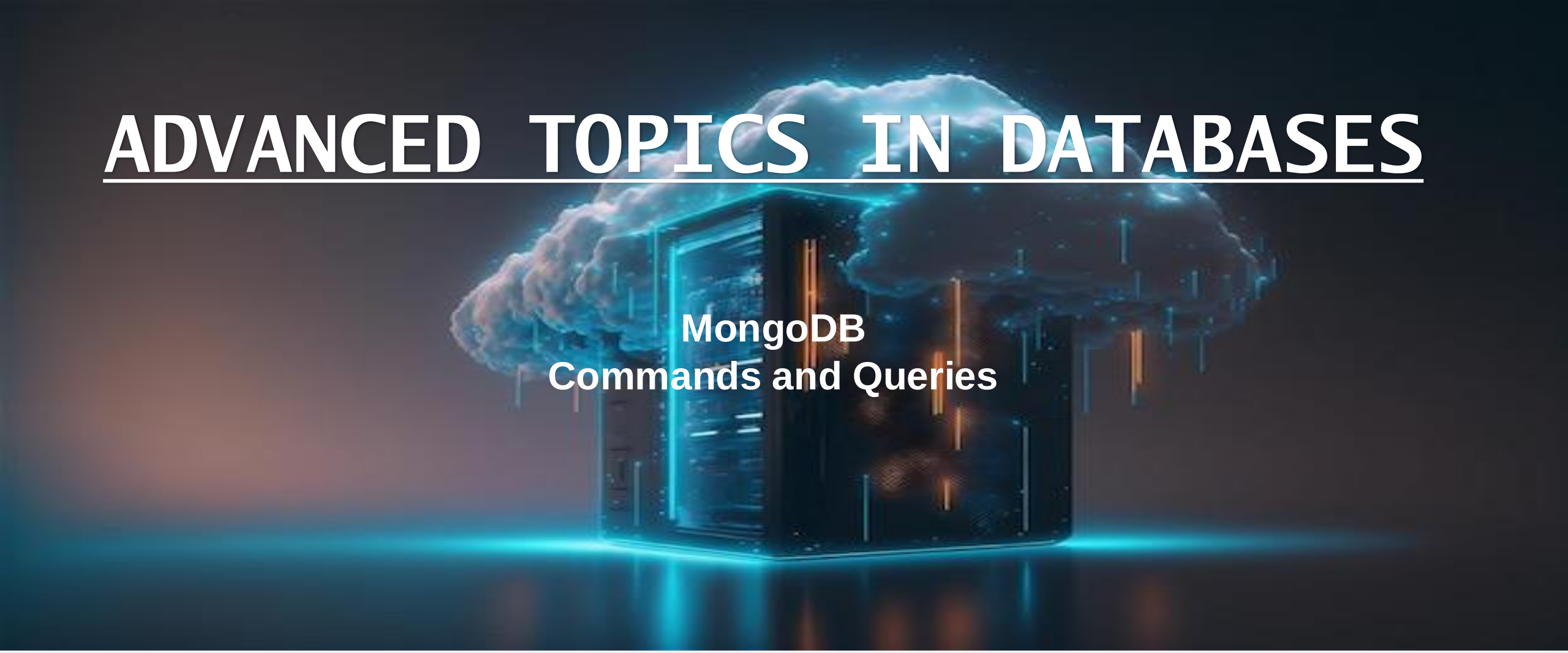


ADVANCED TOPICS IN DATABASES

MongoDB
Commands and Queries

Master in Informatics Engineering
Data Engineering

Informatics Engineering Department

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MongoDB: Terminology

RDBMS	MongoDB
database instance	MongoDB instance
schema	database
table	collection
row	document
rowid	

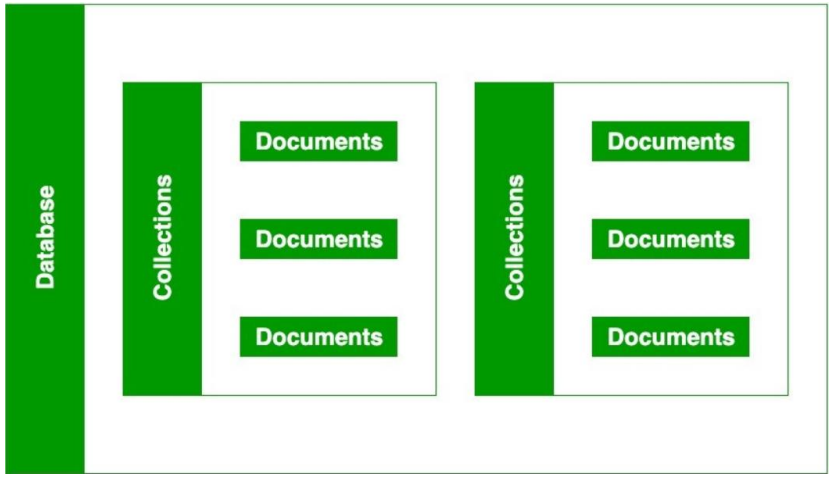
collection

- each JSON document:
  - belongs to a collection
  - has a field _id
    - unique within the collection
- each collection:
  - belongs to a “database”

- http://www.mongodb.org/

Databases e Collections

- Each instance of MongoDB can manage multiple databases
- Each database is composed of a set of collections
- Each collection contains a set of documents
  - The documents of each collection represent similar“objects”
  - can enforce document validation rules for a collection during update and insert operations.



