SAPIENZA
UNIVERSITÀ DI ROMA

MongoDB

Daniele Pantaleone
Giacomo Ronconi

Corso di laurea magistrale in Data Science
2016/2017

Introduction to mongoDB

- MongoDB is an opensource NO-SQL document-oriented database system.
- Eschew the traditional table-based relational database structure in favor of JSON-like *document* format with dynamic schemas (MongoDB calls the format BSON), making the integration of data in certain types of applications easier and faster.
- It can be considered part of the Aggregate NO-SQL database system family because it allows to operate on data in units without the necessity of enforcing a general schema.
- It deals perfectly with the 3Vs describing a Big Data environment: it is capable of storing large amount of data (*volume*), processing them in a fast way (*velocity*) without the needs of having homogeneous data (*variety*).

Difference with *key-value* database systems

- MongoDB is consistent with the definition of document database system because while not enforcing a general schema, the datatypes need to be consistent with the ones of the BSON format. So, it is possible to highlight a structure of the aggregate without the need of an application doing any type of interpretation.

- On the contrary, in *key-value* database systems (like Amazon's Dynamo Database), the aggregate is just a BLOB of bits, and the application is the only responsible of data interpretation.

Data Model

Documents

- MongoDB stores data in the form of *documents*. A document is a JSON-like data structure composed of field-and-value pairs: documents are analogous to structures in programming languages that associate keys with values (e.g. dictionaries, hashes, maps, associative arrays, etc.)

- MongoDB stores document on disk in the BSON serialization format. BSON is a binary representation of JSON documents which contains more data types than JSON does.

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

A MongoDB document

Data Model

Documents

- The maximum BSON document size is **16 MB**. The maximum document size helps ensure that a single document cannot use excessive amount or RAM or, during transmission, excessive amount of bandwidth.

- Documents in MongoDB have the following restrictions on field names:

  - The field name **_id** is reserved for use as primary key: it's value must be unique in the collection and may be of any type other than an array
  - The field names cannot start with the **$** character
  - The field names cannot contain the **.** character

- Every document in MongoDB has attached a primary key: if the client sends a document without and **_id** field, MongoDB will add the **_id** field and generate the **ObjectId**.

Data Model

Collections

- MongoDB stores all documents in *collections*. A collection is the equivalent of an RDBMS table and it exists withing a single database.

- MongoDB collections do not enforce a schema (unlike RDBMS): documents within a collections can have different fields. However, typically all documents in a collection have similar or related purpose.

Data Model

Database references

- MongoDB doesn't support joins. In MongoDB some data is *denormalized* or stored with related data in documents to remove the needs for joins. However in some cases it makes sense to store related information in separate documents, typically in different collections or databases.

- MongoDB applications of of two methods for relating documents:

  - Manual references
  - DBRefs

Data Model

Database references – Manual references

- **Manual references** refers to the practice of including one document **_id** field in another one: in this way the appication can issue a second query to resolve the referenced fields as needed.

```
original_id = ObjectId()

db.places.insert({
   "_id": original_id,
   "name": "Broadway Center",
   "url": "bc.example.net"
})

db.people.insert({
   "name": "Erin",
   "places_id": original_id,
   "url":  "bc.example.net/Erin"
})
```

Example of manual reference

The only limitation of manual linking is that these references do not convey the database and collection name. If there is the need of linking documents in a collections with documents of other collections **DBRefs** can be used.

Data Model

Database references – DBRefs

- **DBRefs** are conventions for representing a document, rather than a specific reference type. They include the name of the collection, and in some cases the database, in addition to the value from the **_id** field.

```
{ "$ref" : <value>, "$id" : <value>, "$db" : <value> }
```

DBRef document format

```
{
 "_id" : ObjectId("5126bbf64aed4daf9e2ab771"),
 "creator" : {
            "$ref" : "creators",
            "$id" : ObjectId("5126bc054aed4daf9e2ab772"),
            "$db" : "users"
          }
}
```

Example of DBRef document which points to a document in the *creators* collection of the *users* database that has ObjectId("5126bc054aed4daf9e2ab772") in its **_id** field

Data Model

GridFS

- **GridFS** is a specification for storing and retrieving files that exceed the BSON document size limit of 16 MB.

- Instead of storing a file in a single document, GridFS divides a file into parts, or chunks, and stores each of those chunks as a separate document (by default GridFS limits chunk size to **256 KB**).

- GridFS store files in two collections: **chunks** to store the binary chunks, and **files** to store the file's metadata. Those collections are stored in a common bucket by prefixing them with the same bucket name: by default GridFS stores files in two collections with names prefixed by **fs** bucket.

- When you query a GridFS store for a file, the driver or client will reassemble the chunks as needed. Is it possible to access information from arbitrary sections of files, which allows you to skip into the middle of a *video* or an *audio* file.

