MEC ITALY

MONGODB PLAYGROUND

ALFIO FERRARA, STEFANO MONTANELLI

Department of Computer Science

Via Comelico 39, 20135 Milano

{alfio.ferrara,stefano.montanelli}@unimi.it
http://islab.di.unimi.it/

SOFTWARE TOOLS

- **MongoDB server and client (v. 3.2.*)**
  https://www.mongodb.com/
- **Python 2.7**
  https://www.python.org/download/releases/2.7/
- **Java SE 1.7 (or higher)**
  https://www.oracle.com/java/index.html
- **Python JSON libs**
  https://docs.python.org/2/library/json.html
- **Java JSON libs**
  https://github.com/stleary/JSON-java
  https://github.com/google/gson
- **Python MongoDB API (pymongo)**
  https://api.mongodb.com/python/current/
- **Java MongoDB API (java driver)**
  http://mongodb.github.io/mongo-java-driver/3.2/

CHECK THE ENVIRONMENT

MONGODB SERVER AND CLIENT

- **Laptop**: start server on localhost and then type **mongo** on your favourite shell console

PYTHON

```
import pymongo
```

JAVA

```
import org.json.*
```

JSON IN A NUTSHELL

JSON means **J**ava**S**cript **O**bject **N**otation

Basic notions about syntax

- Data is organized in **key**/**value** pairs
- Data is separated by commas, objects are denoted by curly brackets, arrays are denoted by square brackets

```
{"game": "GTAV", "platform": "PS4", "genre": ["open-world", "shooting"], "price": 69.99}
```

JSON VALUES (I)

A **value** corresponding to a key can be:

A number (integer or floating point)

```
{"price": 69.99, "version": 5}
```

A string (in double quotes)

```
{"game": "GTAV"}
```

A Boolean (true or false)

```
{"game": "GTAV", "in_collection": true}
```

JSON VALUES (II)

A **value** corresponding to a key can be:

An array (in square brackets)

```
{"genre": ["open-world", "shooting"]}
```

An object (in curly braces)

```
{"game": "GTAV",
 "owner": {
            "first_name": "gino",
            "last_name": "rossi"
        }}
```

The null value

```
{"game": "GTAV", "year": null}
```

NESTED OBJECTS

JSON objects can be nested at any level of depth

```
{"game_shop": {
        "name": "Gino's shop",
        "address": {
            "street": "via dei giochi",
            "number": 16,
            "city": "Milan",
            "country": "Italy"
        },
        "games": [
                {"game": "GTAV", "platform": "PS4",
                "genre": ["open-world", "shooting"], "price": 69.99,
                "reviews": [
                    {"evaluation": 5, "date": ISODate("2015-06-29T00:00:00Z"),
                        "reviewer": {
                            "first_name": "Mario", "last_name": "Bianchi"
                        }
                    },
                    {"evaluation": 4, "date": ISODate("2016-07-16T00:00:00Z"),
                        "reviewer": {
                            "first_name": "Maria", "last_name": "Verdi"
                        }
```

MISSING INFORMATION

Game

ID	name	platform	price
1	GTAV	PS4	69.99
2	Fallout4	PS4	NULL
3	Dying Light	PS4	NULL

```
{"_id": 1, "name": "GTAV", "platform": "PS4", "price": 69.99}
{"_id": 2, "name": "Fallout4", "platform": "PS4", "price": null}
{"_id": 3, "name": "Dying Light", "platform": "PS4"}

> db.games_example.find({"price": {"$gte": 40}})
{ "_id" : 1, "name" : "GTAV", "platform" : "PS4", "price" : 69.99 }

> db.games_example.find({"price": null})
{ "_id" : 2, "name" : "Fallout4", "platform" : "PS4", "price" : null }
{ "_id" : 3, "name" : "Dying Light", "platform" : "PS4" }

> db.games_example.find({"price": {"$exists": true}})
{ "_id" : 1, "name" : "GTAV", "platform" : "PS4", "price" : 69.99 }
{ "_id" : 2, "name" : "Fallout4", "platform" : "PS4", "price" : null }
```

READ JSON DATA

USE LIBS!

PYTHON

```python
import json
in_file = 'data/games.json'
with open(in_file, 'rU') as i_file:
    json.load(i_file, encoding='utf-8')
```

JAVA

```java
import org.json.*;
String inFile = "data/games.json";
try {
    String contents = new String(readAllBytes(get(inFile)));
    JSONObject data = new JSONObject(contents);
} catch (IOException e) {
    e.printStackTrace();
}
```

WRITE JSON DATA

USE LIBS AGAIN!