Databases

- Show the list of available databases

```
show databases
```

- Select the database you are interested in

```
use <database-name>
```

E.g. --> use MySchemadb

- Create a database and a collection inside the database
  - To select the database by using the command “use <database name>”
  - then, we can create a collection
    - MongoDB creates a collection implicitly when the collection is first referenced in a command

Databases

- Delete/Drop a database
  - Select the database by using “use <database name>”
  - Execute the command

```
db.dropDatabase()
```

E.g.,

```
use MySchemadb;
db.dropDatabase();
```

Collections

- A collection stores documents, uniquely identified by a document "_id"
- Create collections

```
db.createCollection(<collection name>, <options>);
```

- The collection is associated with the current database. Always select the database before creating a collection.
- Options related to the collection size and indexing, e.g., to create a capped collection, or to create a new collection that uses document validation

db.createCollection("authors", {capped: true});

db.createCollection("logs", { capped: true, size: 500000 }); // size is in bytes.

- Show collections

```
show collections
```

- Drop collections

```
db.<collection_name>.drop()
```

db.authors.drop()

C.R.U.D. Operations

- Create

```
db.users.insertOne(                 ← collection
   {
      name: "sue",                  ← field: value
      age: 26,                      ← field: value
      status: "pending"             ← field: value
   }                                  } document
)
```

- Read

```
db.users.find(                      ← collection
   { age: { $gt: 18 } },            ← query criteria
   { name: 1, address: 1 }          ← projection
).limit(5)                          ← cursor modifier
```

- Update

```
db.users.updateMany(                ← collection
   { age: { $lt: 18 } },            ← update filter
   { $set: { status: "reject" } }   ← update action
)
```

- Delete

```
db.users.deleteMany(                ← collection
   { status: "reject" }             ← delete filter
)
```

Inserts

db.inventory.insert({ _id: 10, type: "misc", item: "card", qty: 15 })

Inserts a document with three fields into collection inventory

User-specified _id field

db.inventory.insert({ type: "book", item: "journal" })

The database generates _id field

db.inventory.find()

{ "_id": ObjectId("58e209ecb3e168f1d3915300"), type: "book", item: "journal" }

Inserts

db.<collection name>.insertOne({<set of the field:value pairs of the new document>});

Example1

```
db.people.insertOne( {
        user_id: "abc124",
        age: 45,
        favorite_colors: ["blue", "green"]
} );
```

Favorite_colors is an array

Example2

```
db.people.insertOne( {
        user_id: "abc124",
        age: 45,
        address: {
             street: "my street",
             city: "my city"
        }
} );
```

Nested document

db.<collection name>.insertMany([<comma separated list of documents>]);

```
db.products.insertMany( [
      { user_id: "abc123", age: 30, status: "A"},
      { user_id: "abc456", age: 40, status: "A"},
      { user_id: "abc789", age: 50, status: "B"}
   ] );
```

Delete

- Delete existing data, in MongoDB corresponds to the deletion of the associated document.
  - Conditional delete
  - Multiple delete

DELETE FROM people WHERE status = "D"	db.people.deleteMany({ status: "D" })
DELETE FROM people	db.people.deleteMany({})

Updates

- E.g.,

```
db.inventory.updateMany(
    { "qty": { $lt: 50 } },
    {
      $set: { "size.uom": "in", status: "P" },
      $currentDate: { lastModified: true }
    }
)
```

This operation updates all documents with qty <50

It sets the value of the size.uom field to "in", the value of the status field to "P", and the value of the lastModified field to the current date.

