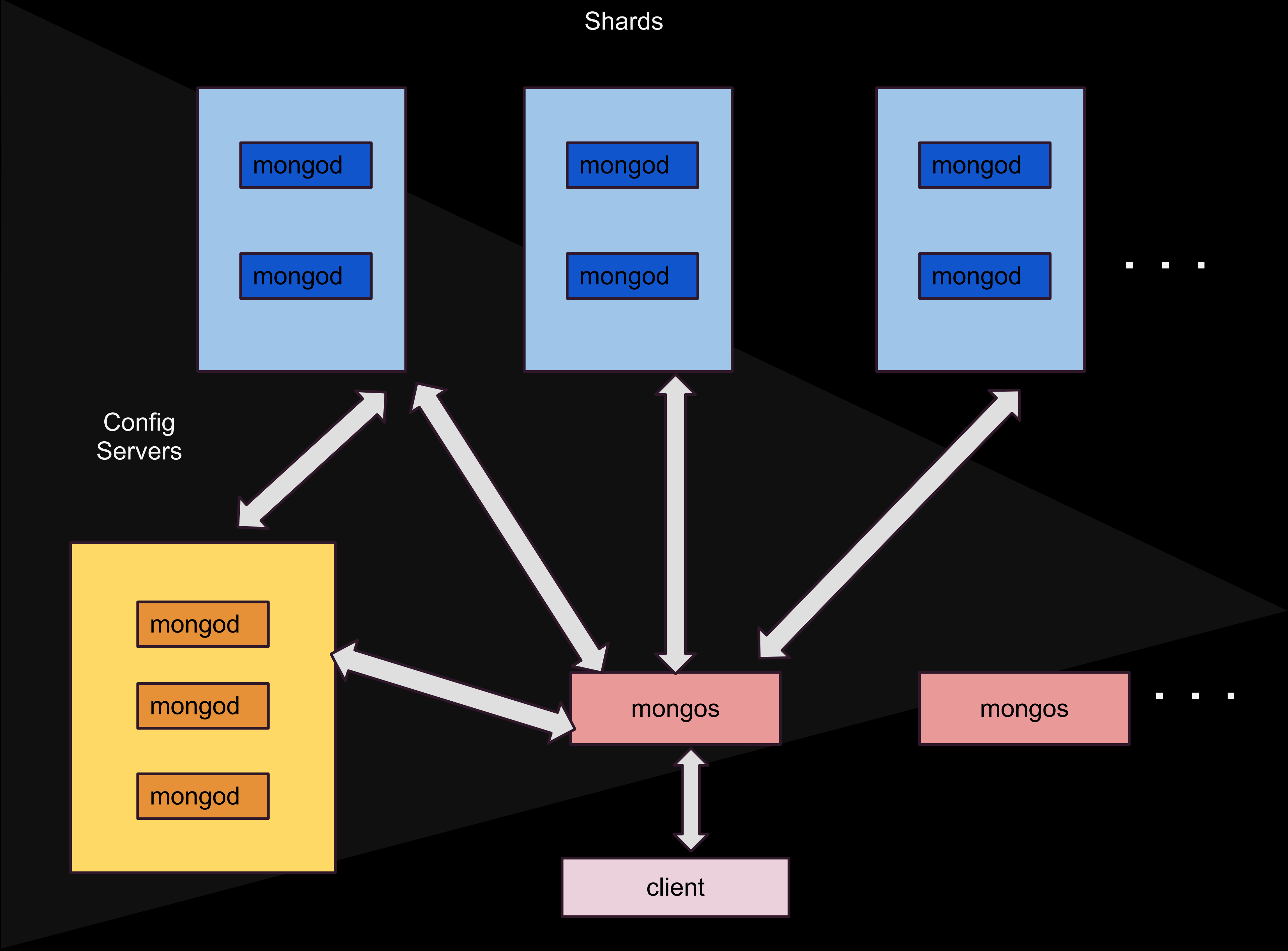
- Replication ensures redundancy, failover and backup
- Replica Set contains mongod instances that replicate one another to provide automated failover
- One of mongod is designated as primary and others are secondary
- Writes are directed to primary
- Secondary members replicate from primary asynchronously.
- This may sometimes lead to consistency issues.

Replica Sets



Sharding

- Sharding is used for scaling.
- Balances data and load across machines
- Provides additional write capacity
- Increases potential amount of data in working set.


Shards
mongod
mongod
mongod
mongod
mongod
mongod
. . .
Config Servers
mongod
mongod
mongod
mongos
mongos
. . .
client

AGGREGATION

- Aggregation Framework provides a means to calculate aggregated values.
- Achieved by - Pipelines and Expressions
- Complex aggregation tasks are handled by Map-Reduce.

