Politecnico di Torino

Introduction to MongoDB

DANIELE APILETTI

POLITECNICO DI TORINO

Introduction

- The leader in the NoSQL Document-based databases
- Full of features, beyond NoSQL:
  - High performance
  - High availability
  - Native scalability
  - High flexibility
  - Open source

Terminology – Approximate mapping

Relational database	MongoDB
Table	Collection
Record	Document
Column	Field

Document Data Design

- High-level, business-ready representation of the data
  - Records are stored into BSON Documents
    - BSON is a binary representation of JSON documents
    - field-value pairs
    - may be nested

```
{
_id: <ObjectID1>,
username: "123xyz",
contact: {
          phone: 1234567890,
          email: "xyz@email.com",
         }
access:  {
          level: 5,
          group: "dev",
         }
}
```

Embedded Sub-Document

Embedded Sub-Document

```
{
   _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)
}
```

Document Data Design

- High-level, business-ready representation of the data
- Mapping into developer-language objects
  - date, timestamp, array, sub-documents, etc.
- Field names
  - The field name **_id** is reserved for use as a **primary key**; its value must be unique in the collection, is **immutable**, possibly autogenerated, and may be of any type other than an array.
  - Field names **cannot** contain the **null** character.
  - The server **permits** storage of field names that contain dots (.) and dollar signs ($)
  - BSON documents may have more than one field with **the same name**. Most MongoDB interfaces, however, represent MongoDB with a structure (e.g., a hash table) that does not support duplicate field names.
  - The maximum BSON document size is **16 megabytes**. To store documents larger than the maximum size, MongoDB provides GridFS.
  - Unlike JavaScript objects, the fields in a BSON document are **ordered.**

Politecnico di Torino

Databases and collections.

Create and delete operations

Databases and 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"
  - However, remember that MongoDB is **schema-less**
  - You are not required to define the schema of the documents a-priori and objects of the same collections can be characterized by different fields
  - Starting in MongoDB 3.2, you can enforce **document validation** rules for a collection during update and insert operations.

Databases and Collections

- Show the list of available databases

```
show databases
```

- Select the database you are interested in

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

- E.g.
  - `use deliverydb`

Databases and Collections

- Create a database and a collection inside the database
  - Select the database by using the command "use <database name>"
  - Then, create a collection
    - MongoDB creates a collection implicitly when the collection is first referenced in a command
- Delete/Drop a database
  - Select the database by using "use <database name>"
  - Execute the command

```
db.dropDatabase()
```

E.g.,

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

Databases and 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

- E.g.,
  - `db.createCollection("authors", {capped: true});`

Databases and Collections

- Show collections

```
show collections
```

- Drop collections

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

- E.g.
  - `db.authors.drop()`

C.R.U.D. Operations

- **C**reate

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

- **R**ead

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

- **U**pdate

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

- **D**elete

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

Create: insert **one** document

- Insert a single document in a collection

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

- E.g.,

```
db.people.insertOne( {
        user_id: "abc123",
        age: 55,
        status: "A"
    } );
```

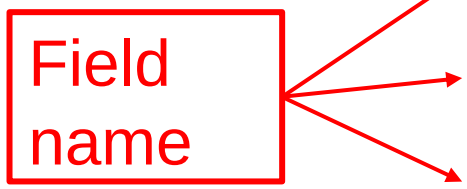
Create: insert **one** document

- Insert a single document in a collection

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

- E.g.,

```
db.people.insertOne( {
        user_id: "abc123",
        age: 55,
        status: "A"
    } );
```



Create: insert **one** document

- Insert a single document in a collection

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

- E.g.

```
db.people.insertOne( {
        user_id: "abc123",
        age: 55,
        status: "A"
    } );
```

Create: insert **one** document

- Insert a single document in a collection

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

Now people contains a new document representing a user with:

```
user_id: "abc123",
age: 55
status: "A"
```

Create: insert **one** document

- E.g.,

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

Now people contains a new document representing a user with:

`user_id: "abc124", age: 45` and an array `favorite_colors` containing the values `"blue"` and `"green"`

Create: insert **one** document

- E.g.,

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

Nested document

Example of a document containing a nested document

Create: insert **many** documents

- Insert multiple documents in a single statement:

```
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"}
   ] );
```

Create: insert **many** documents

- Insert many documents with one single command

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

- E.g.,

```
db.people.insertMany([
  {user_id: "abc123", age: 55, status: "A"},
  {user_id: "abc124", age: 45, favorite_colors: ["blue", "green"]}
] );
```

Delete

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

MySQL clause	MongoDB operator
`DELETE FROM`	`deleteMany()`

Delete

MySQL clause	MongoDB operator
`DELETE FROM`	`deleteMany()`

```
DELETE FROM people
WHERE status = "D"
```

```
db.people.deleteMany(
   { status: "D" }
)
```

Delete

MySQL clause	MongoDB operator
`DELETE FROM`	`deleteMany()`

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

`DELETE FROM people`	`db.people.deleteMany({})`

Politecnico di Torino

DBMG

Databases and collections.
Querying data (find operations)

Query language

- Most of the operations available in SQL language can be expressend in MongoDB language

MySQL clause	MongoDB operator
`SELECT`	`find()`

`SELECT` `*` `FROM people`	`db.people.`**`find()`**

Read data from documents

- Select documents

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

Read data from documents (Filter conditions)

- Select documents

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

- Select the documents satisfying the specified conditions and specifically only the fields specified in fields of interest
  - `<conditions>` are optional
    - conditions take a document with the form:

      ```
      {field1 : <value>, field2 : <value> ... }
      ```

    - Conditions may specify a value or a regular expression

Read data from documents (Project fields)

- Select documents

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

- Select the documents satisfying the specified conditions and specifically only the fields specified in fields of interest
  - `<fields of interest>` are optional
    - projections take a document with the form:

      `{field1 : <value>, field2 : <value> ... }`

    - **1**/true to include the field, **0**/false to exclude the field

find() operator (1)

```
SELECT id,
       user_id,
       status
FROM people
```

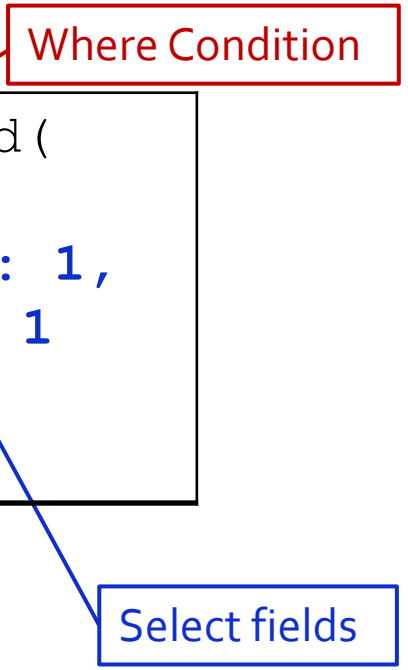
```
db.people.find(
    { },
    { user_id: 1,
      status: 1
    }
)
```

find() operator (2)

MySQL clause	MongoDB operator
SELECT	find()

```
SELECT id,
       user_id,
       status
FROM people
```

```
db.people.find(
    { },
    { user_id: 1,
      status: 1
    }
)
```



find() operator (3)

MySQL clause	MongoDB operator
SELECT	find()
WHERE	find({<WHERE CONDITIONS>})

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

find() operator (4)

MySQL clause	MongoDB operator
SELECT	find()
WHERE	find({<WHERE CONDITIONS>})

```
SELECT user_id, status
FROM people
WHERE status = "A"
```

```
db.people.find(
    { status: "A" },
    { user_id: 1,
      status: 1,
      _id: 0
    }
)
```

Where Condition

Selection fields

By default, the `_id` field is always returned.
To remove it, you must explicitly indicate `_id: 0`

find() operator (5)

MySQL clause	MongoDB operator
SELECT	find()
WHERE	find({<WHERE CONDITIONS>})

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

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

Read data from one document

- Select a single document

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

- Select one document that satisfies the specified query criteria.
  - If multiple documents satisfy the query, it returns the first one according to the natural order which reflects the order of documents on the disk.

(No) joins

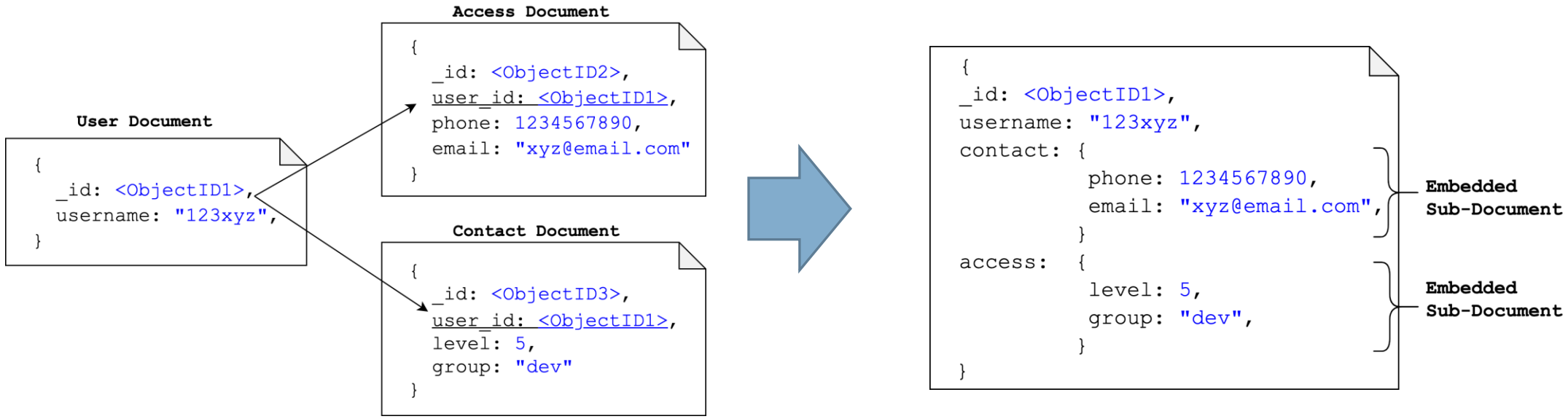
- No join operator exists (but `$lookup`)
  - You must write a program that
    - Selects the documents of the first collection you are interested in
    - Iterates over the documents returned by the first step, by using the loop statement provided by the programming language you are using
    - Executes one query for each of them to retrieve the corresponding document(s) in the other collection

https://docs.mongodb.com/manual/reference/operator/aggregation/lookup

(No) joins

- **(no) joins**
  - Relations among documents/records are provided by
    - Object_ID (_id), named "**Manual reference**" in MongoDB, a second query is required
    - **DBRef**, a standard approach across collections and databases (check the driver compatibility)

