Data Management

2023/2024

MongoDB

Applications of Document-based DBs

SAPIENZA
UNIVERSITÀ DI ROMA

Tutor
Roberto Maria Delfino

delfino@diag.uniroma1.it

MongoDB

Open source NoSQL Database Management System based on aggregates (**documents**).

Tools used today:

- MongoDB Community Server
- MongoDB Compass (GUI client)
- pymongo Python library

JSON vs MongoDB BSON

In MongoDB documents are represented in **BSON** (Binary JSON).

BSON maintains the JSON structure of documents based on key-value pairs, but it adds support for additional data types, such as:

`Date, Timestamp, ObjectId, Double, Long int, Regular expressions.`

Example:

```
{
    _id: ObjectId("5099803df3f4948bd2f98391"),
    name: { first: "Alan", last: "Turing" },
    birth: new Date("1912-06-23"),
    death: new Date("1954-06-07"),
    contribs: [ "Turing machine", "Turing test", "Turingery" ],
    views : NumberLong("1250000")
}
```

Datatypes

The shell of MongoDB (**mongosh**) treats numbers as 64-bit floating-point double values by default. If you write an integer in the shell, it is internally stored as a double.

NumberLong returns 64-bit signed integers. To avoid having Mongo interpreting a number as a float, the value has to be encapsulated into double quotes, (e.g., `NumberLong("200")`.

NumberInt acts similarly, but it returns 32-bit signed integers.

NumberDecilmal returns a 128-bit decimal-based floating point values. This might come up useful when high precision is needed (e.g., scientific calculations), since it allows to emulate decimal rounding with exact precision.

Datatypes

The `Date` datatype is stored as a 64 bit integer representing the number of milliseconds since the Unix epoch (Jan 1, 1970). This allows to represent every timestamp in a range of ~585 million years.

The `ObjectId` is a 12-byte BSON type, obtained as follows:

- 4 bytes representing the seconds since the Unix epoch
- 5 bytes representing a (pseudo)random value
- 3 bytes counter, starting with random value

You can generate an ObjectId for a document with `ObjectId()`, which can also take a hexadecimal or an integer value.

Managing databases

There is no shell create command allowing to create a new database. Instead, a database can be created when referenced by means of the command

```
>use mydatabase
```

The list of databases can be accessed via the commands

`>show dbs` or `>show databases`

Notice: MongoDB actually creates a database only when data is inserted in it.

Collections

In MongoDB documents are grouped into **collections**. To make a comparison with the relational model, collections play a role which is similar to that of tables.

A database can contain multiple collections. Each collection contains documents.

As a NoSQL system MongoDB does not enforce a real **schema**, but a collection should typically contain documents sharing similar structures due to similar or related purposes.

Collections can be created in mongosh with the command

```
db.createCollection(“mycollection”)
```

A collection is implicitly created when first referenced.

Capped Collections

In some cases, we might want to limit the size of a collection in terms of either the memory space it takes, or in terms of the number of documents it contains. This constraints can be expressed by creating **capped collections**:

```
db.createCollection("col", {capped: true, size: 6142800, max: 10000})
```

In the capped collection "`col`", documents are automatically overwritten according to a FIFO policy when it reaches the maximum size of 6142800 bytes or the maximum number of 10000 documents. Size has priority over number limit and it is mandatory for capped collections.

Capped collections do not allow for the deletion of single documents and they can cause the failure of update operations if they make the collection exceed the limits.

Documents

A document in MongoDB has a fixed size of 16MB. If you need to store larger data, like media, you should use GridFS instead.

Every document must obey the following rules on the names of its fields:

- The field `_id` is reserved and it is used as a **primary key**. Its value must be **unique in the collection** and it can be of any type other than array.
- Field names should not start with the `$` character.
- Field names should not contain the `.` character.

Every document must have a primary key, which is automatically created and added as an `_id` field with a corresponding `ObjectId` as value if not explicitly specified

“Schema” of a schemaless system

Despite the notion of schema as intended in RDBMSs is not present in MongoDB, the system still offers ways to define some collection-wise constraints:

- **Schema Validation** allows to define some constraints on the structure of the documents belonging to a given collection, like type constraints, mandatory fields, etc.

- **Unique Indexes** give the possibility to create B-trees index structures over the desired fields.