Updates

```
db.inventory.update(
          { type: "book", item : "journal" },
           { $set: { qty: 10 } },
           { upsert: true }  )
```

Finds all docs matching query

`{ type: "book", item : "journal" }`

and sets the field `{ qty: 10 }`

`upsert: true`

if no document in the inventory collection matches

creates a new document (generated _id)

it contains fields _id, type, item, qty

MongoDB operator

```
db.<table>.updateMany(
   { <condition> },
   { $set: {<statement>} }
)
```

Querying: Basics

- Mongo query language
- A MongoDB query:
  - Targets a specific collection of documents
  - Specifies criteria that identify the returned documents
  - May include a projection to specify returned fields
  - May impose limits, sort, orders, …
- Basic query - all documents in the collection:

Oracle clause	MongoDB operator
SELECT	find()

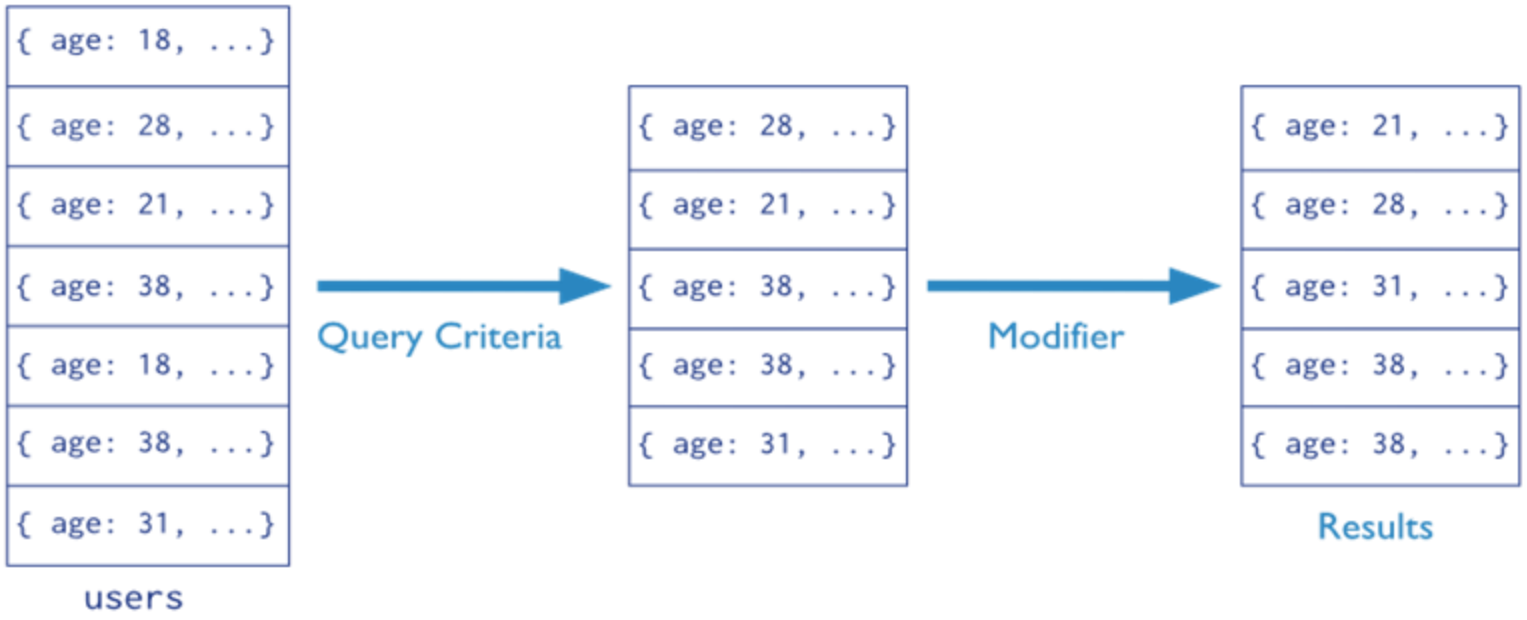
SELECT * FROM users	db.users.**find()**

```
db.<collection name>.find( {<conditions>}, {<fields of interest>} );
```

Querying: Example

Collection — Query Criteria — Modifier

```
db.users.find( { age: { $gt: 18 } } ).sort( {age: 1 } )
```



db.collection.find()gives back a cursor. It can be used to iterate over the result or as input for next operations.

- `cursor.sort()`
- `cursor.count()`
- `cursor.limit()`
- `cursor.max()`
- `cursor.min()`
- `cursor.pretty()`

Querying

MySQL clause	MongoDB operator
COUNT	count()or find().count()

SELECT COUNT(*) FROM people	db.people.count() or db.people.find().count()
SELECT COUNT(*) WHERE status = "A" FROM people	db.people.count(status: "A")} or db.people.find({status: "A"}).count()
SELECT COUNT(*) FROM people WHERE **age > 30**	db.people.count(**{ age: { $gt: 30 } }**)

Querying: Selection

db.<collection name>.find({<conditions>}, {<fields of interest>});

db.inventory.find({ type: "snacks" })

All documents from collection inventory where the type field has the value snacks

db.inventory.find({ type: { $in: ['food', 'snacks'] } })

All inventory docs where the type field is either food or snacks

db.inventory.find({ type: 'food', price: { $lt: 9.95 } })