Indexes

- Indexing allows documents to be located quickly based on the values stored in certain specified fields.
- It uses B-tree data structure to Index
- Indexes are defined on per collection
- A collection can have at most 64 indexes
- Every query uses only one index
- An index “covers” a query if:
  - all the fields in the query are part of that index
  - all the fields returned in the documents that match the query are in the same index.

ACID in MongoDB

- Atomicity: Ensured at single document level.
- Consistency: Eventually consistent reads, from a replica set are only possible with a write concern that permits reads from secondary members.
- Isolation: The multi update/write to multiple documents is not atomic and may interleave with other write operations. The isolation operator isolates the update/write operation and blocks other write operations during update.

ACID in MongoDB contd..

- Durability:
  - MongoDB uses write ahead logging to an on-disk journal to guarantee write operation, durability and to provide crash resiliency.
  - Before applying a change to the data files, MongoDB writes the change operation to the journal.
  - there is an up-to 100 millisecond window between journal commits where the write operation is not fully durable.
  - Requiring journaled write concern in a replica set only requires a journal commit of the write operation to the primary of the set regardless of the level of replica acknowledged write concern.

Programing in MongoDB

- MongoDB operations:
    - Insert
    - Delete
    - Update
- Operations in Java
- Operations in PHP

Insert

Method to insert a document into MongoDB collection:
Syntax: **db.collection.insert(<document>)**

Example:

If the collection ‘**bios**’ does not exist, then the insert operation will create this collection:

```
db.bios.insert(
  {     _id: 1,
        name: { first: 'John', last: 'Backus' },
        birth: new Date('Dec 03, 1924'),
        death: new Date('Mar 17, 2007'),
        contribs: [ 'Fortran', 'ALGOL', 'Backus-Naur Form', 'FP' ],
        awards: [
                {award: 'W.W. McDowell Award',
                year: 1967,
                by: 'IEEE Computer Society'
                },
                {award: 'Draper Prize',
                year: 1993,
                by: 'National Academy of Engineering'
                }]
})
```

Insert

To confirm the insert query the bios collection:

db.bios.find()

It will return the content of the collection.

If the document does not contain an id, we can find it doing

db.bios.find({ name: { first: 'John', last: 'McCarthy' } })

The returned value document contains an _id field with the generated ObjectId:

```
{"_id" : ObjectId("50a1880488d113a4ae94a94a"),
 "name" : { "first" : "John", "last" : "McCarthy" },
 "birth" : ISODate("1927-09-04T04:00:00Z"),
 "death" : ISODate("2011-12-24T05:00:00Z"),
 "contribs" : [ "Lisp", "Artificial Intelligence", "ALGOL" ],
 "awards" : [     { "award" : "Turing Award",
                    "year" : 1971,
                    "by" : "ACM"
                    },
                    {"award" : "National Medal of Science",
                    "year" : 1990,
                    "by" : "National Science Foundation"
               }]
}
```

Insert

save() method is identical to an update operation with the upsert flag
The save() method performs an insert if the document to save does not contain the _id field.

```
db.bios.save(
 {     name: { first: 'Guido', last: 'van Rossum'},
       birth: new Date('Jan 31, 1956'),
       contribs: [ 'Python' ],
       awards: [
              {
              award: 'Award for the Advancement of Free Software',
               year: 2001,
              by: 'Free Software Foundation'
              },{
              award: 'NLUUG Award',
              year: 2003,
              by: 'NLUUG'
              }]
})
```

Delete

remove() deletes documents from the collection

Syntax: **db.collection.remove(<query>, <justOne>)**

In SQL: The remove() method is analogous to the DELETE statement, and:
<**query**> corresponds to the WHERE statement, and
<**justOne**> akes a Boolean and has the same effect as LIMIT 1.

If you do not specify a query, remove() removes all documents from a collection, but does not remove the indexes.

drop() to delete an entire collection form the DB

Delete

Deletes all documents from the bios collection where the subdocument *name contains a field first whose value starts with G*:

db.bios.remove({ 'name.first' : /^G/ })

The following operation *deletes a single document from the bios collection where the turing field equals true*:

db.bios.remove({ turing: true }, 1)

Delete all documents from the bios collection:

db.bios.remove()

Update

Syntax: **db.collection.update(<query>, <update>, <options>)**

In SQL the update() method corresponds to the UPDATE operation in SQL:

<**query**> argument corresponds to the WHERE statement, and <**update**> corresponds to the SET ... statement.

update() correspond to the SQL UPDATE statement with the LIMIT 1.