Data Model

*GridFS – The **chunks** collection*

- Each document in the **chunks** collection represent a distinct chunk of a file as represented in the GridFS store.

```
{
  "_id" : <ObjectId>,
  "files_id" : <ObjectId>,
  "n" : <num>,
  "data" : <binary>
}
```

Prototype document from the *chunks* collection

Field	Description
_id	The unique **ObjectId** of the chunk
files_id	The **_id** of the parent document as specified in the **files** collection
n	The sequence number of the chunk (index start from 0)
data	The chunk's payload as **BSON** bynary type

Data Model

*GridFS – The **files** collection*

- Each document in the **files** collection represents a file in the GridFS store.

- Documents in the files collection contain some or all of the following fields. Applications may create additional arbitrary fields.

```
{
  "_id" : <ObjectId>,
  "length" : <num>,
  "chunkSize" : <num>
  "uploadDate" : <timestamp>
  "md5" : <hash>

  "filename" : <string>,
  "contentType" : <string>,
  "aliases" : <string array>,
  "metadata" : <dataObject>,
}
```

Prototype document from the *files* collection

Data Model

*GridFS – The **files** collection*

Field	Description
_id	The unique ID for this document. The **_id** is of the data type you chose for the original document. The default is BSON **ObjectId**.
length	The size of the document in bytes.
chunkSize	The size of each chunk. GridFS divides the document into chunks of the size specified here. The default size is 256 KB.
uploadDate	The date the document was first stored by GridFS. This value uses the **Date** datatype.
md5	An md5 has returned by the file md5 API. The value has a **String** type.
filename	Optional. A human-readable name for the document.
contentType	Optional. A valid MIME type for the document.
aliases	An array of alias strings.
metadata	Optional. Any additional information you want to store.

Data Model

ObjectId

- The **ObjectId** is a 12-byte BSON type, constructed using:
  - a 4-byte value representing the seconds since the Unix Epoch
  - a 3-byte machine identifier
  - a 2-bye process id
  - a 3-byte counter, starting with random value
- In MongoDB documents stored in a collection require a unique **_id** field that acts as primary key. Since ObjectIds are small most likely unique, and fast to generate, MongoDB uses them as default value for the **_id** field if such field is not specified.

Data Model

SQL to MongoDB mapping chart

- The following table presents the various SQL terminology and concepts and the corresponding MongoDB terminology and concepts:

SQL Term/Concept	MongoDB Term/Concept
database	database
table	collection
row	document
column	field
index	index
table joins	embedded documents and linking
primary key	primary key
aggregation (e.g. group by)	aggregation pipeline

CRUD Operations

Insert documents

- In MongoDB, the db.collection.insert() method adds new documents into a collection. In addition, both db.collection.update() and db.collection.save() methods can also add new documents through an operation called *upsert*. The *upsert* is an operation which performs either an update of an existing document or an insert of a new document if the document to modify does not exists.

- The **db.collection.insert()** method has the following behavior:

  - If the collection doesn't exists, the **insert()** method will create the collection.
  - If the document doesn't specify an **_id** field, then MongoDB will add the **_id** field and assign a unique ObjectId for the document before inserting.
  - If the document specifies a new field then the **insert()** method inserts the document with the new field. This requires no changes to the Data Model for the collection or the existing document.

```
db.collection.insert(<document>)
```

Document *insert* function prototype

CRUD Operations

Update documents

- The **db.collection.update()** method modifies an existing document or documents in a collection. The method can modify specific fields of an existing document or documents, or replace an existing document entirely, depending on the update parameter.

- By default the **update()** method updates a **single** document. If the **multi** option is set to **true**, the method updates all the documents matching the **query** criteria.

- If the upsert parameter is set to **true**, the **update()** method creates a new document when no document matches the **query** criteria.

```
db.collection.update(
  <query>,
  <update>,
  { upsert: <boolean>, multi: <boolean> }
)
```

Document *update* function prototype

CRUD Operations

Update documents

- If the **<update>** document contains update operator expressions, such those using the **$set** operator, then:
  - The **<update>** document must contain only update operator expressions
  - The **update()** methodupdates only the corresponding fields in the document

```
db.books.update(
  { item: "Divine Comedy" },
  { $set: { price: 18 }, $inc: { stock: 5 } }
)
```

Example of document *update:* this will update a document in the *books* collection *changing* the **price** field to 18 and *incrementing* the **stock** field by 5 units

- If the **<update>** document contains only **field:value** expressions, then:
  - The **update()** method replaces the matching document with the **<update>** one
  - The **update()** method cannot update multiple documents

CRUD Operations

Save documents

- The **db.collection.save()** method updates an existing document or inserts a new document, depending on it's document parameter.