All ... where the type field is food and the price is less than 9.95

Name	Description
`$eq` or :	Matches values that are equal to a specified value
`$gt`	Matches values that are greater than a specified value
`$gte`	Matches values that are greater than or equal to a specified value
`$in`	Matches any of the values specified in an array
`$lt`	Matches values that are less than a specified value
`$lte`	Matches values that are less than or equal to a specified value
`$ne`	Matches all values that are not equal to a specified value, **including documents that do not contain the field.**
`$nin`	Matches none of the values specified in an array

Querying: Selection

SELECT * FROM people WHERE **age > 25**	db.people.find({ **age: { $gt: 25 }** })

SELECT * FROM people WHERE status = "A" **AND** age = 50	db.people.find({ status: "A", age: 50 })

SELECT * FROM people WHERE status = "A" **OR** age = 50	db.people.find({ **$or**: **[** { status: "A" } , { age: 50 } **]** })

	db.people.find({"address.city":"Rome" })

```
{ _id: "A",
  address: {
        street: "Via Torino",
        number: "123/B",
        city: "Rome",
        code: "00184"
    }
}
```

nested document

```
db.inventory.find( { item: null } )          // equality filter

db.inventory.find( { item : { $exists: false } } )   // existence filter

db.inventory.find( { item : { $type: 10 } } )        // type filter
```

Note:

- Item: null → matches documents that either contain the item field whose value is null or that do not contain the item field
- Item: {$exists: false} → matches documents that do not contain the item field

Querying- examples

- Count

```
db.people. count({ age: 32 })
```

- Comparison

```
db.people. find({ age: {$gt: 32 })                  // or equivalently with $gte, $lt, $lte,

db.people.find({ age: {$in: [32, 40] })             // returns all documents having age either 32 or 40

db.people.find({ age: { $gt: 25, $lte: 50 } })      //returns all documents having age > 25 and age <= 50
```

- Logical

```
db.people.find({ name: {$not: {$eq: "Max" } } })

db.people.find({ $or: [ {age: 32}, {age: 33} ] } )
```

Querying - examples

```
db.items.find({
    $and: [
        {$or: [{qty: {$lt: 15}}, {qty: {$gt: 50}} ]},
        {$or: [{sale: true}, {price: {$lt: 5}} ]}
    ]
```

This query returns documents (items) that satisfy **both** these conditions:

1. Quantity sold either less than 15 **or** greater than 50
2. Either the item is on sale (field “sale”: true) **or** its price is less than 5

Querying - examples

- Embedded Documents

  Select all documents where the field size equals the **exact document** { h: 14, w: 21, uom: "cm" }

  ```
  db.inventory.find( { size: { h: 14, w: 21, uom: "cm" } } )
  ```

  To specify a query condition on fields in an embedded/nested document, use **dot notation**

  ```
  db.inventory.find( { "size.uom": "in" } )
  ```

  Dot notation and comparison operator

  ```
  db.inventory.find( { "size.h": { $lt: 15 } } )
  ```

Querying - examples

Array

Query for all documents where the field tags value is an array with exactly two specific elements

```
db.inventory.find( { tags: ["red", "black"] } )
```
→ Item list **order is important**

```
db.inventory.find( { tags: { $all: ["red", "black"] } } )
```
→ List order does **not** important

The following queries **return different** results, i.e., **they are not equivalent**

```
db.inventory.find( { tags: ["red", "black"] } )
db.inventory.find( { tags: ["black", "red"] } )
```

The following queries **return the same results**, i.e., **they are equivalent**

```
db.inventory.find( { tags: { $all: ["red", "black"] } } )
db.inventory.find( { tags: { $all: ["black", "red"] } } )
```

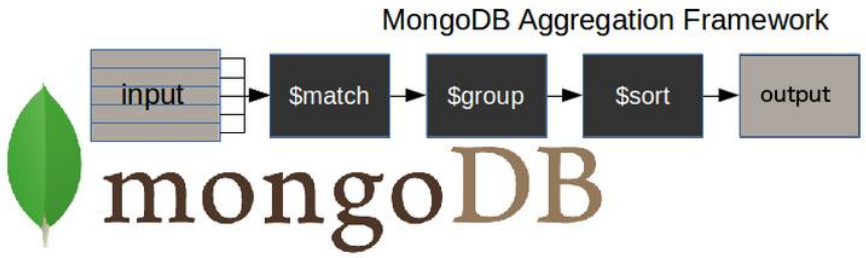
Aggregation Framework

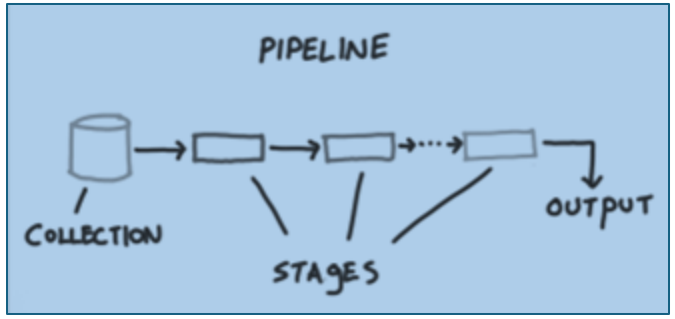
- Allows you to define data processing pipeline
  - Pipelines are made up of processing stages
  - Each step transforms the data as it moves through itPode ser usada para:
  - Find documents
  - Modify the Structure of the Query Response Documents
  - Generate aggregated results
  - Modify Documents in DB
  - create new documents in the DB.

https://docs.mongodb.com/manual/aggregation/#aggregation-framework

Aggregation Framework

- A typical MongoDB aggregation pipeline.





