Data Management for Data Science

Master of Science in Data Science
Facoltà di Ing. dell'Informazione, Informatica e Statistica
Sapienza Università di Roma

AA 2018/2019

a MongoDB overview

Domenico Lembo
Dipartimento di Ingegneria Informatica,
Automatica e Gestionale A. Ruberti

JSON vs. MongoDB BSON

- MongoDB represents JSON documents in binary-encoded format called BSON
- while keeping JSON’s essential key/value pair nature, BSON adds support for a number of additional data types, such as
  - Date, Timestamp, ObjectId, Double, Regular expressions, JavaScripts, Int (32-bit integer), Long int (64-bit integer)

```
{
    _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")
}
```

A MongoDB document

MongoDB Datatypes hints

- Internally, **`Date`** objects are stored as a 64 bit integer representing the number of milliseconds since the Unix epoch (Jan 1, 1970), which results in a representable date range of about 290 millions years into the past and future.

- **`NumberLong`** is a 64 bit signed integer. You must include quotation marks or it will be interpreted as a floating point number, resulting in a loss of accuracy (that is, when retrieving an unquoted numberlong, its value might be changed, whereas quoted numbers retain their accuracy)

➢ 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-byte process id
  - ➢ a 3-byte counter, starting with random value

  To generate a new ObjectId *through shell commands*, you can use `ObjectId()` (to get a random value), or `ObjectId("v")`, where v is a hexadecimal value.

Collections

- MongoDB stores all documents in *collections*. A collection is the equivalent of an RDBMS table and it exists within a single database.
- MongoDB collections do not enforce a schema (unlike RDBMSs): documents within a collections can have different fields. However, typically all documents in a collection have similar or related purpose.
- The command `db.createCollection("mycollection")`, creates a collection named `mycollection` with default parameters.
- The command

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

  creates a collection with an index on the field `_id`, which is capped i.e., it is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size, with a maximum size of `6142800` bytes (If capped is true, then the size has to be specified), and such that 10000 is the maximum number of documents allowed in the collection.

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 the `_id` field, MongoDB will add the `_id` field and generate the `ObjectId`.

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
primary key	primary key
table joins	embedded documents and linking / lookup (see next slides)
aggregation (e.g. group by)	aggregation pipeline (see next slides)

MongDB – getting started

- MongDB comes with a command-line environment to create and manipulates databases, collections, documents, etc. To start this interpreter launch the `mongo` command from a shell (in a different shell start the server with the command `mongod`)
- Some useful commands are
  - `show dbs` to show databases managed by the server
  - `use` *`myDB`* to use a database. If the mentioned database does not exist, it will be created after creating some content in it.
  - `db.createCollection("mycol")`, creates a collection named `mycol` with default parameters
  - `show collections` to show all collections in the database in use
  - `db.`*`mycol`*`.drop()` to eliminate a collection (named *`mycol`*) and all the associated indexes and documents

CRUD Operations
Insert documents

- In MongoDB, the `db.mycol.insert()` method adds new documents into a collection (*mycol*). In addition, both `db.mycol.update()` and `db.mycol.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 to the document before inserting.

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

Document *insert* function prototype

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

Example

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` (`false` is the default), 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 (simplified)

CRUD Operations
Update documents

- If the `<update>` document contains update operators, such as the `$set` operator, then:
  - The `<update>` document must contain only update operator expressions, which use: `$set, $rename, $unset, $mul`, etc.
  - The `update()` method updates 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>` document. The `update()` method does not replace the `_id` value.
  - The `update()` method cannot update multiple documents (the option `{multi:true}` is not allowed in this case)

CRUD Operations
Save documents

- The `db.collection.save()` method updates an existing document or inserts a new document, depending on some input parameters.
- 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. It is possible to remove all documents from a collection, remove all documents matching a given condition, or limit the operation to remove just a single document.

- If the `remove()` method is invoked with an empty document as parameter, all the documents of the collection will be removed (it doesn’t remove the indexes). The `db.collection.drop()` method, instead, drops the entire collection, including indexes.

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

Document *remove* function prototype. The default for `justOne` is `false`

```
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. The users can specify either the fields she wants to return or the fields she does not want to return (no mix of 0 and 1 is allowed)

```
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 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 documents in the *products* collection whose **qty** field value is greater than **25**: **_id** and **qty** will be excluded from the result

Data Model
Database references

- 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 offers 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

Notice that these references do not convey the database and the collection name. Their usage may thus be limited. In their place, **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

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 a 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