PYTHON

```python
import json
data = {'game': 'GTAV'}
out_file = 'data/games.json'
with open(out_file, 'wb') as out:
    json.dump(data, out, encoding='utf-8')
```

JAVA

```java
import org.json.*;
String outFile = "data/games.json";
JSONObject obj = new JSONObject();
obj.put("game", "GTAV");
try {
    FileWriter file = new FileWriter(outFile);
    file.write(obj.toJSONString());
} catch (IOException e) {
    e.printStackTrace();
}
```

Exercise 1

- Download http://islab.di.unimi.it/~alfio/mdbexe/games.json.gz
- Read it in memory using your favourite programming language
- Understand data structure
- What kind of games? How many per kind?

JSON VS BSON

MongoDB represents JSON documents in binary-encoded format called BSON behind the scenes. BSON extends the JSON model to provide additional data types and to be efficient for encoding and decoding within different languages.

https://www.mongodb.com/json-and-bson

Go to MongoDB

MongoDB documentation is awesome. So **use the docs!**
https://docs.mongodb.com/manual/

MongoDB is made of **databases** and **collections**. Forget CREATE DATABASE and CREATE COLLECTION stuff. If you need a DB and a collection **just start using it.**

CONSOLE COMMANDS TO SURVIVE

```
> show dbs
> use [db_name]
> db.getCollectionNames()
> db.[collection_name].[function]
          (e.g., insert(), update(), remove(), find(), count(), aggregate())
> db.[collection_name].drop()
> db.dropDatabase()
```

MONGODB BACKUP AND RESTORE

```
> mongodump --host [host] --db [db_name] --collection [collection_name] --out [path]
```

```
> mongorestore --collection [collection_name] --db [db_name] [path/file.bson]
```

```
> mongorestore [path/db_dir]
```

Use the mongo console to save your data and transfer to a different machine

PUT DATA IN MONGODB PROGRAMMATICALLY

PYTHON

```
import pymongo
db = pymongo.MongoClient(host_name)[db_name] #create client and select db
db[collection_name].insert_one({'key': 'value', ...}) #insert single record
db[collection_name].insert_many([{'key': 'value', ...}, {'key': 'value', ...}, ...])
            #insert batch
```

JAVA

```
import com.mongodb.*;
Mongo mongo = new Mongo(host_name, 27017);
DB db = mongo.getDB(db_name);
DBCollection a_collection = db.getCollection(collection_name);
JSONObject j = ...;
DBObject o = (DBObject) JSON.parse(j.toString()); //take care of mapping from JSON to BSON
a_collection.insert(o); //or even a List of DBObject for batch insert
```

Exercise 2

Get data from games.json and put it in two collections, one for videogames and the other for boardgames

Note that the field 'date' in 'ratings' is encoded as a String in JSON but should be loaded as a Date in the DB

Check your results with respect to this dump: http://islab.di.unimi.it/~alfio/mdbexe/ginfo.tgz

CRUD OPERATIONS AND QUERIES!

https://docs.mongodb.com/manual/tutorial/query-documents/

Understand the notion of MongoDB cursor and cursor timeout

Understand how query operators work

PROGRAMMATICALLY

```
> db.collection.find({'name': 'gino'}) #MongoDB (see findOne)

cursor = db[collection].find({'name': 'gino'})
            #pymongo: cursor is iterable (find_one returns dict)

DBCursor cursor = coll.find(new BasicDBObject("name", "gino");
            #Java: cursor.hasNext() and cursor.next()
```

Exercise(s) 3.1

Define and execute the following queries

- Q1: Find videogames that can be played by 2 people
- Q2: Find title of boardgames that can be played by 2 people
- Q3: Find title and year of boardgames that can be played by 2 people sorted by year descending and title ascending
- Q4: Find titles videogames of category **Action**
- Q5: Find titles, year, and categories of videogames either of category **Action** or **Adventure**
- Q6: Find title and categories of videogames that are of type **Action AND Adventure**

EXERCISE(S) 3.2

Define and execute the following queries

- Q7: Find title of boardgames published by Avalon Games and playable in no more than 120 min
- Q8: Find title of videogames in which the second category is Action
- Q9: Find title and votes of videogames that received at least one evaluation of 8 or more
- Q10: Find titles and platforms of videogames that are sold by Amazon at less than 40
- Q11: Find titles of videogames for which we have ratings
- Q12: Find titles of boardgames where minimum number of players is between 2 and 4
  - use .explain("executionStats") to examine results
  - try db.boardgames.createIndex({min_players: 1})
  - use .explain("executionStats") to examine results again

DB DUMP

Download: http://islab.di.unimi.it/~alfio/mdbexe/ginfo_games.tgz

```
mongorestore --db [your db name] [path/to/dump/folder/]
```

AGGREGATION FRAMEWORK

Aggregations operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result.