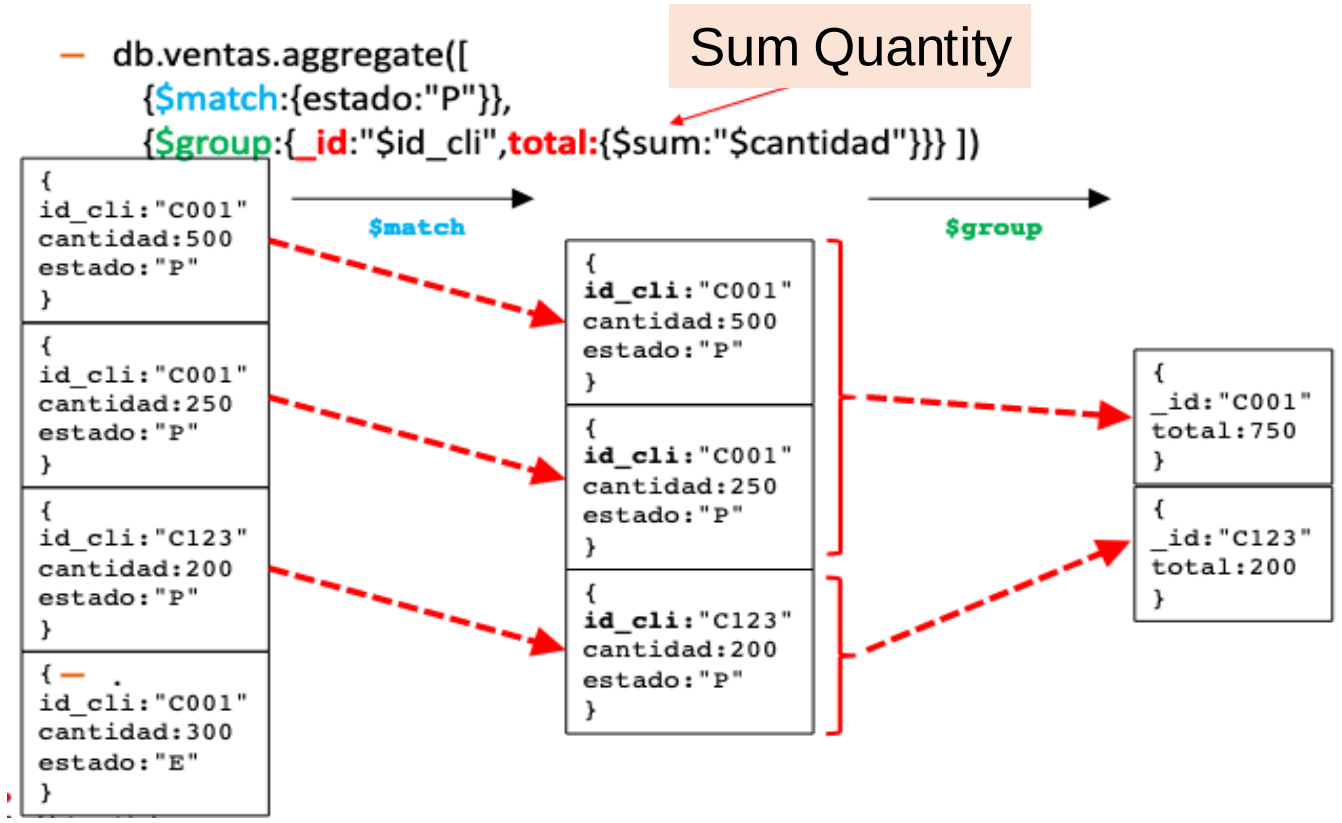
  - $match – filters the documents we need to work with, the ones that meet our needs
  - $group – does the aggregation work
  - $sort – classifies the resulting documents in the way we require (ascending or descending)

The pipeline input can be only one collection, where others can later be merged into the pipeline.

The pipeline performs successive transformations in the data until our goal is achieved.

Aggregation Framework

Exemplification



Aggregation Framework

- General shape of the pipeline:

```
db.collection.aggregate( [ { <etapa1> },{ <etapa2> }, ... ] )
```

- Examples of types of steps that can be used with aggregate:
  - $match: Moves the documents in the stream that match the filter to the next stage of the pipeline.
  - $group: Group documents and applies aggregation operations on the groups;
  - $project: Allows you to add new attributes or remove existing attributes. Generates an output document for each input
  - $sort: sort the documents by the chosen key;
  - $limit: passes the first n input documents to the pipeline (where the chosen threshold is);
  - $unwin : Expands an array, generating an output document for each input in the array.