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



https://docs.mongodb.com/manual/reference/database-references/

Comparison query operators

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

Comparison operators (>)

MySQL	MongoDB	Description
>	**$gt**	**greater than**

```
SELECT *
FROM people
WHERE age > 25
```

```
db.people.find(
    { age: { $gt: 25 } }
)
```

Comparison operators (>=)

MySQL	MongoDB	Description
>	$gt	greater than
>=	**$gte**	**greater equal then**

```
SELECT *
FROM people
WHERE age >= 25
```

```
db.people.find(
    { age: { $gte: 25 } }
)
```

Comparison operators (<)

MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	**$lt**	**less than**

```
SELECT *
FROM people
WHERE age < 25
```

```
db.people.find(
    { age: { $lt: 25 } }
)
```

Comparison operators (<=)

MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	$lt	less than
<=	**$lte**	**less equal then**

```
SELECT *
FROM people
WHERE age <= 25
```

```
db.people.find(
    { age: { $lte: 25 } }
)
```

Comparison operators (=)

MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	$lt	less than
<=	$lte	less equal then
=	**$eq**	**equal to** The $eq expression is equivalent to { field: <value> }.

```
SELECT *
FROM people
WHERE age = 25
```

```
db.people.find(
    { age: { $eq: 25 } }
)
```

Comparison operators (!=)

MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	$lt	less than
<=	$lte	less equal then
=	$eq	equal to
!=	**$ne**	**Not equal to**

```
SELECT *
FROM people
WHERE age != 25
```

```
db.people.find(
    { age: { $ne: 25 } }
)
```

Conditional operators

- To specify multiple conditions, **conditional operators** are used
- MongoDB offers the same functionalities of MySQL with a different syntax.

MySQL	MongoDB	Description
AND	,	Both verified
OR	**$or**	At least one verified

Conditional operators (AND)

MySQL	MongoDB	Description
AND	**,**	**Both verified**

```
SELECT *
FROM people
WHERE status = "A"
AND age = 50
```

```
db.people.find(
    { status: "A",
      age: 50 }
)
```

Conditional operators (OR)

MySQL	MongoDB	Description
AND	,	Both verified
OR	**$or**	**At least one verified**

```
SELECT *
FROM people
WHERE status = "A"
OR age = 50
```

```
db.people.find(
{ $or:
  [ { status: "A" } ,
    { age: 50 }
  ]
}
)
```

Type of read operations (1)

- 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} ] } )
```

Type of read operations (2)

```
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

Type of read operations (3)

- Element

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

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

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

N

  - 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

- Aggregation → Slides on "Data aggregation"

Type of read operations (4)

- Embedded Documents

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

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

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

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

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

Dot notation and comparison operator

Cursor

- `db.collection.find()` gives back a cursor. It can be used to iterate over the result or as input for next operations.
- E.g.,
  - `cursor.sort()`
  - `cursor.count()`
  - `cursor.forEach() //shell method`
  - `cursor.limit()`
  - `cursor.max()`
  - `cursor.min()`
  - `cursor.pretty()`

Cursor: sorting data

- Sort is a cursor method
- Sort documents
  - `sort( {<list of field:value pairs>} );`
  - field specifies which filed is used to sort the returned documents
  - value = -1 descending order
  - Value = 1 ascending order
- Multiple field: value pairs can be specified
  - Documents are sort based on the first field
  - In case of ties, the second specified field is considered

Cursor: sorting data

- Sorting data with respect to a given field in sort() operator

MySQL clause	MongoDB operator
`ORDER BY`	`sort()`

```
SELECT *
FROM people
WHERE status = "A"
ORDER BY age ASC
```

```
db.people.find(
  { status: "A" }
).sort( { age: 1 } )
```

- Returns all documents having status="A". The result is sorted in ascending age order

Cursor: sorting data

- Sorting data with respect to a given field in sort() operator

MySQL clause	MongoDB operator
`ORDER BY`	`sort()`

`SELECT *`
`FROM people`
`WHERE status = "A"`
`ORDER BY age ASC`
	`db.people.find(`
	`  { status: "A" }`
	`).sort( { age: 1 } )`

`SELECT *`
`FROM people`
`WHERE status = "A"`
`ORDER BY age DESC`
	`db.people.find(`
	`  { status: "A" }`
	`).sort( { age: -1 } )`

- Returns all documents having status="A". The result is sorted in ascending age order
- Returns all documents having status = "A". The result is sorted in descending age order

Cursor: counting

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

`SELECT COUNT(*)`
`FROM people`
	`db.people.count()`
	or
	`db.people.find().count()`

Cursor: counting

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()

Cursor: counting

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 } }**)

Similar to the find() operator, count() can embed conditional statements.

Cursor: forEach()

- `forEach` applies a JavaScript function to apply to each document from the cursor.

```
db.people.find({status: "A"}).forEach(
   function(myDoc){
         print( "user:"+myDoc.name );
   })
```

- Select documents with status="A" and print the document name.

Politecnico di Torino

Databases and collections.

Update operations

Document update

- Back at the C.R.U.D. operations, we can now see how documents can be updated using:

```
db.collection.updateOne(<filter>, <update>, <options>)
```

```
db.collection.updateMany(<filter>, <update>, <options>)
```

  - `<filter>` = filter condition. It specifies which documents must be updated
  - `<update>` = specifies which fields must be updated and their new values
  - `<options>` = specific update options

Document update

- 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.

Updating data

- Tuples to be updated should be selected using the WHERE statements

MySQL clause	MongoDB operator
UPDATE <table> SET <statement> WHERE <condition>	db.<table>.updateMany({ <condition> }, { $set: {<statement>} })

Updating data

MySQL clause	MongoDB operator
UPDATE <table> SET <statement> WHERE <condition>	db.<table>.updateMany({ <condition> }, { $set: {<statement>}})
UPDATE people SET status = "C" WHERE age > 25	db.people.updateMany({age: { $gt: 25 } }, {$set: { status: "C"}})

Updating data

MySQL clause	MongoDB operator
UPDATE <table> SET <statement> WHERE <condition>	db.<table>.updateMany({ <condition> }, { $set: {<statement>}})
UPDATE people SET status = "C" WHERE age > 25	db.people.updateMany({age: { $gt: 25 } }, {$set: { status: "C"}})
UPDATE people SET age = **age + 3** WHERE status = "A"	db.people.updateMany({ status: "A" } , { **$inc: { age: 3 }** })

The $inc operator increments a field by a specified value

Politecnico di Torino

Data aggregation pipeline

General concepts

- Documents enter a multi-stage pipeline that transforms the **documents of a collection** into an aggregated result
- Pipeline **stages** can appear **multiple** times in the pipeline
  - exceptions *$out*, *$merge*, and *$geoNear* stages
- Pipeline expressions can **only** operate on the **current document** in the pipeline and cannot refer to data from other documents: expression operations provide in-memory transformation of documents (max 100 Mb of RAM per stage).
- Generally, expressions are **stateless** and are only evaluated when seen by the aggregation process with one exception: accumulator expressions used in the *$group* stage (e.g. totals, maximums, minimums, and related data).
- The aggregation pipeline provides an alternative to ***map-reduce*** and may be the preferred solution for aggregation tasks since MongoDB introduced the *$accumulator* and *$function* aggregation operators starting in version 4.4

Aggregation Framework

SQL	MongoDB
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
//LIMIT	$limit
SUM	$sum
COUNT	$sum

Aggregation pipeline

- Aggregate functions can be applied to collections to group documents

```
db.collection.aggregate( { <set of stages> })
```

  - Common stages: `$match, $group ..`
  - The aggregate function allows applying aggregating functions (e.g. sum, average, ..)
  - It can be combined with an initial definition of groups based on the grouping fields

Aggregation example (1)

```
db.people.aggregate( [
    { $group: { _id: null,
                mytotal: { $sum: "$age" },
                mycount: { $sum: 1 }
              }
    }
] )
```

- Considers all documents of people and
  - sum the values of their age
  - sum a set of ones (one for each document)
- The returned value is associated with a field called “mytotal” and a field “mycount”

Aggregation example (2)

```
db.people.aggregate( [
    { $group: { _id: null,
                myaverage: { $avg: "$age" },
                mytotal: { $sum: "$age" }
              }
    }
] )
```

- Considers all documents of people and computes
  - sum of age
  - average of age

Aggregation example (3)

```
db.people.aggregate( [
    { $match: {status: "A"} } ,
    { $group: { _id: null,
                count: { $sum: 1 }
              }
    }
] )
```

Where conditions

- Counts the number of documents in people with status equal to “A”

Aggregation in “Group By”