AGGREGATION PIPELINE

The aggregation pipeline is a **sequence of operations on data** where the only condition is that data returned from operation **i** are compatible with the input required from operation **i + 1**

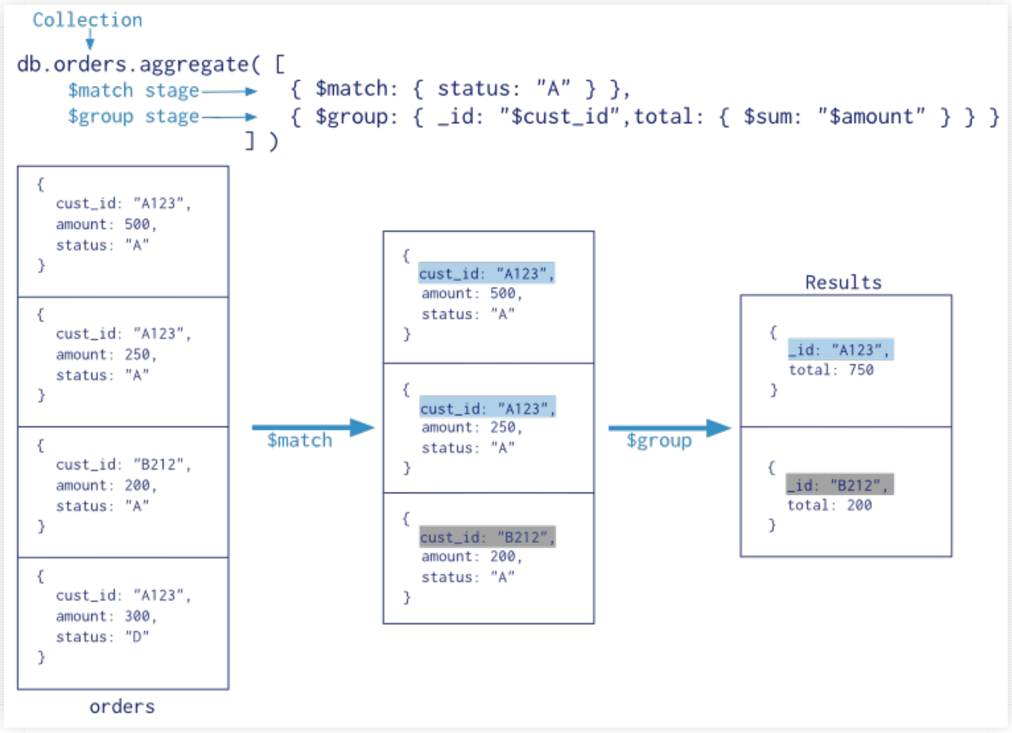
Remember?

$$\pi_{A,B}(\sigma_{c=v}(R \bowtie_{x=y} S))$$

PIPELINE

An aggregation pipeline has the following structure:

db.collection.aggregate([O1, O2, ..., On])

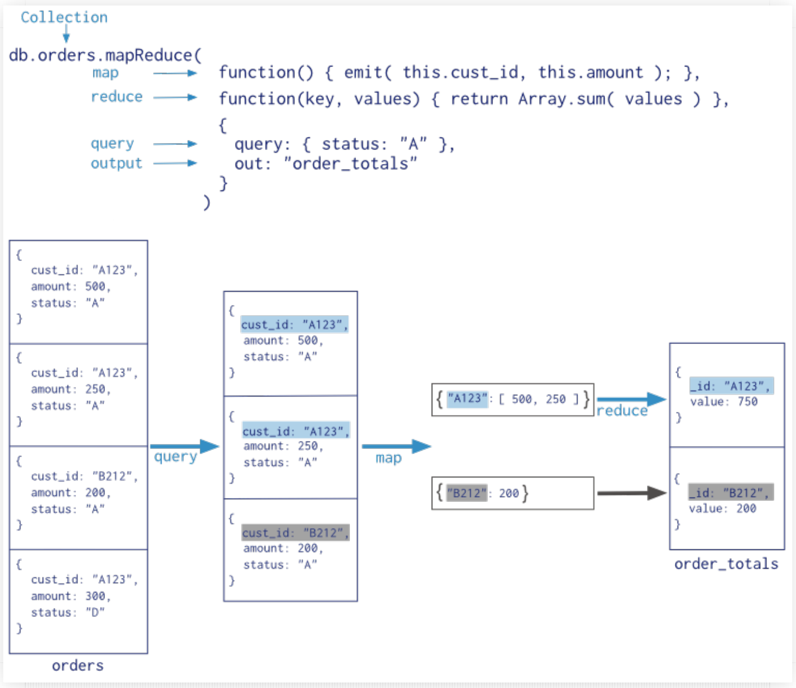


```
Collection
db.orders.aggregate( [
    $match stage ──►  { $match: { status: "A" } },
    $group stage ──►  { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
] )
```