- If the document doesn't contain an **_id** field, then the **save()** method performs an **insert()**. During the operation, the **ObjectId** is generated and assigned to the **_id** field.

- If the document contains an **_id** field then the save performs an upsert, querying the collection on the **_id** field. If a document doesn't exist with the specified **_id** value, then the **save()** method performs an **insert()**. If a document exists with the specified **_id** value, then the **save()** method performs an **update()** that replaces all the fields in the existing document with the fields from the given one.

```
db.collection.save(<document>)
```

Document *save* function prototype

CRUD Operations

Remove documents

- The **db.collection.remove()** method, removes documents from a collection. Is it possible to remove all documents from a collection, removes documents matching a given condition, or limit the operation to remove just a single document.

- If the **remove()** method is invoked without specifying query parameters, all the documents of the collection will be removed (it doesn't remove the indexes). If this is the case, it may be more efficient to use the **db.collection.drop()** method, which drops the entire collection, including indexes and then recreate the collection and rebuild the indexes.

```
db.collection.remove(<query>, justOne: <boolean>)
```

Document *remove* function prototype

```
db.inventory.remove({type: "food"})
```

Document *remove* function which removes all documents from the **inventory** collection where the **type** field equals **food**

CRUD Operations

Query documents

- The **db.collection.find()** method retrieves documents from a collection: it returns a cursor which can be used to iterate over the retrieved documents.

- The method specifies two optional parameters:

  - **criteria**: a document which specifies selection criteria using Query Operators. To return all documents within a collection, omit this parameter or pass an empty document.
  - **projection**: a document which specifies the fields to return using Projection Operators. To return all fields in the matching document, omit this parameter. The projection document format is of the type **{ <field> : <boolean>, ... }** where the boolean value indicates whether to return or not the specified field.

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

Document *find* function prototype

CRUD Operations

Query documents

```
db.products.find()
```

Example of *find* function usage which retrieves all the documents within a collection

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

Example of *find* function which retrieves all the documents in the *products* collection whose **qty** field value is greater than **25**

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

Example of *find* function which retrieves all the documents in the *products* collection whose **qty** field value is greater than **25**: only the **_id**, **item** and **qty** fields will be returned

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

Example of *find* function which retrieves all the documents in the *products* collection whose **qty** field value is greater than **25**: The **_id** and **qty** fields will be excluded from the result set

Aggregation

Introduction

- *Aggregations* are operations that process data records and return computed results. MongoDB provides a rich set of aggregation operations that examine and perform calculations on the data sets.

- Like queries, aggregation operations in MongoDB use collection of documents as input, and return results in the form of one or more documents.

- MongoDB provides two Aggregation Modalities:

  - **Aggregation Pipeline**
  - **Map Reduce**

Aggregation

Aggregation Pipeline

- The *aggregation pipeline* is a new framework for data aggregation modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated results.

- The aggregation pipeline provides an alternative to *map-reduce* and may be the preferred solution for many aggregation tasks where the complexity of map-reduce may be unwarranted.