MySQL clause	MongoDB operator
GROUP BY	aggregate($group)

```
SELECT status,
       AVG(age) AS total
FROM people
GROUP BY status
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $avg: "$age" }
     }
   }
] )
```

Aggregation in “Group By”

MySQL clause	MongoDB operator
GROUP BY	aggregate($group)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
```

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

Group field

Aggregation in “Group By”

MySQL clause	MongoDB operator
GROUP BY	aggregate($group)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
```

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

Group field

Aggregation function

Aggregation in “Group By + Having”

MySQL clause	MongoDB operator
HAVING	aggregate($group, $match)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   },
  { $match: { total: { $gt: 1000 } } }
] )
```

Aggregation in "Group By + Having"

MySQL clause	MongoDB operator
HAVING	aggregate($group, $match)

```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   },
  { $match: { total: { $gt: 1000 } } }
] )
```

Group stage: Specify the aggregation field and the aggregation function

Aggregation in "Group By + Having"

MySQL clause	MongoDB operator
HAVING	aggregate($group, $match)

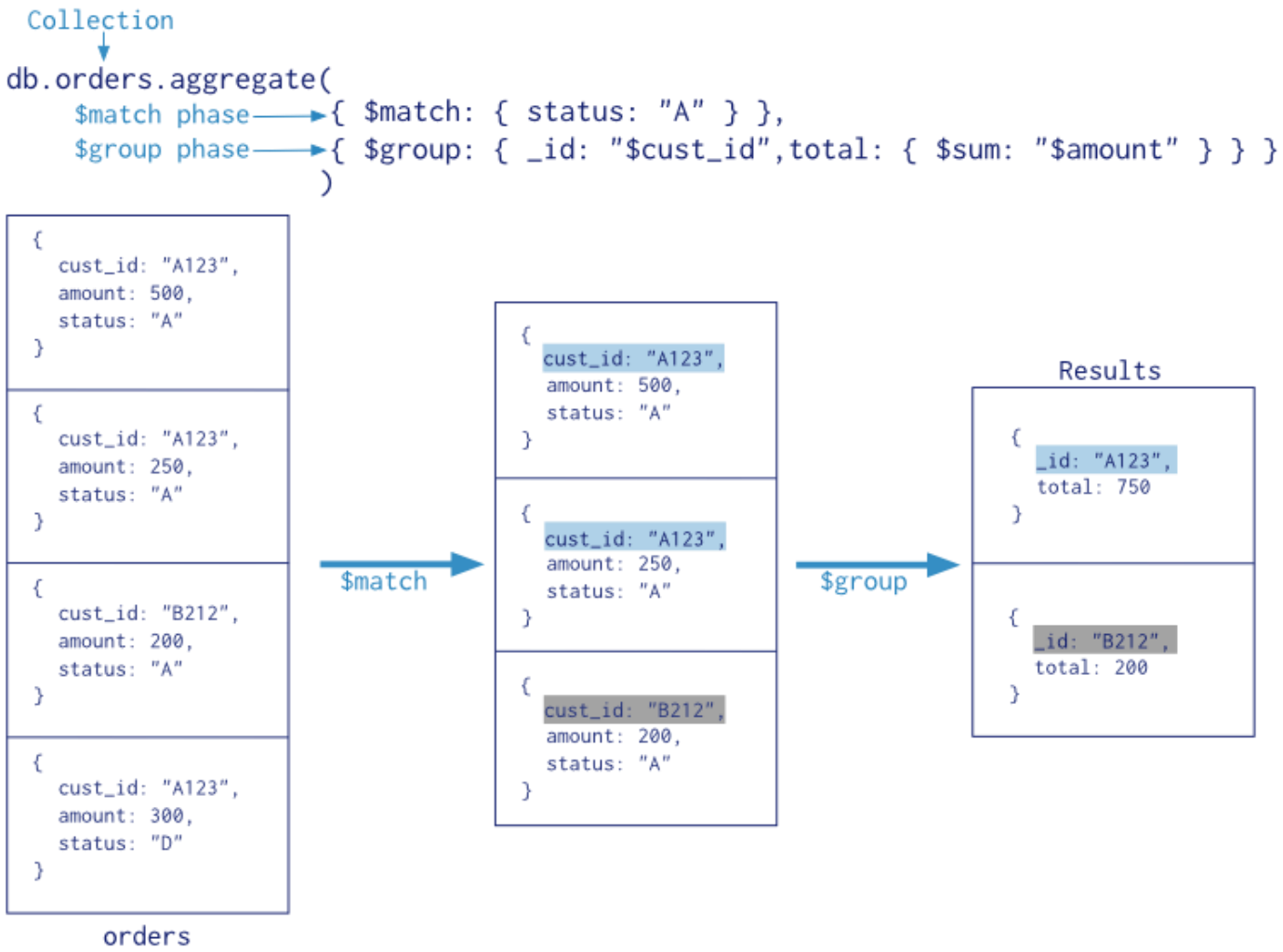
```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
db.orders.aggregate( [
   {
     $group: {
        _id: "$status",
        total: { $sum: "$age" }
     }
   },
   { $match: { total: { $gt: 1000 } } }
] )
```

Group stage: Specify the aggregation field and the aggregation function

Match Stage: specify the condition as in `HAVING`

Aggregation at a glance



Pipeline stages (1)

Stage	Description
$addFields	Adds **new fields** to documents. Reshapes each document by adding new fields to output documents that will contain both the existing fields from the input documents and the newly added fields.
$bucket	Categorizes incoming documents **into groups,** called buckets, based on a specified expression and bucket boundaries. On the contrary**, $group** creates a "bucket" for each value of the group field.
$bucketAuto	Categorizes incoming documents into a specific number of groups, called buckets, based on a specified expression. Bucket boundaries are automatically determined in an attempt to **evenly distribute** the documents into the specified number of buckets.
$collStats	Returns statistics regarding a collection or view (it must be the **first stage**)
$count	Passes a document to the next stage that contains a count of the input **number of documents** to the stage (same as $group+$project)

Pipeline stages (2)

Stage	Description
$facet	Processes **multiple aggregation pipelines** within a single stage on the same set of input documents. Enables the creation of multi-faceted aggregations capable of characterizing data across multiple dimensions. Input documents are passed to the $facet stage only once, without needing multiple retrieval.
$geoNear	Returns an ordered stream of documents based on the **proximity** to a geospatial point. The output documents include an additional **distance** field. It must in the **first stage** only.
$graphLookup	Performs a **recursive search** on a collection. To each output document, adds a new array field that contains the traversal results of the recursive search for that document.

Example

```
db.employees.aggregate( [
  {
    $graphLookup: {
      from: "employees",
      startWith: "$reportsTo",
      connectFromField: "reportsTo",
      connectToField: "name",
      as: "reportingHierarchy"
    }
  }
] )
```

- The $graphLookup operation recursively matches on the **reportsTo** and **name** fields in the employees collection, returning the **reporting hierarchy** for each person.
- Returns a list of documents such as

```
{
  "_id" : 5,
  "name" : "Asya",
  "reportsTo" : "Ron",
  "reportingHierarchy" : [
    { "_id" : 1, "name" : "Dev" },
    { "_id" : 2, "name" : "Eliot", "reportsTo" : "Dev" },
    { "_id" : 3, "name" : "Ron", "reportsTo" : "Eliot" }
  ]
}
```

original document

Pipeline stages (3)

Stage	Description
$group	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.
$indexStats	Returns statistics regarding the use of each index for the collection.
$limit	Passes the first n documents unmodified to the pipeline where n is the specified limit. For each input document, outputs either one document (for the first n documents) or zero documents (after the first n documents).
$lookup	Performs a **join** to another collection in the same database to filter in documents from the "joined" collection for processing. To each input document, the $lookup stage adds a new array field whose elements are the matching documents from the "joined" collection. The $lookup stage passes these reshaped documents to the next stage.

Pipeline stages (4)

Stage	Description
$match	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
$merge	Writes the resulting documents of the aggregation pipeline to a collection. The stage can incorporate (insert new documents, merge documents, replace documents, keep existing documents, fail the operation, process documents with a custom update pipeline) the results into an output collection. To use the $merge stage, it must be the last stage in the pipeline.
$out	Writes the resulting documents of the aggregation pipeline to a collection. To use the $out stage, it must be the last stage in the pipeline.
$project	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.

Pipeline stages (5)

Stage	Description
$sample	Randomly selects the specified number of documents from its input.
$set	Adds new fields to documents. Similar to $project, $set reshapes each document in the stream; specifically, by adding new fields to output documents that contain both the existing fields from the input documents and the newly added fields. $set is an alias for $**addFields** stage. If the name of the new field is the same as an existing field name (including _id), $set **overwrites** the existing value of that field with the value of the specified expression.
$skip	Skips the first n documents where n is the specified skip number and passes the remaining documents unmodified to the pipeline. For each input document, outputs either zero documents (for the first n documents) or one document (if after the first n documents).
$sort	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.

Pipeline stages (6)

Stage	Description
$sortByCount	Groups incoming documents based on the value of a specified expression, then computes the count of documents in each distinct group.
$unset	Removes/excludes fields from documents.
$unwind	Deconstructs an array field from the input documents to output a document for each element. Each output document replaces the array with an element value. For each input document, outputs n documents where n is the number of array elements and can be zero for an empty array.

Politecnico di Torino

DBMG

Data aggregation examples

Data Model

- Given the following collection of books

