NoSQL document-oriented

Matteo Francia, Ph.D.

m.francia@unibo.it

Who am I?

Matteo Francia, Ph.D.

- Assistant professor @ UniBO
- BIG: Business Intelligence Group (http://big.csr.unibo.it/)

Help me:

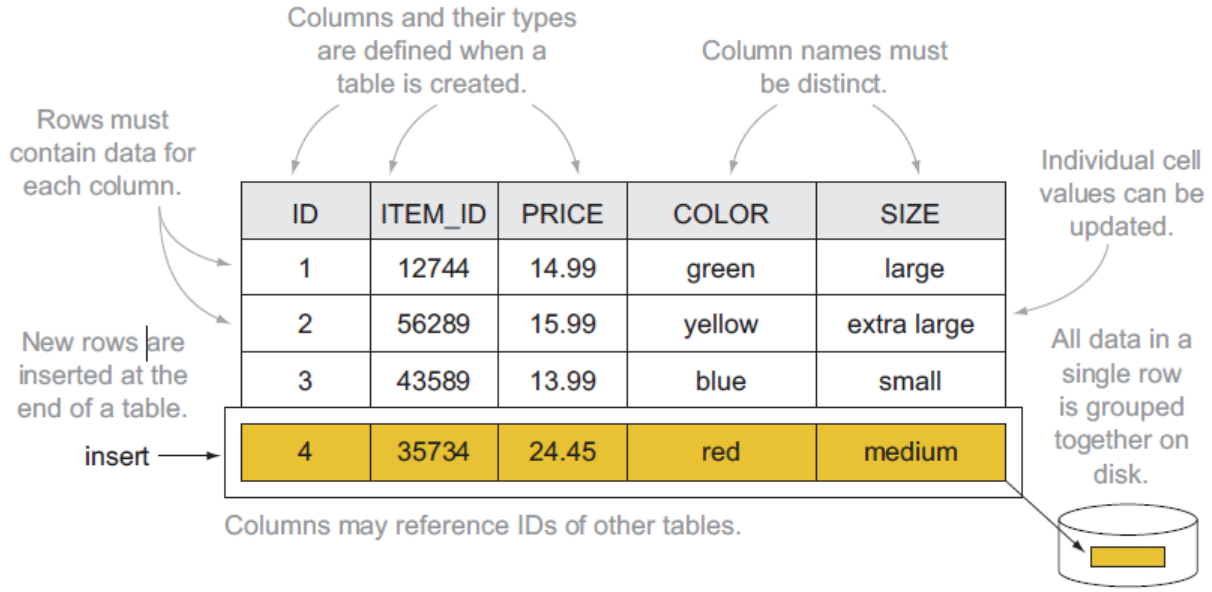
- Keep this class interactive!
- Do not be afraid to ask (also for language related issues :))

Thanks to Enrico Gallinucci for these slides

Why NoSQL?

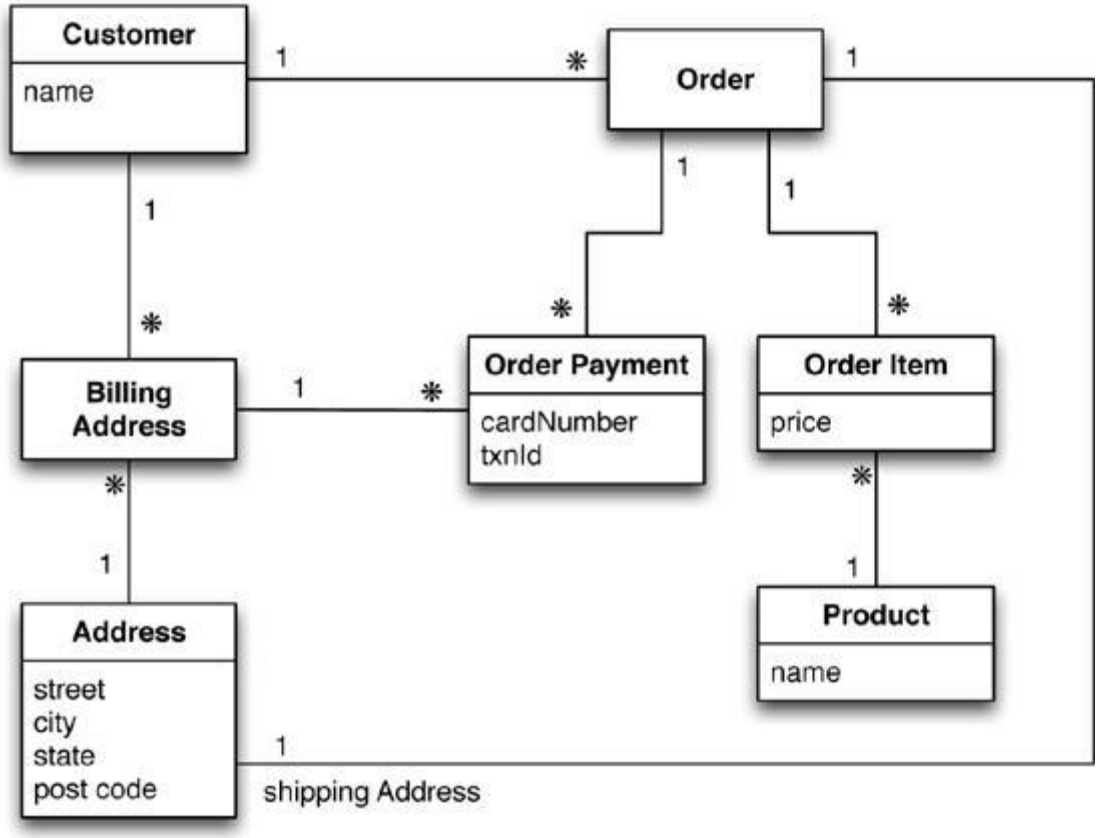
Relational: data model

Based on tables and rows



ID	ITEM_ID	PRICE	COLOR	SIZE
1	12744	14.99	green	large
2	56289	15.99	yellow	extra large
3	43589	13.99	blue	small
4	35734	24.45	red	medium

Running example



Data modeling example: relational model

Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

Address

Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Strengths of RDBMSs

ACID properties

- Provides guarantees in terms of consistency and concurrent accesses

Data integration and normalization of schemas

- Several application can share and reuse the same information

Standard model and query language

- The relational model and SQL are very well-known standards
- The same theoretical background is shared by the different implementations

Robustness

- Have been used for over 40 years

Weaknesses of RDBMS

Impedance mismatch

- Data are stored according to the relational model, but applications to modify them typically rely on the object-oriented model
- Many solutions, no standard
  - E.g.: Object Oriented DBMS (OODBMS), Object-Relational DBMS (ORDBMS), Object-Relational Mapping (ORM) frameworks

Painful scaling-out

- Not suited for a cluster architecture
- Distributing an RDBMS is neither easy nor cheap (e.g., Oracle RAC)

Consistency vs latency

- Consistency is a must – even at the expense of latency
- Today's applications require high reading/writing throughput with low latency

Schema rigidity

- Schema evolution is often expensive

The meaning of NoSQL

1998 Carlo Strozzi

- RDBMS open-source with a query language different from SQL

2009 Meetup in San Francisco

- Discussion about projects inspired by new databases from Google e Amazon
- Participants: Voldemort, Cassandra, Dynamite, HBase, Hypertable, CouchDB, MongoDB

NoSQL is a DBMS (DataBase Management System) in which the persistency data model is not relational (RDBMS)

- NoSQL = Not Only SQL
- According to Strozzi: NoREL would have been more appropriate

The first NoSQL systems

LiveJournal, 2003

- Goal: reduce the number of queries on a DB from a pool of web servers
- Solution: Memcached, designed to keep queries and results in RAM

Google, 2005

- Goal: handle Big Data (web indexing, Maps, Gmail, etc.)
- Solution: BigTable, designed for scalability and high performance on Petabytes of data

Amazon, 2007

- Goal: ensure availability and reliability of its e-commerce service 24/7
- Solution: DynamoDB, characterized by strong simplicity for data storage and manipulation

NoSQL common features

Not just rows and tables

- Several data model adopted to store and manipulate data

Freedom from joins

- Joins are either not supported or discouraged

Freedom from rigid schemas

- Data can be stored or queried without pre-defining a schema (*schemaless* or *soft-schema*)

Distributed, shared-nothing architecture

- Trivial scalability in a distributed environment with no performance decay
- Each workstation uses its own disks and RAM

SQL is dead, long live SQL!