Groupings & Aggregations

Exemplo:

```
db.universities.aggregate([
                { $match : { name :  'Isep' } },
                 { $unwind : '$students' },
                { $project : { _id : 0, 'students.year' : 1, 'students.number' : 1 } },
                { $sort : { 'students.number' : -1 } }
 ]).pretty()
```

Groupings & Aggregations

- A $group step allows you to do groupings and aggregations

Examples:

  - Sum of all product values in an order;
  - Average grades of students in a subject.

- These operations resemble the group by, having, count, avg, sum, max, and min of the SQL language

https://docs.mongodb.com/manual/aggregation/#aggregation-framework

Simple aggregations

General form of simple aggregations (without grouping):

```
db.collection.aggregate(
    { $group :
        { _id : null,
          <nome_resultado_agregação1>:{
                  <função_agregação1>:"$<atributo_a_agregar>"},
          <nome_resultado_agregação2>:{
                  <função_agregação2>:"$<atributo_a_agregar>"},
          ...
        }
    }
);
```

Aggregation example

```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [
    -74.016323,
    40.710537
  ]
}
```

```
db.zipcodes.aggregate( [
   { $group: { _id: null,
                totalPop: { $sum: "$pop" }
               }
    }
] )
```

Query Result

```
{ "_id" : null, "totalPop" : 248408400 }
```

https://docs.mongodb.com/manual/tutorial/aggregation-zip-code-data-set/

Agregations

General form of aggregations with clustering:

```
db.collection.aggregate(
    { $group :
        { _id : <atributo(s)_agrupador(es)>,
          <nome_resultado_agregação1>:{
                  <função_agregação1>:"$<atributo_a_agregar>"},
          <nome_resultado_agregação2>:{
                  <função_agregação2>:"$<atributo_a_agregar>"},
          ...
        }
    }
);
```

Example

- For each state, calculate the total population and the average population;

```
db.zipcodes.aggregate( [
   { $group: { _id: "$state",
                    totalPop: { $sum: "$pop" },
                    mediaPop: { $avg: "$pop" }
                  }
  }
] )
```

Example

- For each city in the state of New York, calculate the total population

```
db.zipcodes.aggregate(
   [
      { $match: { state: "NY"}},
      { $group: { _id: "$city",
                  cityTotalPop: { $sum: "$pop" }
                }
      }
   ]
);
```

SELECT city, SUM(pop) AS cityTotalPop

FROM zipcodes

Where state like ´NY´

GROUP BY city;

Example

```
db.zipcodes.aggregate( [
  { $group: { _id: "$state",
                totalPop: { $sum: "$pop" }
              }
   },
  { $match: { totalPop: { $gte: 10*1000*1000 } } }
] )
```

```
SELECT state, SUM(pop) AS totalPop
FROM zipcodes
GROUP BY state
HAVING totalPop >= (10*1000*1000)
```

Example

- Group by state and city

```
db.zipcodes.aggregate( [
  { $group: { _id:  {gstate:"$state",
gcity:”$city”},
                cityPop: { $sum: "$pop" }
                }
   }
] )
```

Example

1. List user names in uppercase letters and in alphabetical order.

```
db.users.aggregate(
  [
    { $project : { name:{$toUpper:"$_id"} , _id:0 } },
    { $sort : { name : 1 } }
  ]
)
```

```
{
  "name" : "JANE"
},
{
  "name" : "JILL"
},
{
  "name" : "JOE"
}
```

The operator - $project:

– Creates a new field called Name.

– Converts the value from _id to uppercase, with the $toUpper operator. Then $project creates a new field, called name to hold this value.

– Suppresses the ID field. The $project will pass the _id field by default unless explicitly suppressed. •

- The $sort operator sorts the results by the name field.

Example

2. List how many people have joined the club in each month of the year.

```
db.users.aggregate(
  [
    { $project : { month_joined : { $month : "$joined" } } } ,
    { $group : { _id : {month_joined:"$month_joined"} , number : { $sum : 1 } } },
    { $sort : { "_id.month_joined" : 1 } }
  ]
)
```

•**The $project operator creates a new attribute** called month_joined.

The $month operator converts the attribute values to integer representations of the month (1,2,...). The $project operator then assigns the values to **the month_joined attribute**.

The operator $group collects all documents with a certain value of month_joined and counts how many documents There are no questions for that value. Specifically, for each unique value, $group creates a new document "per month" with two attributes:

_id, which contains a document with the month_joined attribute and its value.

– number, which is a generated attribute. **The $sum operator increments this** attribute by 1 for each document containing the value month_joined provided.

