POLITECNICO
MILANO 1863

Documental Database - MongoDB

Andrea Tocchetti
andrea.tocchetti@polimi.it

MongoDB - Introduction

MongoDB is a document-oriented database that stores data within **Collections** as **Documents**.

Documents consist of key-value pairs which are the basic unit of data in MongoDB.

Collections contain sets of documents. **Databases** are made by one or more collections.

MongoDB - Document Structure

```
{
  "_id":
      ObjectId("5ea64532a7678a3b27d34e00"),
  "name": {
     "first": "Mark",
     "last": "Jonhson"
  },
  "age": 37,
  "birth_date": {
     "$date": "1986-03-01T00:00:00.000Z"
  },
  "address": {
     "street": "Giuseppe Ponzio",
     "city": "Milan",
     "postal_code": "20133"
  },
  "mobile_phone_numbers": [
     {"phone_number": "+39 XXX 3456 778",
      "service_provider": "Wind"},
     {"phone_number": "+39 YYY 9874 123",
      "service_provider": "TIM"}
  ]
}
```

ObjectId

Complex Structure - Sub-document

Simple Field

Complex Field

Complex Structure - Collection

MongoDB - The ObjectId Type

ObjectId is the type associated with the predefined field created by MongoDB to uniquely identify the documents within a collection, like a **Primary Key** in a relational database.

Such field is always named **_id**.

The 12-byte **ObjectId** value consists of three different elements.

- A **4-byte timestamp value**, representing the value creation, measured in seconds since the Unix epoch.
- A **5-byte random value** generated once per process. This random value is unique to the machine and process.
- A **3-byte incrementing counter**, initialized to a random value.

MongoDB - Create Documents

A document can be added to a collection using the **InsertOne(...)** method.
Multiple documents can be added using the **InsertMany(...)** method instead.

While the first one accepts only one document, the latter may involve a list of comma-separated documents to be added to a specific collection.

```
db.people_collection.insertOne({
    "name": {"first": "Mike", "last": "Ruby"},
    "age": 63
})
```

Simple Attribute

Complex Structure - Sub-document

MongoDB - Create Indexes

Indexes are data structures that **store** a small portion of the collection's data set in an easy to traverse form, ordered by the value of the field. **Indexes** support the efficient execution of some types of queries.

Indexes are created with the **createIndex(...)** operator which accepts a list of the fields with respect to which create the index and their corresponding ordering, i.e., ascending (**1**) or descending (**-1**).

```
db.people_collection.createIndex(
    {"name.last": -1, "name.first": 1, "age": 1}
)
```

List of Fields

MongoDB - Nested Documents & Structures

Documents can contain complex structures, like **sub-documents** or even **collections** of documents. In both cases, access to these structures is achieved through the **dot notation**.

When accessing a sub-document, the chosen attribute is accessed directly. Instead, when accessing collections, the chosen attribute is accessed for each of the documents included in the collection.

```
"address": {
    "street": "Giuseppe Ponzio",
    "city": "Milan",
    "postal_code": "20133"
},
"mobile_phone_numbers": [
    {"phone_number": "+39 XXX 3456 778",
     "service_provider": "Wind"},
    {"phone_number": "+39 YYY 9874 123",
     "service_provider": "TIM"}
]
```

Complex Structure - Sub-document

Complex Structure - Collection

MongoDB - Collect Documents & Filtering

A document can be collected using the **findOne(...)** method. It collects the first document that satisfies one or more conditions defined in a **filter**.

Multiple documents can be collected using the **find(...)** method. It behaves exactly like its individual counterpart, although it collects all the documents rather than the first one.

```
db.people_collection.find(
    { "name.first": "Mark", "name.last": "Jonhson" }
)
```

Filter

Whenever it is necessary to return the number of documents collected instead of the documents themselves, the **countDocuments(...)** method can be applied.

```
db.people_collection.countDocuments(
    { "name.first": "Mark", "name.last": "Jonhson" }
)
```

Filter

MongoDB - Update Documents

A document can be updated using the **updateOne(...)** method. It collects the documents that satisfy one or more conditions defined in a **filter** and updates the first one found according to a list of comma-separated fields' updates.

Multiple documents can be updated using the **updateMany(...)** method. It behaves exactly like its individual counterpart, although it updates all the collected documents rather than just the first one.

```
db.people_collection.updateOne(
  { "age": 63},
  { "$set": {"age": 15} }
)
```

Filter

Fields' Updates

MongoDB - Delete Documents

A document can be deleted using the **deleteOne(...)** method. It collects the documents that satisfy one or more conditions defined in a **filter** and deletes the first one found.

Multiple documents can be deleted using the **deleteMany(...)** method. It behaves exactly like its counterpart, although it deletes all the collected documents.

```
db.people_collection.deleteOne(
    {"age": 15}
)
```

Filter

MongoDB - Projections

When collecting documents, it is possible to restrict, explicit, or expand the fields to be returned through **projections**. **Projections** are lists of key-value pairs made by the field name and a boolean value representing whether the field will be returned (**1**) or not (**0**).