```
{
 "title":"MongoDb Guide2",
 "tag":["mongodb","guide","database"],
 "n":200,
 "review_score": 2.2,
 "price":[ {"v": 22.22, "c": "€", "country": "IT"},
           {"v": 22.00, "c": "£", "country": "UK"}
         ],
 "author": {
    "_id": 1,
    "name":"Mario",
    "surname": "Rossi"}
}
{_id:ObjectId("5fb29b175b99900c3fa24293",
 title:”Developing with Python",
 tag:[”python”,”guide”,“programming”],
 n:352,
 review_score:4.6,
 price:[{v: 24.99, c: “€”,  country: “IT”},
        {v: 19.49, c: “£”, country:”UK”} ],
 author: {_id: 2,
          name:”John”,
          surname: “Black”}
}, …
```

price currency

price value

number of pages

Example 1

- For each country, select the average price and the average review_score.
- The review score should be rounded down.
- Show the first 20 results with a total number of books higher than 50.

$unwind

```
db.book.aggregate( [
  { $unwind: "$price" },
] )
```

Build a document for each entry of the **price** array

Result - **$unwind**

{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : ["mongodb", "guide", "database"], "n" : 100, "review_score" : 4.3, "price" : { "v" : 19.99, "c" : " € ", "country" : "IT" }, "author" : { "_id" : 1, "name" : "Mario", "surname" : "Rossi" } }

{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : ["mongodb", "guide", "database"], "n" : 100, "review_score" : 4.3, "price" : { "v" : 18, "c" : "£", "country" : "UK" }, "author" : { "_id" : 1, "name" : "Mario", "surname" : "Rossi" } }

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python ", "tag" : ["python", "guide", "programming"], "n" : 352, "review_score" : 4.6, "price" : { "v" : 24.99, "c" : " € ", "country" : "IT" }, "author" : { "_id" : 2, "name" : "John", "surname" : "Black" } }

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python ", "tag" : ["python", "guide", "programming"], "n" : 352, "review_score" : 4.6, "price" : { "v" : 19.49, "c" : "£", "country" : "UK" }, "author" : { "_id" : 2, "name" : "John", "surname" : "Black" } }

…

$group

```
db.book.aggregate( [
  { $unwind: "$price" } ,
  { $group: { _id: "$price.country"},
              avg_price: { $avg: " $price.v" ,
              bookcount: {$sum:1},
              review: {$avg: " $review_score"}
            }
  }
] )
```

dot notation to access the value of the **embedded** document fields

count the number of books (number of documents)

Result - `$group`

{ "_id" : "**UK**", "avg_price" : 18.75, "bookcount": **150**, "review": 4.3}

{ "_id" : "**IT**", "avg_price" : 22.49, "bookcount": **132**, "review": 3.9}

{ "_id" : "**US**", "avg_price" : 22.49, "bookcount": **49**, "review": 4.2}

…

$match

```
db.book.aggregate( [
  { $unwind: '$price' } ,
  { $group: { _id: '$price.country',
              avg_price: { $avg: '$price.v' },
              bookcount: {$sum:1},
              review: {$avg: '$review_score'}
            }
  },
 {$match: { bookcount: { $gte: 50 } } },
] )
```

Filter the documents where **bookcount** is greater than 50

Result - `$match`

```
{ "_id" : "UK", "avg_price" : 18.75, "bookcount": 150, "review": 4.3}
{ "_id" : "IT", "avg_price" : 22.49, "bookcount": 132, "review":  3.9}
...
```

$project

```
db.book.aggregate( [
  { $unwind: '$price' } ,
  { $group: { _id: '$price.country',
                avg_price: { $avg: '$price.v' },
                bookcount: {$sum:1},
                review: {$avg: '$review_score'}
              }
  },
 {$match: { bookcount: { $gte: 50 } } },
 {$project: {avg_price: 1, review: { $floor: '$review' }}},
] )
```

round down the review score

Result - `$project`

```
{ "_id" : "UK", "avg_price" : 18.75, "review": 4}
{ "_id" : "IT", "avg_price" : 22.49, "review" :  3}
...
```

$limit

```
db.book.aggregate( [
  { $unwind: '$price' } ,
  { $group: { _id: '$price.country',
                avg_price: { $avg: '$price.v' },
                bookcount: {$sum:1},
                review: {$avg: '$review_score'}
              }
  },
 {$match: { bookcount: { $gte: 50 } } },
 {$project: {avg_price: 1, review: { $floor: '$review' }}},
 {$limit:20}
] )
```

Limit the results to the first 20 documents

Example 2

- Compute the 95 percentile of the number of pages,
- only for the books that contain the tag “guide”.

$match

```
db.book.aggregate( [
   {$match: { tag : "guide" } }
] )
```

select documents containing "guide" in the tag array, compare with tag:["guide"]

Result - `$match`

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python", "tag" : ["python", "guide", "programming"], "n" : 352, "review_score" : 4.6, "price" : [{ "v" : 24.99, "c" : "€", "country" : "IT" }, { "v" : 19.49, "c" : "£", "country" : "UK" }], "author" : { "_id" : 1, "name" : "John", "surname" : "Black" } }

{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : ["mongodb", "guide", "database"], "n" : 100, "review_score" : 4.3, "price" : [{ "v" : 19.99, "c" : "€", "country" : "IT" }, { "v" : 18, "c" : "£", "country" : "UK" }], "author" : { "_id" : 1, "name" : "Mario", "surname" : "Rossi" } }

…

$sort

```
db.book.aggregate( [
 {$match: { tag : "guide"} },
{$sort : { n: 1} }
] )
```

sort the documents in ascending order according to the value of the **n** field, which stores the number of pages of each book

Result - `$sort`

{ "_id" : ObjectId("5fb29ae15b99900c3fa24292"), "title" : "MongoDb guide", "tag" : ["mongodb", "guide", "database"], **"n" : 100**, "review_score" : 4.3, "price" : [{ "v" : 19.99, "c" : "€", "country" : "IT" }, { "v" : 18, "c" : "£", "country" : "UK" }], "author" : { "_id" : 1, "name" : "Mario", "surname" : "Rossi" } }

{ "_id" : ObjectId("5fb29b175b99900c3fa24293"), "title" : " Developing with Python", "tag" : ["python", "guide", "programming"], **"n" : 352**, "review_score" : 4.6, "price" : [{ "v" : 24.99, "c" : "€", "country" : "IT" }, { "v" : 19.49, "c" : "£", "country" : "UK" }], "author" : { "_id" : 1, "name" : "John", "surname" : "Black" } }

…

$group + $push

```
db.book.aggregate( [
  {$match: { tag : "guide"} },
  {$sort : { n: 1} },
  {$group: {_id:null, value: {$push: "$n"}}}
] )
```

group all the records together inside a single document (**_id:null**), which contains an array with all the values of **n** of all the records

Result - `$group + $push`

```
{ "_id": null, "value": [100, 352, ...]}
```

$project + $arrayElemAt

```
db.book.aggregate( [
 {$match: { tag : "guide"} },
 {$sort : { n: 1} },
 {$group: {_id:null, value: {$push: "$n"}}},
 {$project:
            {"n95p": {$arrayElemAt:
                       ["$value",
                       {$floor: {$multiply: [0.95, {$size: "$value"}]}}
                      ]
            } }
}
] )
```

get the value of the array at a given index
with { $arrayElemAt: [<array>, <idx>] }

compute the index at 95% of the array length

Result - `$project + $arrayElemAt`