- Some systems do adopt SQL (or a SQL-like language)

NoSQL in the Big Data world

NoSQL systems are mainly used for operational workloads (OLTP)

- Optimized for high read and write throughput on small amounts of data

Big Data technologies are mainly used for analytical workloads (OLAP)

- Optimized for high read throughput on large amounts of data

Can NoSQL systems be used for OLAP?

- Possibly, but through Big Data analytical tools (e.g., Spark)

NoSQL: many models

One of the key challenges is to understand which one fits best with the required application

Modello	Descrizione	Casi d'uso
Key-value	Associates any kind of value to a string	Dictionary, lookup table, cache, file and images storage
Document	Stores hierarchical data in a tree-like structure	Documents, anything that fits into a hierarchical structure
Wide-column	Stores sparse matrixes where a cell is identified by the row and column keys	Crawling, high-variability systems, sparse matrixes
Graph	Stores vertices and arches	Social network queries, inference, pattern matching

The documental model

Document: data model

Each DB contains one or more collections (corresponding to tables)

Each collection contains a list of documents (usually JSON)

- Documents are hierarchically structured

Each document contains a set of fields

- The ID is mandatory

Each field corresponds to a key-value pair

- Key: unque string in the document
- Value: either simple (string, number, boolean) or complex (object, array, BLOB)
  - A complex field can contain other field

```
{
    "_id": 1234,
    "name": "Enrico",
    "address": {
        "city": "Cesena",
        "postalCode": 47522
    },
    "contacts": [ {
        "type": "office",
        "contact": "0547-338835"
    }, {
        "type": "skype",
        "contact": "egallinucci"
    } ]
}
```

Document: querying

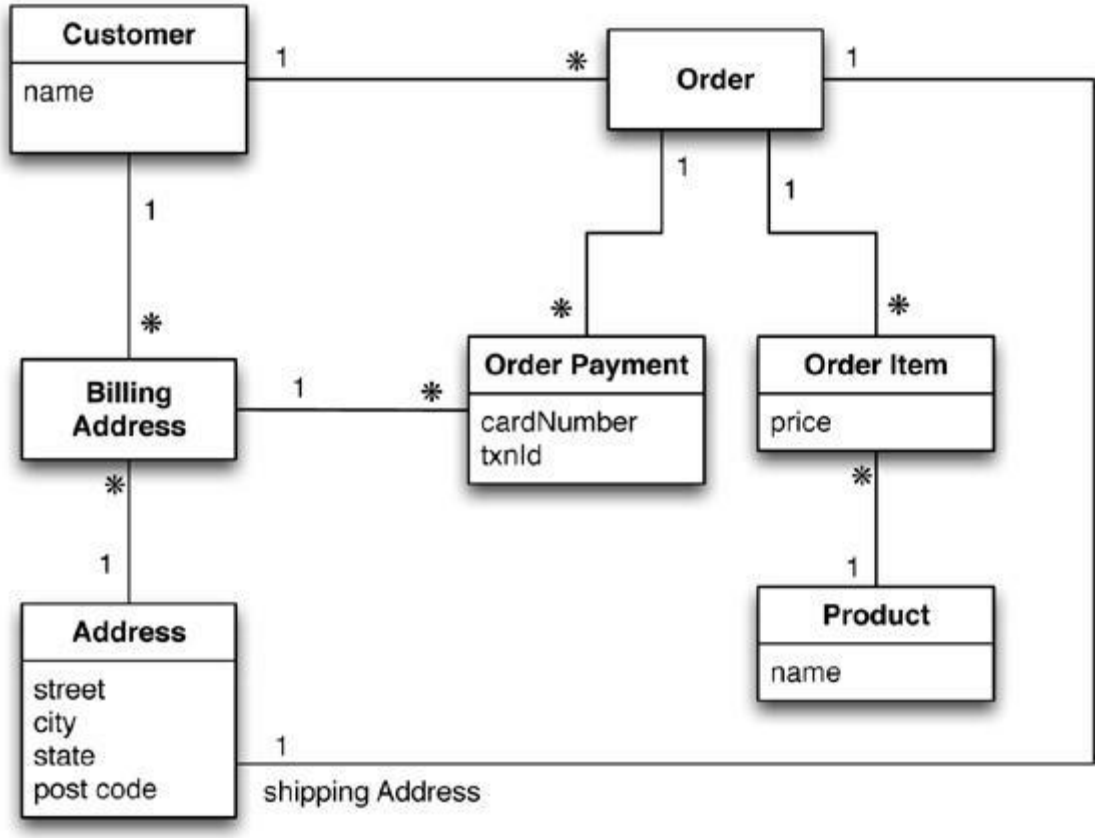
The query language is quite expressive

- Can create indexes on fields
- Can filter on the fields
- Can return more documents with one query
- Can select which fields to project
- Can update specific fields

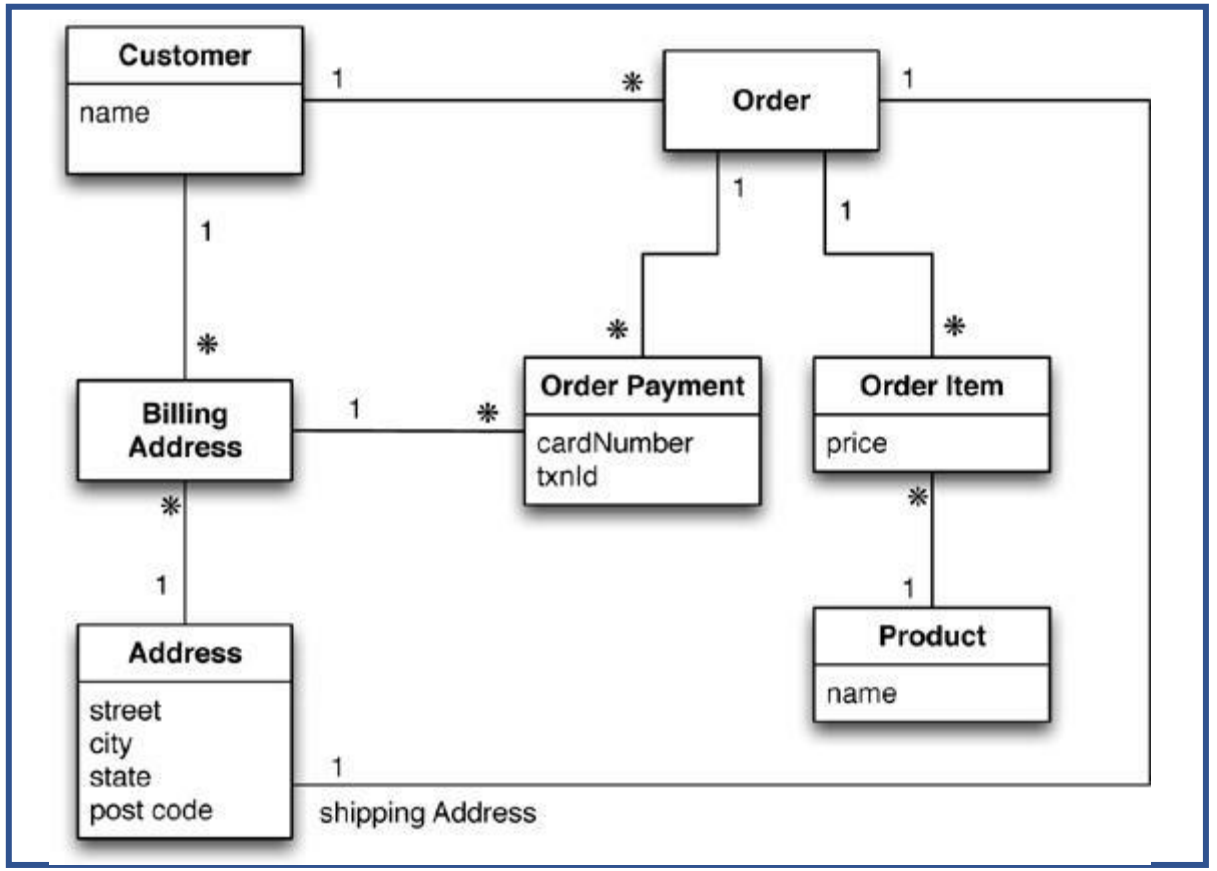
Different implementations, different functionalities