Example of a two-step grouping & aggregation

```
db.zipcodes.aggregate( [
  { $group: { _id: { state: "$state", city: "$city" },
                      cityPop: { $sum: "$pop" }
              }
  },

  { $group: { _id: "$_id.state",
              avgCityPop: { $avg: "$cityPop" }
              }
  }
] )
```

This block groups zip codes by state and city and calculates the total population of each state.
Block result: set of triples (state, city, cityPop)

This block groups the result of the previous block by state and calculates the average of the population in each group.
Block Result: Set of Doubles (State, avgCityPop

Example of a multi-step pipeline

```
db.zipcodes.aggregate( [
   { $group:
      { _id: { gState: "$state", gCity: "$city" },
        cityPop: { $sum: "$pop" } } },
   { $sort: { cityPop: 1 } },
   { $group:
      { _id : "$_id.gState",
        biggestCity:  { $last: "$_id.gCity" },  biggestPop:   { $last: "$cityPop" },
        smallestCity: { $first: "$_id.gCity" }, smallestPop:  { $first: "$cityPop" } } },
   { $project:
      { _id: 0,
        state: "$_id",
        biggestCity:  { name: "$biggestCity",  pop: "$biggestPop" },
        smallestCity: { name: "$smallestCity", pop: "$smallestPop" } } }
] );
```

For each state, it gets the city with the highest and lowest population. It also shows the population

Example of a multi-step pipeline

```
db.zipcodes.aggregate( [
    { $group:
       { _id: { gState: "$state", gCity: "$city" },
         cityPop: { $sum: "$pop" } } },
    { $sort: { cityPop: 1 } },
    { $group:
       { _id : "$_id.gState",
         biggestCity:  { $last: "$_id.gCity" },  biggestPop:   { $last: "$cityPop" },
         smallestCity: { $first: "$_id.gCity" }, smallestPop:  { $first: "$cityPop" } } },
    { $project:
       { _id: 0,
         state: "$_id",
         biggestCity:  { name: "$biggestCity",  pop: "$biggestPop" },
         smallestCity: { name: "$smallestCity", pop: "$smallestPop" } } }
] );
```

Get the population of each city

Sorts the response of the previous block in crescent order of the population

From the answer of the previous block, it obtains the largest and smallest population and their respective cities

Formats the documents from the previous block response to the final result

Aggregation functions

Examples of functions that can be used in step $group:

- Same as SQL: $sum, $avg,$min,$max
- $push : return a vector with all the values that appear in the group for the chosen field;
- $addToSet
- $first: returns the value of an attribute from the first document in an ordered set of documents;
- $last: returns the value of an attribute from the last document in an ordered set of documents;

Example - $push and $addToSet

For each state, it returns a vector with all the cities in the state with repeats and another vector with the cities without repeats

```
db.zipcodes.aggregate(
   [
      { $group: { _id: "$state",
                  bagOfCities: { $push: "$city" },
                  setOfCities: { $addToSet: "$city" }
                }
      }
   ]
);
```

SQL vs Mongodb

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM()	$sum
COUNT()	$sum $sortByCount
join	$lookup
SELECT INTO NEW_TABLE	$out
MERGE INTO TABLE	$merge (Available starting in MongoDB 4.2)
UNION ALL	$unionWith (Available starting in MongoDB 4.4)

Examples: SQL vs Mongodb

SQL Example	MongoDB Example	Description

`SELECT COUNT(*) AS count`
`FROM orders`
	`db.orders.aggregate( [`
	`  {`
	`  $group: {`
	`  _id: null,`
	`  count: { $sum: 1 }`
	`  }`
	`  }`
	`] )`
		Count all records from `orders`

`SELECT cust_id,`
`  SUM(price) AS total`
`FROM orders`
`GROUP BY cust_id`
	`db.orders.aggregate( [`
	`  {`
	`  $group: {`
	`  _id: "$cust_id",`
	`  total: { $sum: "$price" }`
	`  }`
	`  }`
	`] )`
		For each unique `cust_id`, sum the `price` field.

Examples: SQL vs Mongodb

```sql
SELECT cust_id,
       ord_date,
       SUM(price) AS total
FROM orders
GROUP BY cust_id,
         ord_date
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: {
           cust_id: "$cust_id",
           ord_date: { $dateToString:
              format: "%Y-%m-%d",
              date: "$ord_date"
           }}
        },
        total: { $sum: "$price" }
     }
   }
] )
```

For each unique `cust_id`, `ord_date` grouping, sum the `price` field. Excludes the time portion of the date.