```
{ "_id" : null, "n95p" : 420 }
```

Example 3

- Compute the median of the review_score,
- only for the books having at least a price whose value is higher than 20.0.

Solution

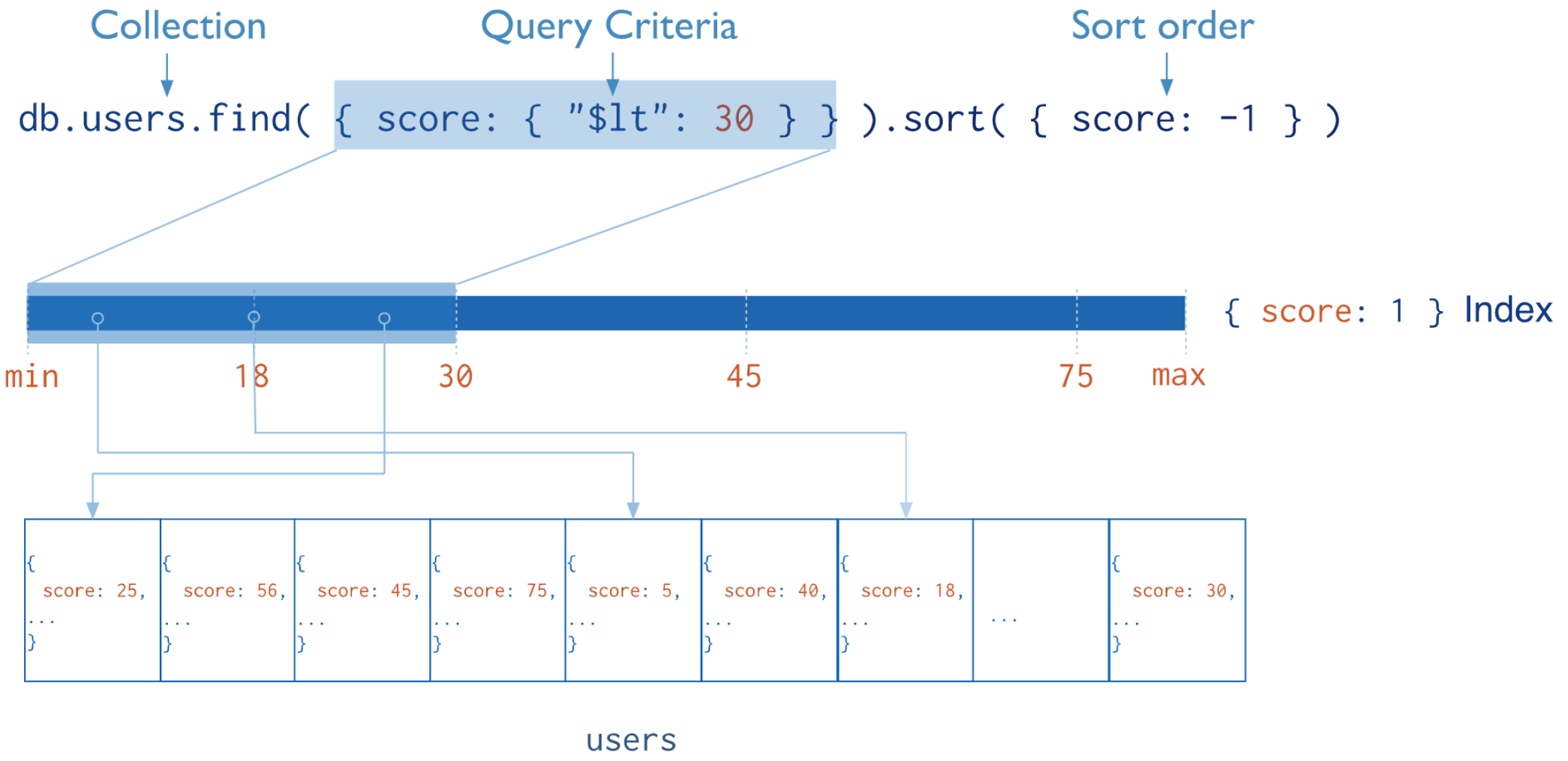
```
db.book.aggregate( [
 {$match: {'price.v' : { $gt: 20 }} },
 {$sort : {review_score: 1} },
 {$group: {_id:null, rsList: {$push: '$review_score'}}},
 {$project:
            {'median': {$arrayElemAt:
                        ['$rsList',
                        {$floor: {$multiply: [0.5, {$size: '$rsList'}]}}
                    ]
            } }
}
] )
```

Indexing

Indexes

- Without indexes, MongoDB must perform a **collection scan**, i.e. scan every document in a collection, to select those documents that match the query statement.
- Indexes are data structures that store a small portion of the collection's data set in a form easy to traverse.
- They store **ordered values of a specific field**, or set of fields, in order to efficiently support
  - equality matches,
  - range-based queries and
  - sorting operations.

Indexes


Collection
Query Criteria
Sort order
db.users.find({ score: { "$lt": 30 } }).sort({ score: -1 })
{ score: 1 } Index
min
18
30
45
75
max
{ score: 25, ... }
{ score: 56, ... }
{ score: 45, ... }
{ score: 75, ... }
{ score: 5, ... }
{ score: 40, ... }
{ score: 18, ... }
...
{ score: 30, ... }
users

Indexes

- MongoDB creates a unique index on the **_id** field during the creation of a collection.
- The **_id** index prevents clients from inserting two documents with the same value for the **_id** field.
- You cannot drop this index on the **_id** field.

Create new indexes

- Creating an index

```
db.collection.createIndex(<index keys>, <options>)
```

  - Before v. 3.0 use `db.collection.ensureIndex()`

- Options include:
  - `name` - a mnemonic name given by the user, you cannot rename an index once created, instead, you must drop and re-create the index with a new name
  - `unique` - whether to accept or not insertion of documents with duplicate keys,
  - `background`, `dropDups`, …

Indexes

- MongoDB provides different data-type indexes
  - Single field indexes
  - Compound field indexes
  - Multikey indexes (to index the content stored in **arrays**, MongoDB creates separate index entries for every element of the array)
  - Geospatial indexes (2d indexes with **planar** and 2dsphere with **spherical** geometry)
  - Text indexes (searching for **string** content in a collection, they do not store language-specific stop words, e.g., "the", "a", "or‟, and stem the words in a collection to only store root words
  - Hashed indexes (indexes the hash of the value of a field, they have a more random distribution of values along their range, but only support equality matches and cannot support range-based queries)

Indexes

- Single field indexes
  - Support user-defined ascending/descending indexes on a single field of a document
- E.g.,
  - `db.orders.createIndex( {orderDate: 1} )`
- Compound field indexes
  - Support user-defined indexes on a set of fields
- E.g.,
  - `db.orders.createIndex( {orderDate: 1,  zipcode: -1} )`

Indexes

- MongoDB supports efficient queries of geospatial data
- Geospatial data are stored as:
  - GeoJSON objects: embedded document { <type>, <coordinate> }
    - E.g., location: `{type: "Point", coordinates: [-73.856, 40.848]}`
  - Legacy coordinate pairs: array or embedded document
    - `point: [-73.856, 40.848]`
- Fields with 2dsphere indexes must hold geometry data in the form of coordinate pairs or GeoJSON data.
  - If you attempt to insert a document with non-geometry data in a 2dsphere indexed field, or build a 2dsphere index on a collection where the indexed field has non-geometry data, the operation will fail.

Indexes

- Geospatial indexes
  - Two type of geospatial indexes are provided: `2d` and `2dsphere`
- A `2dsphere` index supports queries that calculate geometries on an earth-like sphere
- Use a `2d` index for data stored as points on a two-dimensional plane.
- E.g.,
  - `db.places.createIndex( {location: "2dsphere"} )`
- Geospatial query operators
  - `$geoIntersects`, `$geoWithin`, `$near`, `$nearSphere`

Indexes

- `$near` syntax:

```
{
    <location field>: {
      $near: {
        $geometry: {
           type: "Point" ,
           coordinates: [ <longitude> , <latitude> ]
        },
        $maxDistance: <distance in meters>,
        $minDistance: <distance in meters>
      }
    }
}
```

Indexes

- E.g.,
  - `db.places.createIndex( {location: “2dsphere”} )`
- Geospatial query operators
  - `$geoIntersects`, `$geoWithin`, `$near`, `$nearSphere`
- Geopatial aggregation stage
  - `$near`

Indexes

- E.g.,
  - ```
    db.places.find({location:
        {$near:
            {$geometry: {
               type: "Point",
               coordinates: [ -73.96, 40.78 ] },
              $maxDistance: 5000}
        }})
    ```
  - Find all the places within 5000 meters from the specified GeoJSON point, sorted in order from nearest to furthest
    ```

Indexes

- Text indexes
  - Support efficient searching for string content in a collection
  - Text indexes store only *root words* (no language-specific *stop words* or *stem*)
- E.g.,

```
db.reviews.createIndex( {comment: "text"} )
```

  - Wildcard (`$**`) allows MongoDB to index every field that contains string data
  - E.g.,

```
db.reviews.createIndex( {"$**": "text"} )
```

VIEWS

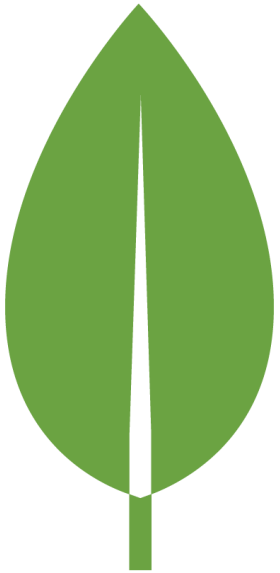
- A queryable object whose contents are defined by an **aggregation** pipeline on other **collections** or **views**.
- MongoDB does not persist the view contents to disk. A view's content is **computed on-demand**.
- Starting in version 4.2, MongoDB adds the $merge stage for the aggregation pipeline to create on-demand **materialized views**, where the content of the output collection can be updated each time the pipeline is run.
- **Read-only** views from existing collections or other views. E.g.:
  - excludes private or confidential data from a collection of employee data
  - adds computed fields from a collection of metrics
  - joins data from two different related collections

```
db.runCommand( {
  create: <view>, viewOn: <source>, pipeline: <pipeline>, collation: <collation> } )
```

- Restrictions
  - immutable Name
  - you can modify a view either by dropping and recreating the view or using the *collMod* comman