- Some enable (possibly materialized) views
- Some enable MapReduce queries
- Some provide connectors to Big Data tools (e.g., Spark, Hive)
- Some provide *full-text search* capabilities

Running example



Running example: workload #1

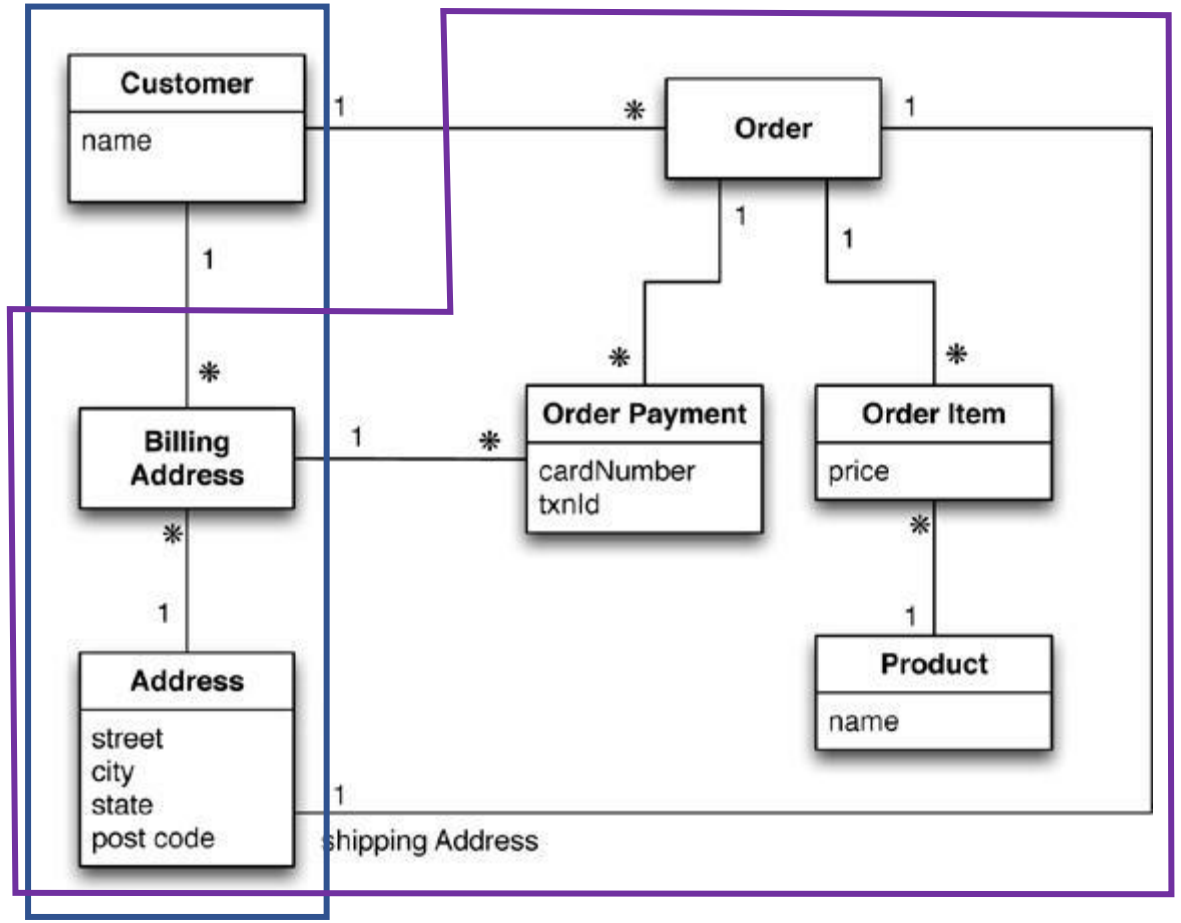


Data modeling example: document model (1)

Customer collection

```
{
   "_id": 1,
   "name": "Martin",
   "adrs": [
      {"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007},
      {"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}
   ],
   "orders": [ {
      "orderpayments":[
         {"card":477, "billadrs": {"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}},
         {"card":457, "billadrs": {"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}}
      ],
      "products":[
         {"id":1, "name":"Cola", "price":12.4},
         {"id":2, "name":"Fanta", "price":14.4}
      ],
      "shipAdrs": {"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}
   }]
}
```

Running example: workload #2



Data modeling example: document model (2)

```
{
   "_id": 1,
   "name": "Martin",
   "adrs": [
      {"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007},
      {"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}
   ]
}
```

Customer collection

```
{
   "_id": 1,
   "customer":1,
   "orderpayments":[
      {"card":477, "billadrs":{"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}},
      {"card":457, "billadrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}}
   ],
   "products": [
      {"id":1, "name":"Cola", "price":12.4},
      {"id":2, "name":"Fanta", "price":14.4}
   ],
   "shipAdrs": {"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}
}
```

Order collection

Modeling documents

Aggregate data modeling

- An aggregate is a set of objects that are linked together and that are treated in bulk
- An aggregate is a unit for data manipulation and consistency management

Aggregates:

- + Facilitate software developers, who often manipulate data through aggregated structures
- + Easy to manage in a distributed system
  - Data that needs to be manipulated together (e.g., orders and their details) are modeled in the same aggregate – and therefore reside in the same node
- - Duplication of data stored in nested layers (and risk of inconsistencies)

There is no universal strategy for defining aggregates

- It depends on how you intend to manipulate the data

Use Cases

- Event logs / web services
  - Central repositories for storing event logs of different applications
- CMS, blogging platforms
  - Absence of a default scheme
- Web Analytics o Real-Time
  - Analytics Indexing of textual content real-time sentiment analysis, social media monitoring
- Ecommerce Applications
  - Flexibility on the scheme to store products and orders
  - Evolve your data model without incurring refactory or migration costs

Downsides to consider

- Aggregate data modeling source of duplication and inconsistencies
- Schemaless is an advantage in writing, not reading
- Not ideal for an analytical workload (OLAP)

Use Cases

Advertising

- MongoDB was born as a banner ad management system
  - Service must be available 24/7 and very efficient
  - Complex rules needed to find the right banner based on the interests of the person
  - Need to manage different types of CEOs and to have detailed reporting

Internet of Things

- Real-time management of data generated by sensors
- FIWARE smart data models: https://www.fiware.org/smart-data-models/

(There is not time)
To talk about

Sharding

- Data partitioning
- Data replication
- Master/Slave ReplicaSet
- Eventual consistency (CAP)

Polyglot persistence

DB administration

GUI

- MongoDB Compass
- MongoDB Atlas
- Studio 3T

Integration with 3rd party applications

Getting started with MongoDB

Introduction

Document-oriented databases replace the concept of row with a more flexible model: the document

- Represent complex hierarchical relationships in a single document
- There is no default schema

Some of the main features:

- Many indexes available: compounds, geo-spatial, full-text
- An aggregation pipeline mechanism to build complex aggregations through concatenation
- Different collection types: time-to-live, fixed-size
- Ability to use scripts in the Javascript language to manipulate data

Documents

Documents are basically JSON objects

- It is possible to use data types that JSON formalism does not provide.
- Recursively defined as objects composed of key-value ordered pairs, where:
  - The **key** is a case-sensitive string
    - You cannot use the characters "." and "$“
    - No two identical keys can exist within the same object
  - The **value** can be of several types:
    - A simple type (e.g., string, number, date, etc.)
    - Another object
    - An array of values
  - The key order is not important
- In each document a special field is automatically inserted
  - **_id**, whose value is unique within the collection (primary key)

Documents

An example

```
{
   "_id": ObjectId("5037ee4a1084eb3ffeef7228"),
   "info": {
      "name": "Henry", "dateBirth" : ISODate("1988-08-04T20:42:00.000Z")
   },
   "interests": ["Soccer", "Travels", "TV Series"],
   "teaching": [
      { "course": "Big Data", "employer": "University of Bologna" },
      {"course": "Introduction to NoSQL databases", "typology": "IFTS" }
   ]
}
```

Collections

A collection consists of a set of documents

- There is no schema (schemaless)

Why create multiple collections instead of keeping just one?

- Comfort
- Performance
- Data locality
- Different indexes in different collections

A collection is identified by a name

- You cannot use the "$" character
- You can use the ".", in particular to conceptually organize collections into sub-collections.
- E.g., blog.posts, blog.authors, etc.

Database

A MongoDB instance can contain many DBs, each DB hosts collections

- Each database has its own permission management mechanism
- Typically, you use a database for each application
- Databases are identified by a name
- There are many restrictions on characters (use ASCII alphanumeric characters)

Database

Some databases are reserved

admin

- It is the main database in terms of authentication
- Users assigned to this database can also access all others
- Some commands can only be executed from this database (e.g., shut down the server)

local

- In a cluster, there is one for each machine where MongoDB is installed
- Can be used to store data that should not be distributed

config

- Stores useful information for use in distributed mode

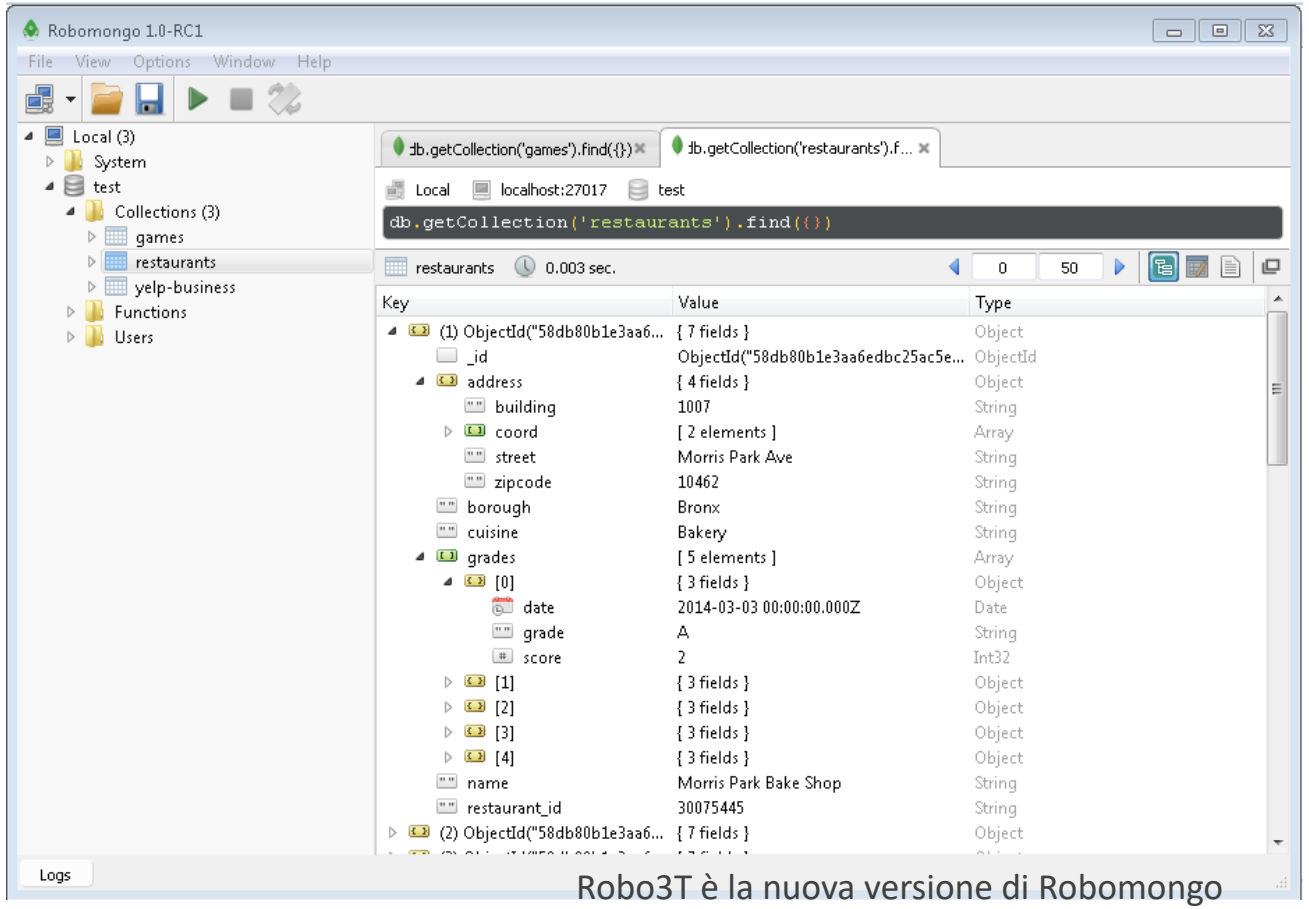
Connection via Robo3T

Robo3T?

- Simplifies management and Database navigation
- Embed a MongoDB shell

Parameters:

- URL: localhost
- Port: 27018 (default is 27017)