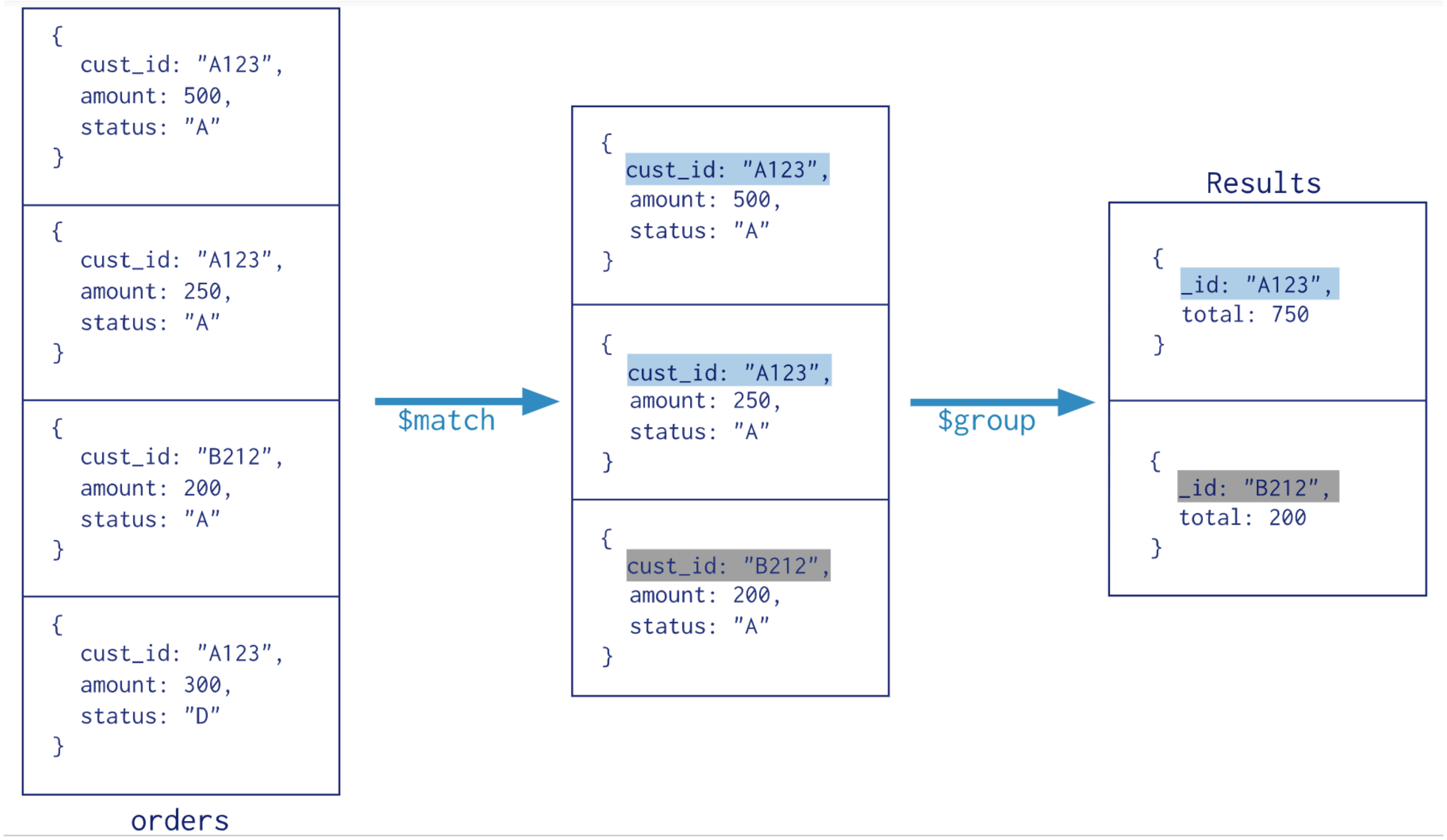
- The *aggregation pipeline* is a framework for data aggregation modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated results.
- The aggregation pipeline provides an alternative to *map-reduce* and may be the preferred 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 computed summing the **amount** fields of the documents with the same **_id**

Aggregation

Aggregation Pipeline



{ 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 }

Aggregation
Aggregation Pipeline

- The *aggregation pipeline* starts processing the documents of the collection and passes 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
- Accumulators as $sum, $avg, $min, $max are available for the $group operator
- The *aggregate* command operates on a single collection.
- Hint: use *match* operator at the beginning of the pipeline.

Aggregation

Lookup in the Aggregation Pipeline as a means for join

- Assume to have executed the following
  db.people.insert({ "_id" : 1, "name" : "John Lennon", "city" : "NY" })
  db.places.insert({ "_id" : 1, "cityName" : "NY", "state" : "New York" })

- We now want to 'join' the field city in people with the field cityName of places

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

`From` indicates the collection to which the 'calling' collection wants to join (the foreign collection), `localField` indicates the field of the calling collection, `foreignField` indicates the join field of the foreign collection, `as` is the name of the additional key in the result:

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

Aggregation

Single Purpose Aggregation Operations

- MongoDB provides a set of specific operations for aggregation. Among them we used:
  - **count**: return the number of document that match a query
  - **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.
  - **Lookup:** it allows for limited forms of outer joins