Examples: SQL vs Mongodb

```sql
SELECT cust_id,
       ord_date,
       SUM(price) AS total
FROM orders
GROUP BY cust_id,
         ord_date
HAVING total > 250
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: {
           cust_id: "$cust_id",
           ord_date: { $dateToString
              format: "%Y-%m-%d",
              date: "$ord_date"
           }}
        },
        total: { $sum: "$price" }
     }
   },
   { $match: { total: { $gt: 250 } }
] )
```

For each unique `cust_id`, `ord_date` grouping, sum the `price` field and return only where the sum is greater than 250. Excludes the time portion of the date.

Examples: SQL vs Mongodb

```sql
SELECT cust_id,
       SUM(price) as total
FROM orders
WHERE status = 'A'
GROUP BY cust_id
HAVING total > 250
```

```
db.orders.aggregate( [
   { $match: { status: 'A' } },
   {
     $group: {
        _id: "$cust_id",
        total: { $sum: "$price" }
     }
   },
   { $match: { total: { $gt: 250 } } }
] )
```

For each unique `cust_id` with status `A`, sum the `price` field and return only where the sum is greater than 250.

Examples: SQL vs Mongodb

```
SELECT COUNT(*)
FROM (SELECT cust_id,
             ord_date
      FROM orders
      GROUP BY cust_id,
               ord_date)
      as DerivedTable
```

```
db.orders.aggregate( [
    {
      $group: {
         _id: {
            cust_id: "$cust_id",
            ord_date: { $dateToString
               format: "%Y-%m-%d",
               date: "$ord_date"
            }}
         }
      },
    {
      $group: {
         _id: null,
         count: { $sum: 1 }
      }
    }
] )
```

Count the number of distinct `cust_id`, `ord_date` groupings. Excludes the time portion of the date.

Join of collections

- Step $lookup does an outer join to the left, with an equal condition between the value of an attribute of the documents in the input collection and the value of a field of the documents in the collection to be "joined"

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

```
SELECT *, <output array field>
FROM collection
WHERE <output array field> IN (SELECT *
                               FROM <collection to join>
                              WHERE <foreignField> = <collection.localField>
);
```

Examples

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

```
db.pessoas.aggregate([
    { $lookup: {
            from: "compras",
            localField: "_id",
            foreignField: "id_comprador",
            as: "historico_compras" } },
    { $match: { "historico_compras": { $ne: [] } } }
]);
```

Join Conditions with Subqueries

```
{
   $lookup:
      {
         from: <joined collection>,
         let: { <var_1>: <expression>, …, <var_n>: <expression> },
         pipeline: [ <pipeline to run on joined collection> ],
         as: <output array field>
      }
}
```

Field	Description
from	Specifies the collection in the same database to perform the join operation. Starting in MongoDB 5.1, the from collection can be sharded.
let	Optional. Specifies variables to use in the pipeline stages. Use the variable expressions to access the fields from the joined collection's documents that are input to the pipeline.

```
SELECT *, <output array field>
FROM collection
WHERE <output array field> IN (
  SELECT <documents as determined from the pipeline>
  FROM <collection to join>
  WHERE <pipeline>
);
```

To reference variables in the stages pipeline steps, the following syntax is used: "$$<variable>"

Example of a pipeline involving join

Query: Get people's data and data on purchases they've made from April 2020

```
db.pessoas.aggregate([
   { $lookup: {
         from: "compras",
         let: { id_pessoa: "$_id" },
         pipeline: [
           { $match: { $expr:
               { $and: [
                   { $eq: [ "$id_comprador",  "$$id_pessoa" ] },
                   { $gte: [ "$data",  ISODate("2020-04-01") ] } ] } } },
           { $project: { "id_comprador": 0, "_id": 0 } } ],
         as: "compras_recentes"}},
   { $match: { "compras_recentes": { $ne: [] } } }
]);
```

Defines variables for the weighted collection attributes that will be used in the join condition

Join/Selection Condition to tuples of purchases collection:
id_comprador == id_pessoa
Data_compra >=01/04/2020

Defines the structure of the objects in the Purchases collection that will be included in the join response

Example:

```
db.orders.aggregate( [
  {
    $lookup: {
      from: "restaurants",
      localField: "restaurant_name",
      foreignField: "name",
      let: { orders_drink: "$drink" },
      pipeline: [ {
        $match: {
          $expr: { $in: [ "$$orders_drink", "$beverages" ] }
        }
      } ],
      as: "matches"
    }
  }
] )
```

Joins order and restaurant collections by matching localField orders.restaurant_name to ForeignField restaurant.name.

Matching is performed before the pipeline runs.

Performs an array match$in Between the orders.drink and restaurant.beverages fields. These are accessed using $$orders_drink and $beverages respectively.

```
SELECT *, matches
FROM orders
WHERE matches IN (
    SELECT *
    FROM restaurants
    WHERE restaurants.name = orders.restaurant_name
    AND restaurants.beverages = orders.drink
);
```

References

MongoDB: The Definitive Guide
Kristina Chodorow
O´Reilly

MongoDB Manual: http://docs.mongodb.org/manual/