Robo3T è la nuova versione di Robomongo

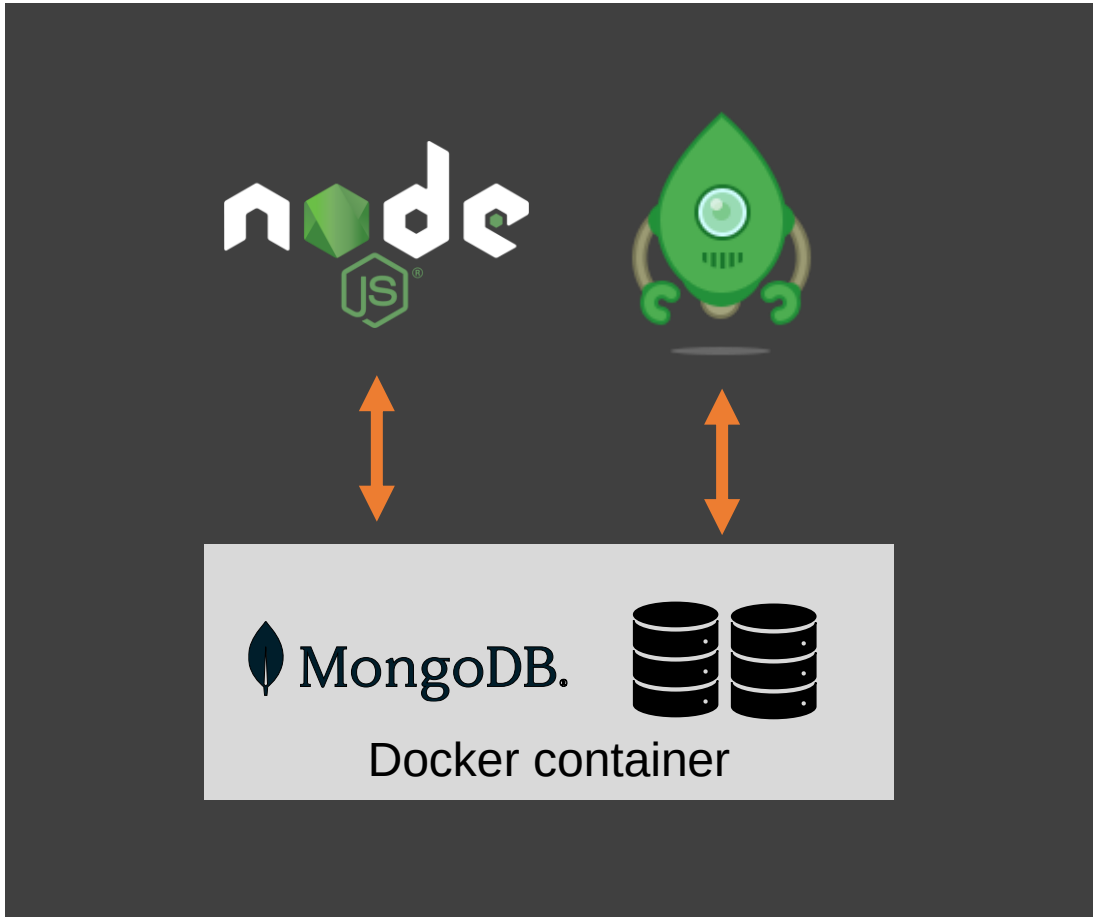
Software components

MongoDB

- Deployed on a docker container
- Already contains the data
- Check `docker-compose.yml`

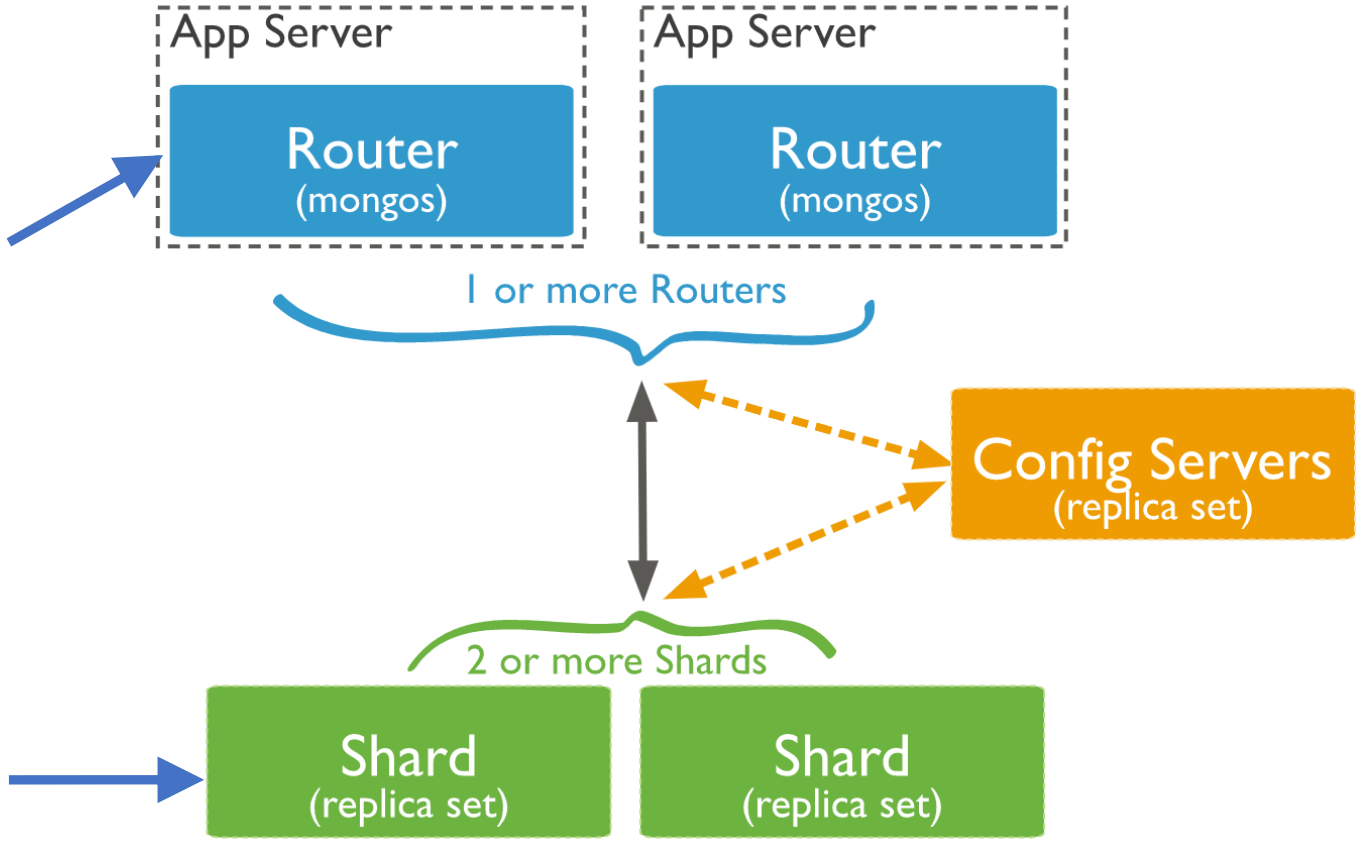
Querying the data, two alternatives:

- Robo3T (external program)
- Nodejs (check `src/`)



Connecting to a cluster

Port 27020
- Access to collections Distributed (Sharded)



Gate 27017 the 27018
- Access to local collections
- Access to local portions of the distributed collections

Collections for lab (1)

Restaurants

- https://raw.githubusercontent.com/mongodb/docs-assets/primer-dataset/primer-dataset.json
- 25359 documents related to Restaurants (name, address, type of cuisine, votes)

NBA games http://bit.ly/1gAatZK

- 31686 documents relating to 30 years of NBA games (date, teams, stats)

Yelp

- https://www.yelp.com/dataset_challenge
- Real data made available to scientific research
- More than $50,000 distributed in competitions, more than 100 academic papers

```
mongorestore --collection games --db test C:\games.bson
```

Collections for lab (2)

NBA Statistics

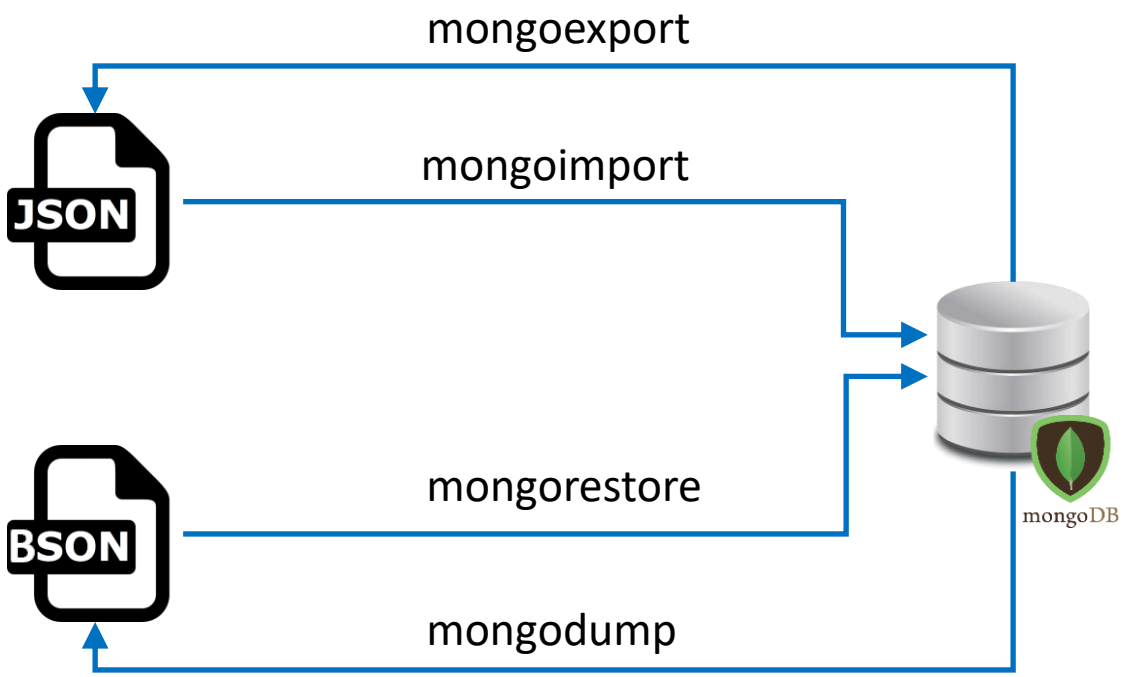
- http://www.mediafire.com/file/ju52cn1eadiydz6/NBA2016.json
- 1 document containing statistics for the 2016/17 season for all players and teams
- Modified to separate player and team statistics into two files