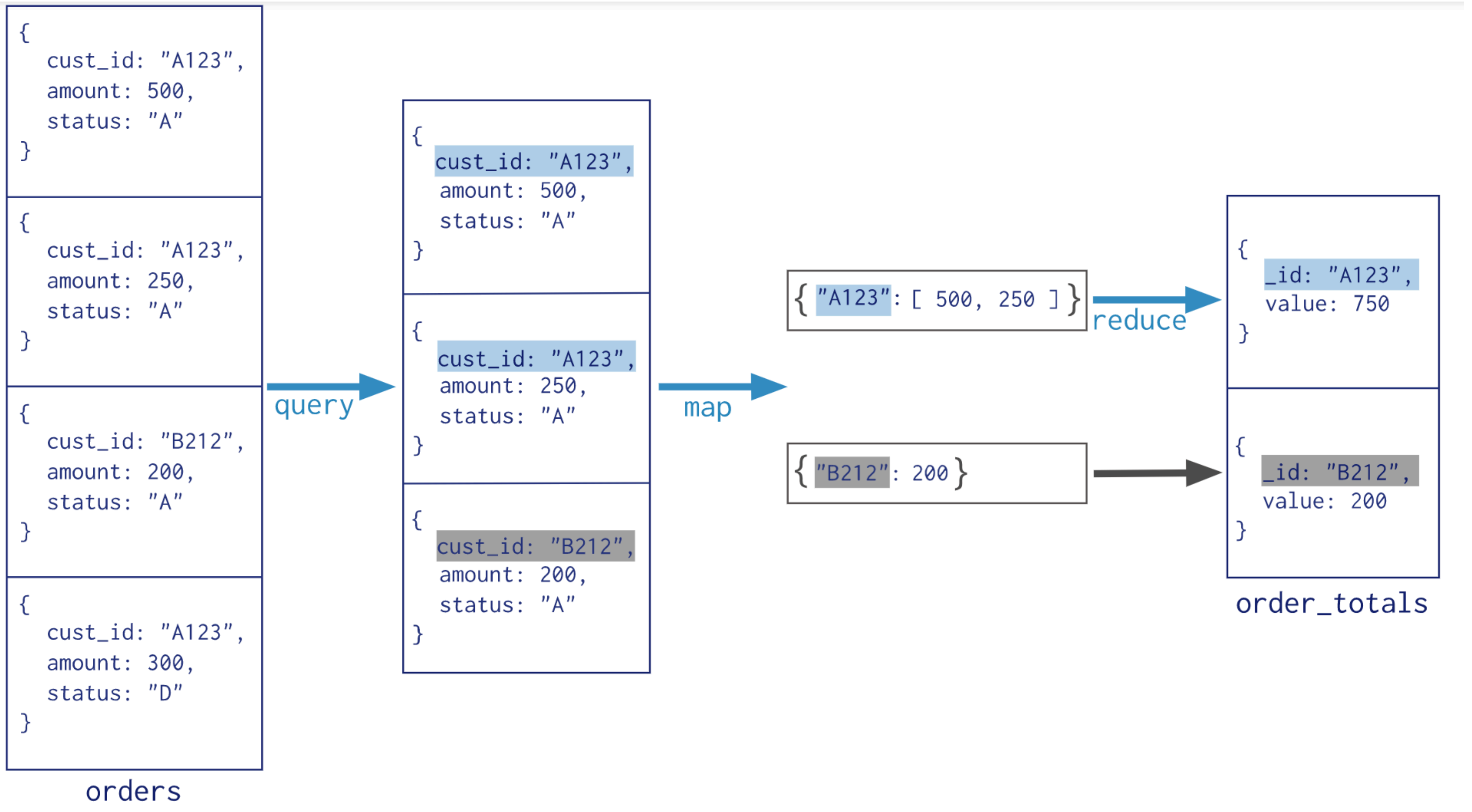
Aggregation

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: applied 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 – Map-Reduce


{ 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

Indexes

Introduction

- Indexes are used by MongoDB to answer more efficiently queries.
- Without indexes MongoDB have to scan the whole collection.
- 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.

Replication

Introduction

- MongoDB supports replication of a collection to increase availability and safetiness.
- Multiple server run several mongod instances over the same dataset.
- The primary replica receives the writes and send the update request to the other replicas to ensure consistency.
- If the primary goes down (does not respond for 10 seconds), the secondaries elect a new primary among them.
- By default client reads from primary but they can set preferences to read from secondaries.

Sharding
Introduction

- Sharding is a method to store data on several machines. This make Big Data Sets more affordable.
- Server are less stressed by clients and the throughput is augmented because each server has a part of the entire database
- MongoDB supports sharding through a *sharded cluster* composed by:
  - Shards : store the data using a *replica set.*
  - Query routers or *mongos*: each of these interacts with a client to direct the requests to the right shard or shards; then collect the answers and return them to the client.
  - Config servers: store the cluster metadata to map the dataset to the shards.
- To shard a collection you need to select a **shard key** (should be indexed). MongoDB divides the keys in **chunks** and distribute them across the shards.

Sharding

Introduction

- To divide shard key values in chunks MongoDB uses **ranged based partitioning** (suited for range queries) or **hash based partitioning** (more random distribution and higher performance for single queries)

- To manage new data two background processes keep the chunks small and equally distributed along the shards.

Acknowledgements

The previous slides on MongoDB are adapted from similar slides authored by Daniele Pantaleone and Giacomo Ronconi for their final project for the exam on “Big Data Management” A.Y. 2014-15.