Schema Validator

The schema constraints can be specified through the $jsonSchema operator

```
db.createCollection("students", {
    validator: { $jsonSchema: {
    required: [ "name", "year", "major", "address" ],
        properties: {
            name: { bsonType: "string" },
            year: { bsonType: "int",
                        minimum: 1900,
                        maximum: 3017 },
             major: { enum: [ "Math", "English", "CS", "History", null ] },
             address: { bsonType: "object",
                        required: [ "city" ],
                             properties: {
                             street: { bsonType: "string" },
                             city:   { bsonType: "string" } } } } } } } )
```

Unique Indexes

By default, an index exists for each collection based on the `_id` field.

Nonetheless, it is possible to create additional indexes (B-trees) with the command

```
db.mycollection.createIndex({field1: 1, field2: 1})
```

In the previous example an index is created in ascending order (-1 for descending order) on `field1` and `field2`.

It is possible to specify uniqueness for an index (thus acting as a primary key in RDBMS)

```
db.mycollection.createIndex({field: -1}, {unique: true)
```

The index over the `_id` field is unique.

SQL to MongoDB mapping chart

SQL	MongoDB
database	database
table	collection
row	document
column	field
index	index
primary key	unique index
table join	embedded documents and linking/lookup
aggregation (e.g., group by)	aggregation pipeline

CRUD operations: Create

Documents can be written in a collection by means of the following commands:

```
db.mycollection.insertOne(<document>)
```

or

```
db.mycollection.insertMany([<d1>, <d2>, …, <dn>])
```

Example:

```
db.books.insertOne(
    { item: "Divine Comedy",
     author: "Dante Alighieri"
    } )
```

If the collection does not exists, it is created.

CRUD Operations: Read

Documents are retrieved through the `find` method.

```
db.mycollection.find(<criteria>, <projection>)
```

Two optional parameters can be specified:

- `<criteria>`: it is a document specifying the selection criteria by means of query operators (it acts as the WHERE clause in SQL). Omitting criteria makes the method retrieve all documents in the collection.

- `<projection>`: it allows to project the fields that we want to retrieve (similarly to the SELECT clause in SQL). Omitting projection parameters makes the method return all the fields for each document corresponding to the criteria.

CRUD Operations: Read - Examples

```
db.products.find({ qty: { $gt: 25 })
```

Find all the documents where the value for the field `qty` is greater than 25.

```
db.products.find({ qty: { $gt: 25}}, {item: 1, qty: 1})
```

Find the same documents, but return only the `item`, the `quantity` and the `_id` (implicit)

```
db.products.find({qty: { $gt: 25}}, {_id: 0,  qty: 0})
```

Find the same documents and return all fields but `_id` and `qty`.

CRUD Operations: Update

The methods `updateOne` and `updateMany` allow to modify existing document (or even to create them, with the parameter `upsert`).

```
db.mycollection.updateOne(<query>, <update>, {upsert: <>})
```

The `<query>` parameter is similar to the `<criteria>` parameter used in the find methods, in that is used to specify which are the documents of interest by means of query operators.

The `<update>` parameter is used to specify how to modify the documents.

The `upsert` parameter (false by default) can be specified as true to insert the document if it does not exist.

CRUD Operations: Update - Examples

```
db.books.updateOne({item: “Divine Comedy”},
                   { $set: {price: 18}, $inc: {stock: 5}}
```

This operations set the value of the price field to 18, and increments the value of the stock field by 5.

```
db.books.updateOne({item:”Divine Comedy”},
                   {item: “Divina Commedia”,
                    author: “Dante Alighieri”,
                    price: 18,
                    stock: 5)
```

The document can be modified even by explicitly declaring the new values.

CRUD Operations: Delete

Documents can be deleted from collections by using the following commands:

```
db.mycollection.deleteOne(<query>)
```
or
```
db.mycollection.deleteMany(<query>)
```

The `<query>` parameter specifies the document(s) to remove. If no matching parameter is specified, all documents are going to be deleted.

```
db.mycollection.deleteMany({salary: {$lt: 180}})
```

Removes from the collection all the documents having 180 or less as value for the salary field.

Aggregation pipeline

Aggregations are operations that process documents and return computed results. Such operations are executed by means of several operators which MongoDB provides.

Aggregation operators can be combined in what is called an **aggregation pipeline**.

An aggregation pipeline is constituted by a sequence of operations where the input of each intermediate stage is the output of the previous one, and its output is the input of the subsequent operation.