US City Statistics

- https://gist.githubusercontent.com/Miserlou/c5cd8364bf9b2420bb29/raw/2bf258763cdddd704f8ffd3ea9a3e81d25e2c6f6/cities.json
- 1000 documents with statistics on the most populated cities in the United States of America

```
mongoimport --collection nba2016players --db test C:\nba2016players.json --jsonArray
```

Import/export tools



https://www.mongodb.com/basics/bson

Basic commands

Most MongoDB commands are methods of the object **db**

Some examples

- `db.getMongo().getDBs()` – shows the databases in MongoDB
- `db.getCollectionNames()` – shows collection names in the current DB
- `db.getSisterDB("foo")` – switch to FOO database

To work on a collection:

- `db.collectionName.[method]([parameters])`
- `db.getCollection(collectionName).[method]([parameters])`

Main commands

Querying

- Find, FindOne – read data with projections and selections
- Count, Distinct – easy ways to aggregate data
- Aggregate – advanced mode to aggregate data through the concatenations of simpler operations (match, unfold, group, ecc.)

Editing data

- Insert, Delete, Update

MongoDB query language

Find

Allows you to perform queries (queries) on the DB

The basic form is:

```
db.collectionName.find([[objSel],[objProj]])
```

- `collectionName` is name of the collection to be queried
  - equivalent SQL: FROM (but limited to a single collection)
- `objSel` is an optional object that contains filter criteria; equivalent SQL: `WHERE`
- `objProj` is an optional object that contains the selection criteria; equivalent SQL: `SELECT`

Find

MongoDB

```
db.collectionName
  .find(objSel, objProj)
  .sort(attrs)
```

Relational

```
select objProj
from collectionName
where objSel
order by attrs
```

Find

Some examples

db.restaurant.find()

- Returns all documents

db.restaurant.findOne()

- Returns only the first document

db.restaurant.find({cuisine: "Hamburgers"})

- Returns documents where the cuisine attribute (if any) is valued with the string "Hamburgers"

db.restaurant.find({}, {cuisine: 1})

- Returns all documents, but projecting only the cuisine attribute
- In addition to the _id, which is returned by default

db.restaurant.find({cuisine: "Hamburgers"}, {cuisine: 1})

- The combination of selection and projection

Find – projection

If projection is not specified, all attributes of all documents are returned

If a projection is indicated, only the indicated fields are retained – with the exception of the _id field

- It is however possible to exclude the _id field

Syntax

- key: [0, 1]

Where

- key is the name of an attribute
- 1 to keep the attribute
- 0 to exclude the attribute (e.g., l'_id)

Find – selection

A first selection mode takes place through the exact match of the attribute value with a specified value

Examples:

- db.users.find({"age" : 27})
- db.users.find({"username" : "joe"})
- db.users.find({"username" : "joe", "age" : 27})

How to express more complex conditions?

- ~~db.restaurant.find({restaurant_id > 40367790})~~
- It is not possible, because you have to respect the JSON syntax

Find – selection

Complex selection conditions requires nesting new objects

Syntax

- key : { operator: value }

Dove

- operator is a comparison operator according to the syntax of MongoDB (e.g., $gte is "Greather Than or Equal to")
- value is a simple value (e.g., a number or string)
- Some operators require that value is an object, composed of another pair operator : value

Find – comparison operators

Operators

- $gte, $gt – correspond to ≥ e >
- $lte, $lt – correspond to≤ e <
- $ne – corresponds to ≠

Examples

- db.users.find({"age" : {"$gte" : 18} })
- db.users.find({"age" : {"$gte" : 18, "$lte" : 30}})
- db.users.find({"registered" : {"$lt" : new Date("2007-01-01")} })
  - Il formato della data dipende dalla localizzazione
- db.users.find({"username" : {"$ne" : "joe"}})

Find – joining conditions

Operators

- $in, $nin – equivalent to IN and NOT clauses IN of SQL
- $or, $nor, $and – equivalent to their logical operators

Examples

- db.users.find({"user_id": {"$in": [12345, "joe"]}})
- db.raffle.find({"ticket_no": {"$nin": [725, 542, 390]}})

- db.raffle.find({"$or": [{"ticket_no": 725}, {"winner": true}]})
- db.raffle.find({"$or": [{"ticket_no": {"$in": [725, 542, 390]}}, {"winner": true}]})

- db.raffle.find({"$nor": [{"ticket_no": 725}, {"winner": true}]})
- db.raffle.find({"$and": [{"ticket_no": 725}, {"winner": true}]})

Find – joining conditions

There can be different ways to express the same criterion, more or less optimized

Examples

- db.users.find({"$and" : [{"x" : {"$lt" : 5}}, {"x" : 1}]})
- db.users.find({"x" : {"$lt" : 5, "$in" : [1]}})

The optimizer struggles more in the presence of $and and $or operators; If possible, it is best to avoid using them

Find – negation

Operator

- $not

Examples

- db.users.find({"id_num" : {"$not" : {"$mod" : [5, 1]}}})

Find – nulls

Some attributes can have null as a value.

The command `db.c.find({"y" : null})`

- Returns both documents in which key Y exists and is valued at null and documents in which key Y does not exist.

To retrieve the documents in which the key y exists and is valued at null, you must also verify the existence of the key itself:

- `db.c.find({"y" : {"$in" : [null], "$exists" : true}})`

Find – querying arrays

Given a food collection with 3 documents:

- {"_id" : 1, "fruit" : ["apple", "banana", "peach"]}
- {"_id" : 2, "fruit" : ["apple", "kumquat", "orange"]}
- {"_id" : 3, "fruit" : ["cherry", "banana", "apple"]}

Commands

- db.food.find({"fruit" : "banana"})
  match if the array contains banana (returns: 1 and 3)
- db.food.find({fruit : {$all : ["apple", "banana"]}})
  match if the array contains both Apple and Banana (returns: 1 and 3)
- db.food.find({fruit : {$in : ["apple", "banana"]}})
  match if the array contains apple or banana (returns: 1, 2 and 3)

Find – querying arrays

Given a food collection with 3 documents :

- {"_id" : 1, "fruit" : ["apple", "banana", "peach"]}
- {"_id" : 2, "fruit" : ["apple", "kumquat", "orange"]}
- {"_id" : 3, "fruit" : ["cherry", "banana", "apple"]}

Commands

- db.food.find({"fruit" : ["banana", "apple", "peach"]})
  match if the array exactly matches the one indicated (returns: nothing)
- db.food.find({"fruit.2" : "peach"})
  match if the array contains peach at position 2 0-based (returns: 1)
- db.food.find({"fruit" : {"$size" : 3}})
  match if the array contains 3 elements (returns: 1, 2 and 3)

Find – querying arrays

at projection time you can limit the number of array elements that are returned by the query