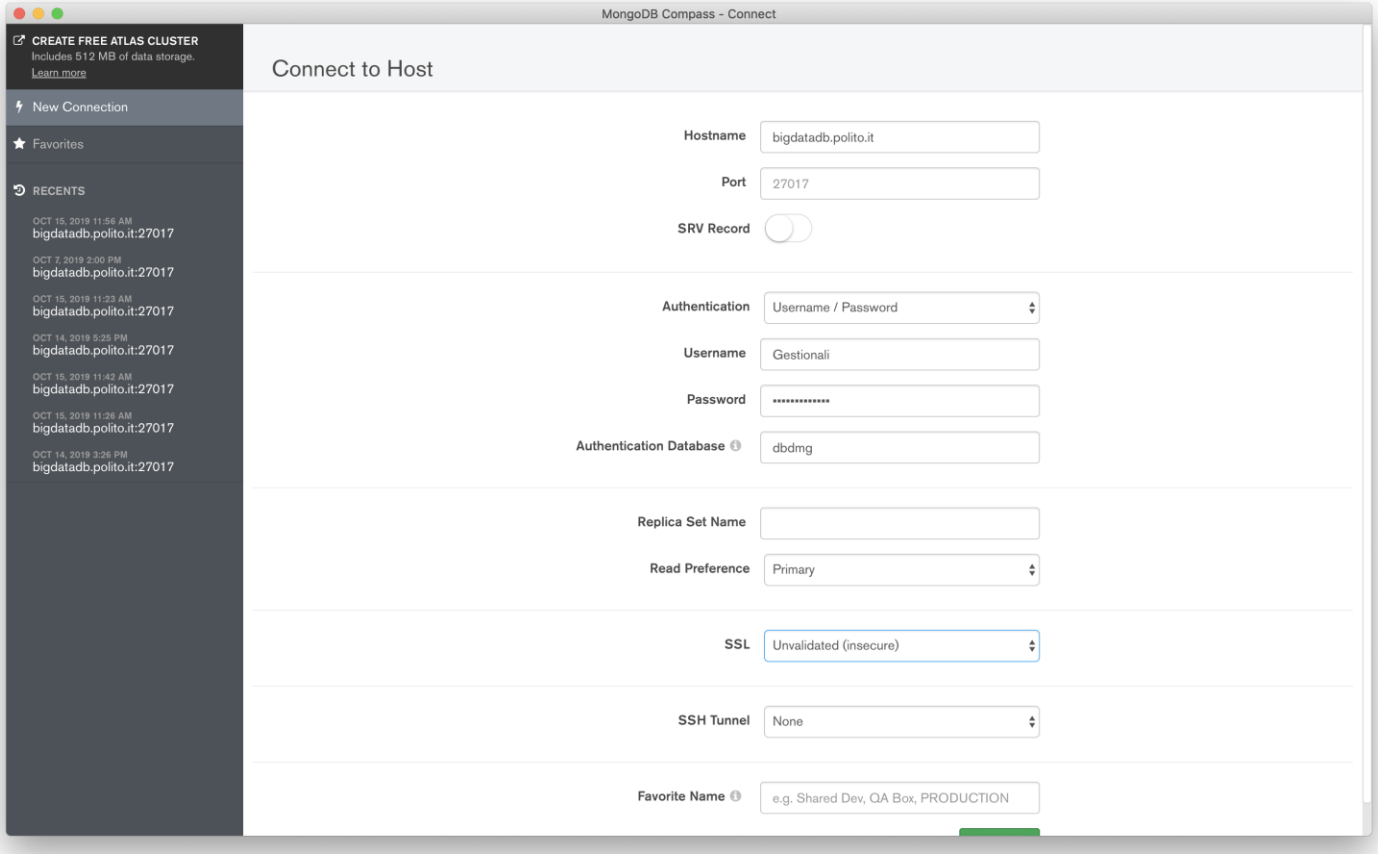
GUI for MongoDB

MongoDB Compass



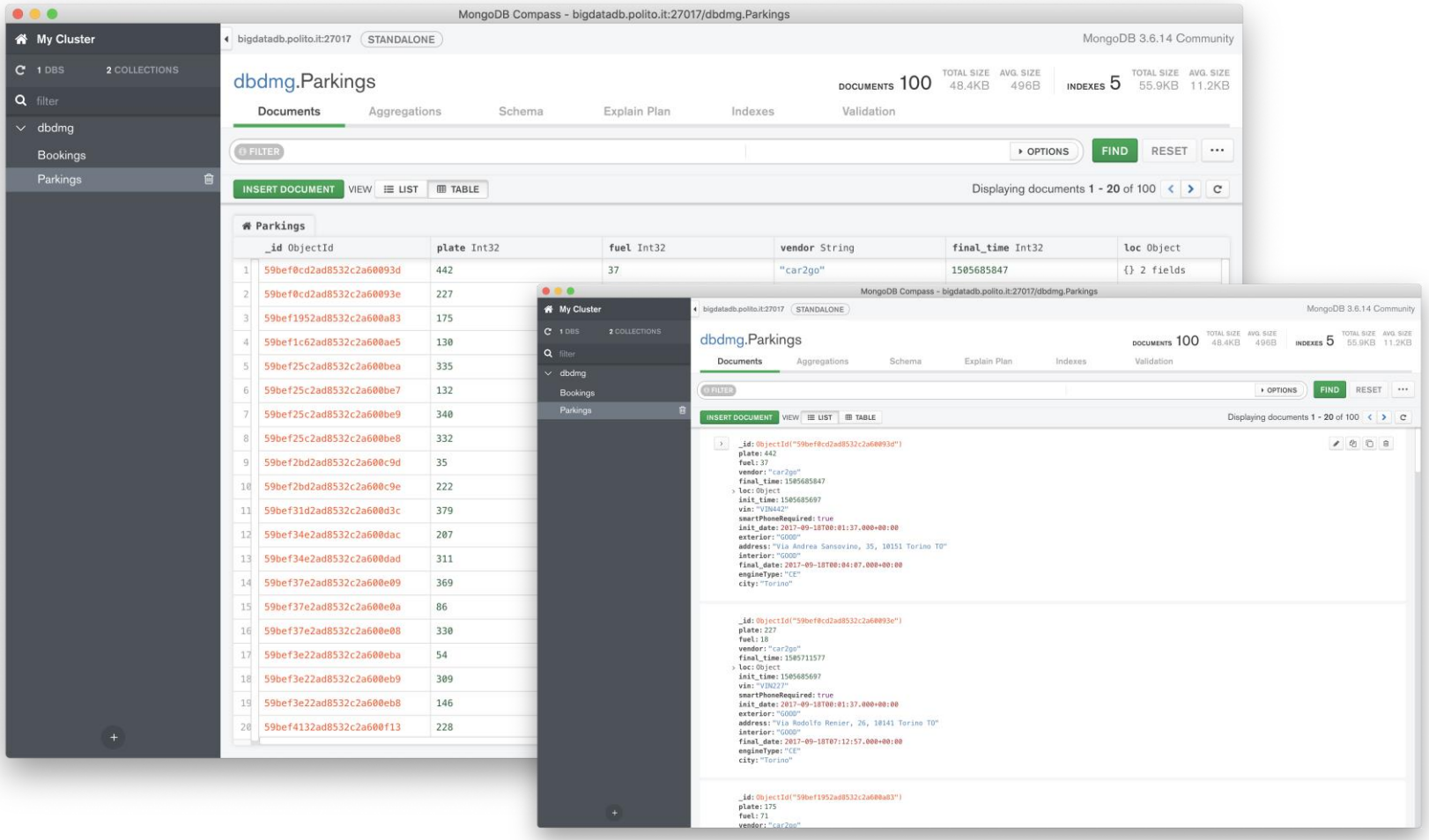
- Visually explore data.
- Available on Linux, Mac, or Windows.
- MongoDB Compass analyzes documents and displays rich structures within collections.
- Visualize, understand, and work with your geospatial data.

MongoDB Compass



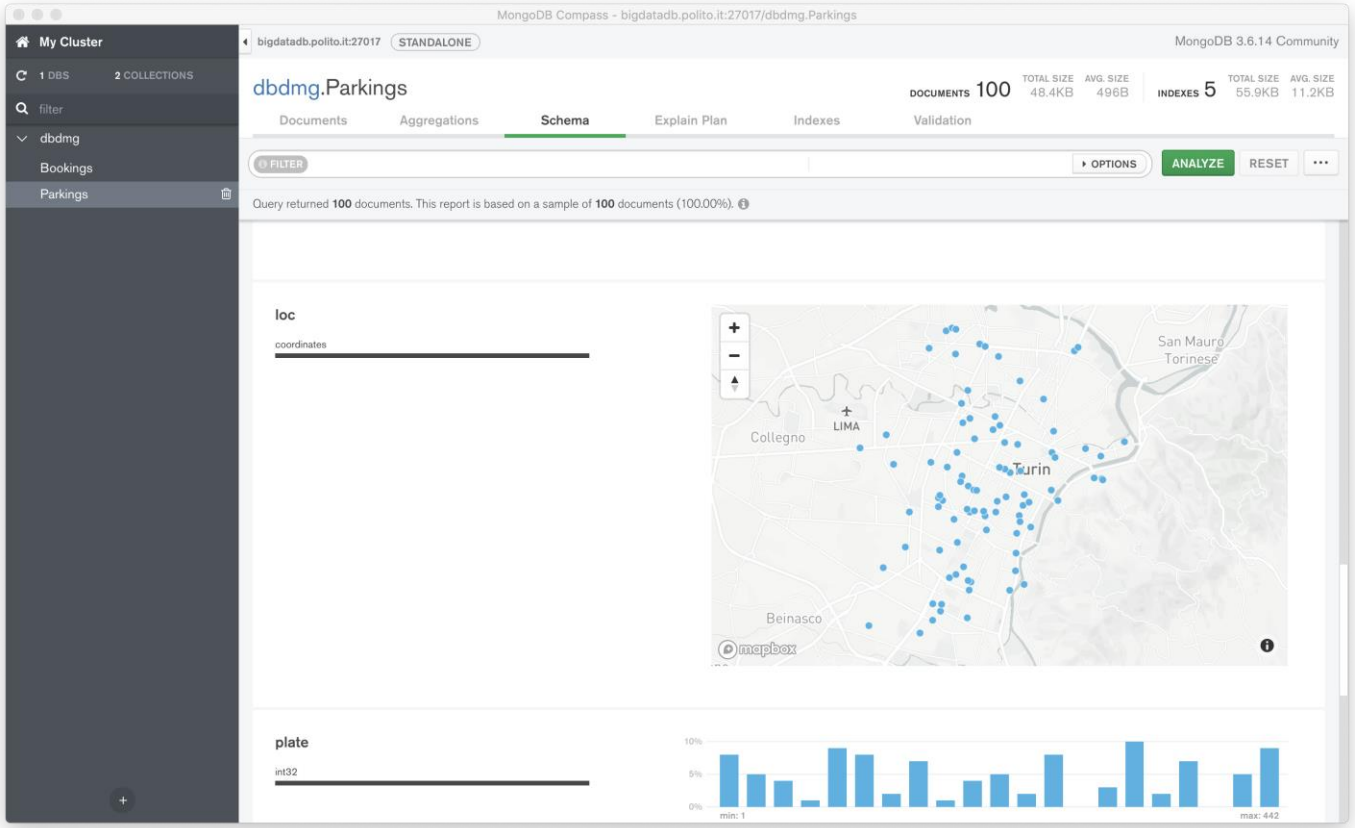
- Connect to local or remote instances of MongoDB.