Update

Modify

Use **$set** to update a value of a field.

The following operation *queries the bios collection for the first document that has an _id field equal to 1* and sets the value of the field middle, in the subdocument name, to Warner:

```
db.bios.update(
  { _id: 1 },
  {
     $set: { 'name.middle': 'Warner' },
  }
)
```

Update

Add new field

If the <update> argument contains fields not currently in the document, the update() method adds the new fields to the document.

The following operation queries the bios collection for the first document that has an _id field equal to 3 and adds to that document a new mbranch field and a new aka field in the subdocument name:

```
db.bios.update(
  { _id: 3 },
  { $set: {
             mbranch: 'Navy',
             'name.aka': 'Amazing Grace'
      }
  }
)
```

Update

Remove field

If the <update> argument contains **$unset** operator, the update() method removes the field from the document.

The following operation *queries the bios collection for the first document that has an _id field equal to 3* and removes the birth field from the document:

```
db.bios.update(
  { _id: 3 },
  { $unset: { birth: 1 } }
)
```

Update

Upsert flag Remove

Upsert flag modifies the behavior of update() from updating existing documents, to inserting data.
These update operations have the use <query> argument to determine the write operation:
If the **query matches** an existing document(s), the operation is an **update**.
If the **query matches no document** in the collection, the operation is an **insert.**

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

Update

If query does not include an _id field, the operation adds the _id field and generates a unique ObjectId for its value.

```
db.bios.update(
  { name: { first: 'Dennis', last: 'Ritchie'} },
  {
        name: { first: 'Dennis', last: 'Ritchie'},
        birth: new Date('Sep 09, 1941'),
        death: new Date('Oct 12, 2011'),
        contribs: [ 'UNIX', 'C' ],
        awards: [
                {       award: 'Turing Award',
                year: 1983,
                by: 'ACM'
                },
                {       award: 'Japan Prize',
                year: 2011,
                by: 'The Japan Prize Foundation'
                }]
  },
  { upsert: true }
)
```

Update

The following operation inserts a new document into the bios collection:

```
db.bios.update(
  {      _id: 7,
         name: { first: 'Ken', last: 'Thompson' }
  },
  {      $set: {
         birth: new Date('Feb 04, 1943'),
         contribs: [ 'UNIX', 'C', 'B', 'UTF-8' ],
         awards: [
                {award: 'Turing Award',
                       year: 1983,
                       by: 'ACM'
                },
                {award: 'Japan Prize',
                       year: 2011,
                       by: 'The Japan Prize Foundation'
                }]
         }
  },
  { upsert: true }
)
```

Operations in java

MongoDB java example:

```
import com.mongodb.*;
import java.util.ArrayList;
public class MongoExample {
   // in the URI. Ex: "mongodb://username:password@localhost:27017/mongoquest"
   private static String uriString = "mongodb://localhost:27017/mongoquest";
 public static void main(String[] args){
        // We opt to use the MongoURI class to access MongoDB connection methods.
        MongoURI uri = new MongoURI(uriString);
        DB database = null;
        DBCollection locations = null;
        try {
           // The MongoURI class can connect and return a database given the URI above.
           database = uri.connectDB();
           // If running in auth mode and have provided user info in your URI, you can use this line.
           // database.authenticate(uri.getUsername(), uri.getPassword());
        } catch(UnknownHostException uhe) { System.out.println("UnknownHostException: " + uhe);}
         catch(MongoException me) { System.out.println("MongoException: " + me);}
```

Operations in Java

MongoDB java example (cont):

```
if (database != null) {
  locations = database.getCollection("locations"); // Retrieve the collection we'll be working with.
            // In this example, we build BasicDBObjects describing two locations, Arganis and Kent.
          // {'name': 'Arganis',
          //  'weather': 'temperate',
          //  'terrain': ['forests', 'plains'],
          //  'benefits': ['lodging', 'trade', 'justice'],
          //  'dangers': ['bandits', 'rebels', 'goblins', 'ghosts']}
            BasicDBObject arganis = new BasicDBObject();
            ArrayList<String> arganisTerrain = new ArrayList<String>();
            ArrayList<String> arganisBenefits = new ArrayList<String>();
            ArrayList<String> arganisDangers = new ArrayList<String>();
            arganis.put("name", "Arganis"); arganis.put("weather", "temperate");
            arganisTerrain.add("forests"); arganisTerrain.add("plains");
            arganis.put("terrain", arganisTerrain);
            arganisBenefits.add("lodging"); arganisBenefits.add("trade"); arganisBenefits.add("justice");
           arganis.put("benefits", arganisBenefits);
            arganisDangers.add("bandits"); arganisDangers.add("rebels"); arganisDangers.add("ghosts");
            arganis.put("dangers", arganisDangers);
```