Given a doc that contains a blog post and comments

- db.blog.posts.find(criteria, {"comments": {"$slice": 10}})
  returns the first 10 comments
- db.blog.posts.find(criteria, {"comments": {"$slice": -10}})
  returns the last 10 comments
- db.blog.posts.find(criteria, {"comments": {"$slice": [23,10]}})
  skip the first 23 documents and return the next 10 (24th to 33rd)
- db.blog.posts.find(criteria, {"comments.$": 1})
  returns comments that meet the selected criteria

Please note: If $slice is the only operator used in the projection, all document fields are returned

Find – querying arrays

When placing a selection with multiple criteria, these are evaluated in AND

- db.test.find({"x": {"$gt":10, "$lt":20}})
- $(\exists x > 10) \wedge (\exists x < 20)$

If x is a simple attribute

- *The document contains an x greater than 10 and less than 20?*

If x is an array

- *The document contains an x greater than 10?*
- *… AND the document contains an x greater than 20?*
- If the document is not empty, it will always be returned

To impose constraints element-wise in an array, you must use $elemMatch

- db.test.find({"x": {"$elemMatch": {"$gt":10, "$lt":20}})

Find – querying objects

Given {"name": {"first":"Joe", "middle":"K" , "last":"Schmoe" }}

There are two modes

- db.people.find({"name" : {"first":"Joe", last":"Schmoe"}})
  Exact match: the object searched for must be equal (in this case, it returns nothing)
- db.people.find({"name.first":"Joe", "name.last":"Schmoe"})
  You can use dot notation to reference individual fields (in this case, it returns the document)

Find – querying arrays of object

Goal: Search for Joe's comments with a score of at least 5

- db.blog.find({"comments" : {"author" : "joe", "score" : {"$gte" : 5}}})
  Wrong: look for the exact match
- db.blog.find({"comments.author" : "joe", "comments.score" : {"$gte" : 5}})
  Wrong: Returns both comments.
  because conditions are evaluated in OR
- db.blog.find({"comments" : {"$elemMatch" : {"author" : "joe", "score" : {"$gte" : 5}}}})
  Correct

Given:

```
{
 "content" : "...",
 "comments" : [{
  "author" : "joe",
  "score" : 3,
  "comment" : "nice post"
 },{
  "author" : "mary",
  "score" : 6,
  "comment" : "terrible post"
 }]
}
```

Find – Javascript scripts

Key-value pair query expressiveness is limited

For particularly complex queries it is possible to use the operator $where to run Javascript code

- db.mycoll.find({$where : function() { return this.date.getMonth() == 11} })

Through scripts you can do almost any type of operation

- For safety reasons, however, the use of the operator is strongly discouraged $where
- End users should NEVER be allowed to run these types of queries

Limit, skip & sort

Limit: return only the first n documents

- db.c.find().limit(3)

Skip: skip the first n documents and return the next ones

- db.c.find().skip(3)

Sort: order results based on one or more attributes

- db.c.find().sort({username : 1, age : -1})
- The sort order can be ascending (1) or descending (-1)

Count

Count the number of documents returned by a query

- db.foo.count()
- db.foo.count({"x" : 1})
- Similar to Find, except for the absence of the selection object

Distinct

distinct returns the distinct values of a field from matching documents

- db.inventory.distinct("item.sku", { dept: "A" })
- Returns the distinct values of the item.sku field to documents where the department is A
- If item.sku is an array, values are also returned to the array of a single document

Aggregation Framework

Aggregate

Aggregation Framework

Apply transformations and aggregations on the documents of a collection

It consists of a series of pipeline operators

- Building blocks that can be freely combined with each other (even several times and in any order) to create more or less complex queries
- Match, Project, Group, Unwind, Sort, Limit, Skip

Not only aggregations: formulate queries that could not be done with Find

- Apply transformations on dates
- Concatenate two or more fields
- Compare the values of two fields
- Return a single element of an array instead of the entire array

Aggregation Framework

Example: in a collection of magazines, I want to know which are the authors who have sold the most

- Project: extract from each document the author of the revisit
- Group: group by author counting the number of occurrences
- Sort: sort in descending order on the number of occurrences
- Limit: keep only the first 5 results

```
db.articles.aggregate([
    {"$project" :{"author" : 1}},
    {"$group" :   {"_id" : "$author", "count" : {"$sum" : 1}}},
    {"$sort" :    {"count" : -1}},
    {"$limit" :   5}
])
```

$match

The $match operator filters documents

Basically like a Find query

- Does not support geospatial operators

Use the operator as soon as possible

- Reduces the number of documents for subsequent operations
- Can take advantage of indexes (may not be usable at later stages)

Examples

- db.restaurants.aggregate([{$match: {cuisine: "Hamburger"} }])
- db.restaurants.find({cuisine: "Hamburger"})

$project

$project selects the fields

- It is much more powerful than the projection in the Find command
- Allows you to extract fields from grafted objects and apply transformations

db.articles.aggregate([{"$project" : {"author" : 1, "_id" : 0}}])

- Returns the author of an article and excludes the _i fieldd

db.users.aggregate([{"$project" : {"userId" : "$_id", "_id" : 0}}])

- Rename the _id field to userId
- In practice, it introduces a new userId field whose value matches the value of _id
- NB: the use of the $ operator in "$_id" allows you to indicate the reference to a field
  - Otherwise, "_id" would be interpreted as a simple value

$project – mathematical expressions

Expressions on 1 or more values: $add, $multiply

- "$add" : [expr1, expr2, ..., exprN]

Expressions on 2 values: $subtract, $divide, $mod

- "$subtract" : [expr1, expr2]
- "$divide": divides the first value by the second
- "$mod": divides the first value by the second and returns the remainder

Example

- db.employees.aggregate([{ "$project" : {
  "totalPay" : { "$subtract" : [{"$add" : ["$salary", "$bonus"]}, "$401k"] }
  } }])
- Returns a calculated field: totalPay = (salary + bonus) - 401k

$project – expressions on dates

There are several expressions that allow you to extract a specific information from a date

- "$year", "$month", "$week"
- "$dayOfYear", "$dayOfMonth", "$dayOfWeek"
- "$hour", "$minute", "$second"

Examples

- db.employees.aggregate([{"$project": {"hiredIn" :{"$month" : "$hireDate"}} }])
  - Returns the month employees were hired
  -
- db.employees.aggregate([{"$project" : {"tenure" :
  { "$subtract" : [{"$year" : new Date()}, {"$year" : "$hireDate“}] }
  } }])
  - Returns the number of years since employees were hired
  - An arithmetic operation between two dates returns a result in milliseconds

$project – expressions on strings

"**$substr**" : [*expr*, *startOffset*, *numToReturn*]

- Returns a substring of the string passed as the first parameter; starts from startOffset and returns numToReturn bytes
- Beware of encoding: a byte may not correspond to a character

"**$concat**" : [*expr1[, expr2, ..., exprN]*]

- Concatenate strings passed as parameters

"**$toLower**", "**$toUpper**"

- Restore the string passed as a parameter in all lowercase or uppercase

Example

- ```
  db.employees.aggregate([{"$project" : {"email" : {"$concat" :
     [{"$substr" : ["$firstName", 0, 1]}, ".", "$lastName", "@example.com"]
  } } }])
  ```
- Returns a string as e.gallinucci@example.com
  ```

$project – logical expressions

Comparison expressions

- "**$cmp**" : [expr1, expr2]
  Compare expr1 with expr2. Returns 0 if they are equal, a negative number if expr1 < expr2, a positive number if expr1 > expr2
- "**$strcasecmp**" : [string1, string2]
  Case-insensitive comparison between two strings
- "**$eq**"/"**$ne**"/"**$gt**"/"**$gte**"/"**$lt**"/"**$lte**" : [expr1, expr2]
  Compare expr1 with expr2 and return true or false

Boolean expressions

- "**$and**", "**$or**" : [*expr1[, expr2, ..., exprN]*]
  True if all ($and) or at least one ($or) of the expressions is true
- "**$not**" : *expr*
  Returns the opposite Boolean value of expr

$project – logical expressions

- "**$cond**" : [*booleanExpr*, *trueExpr*, *falseExpr*]
  If the booleanExpr expression is true, return trueExpr, else falseExpr
- "**$ifNull**" : [*expr*, *replacementExpr*]
  If expr is null, return replacementExpr, otherwise return expr

Example: students are evaluated for 10% on attendance, 30% on questions, 60% on tests; but they score 100 if they are “teachers pet"