{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }
orders
$match
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
$group
Results
{ _id: "A123", total: 750 }
{ _id: "B212", total: 200 }

ALTERNATIVE: MAPREDUCE

Map-reduce operations have two phases: a **map** stage that processes each document and **emits** one or more objects for each input document, and **reduce** phase that **combines** the output of the map operation



```
db.orders.mapReduce(
                      function() { emit( this.cust_id, this.amount ); },
                      function(key, values) { return Array.sum( values ) },
                      {
                        query: { status: "A" },
                        out: "order_totals"
                      }
                    )
```

Collection
map
reduce
query
output
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }
orders
query
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
map
{ "A123": [500, 250] }
{ "B212": 200 }
reduce
{ _id: "A123", value: 750 }
{ _id: "B212", value: 200 }
order_totals

$GROUP

Groups input documents by a **specified identifier expression** and applies the **accumulator expression(s)**, if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.

```
db.games.aggregate([
            {
                $group: {
                    _id: "$type", //GROUP BY clause
                    players: {
                        "$avg": "$max_players" //ACCUMULATION (Aggregate operator)
                    }
                }
            }
            ])
```

Exercise: find average number of minimum and maximum players per year

SPECIAL CASE: ACCUMULATE ON ALL

```
db.games.aggregate([
    {
        $group: {
            _id: null, //GROUP BY clause
            players: {
                "$avg": "$max_players" //ACCUMULATION (Aggregate operator)
            }
            number: {"$sum": 1} // The way we implement COUNT
        }
    }
])
```

MULTIPLE FIELD GROUPING

```
db.games.aggregate([
    {
        $group: {
            _id: {"year": "$year", "type": "$type"}, //GROUP BY clause
            players: {
                "$avg": "$max_players" //ACCUMULATION (Aggregate operator)
            }
        }
    }
])
```

Accumulators

$sum	Returns a sum of numerical values. Ignores non-numeric values.
$avg	Returns an average of numerical values. Ignores non-numeric values.
$first / $last	Returns a value from the first/last document for each group. Order is only defined if the documents are in a defined order.
$max / $min	Returns the highest/lowest expression value for each group.
$push / $addToSet	Returns an array of expression values (**unique for $addToSet**) for each group.
$stdDevPop / $stdDevSamp	Returns the population/sample standard deviation of the input values.

$MATCH

Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).

Exercise: calculate the number of **Action** games per year in the period 2010-2015

$PROJECT

Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.

```
db.games.aggregate([{$project: {publisher: "$publisher.name", year: 1}}])
```

$REDACT

Reshapes each document in the stream by restricting the content for each document based on information stored in the documents themselves. Incorporates the functionality of **$project** and **$match**. Can be used to implement field level redaction. For each input document, outputs either one or zero documents.

$UNWIND

Deconstructs an array field from the input documents to output a document for each element. Each output document replaces the array with an element value. For each input document, outputs n documents where n is the number of array elements and can be zero for an empty array.

Exercise: count games per category

COMMENTED EXERCISE

Find average rating per month and category

```
u1 = {'$unwind': "$categories"}
u2 = {'$unwind': "$ratings"}
p = {'$project': {'month': {'$month': "$ratings.date"},
            'category': "$categories", 'rate': "$ratings.vote"}}
g = {'$group': {
            '_id': {"month": "$month", "category": "$category"},
            "rate": {"$avg": "$rate"}}
    }
pipeline = [u1, u2, p, g]
db['games'].aggregate(pipeline)
```

OTHER OPERATIONS

$sample (3.2)	Randomly selects the specified number of documents from its input.
$lookup (3.2)	Performs a left outer join to another collection in the same database to filter in documents from the "joined" collection for processing.
$out	Writes the resulting documents of the aggregation pipeline to a collection. To use the $out stage, it must be the last stage in the pipeline.

Exercises

- A1: find years when more than 100 games have been published
- A2: find distribution of categories per year