Operations in Java

```
    // {'name': 'Kent',
    //  'weather': 'temperate',
    //  'terrain': ['hills', 'plains'],
    //  'benefits': ['lodging', 'trade']
    //  'dangers': ['bandits', 'rebels', 'famine', 'goblins']}
  BasicDBObject kent = new BasicDBObject();
  ArrayList<String> kentTerrain = new ArrayList<String>();
  ArrayList<String> kentBenefits = new ArrayList<String>();
  ArrayList<String> kentDangers = new ArrayList<String>();
  kent.put("name", "Kent"); kent.put("weather", "temperate");
  kentTerrain.add("hills");kentTerrain.add("plains");
  kent.put("terrain", kentTerrain);
  kentBenefits.add("lodging"); kentBenefits.add("trade");
  kent.put("benefits", kentBenefits);
  kentDangers.add("bandits"); kentDangers.add("rebels"); kentDangers.add("famine");kentDangers.add("goblins");
  kent.put("dangers", kentDangers);
// Pass the BasicDBObjects to the .insert() function in our collection object.
  locations.insert(arganis);
  locations.insert(kent);
```

Operations in Java

```
locations.update(new BasicDBObject("name", "Arganis"),
                        new BasicDBObject("$set",
                                    new BasicDBObject("leader",
                                                "King Argan III")));
        // Query for locations with forests.
        System.out.println("Total number of locations " + locations.count() );
    // Assign the results of a find operation to a DBCursor object.
        // Cursors can be iterated through using familiar next/hasNext logic.
        DBCursor results = locations.find(new BasicDBObject("terrain", "forests"));
        while(results.hasNext()){
            DBObject result = results.next();
             System.out.println((String) result.get("name") + " has forests.");
             System.out.println("Leader (optional) " +  (String) result.get("leader") );
        }
        // Clean up after ourselves.
        locations.drop();
    }}}
        // end
```

Operations in PHP

```php
<?php
$m = new Mongo('mongodb://localhost:27017');
$db = $m->mongoquest;
/*First we get our desired collection.*/
$collection = $db->Spells;
/*$collection = $db->createCollection("Collection_Name", true, 10*1024, 10);*/
/* We insert by first creating an array, and passing that array to the collection's insert function.
We use arrays to construct JSON-like objects.*/
$obj = array('name' => 'Poke', 'level' => 1);
$collection->insert($obj);
$obj2 = array('name' => 'Zap', 'level' => 1);
$collection->insert($obj2);
$obj3 = array('name' => 'Blast', 'level' => 2);
$collection->insert($obj3);
/* At level 1, we only know level 1 spells.*/
echo 'Level 1 spell list:<br/>';
$query = array('level' => 1);
$cursor = $collection->find($query);
foreach($cursor as $obj) {
   echo 'Spell name: ' .$obj['name'] .'<br/>';
}
```

Operations in PHP

```php
/*We can use array syntax in-line to create JSON-like queries.*/
$collection->update(array('name' => 'Poke'), array('$set' => array('flavor' => 'Snick snick!')));
$collection->update(array('name' => 'Zap'), array('$set' => array('flavor' => 'Bzazt!')));
$collection->update(array('name' => 'Blast'), array('$set' => array('flavor' => 'FWOOM!')));

/*query again with flavor!*/
echo '<br/>Level 1 spell list, with flavor:<br/>';
$query2 = array('level' => 1);
$cursor2 = $collection->find($query2);
foreach($cursor2 as $obj2) {
    echo 'Spell name: ' .$obj2['name'];
    echo ' Flavortext: ' .$obj2['flavor'] .'<br/>';
}

/*clean up after ourselves.*/
$collection->drop();

?>
```

Uses Cases

Content Management

Data Warehousing

Social Media Storage

Inventory Management

Products Catalogs

Popular Applications

Craiglist

Firebase

SAP

MTV

Source Forge

EA Sports

Disney

Application

An archive for a few massive forum

Only browse functionality (no insert or update)

Statistics:

- Number of forums: 7
- Average size of each forum 280 MB
- Average of threads in each forum: 25716
- Average number of message in each thread: 110

Schema Design

Collections:
Forums
Threads
Users

Users:
_id
title

Forums:
_id
title
website
description
startdate
endDate
numOfMembers
numOfThreads
numOfMsgs

Threads:
_id
forum
title
numOfMsgs
tags
[comments]
- _id
- author
- date
- post

Rationale for choosing MongoDB

Huge data volumes

Variability in data structure

No complex transactions

In Memory caching for pagination

Demo

QA