```
db.students.aggregate([{"$project" : {"grade" : {"$cond" :
   ["$teachersPet", 100,
      {"$add" : [
         {"$multiply" : [.1, "$attendanceAvg"]},
         {"$multiply" : [.3, "$quizzAvg"]},
         {"$multiply" : [.6, "$testAvg"]}
      ]}
   ]}
} }])
```

$group

$group documents by certain keys and calculate aggregate values

Examples:

- Context: minute-by-minute weather measurements.
  Query: Average humidity per day
- Context: student collection
  Query: Group students by grade
- Context: User collection
  Query: Group users by city and state

The fields you want to group on are the keys to the group

- {"$group" : {"_id" : "$day"}}
- {"$group" : {"_id" : "$grade"}}
- {"$group" : {"_id" : {"state" : "$state", "city" : "$city"}}}

$group - aggregation operators

In addition to the keys on which to group, you can specify one or more operations to calculate aggregate values

- "**$sum**" : value
  Produces the sum of the values
- "**$avg**" : value
  Produces the average of the values

Example

- ```
  db.sales.aggregate([{"$group" : {
     "_id" : "$country",
     "totalRevenue" : {"$avg" : "$revenue"},
     "numSales" : {"$sum" : 1}
  } }])
  ```
  ```

$group - aggregation operators

There are four operators to get the "extremes" of the dataset:

- "**$max**" : *expr ;* **"$min**" : *expr*
  Return respectively the maximum and the minimum value found
- "**$first**" : *expr ;* "**$last**" : *expr*
  They examine only the first and last document to return the value found

Examples

- ```
  db.scores.aggregate([{"$group" : {
     "_id" : "$grade",
     "lowestScore" : {"$min" : "$score"},
     "highestScore" : {"$max" : "$score"}
  } }])
  ```

```
db.scores.aggregate([
   {"$sort" : {"score" : 1} },
   {"$group" : {
      "_id" : "$grade",
      "lowestScore" : {"$first" : "$score"},
      "highestScore" : {"$last" : "$score"}
   }
}])
```
```

$group - aggregation operators

Return an array with the values found in each group

- "**$addToSet**" : *expr*
  Return an array with all distinct values
- "**$push**": *expr*
  Return an array with all found values, even duplicates

Example: returns the distinct products sold on each day

- ```
  db.sales.aggregate([{"$group" : {
     "_id" : { day : { $dayOfYear: "$date"}, year: { $year: "$date" } },
     "itemsSold" : { $addToSet: "$item" }
  } }])
  ```
  ```

$unwind

$unwind flattenS an array, building as many documents as there are elements in the array

- {
  _id: 1,
  categories: ['A','B','C']
  }

{ _id: 1, categories: 'A' },
{ _id: 1, categories: 'B' },
{ _id: 1, categories: 'C' }

Useful for projections and aggregations on the internal elements of arrays

Example: returns Mark's comments

```
db.blog.aggregate([
   {"$project" : {"comments" : "$comments"} },
   {"$unwind" : "$comments"},
   {"$match" : {"comments.author" : "Mark"} }
])
```

$unwind

Example: returns the average of the votes by city

- { _id: 1, nome: "Enrico", città: "Cesena", voti: [1, 2, 3],
  { _id: 2, nome: "Lorenzo", città: "Cesena", voti: [4, 5, 6] },
  { _id: 3, nome: "Matteo", città: "Trieste", voti: [7, 8, 9] }
- db.col.aggregate([
  {"$unwind" : "$voti"},
  {"$group" : {"_id" : "$città", "mediaVoti": { "$avg" : "$voti" } } },
  ])

If the array contains another array, you can cascade the $unwind (first on the outer array, then on the inner array)

$unwind

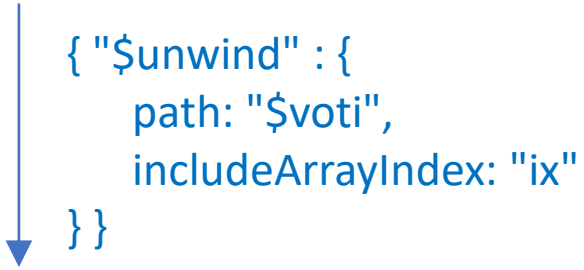
$unwind can also be declared as an object to indicate (in addition to the field to be flattened) some optional parameters

- $unwind: {
  path: <field path>,
  includeArrayIndex: <string>,
  preserveNullAndEmptyArrays: <boolean>
  }
- **path** to the array (as in the simple version)
- **includeArrayIndex** is the name of a new field where you want to extract the positional index of the array
- **preserveNullAndEmptyArrays**, if set to true, allows you to return a document even if the indicated array does not exist (or is null or empty)

$unwind

Example

- { _id: 1, nome: "Enrico", città: "Cesena", voti: [1, 2, 3],
  { _id: 1, nome: "Lorenzo", città: "Cesena", voti: [4, 5, 4] }



- { _id: 1, nome: "Enrico", città: "Cesena", voti: 1, ix: 0 },
  { _id: 1, nome: "Enrico", città: "Cesena", voti: 2, ix: 1 },
  { _id: 1, nome: "Enrico", città: "Cesena", voti: 3, ix: 2 },
  { _id: 2, nome: "Lorenzo", città: "Cesena", voti: 4, ix: 0 },
  { _id: 2, nome: "Lorenzo", città: "Cesena", voti: 5, ix: 1 },
  { _id: 2, nome: "Lorenzo", città: "Cesena", voti: 6, ix: 2 }

$sort, $limit e $skip

$sort, **$limit** and **$skip** work as in find

- To sort many of documents, sort ASAP along the pipeline and have an index in the field
- It is possible to sort on the fields created along the pipeline

Example

```
db.employees.aggregate([
   { "$project" : {
      "compensation" : { "$add" : ["$salary", "$bonus"] },
      "name" : 1
   } },
   { "$sort" : {"compensation" : -1, "name" : 1} }
])
```

$lookup

$lookup (Introduced since version 3.2) performs the left outer join between collections residing in the same database

- A new array field is created in the "primary" collection, containing any corresponding documents in the "secondary" collection.

Syntax:

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

$lookup

orders

- { "_id" : 1, "item" : "abc", "price" : 12, "quantity" : 2 }
- { "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1 }
- { "_id" : 3 }

inventory

- { "_id" : 1, "sku" : "abc", description: "product 1", "instock" : 120 }
- { "_id" : 2, "sku" : "def", description: "product 2", "instock" : 80 }
- { "_id" : 3, "sku" : "ghi", description: "product 3", "instock" : 60 }
- { "_id" : 4, "sku" : "jkl", description: "product 4", "instock" : 70 }
- { "_id" : 5, "sku": null, description: "Incomplete" }
- { "_id" : 6 }

$lookup

Example

- db.orders.aggregate([{
  $lookup: {
  from: "inventory",
  localField: "item",
  foreignField: "sku",
  as: "inventory_docs"
  }
  }])

```
{
 "_id" : 1,
  "item" : "abc",
 "price" : 12,
 "quantity" : 2,
 "inventory_docs" : [
  { "_id" : 1, "sku" : "abc",
    description: "product 1", "instock" : 120 }
 ]
}
..
{
 "_id" : 3,
 "inventory_docs" : [
  { "_id" : 5, "sku" : null, "description" : "Incomplete" },
  { "_id" : 6 }
 ]
}
```

Thank you for your attention!