Whenever a list specifies a subset of fields **to be** returned, the other ones won't be returned. Conversely, whenever a list specifies a subset of fields **not to be** returned, the other ones will be returned by default.

Furthermore, it is possible to shape projections to include fields from subdocuments and arrays or create new fields.

```
db.people_collection.find(
  {"name.first": "Mark", "name.last": "Jonhson"},
  {"birth_date": 1, "mobile_phone_numbers": 1,
   "year": {"$year": "birth_date"} }
)
```

Filter

New (Temporary) Field

Projection

MongoDB - Filters, Projections, and Document Collection

When performing any **find(...)** operation, it is important to notice it can only perform filters and projections in that exact order. Hence, it won't be possible to project and then filter.

```
db.people_collection.find(
  {"name.first": "Mark", "name.last": "Jonhson"},
  {"birth_date": 1, "mobile_phone_numbers": 1,
   "year": {"$year": "birth_date"} }
)
```

Filter

Projection

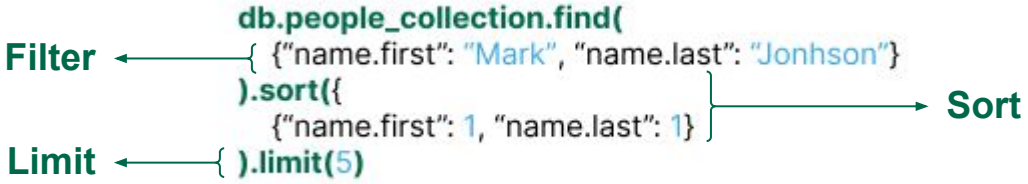
More complex operations can be performed through the **aggregation pipeline** that will be explained later.

MongoDB - Sort & Limit

When collecting documents, it is possible to sort and limit the results. These operations can be performed through the **$sort** and **$limit** stages or using the **sort(...)** and **limit(...)** methods.

The **sort(...)** method (and its equivalent stage) accepts a list of fields and their ordering, i.e. descending (**-1**) and ascending (**1**). The earlier a field is referenced, the more relevant it is for the ordering.

The **limit(...)** method (and its equivalent stage) accepts a number representing the number of elements to collect.



MongoDB - Query Operators

When collecting documents, these are filtered based on conditions evaluated on the value of their fields. Several types of operators can be employed in filtering stages, in particular:

- **Logical Query Operators** – Operators that return documents based on expressions evaluated as true or false.
- **Comparison Query Operators** – Operators that return documents based on value comparisons.
- **Element Query Operators** – Operators that return documents based on field existence or type.
- **Evaluation Query Operators** – Operators that return documents based on evaluations of individual fields or documents.

MongoDB - Logical Query Operators

MongoDB supports multiple different Logical Query Operators, namely:

- **$and** – returns documents that match all the conditions of multiple query expressions.
- **$not** – returns documents that do not match the conditions of a query expression.
- **$nor** – returns documents that do not match at least one condition of multiple query expressions.
- **$or** – returns documents that match at least one condition of multiple query expressions.

Logical Query Operator

```
db.people_collection.find({
  "$or": [
    {"name.first": "Mark", "name.last": "Jonhson"},
    {"age": 37}
  ]
})
```

Conditions

MongoDB - Comparison Query Operators

MongoDB supports multiple Comparison Query Operators, namely:

- **$eq** – matches values equal to a specified value.
- **$gt** (**$gte**) – matches values greater (greater or equal) than a specified value.
- **$lt** (**$lte**) – matches values smaller (smaller or equal) than a specified value.
- **$in** – matches any of the values specified in an array.
- **$ne** – matches values not equal to a specified value.
- **$nin** – matches values not contained in a specified array.

Nested Logical Query Operator

```
db.people_collection.find({
  "$or": [
    "$and": [
      {"age": {"$gt": 19} },
      {"age": {"$lte": 31} } }
    ],
    {"age": {"$nin": [40, 41, 42, 44]}
  ]
})
```

Comparison Query Operator

MongoDB - Element Query Operators

MongoDB supports a few Element Query Operators, namely:

- **$exists** – matches documents with a specified field.
- **$type** – matches documents whose chosen field is of a specified type.

```
db.people_collection.find(
  "$and": [
    {"birth_date": {"$exists": true} },
    {"birth_date": {"$type": "date"} }
  ])
```

Logical Query Operator

Element Query Operator

MongoDB - Evaluation Query Operators

MongoDB supports multiple Evaluation Query Operators, namely:

- **$text** – matches documents based on text search on indexed fields.
- **$regex** – matches documents based on a specified regular expression.
- **$where** – matches documents based on a JavaScript expression.

```
db.people_collection.find(
    {"name.first": {"$regex": "[A-Z][a-z]+"} }
)
```

Evaluation Query Operator

MongoDB - Querying Nested Documents

Filtering operations may behave differently based on the type of complex field a query is accessing (e.g., subdocuments, arrays, etc.).