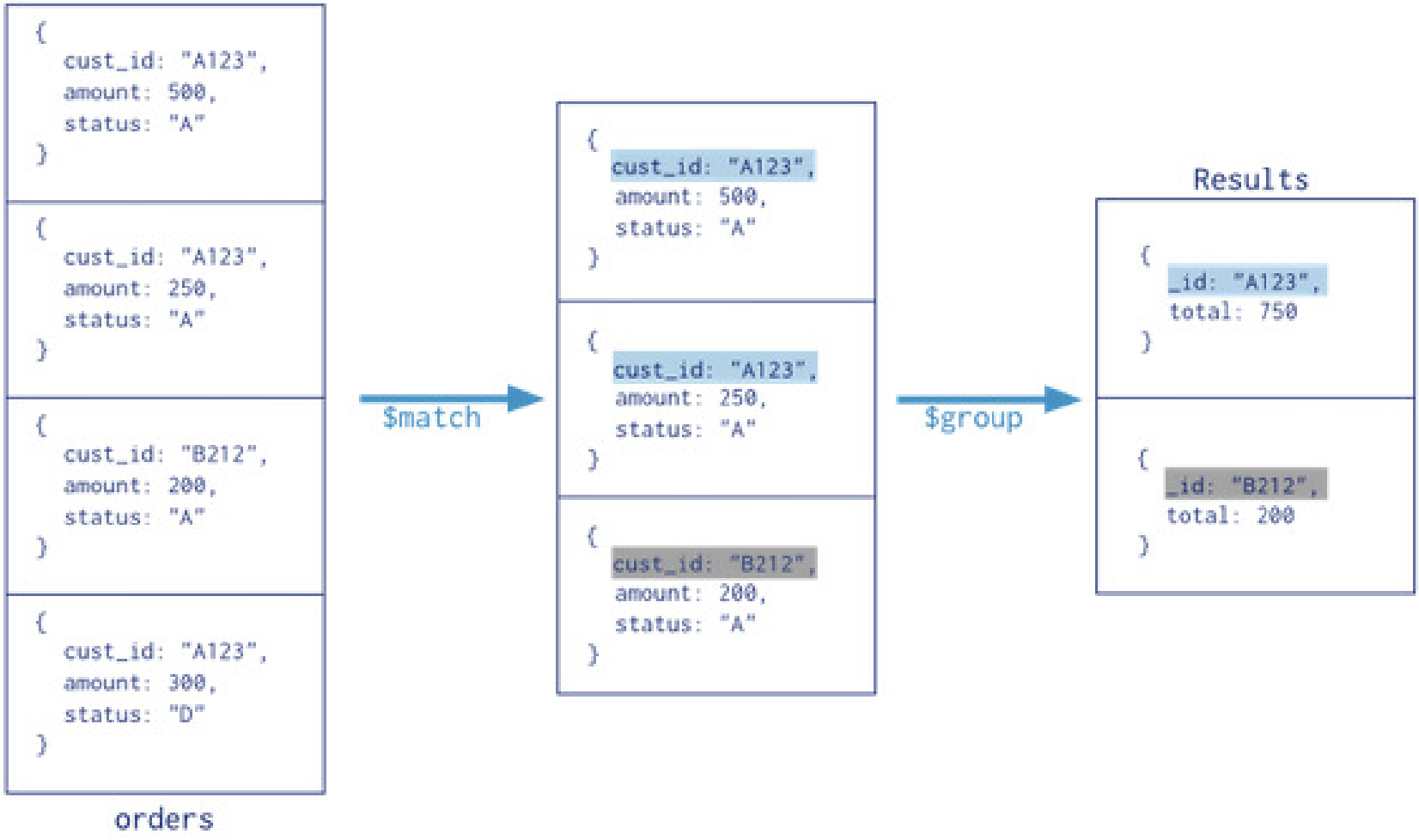
- There are two main phases in order to do the aggregation:

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

Example of *aggregate* function which retrieves all the documents in the *orders* collection whose **status** field value is equal to **A**; the selected documents are then grouped by **_id** and **total** is computedsumming all the **amount** fields of the documents having the same **_id**

Aggregation

Aggregation Pipeline



Aggregation

Aggregation Pipeline

- The *aggregation pipeline* starts processing the documents of the collection and pass the result to the next *Pipeline Operator* in order to get the final result.
- The *operators* can filter out documents (e.g. **$match**) generate new documents (e.g. **$group**) computing the result from the given ones.
- The same *operator* can be used more than once in the pipeline.
- Each *operator* takes as input a pipeline expression that is a document itself containing:
  - Fields
  - Values
  - Operators
- The *aggregate* command operates on a single collection.
- Hint: use *match* operator at the beginning of the pipeline.

Aggregation

Map-Reduce

- Map-Reduce is a paradigm to manage big data in aggregated results.
- Supported by MongoDB with the command *mapReduce*
- *mapReduce* need two functions and an object:
  - A function for the map phase: emit (key,value) pairs
  - A function for the reduce phase: apllied for keys with multiple values
  - And an object for the query and the output
- Map-Reduce is implemented through javascript calls

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

Aggregation

Single Purpose Aggregation Operations

- MongoDB provides a set of specific operations for aggregation:
  - **count**: return the number of document that match a query
  - **distinct**: takes a number of documents that match a query and returns all of the unique values for a field in the matching documents.
  - **group**: takes a number of documents that match a query, and then collects groups of documents based on the value of a field or fields. It returns an array of documents with computed results for each group of documents.

Indexes

Introduction

- Indexes are used by MongoDB to answer more efficiently queries.
- Without indexes MongoDB have to scan the whole collection.
- The idea is very similar to indexes in RDB: indexes are B-trees at the collection level.

Indexes

Indexes Types

- **Default _id:**
  - All MongoDB collections have an index on the **_id** field that exists by default. If applications do not specify a value for **_id** MongoDB will create an **_id** field with an **ObjectId** value.
  - The **_id** index is unique, and prevents clients from inserting two documents with the same value for the **_id** field.
- **Single Field:**
  - User defined index on a single field of a document.
- **Compound Index:**
  - User defined index on multiple fields.

Indexes

Indexes Types

- **Multikey Index:**
  - Used to index the content stored in an array. Allow you to make queries matching the elements of an array.
- **Geospatial Index:**
  - Support queries of geospatial coordinate data.
- **Text Indexes:**
  - Support queries of string skipping language specific stop-words.
- **Hashed Indexes:**
  - Index the hash of the value of a field to increase randomity of distribution.