MongoDB Compass



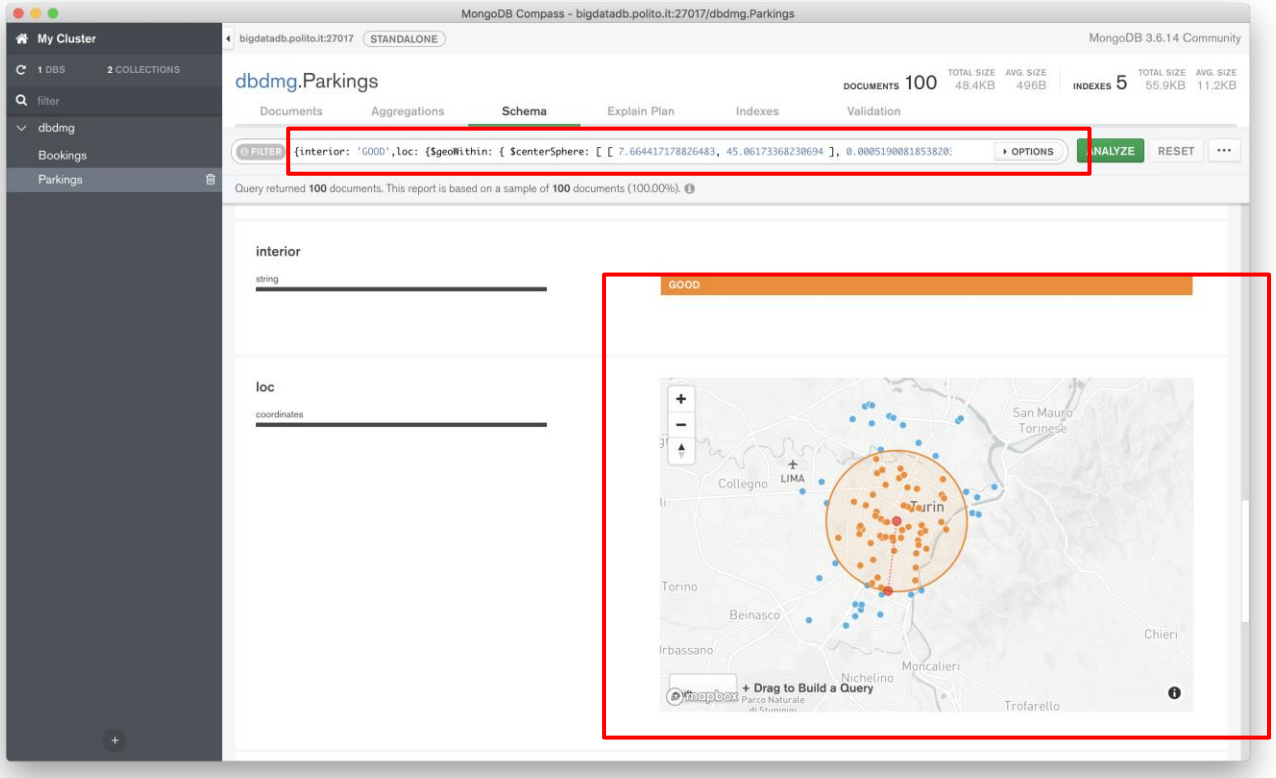
- Get an overview of the data in list or table format.

MongoDB Compass



- Analyze the documents and their fields.
- Native support for geospatial coordinates.

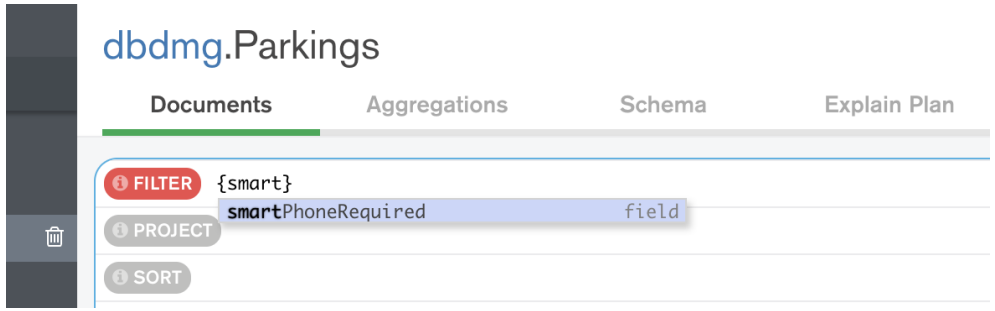
MongoDB Compass



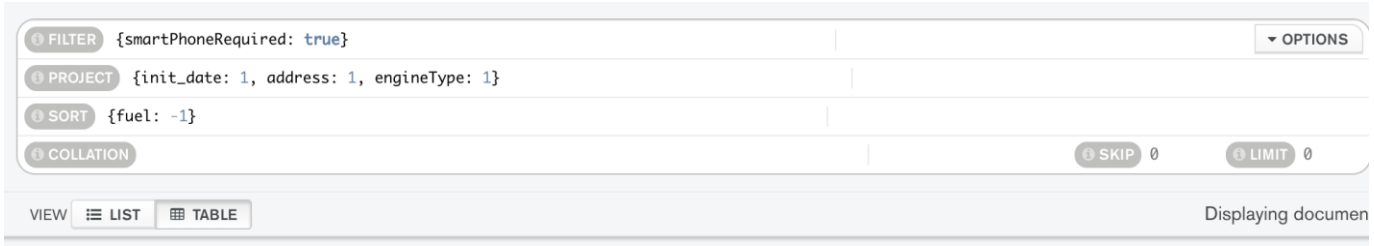
- Visually build the query conditioning on analyzed fields.

MongoDB Compass

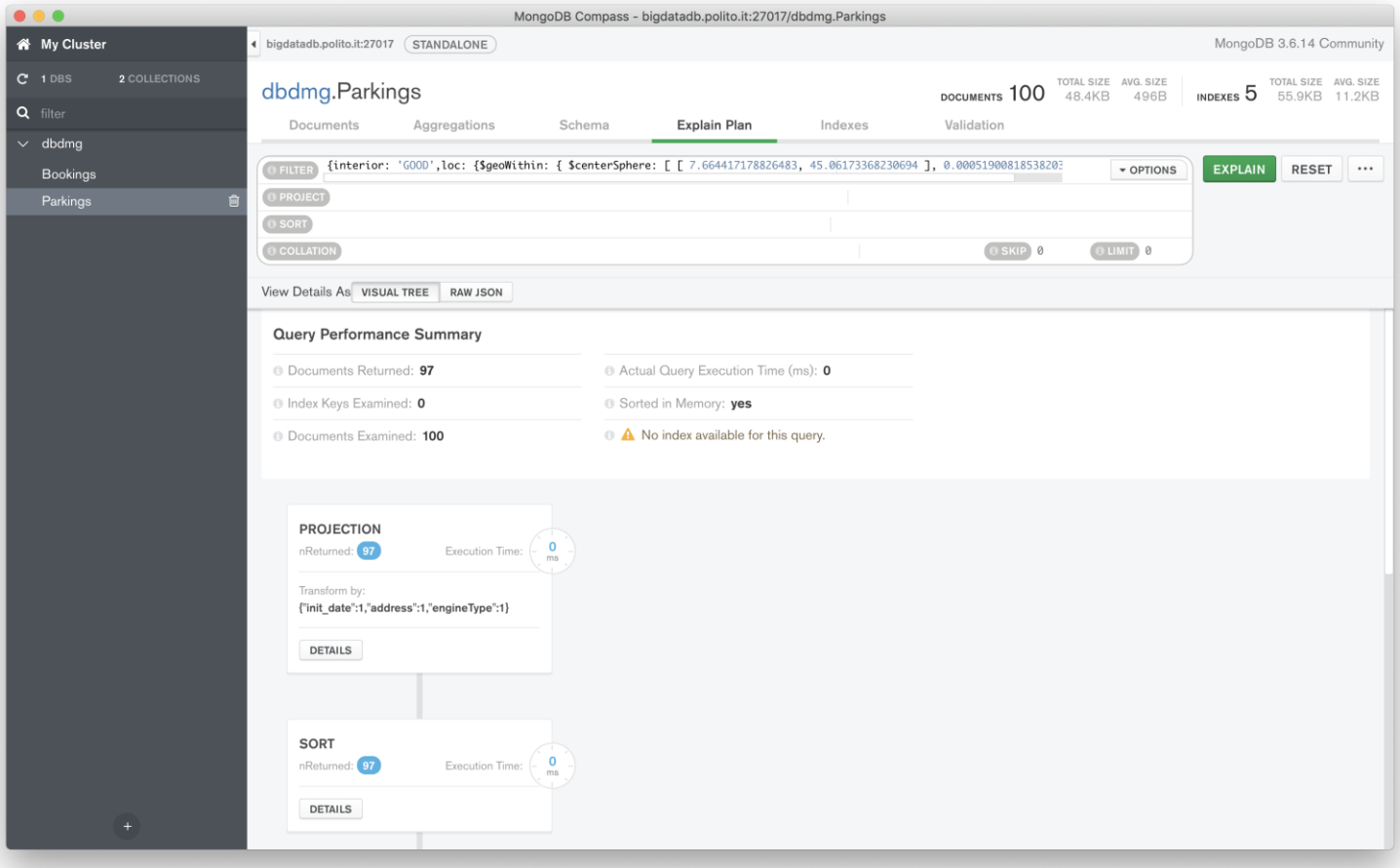
- Autcomplete enabled by default



- Construct the query step by step.