Queries evaluating one or more conditions on the fields of a **subdocument** field are not subject to any particular behaviour change.

On the other hand, queries evaluating a **single** condition on the fields of the documents of an **array** will return the **main document** if **at least one** of the documents in the array satisfies the condition.

```
db.people_collection.find(
    {"mobile_phone_numbers.service_provider": "Wind"}
)
```

→ **Single Condition**

MongoDB - Querying Nested Arrays

Whenever **multiple** conditions are evaluated on the documents in an **array** field, they will be assessed individually on the array's documents, hence returning the main document if, **for each condition**, there exists **at least one** document that satisfies it. It doesn't matter whether there's **only one** document satisfying all conditions or **multiple** documents satisfying one each.

```
db.people_collection.find(
   {"mobile_phone_numbers.service_provider": "Wind",
    "mobile_phone_numbers.phone_number": "+39 YYY 9874 123"}
)
```

Multiple Conditions

MongoDB - Querying Nested Arrays

Whenever a query is targeted at evaluating **multiple** conditions on the fields of the **same** document of an **array**, it is necessary to apply the **$elemMatch** stage. In particular, it matches documents containing an array field with **at least one** document that satisfies **all** the specified query criteria.

```
db.people_collection.find(
  {"mobile_phone_numbers": {
    "$elemMatch": {
      "service_provider": "TIM"
      "phone_number": "+39 YYY 9874 123"
    }
  }
})
```

Multiple Conditions

MongoDB - Unwind

When a collection is made of documents containing arrays, retrieving the array's content may be useful. Applying the **$unwind** stage can achieve such an outcome.

It shapes the collection so that **each** document is replaced with a set of new ones, i.e., **one for each element in the document's array** on which the unwind stage is applied. These new documents contain all the fields from the main one and a field with the name of the array field that contains one of its documents.

When applying **$unwind** or **$group** stages (explained later), it is necessary to apply the **aggregate(...)** method, i.e., a method to compute aggregate values for the documents in a collection.

Aggregate Operator

```
db.people_collection.aggregate([
    {"$unwind": {"path": "$mobile_phone_numbers"} }
])
```

Unwind Stage

MongoDB - Aggregations

Aggregate operations, i.e., operations aimed at grouping with respect to one or more fields, are achieved by applying the **$group** stage within the **aggregate(...)** method. Such a stage requires defining the list of fields to perform the aggregation and the aggregation functions to be applied.

Whenever a **$group** stage is applied, only the fields used to perform the aggregation or created by it will be available in the next stages. MongoDB supports many aggregate functions, e.g., **sum**, **avg**, **min**, **max**, etc.

```
db.people_collection.aggregate([
  {"$unwind": {"path": "$mobile_phone_numbers"} },
  {
    "$group": {
      "_id": {"provider": "$mobile_phone_numbers.service_provider"},
      "phone_numbers_per_provider": {"$sum": 1} }
    }
  }
])
```

Grouping Fields

Aggregated Outcome

MongoDB - Aggregations

Whenever a grouping operation is to be performed on the whole dataset, it is possible to apply a **dummy _id** in the **$group** stage. It's enough to set the _id to **true** or a **fixed value**. The latter is why the **$** in the grouping stage is important!

```
db.people_collection.aggregate([
  {"$unwind": {"path": "$mobile_phone_numbers"} },
  {
    "$group": {
      "_id": true,
      "phone_numbers_per_provider": {"$sum": 1}
    }
  }
])
```

```
db.people_collection.aggregate([
  {"$unwind": {"path": "$mobile_phone_numbers"} },
  {
    "$group": {
      "_id": {"provider": "mobile_phone_numbers.service_provider"},
      "phone_numbers_per_provider": {"$sum": 1}
    }
  }
])
```

MongoDB - Aggregations

Whenever an aggregation pipeline is to be employed, it is necessary to explicitly specify all the different pipeline stages (e.g., filtering, projections, etc.). In particular, besides the previously explained **$group**, **$unwind**, **$sort**, and **$limit** stages, **$match** defines filters while **$project** defines projections. These stages can be applied interchangeably in the aggregation pipeline.

```
db.people_collection.aggregate([
  {"$unwind": {"path": "$mobile_phone_numbers"} },
  {
    "$group": {
      "_id": {"provider": "$mobile_phone_numbers.service_provider"},
      "phone_numbers_per_provider": {"$sum": 1}
    }
  },
  {"$match": {"phone_numbers_per_provider": {"$gt": 10} },
  {"$sort": {"phone_numbers_per_provider": -1} },
  {"$project": {"phone_numbers_per_provider": 0} }
])
```


ANY
Questions?