An aggregation is executed with the command

```
db.mycollection.aggregate([<stage1>,…,<stagen>])
```

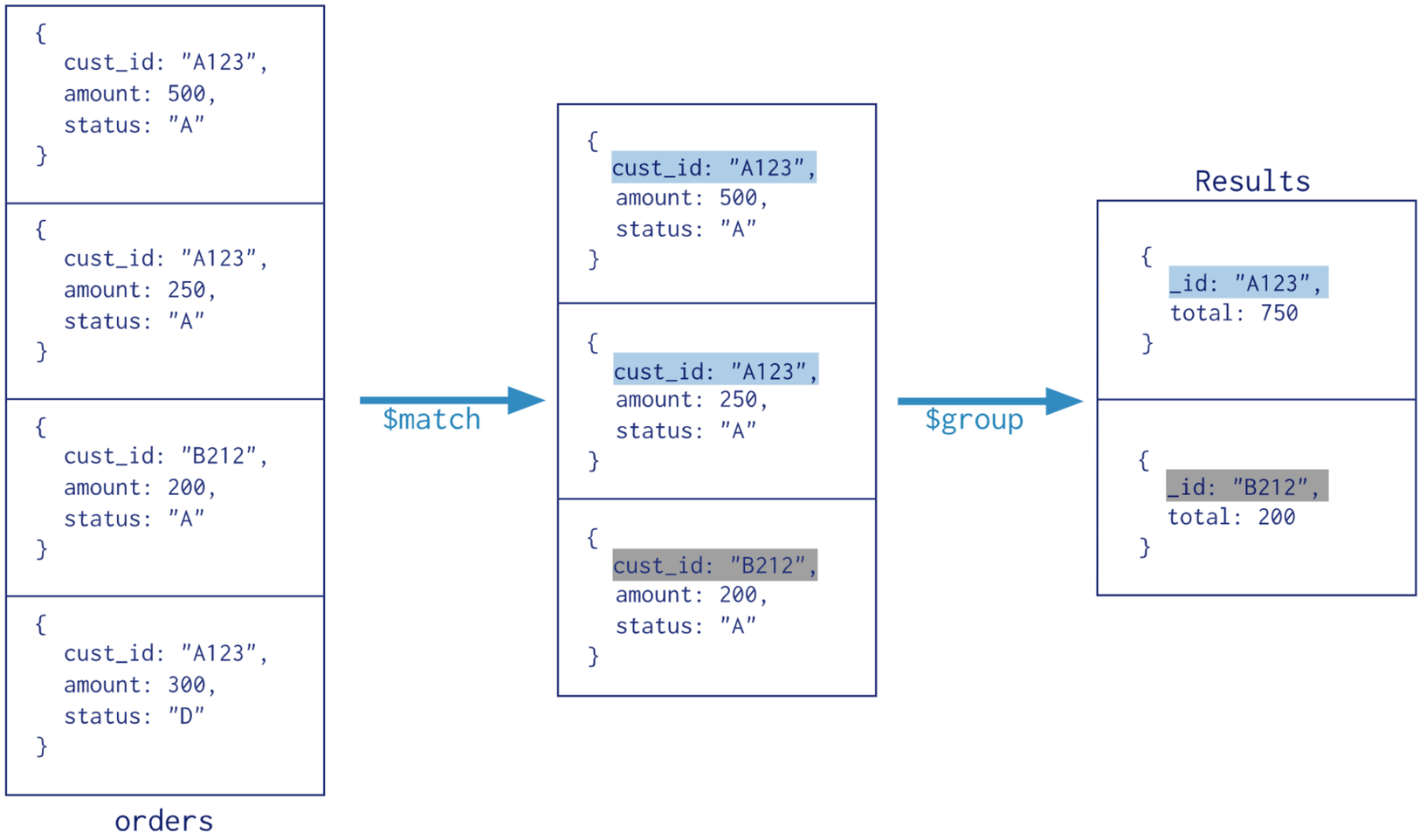
Aggregation pipeline - Example

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

The above command executes an aggregation constituted by two stages.

- `$match`: this selects all documents in the orders collection where the `status` field has value `A` and passes such documents as input for the next stage
- `$group`: it groups documents by the `cust_id` field and computes the `total` field as the sum of all the values appearing in the `amount` field.

Aggregation pipeline - Example



Aggregation as means for JOINs

Assume the following two documents have been added to their respective collections

`{_id: 7, name:”John Lennon”, city:”NY”}` in the collection people

`{_id: 3, cityName:”NY”, state:”New York”}` in the collection places

We can join data from the two collection as follows:

```
db.people.aggregate([
      {$lookup:
        { from:"places",
          localField:"city",
               foreignField:"cityName",
               as:"city_details" }}])
```

The above command produces the following document:

```
{ _id : 1.0, name : "John Lennon", city : "NY",
city_details : [ { _id : 1, cityName : "NY", state : "New York" } ] }
```

DEMO