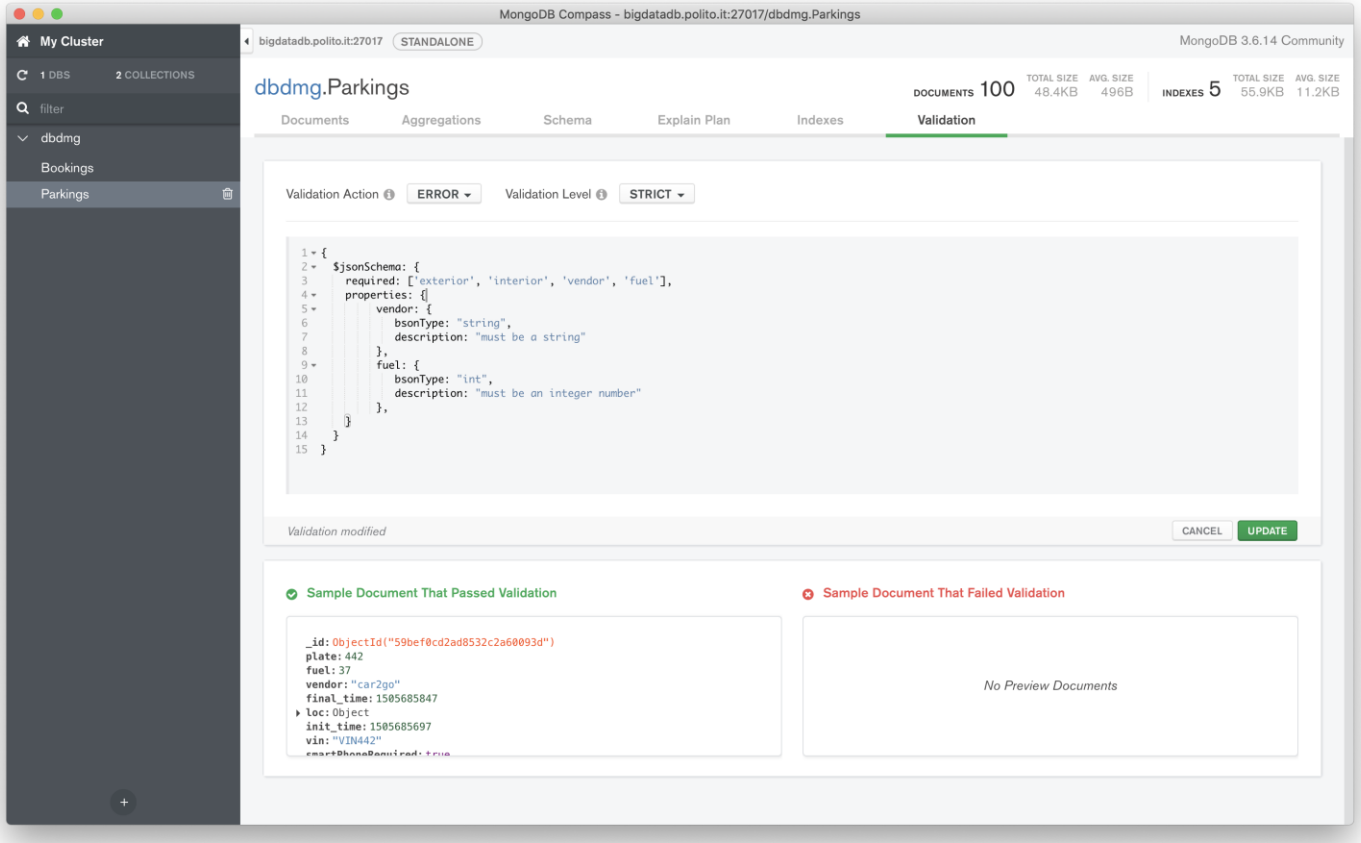


MongoDB Compass



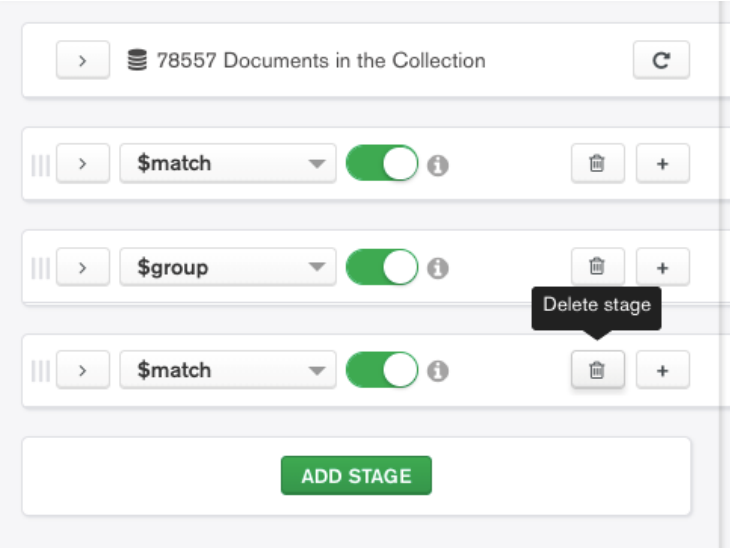
- Analyze query performance and get hints to speed it up.

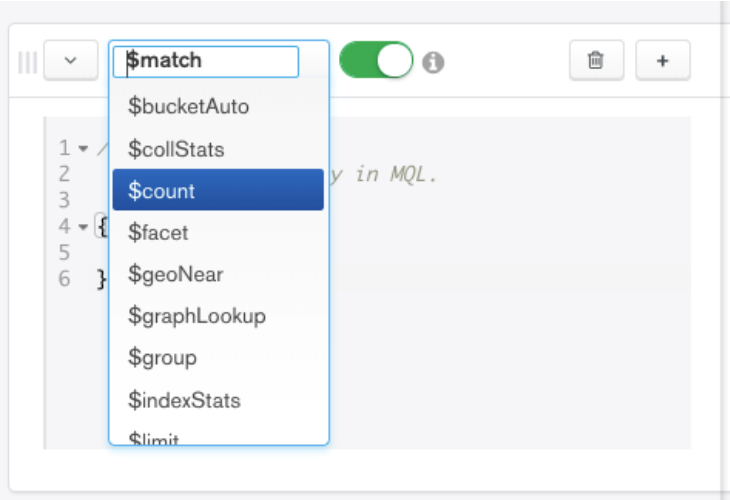
MongoDB Compass



- Specify contraints to validate data
- Find unconsistent documents.

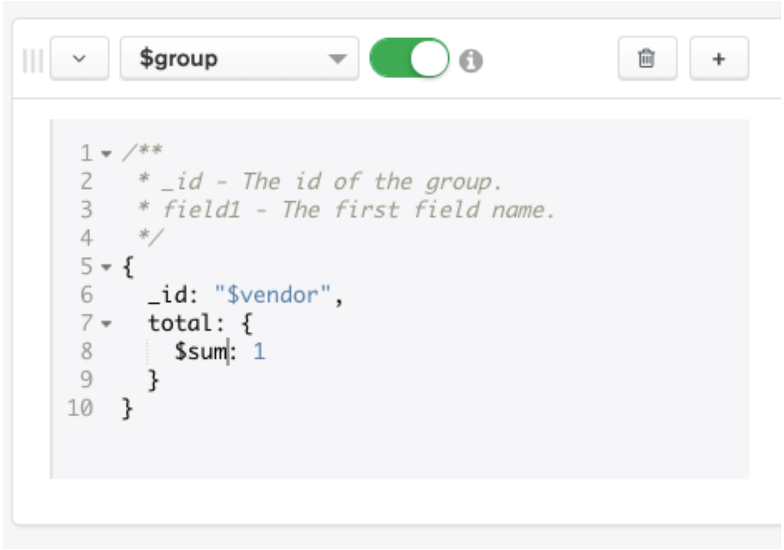
MongoDB Compass: Aggregation



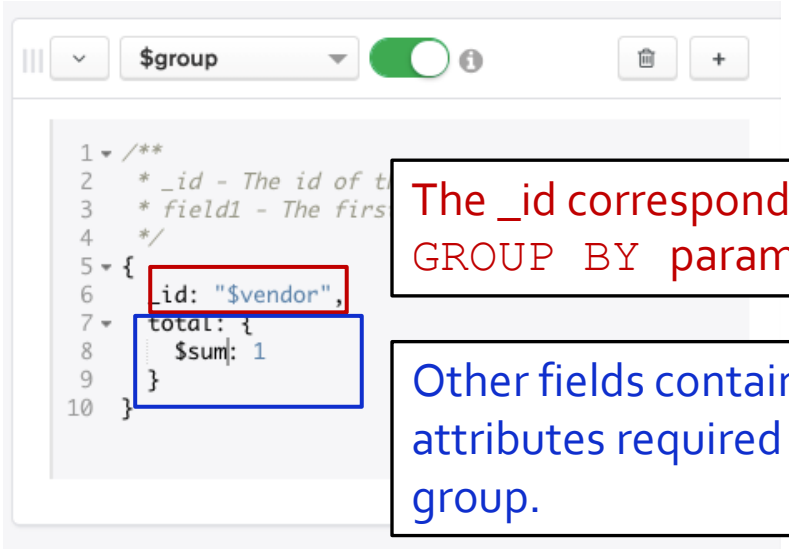


- Build a pipeline consisting of multiple aggregation stages
- Define the filter and aggregation attributes for each operator.

MongoDB Compass: Aggregation stages



MongoDB Compass: Aggregation stages



The _id corresponds to the `GROUP BY` parameter in SQL

Other fields contain the attributes required for each group.

One group for each "vendor".

MongoDB Compass: Pipelines