Introduction to MongoDB

MongoDB: Introduction

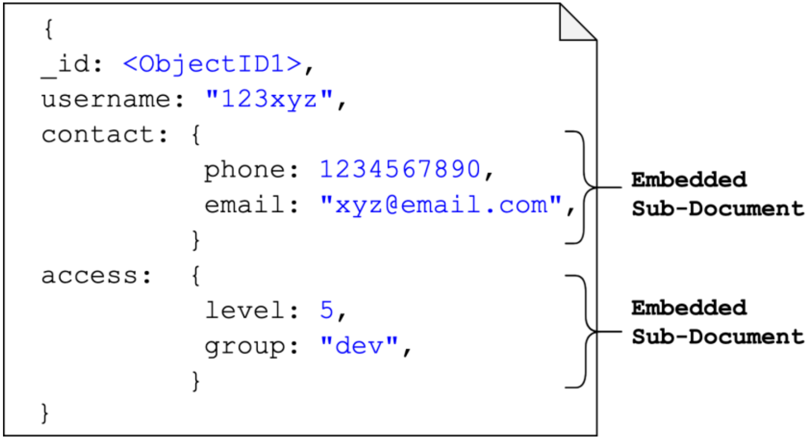
- The leader in the NoSQL Document-based databases
- Full of features, beyond NoSQL
  - High performance
  - High availability
  - Native scalability
  - High flexibility
  - Open source

Terminology – Approximate mapping

Relational database	MongoDB
Table	Collection
Record	Document
Column	Field

MongoDB: Document Data Design

- High-level, business-ready representation of the data
  - Records are stored into Documents
    - field-value pairs
    - similar to JSON objects
    - may be nested



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

MongoDB: Document Data Design

- High-level, business-ready representation of the data
- Flexible and rich syntax, adapting to most use cases
- Mapping into developer-language objects
  - year, month, day, timestamp,
  - lists, sub-documents, etc.

MongoDB: Main features

- Rich query language
  - Documents can be created, read, updated and deleted.
  - The **SQL language** is **not supported**
  - APIs available for many programming languages
    - JavaScript, PHP, Python, Java, C#, ..

MongoDB: 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()**

MongoDB: Read data from documents

- Select documents
  - `db.<collection name>.find( {<conditions>}, {<fields of interest>} );`
- E.g.,

```
db.people.find();
```

  - Returns all documents contained in the people collection

MongoDB: Read data from documents

- 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

MongoDB: Read data from documents

- 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

MongoDB: Read data from documents

- E.g.,

```
db.people.find().pretty();
```

  - No conditions and no fields of interest
    - Returns all documents contained in the people collection
    - `pretty()` displays the results in an easy-to-read format

```
db.people.find({age:55})
```

  - One condition on the value of age
    - Returns all documents having *age* equal to 55

MongoDB: Read data from documents

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

- No conditions, but returns a specific set of fields of interest
  - Returns only **user_id** and **status** of all documents contained in the people collection
  - Default of fields is false, except for _id

```
db.people.find({ status: "A",  age: 55})
```

- `status = “A”` and `age = 55`
  - Returns all documents having **status = “A”** and **age = 55**

MongoDB: find() operator

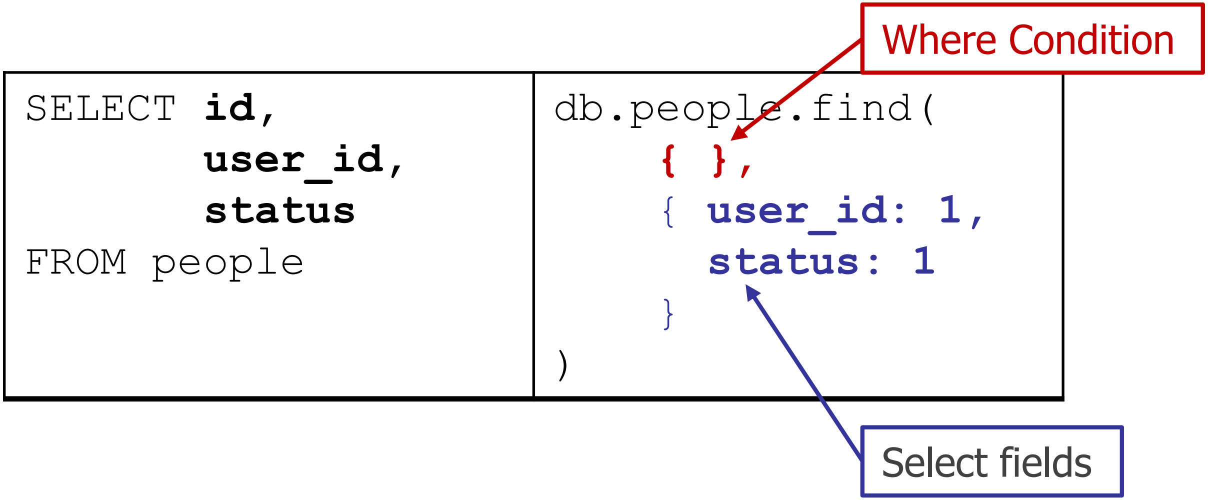
MySQL clause	MongoDB operator
SELECT	find()

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

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

MongoDB: find() operator

MySQL clause	MongoDB operator
SELECT	find()



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

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

MongoDB: find() operator

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

```
SELECT *
FROM people
WHERE status = "A"
```

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

Where Condition

MongoDB: find() operator

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

Where Condition

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

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

Selection fields

By default, the `_id` field is shown.
To remove it from visualization use: `_id: 0`

MongoDB: find() operator

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”
      }
}
```

nested document

MongoDB: Read data from documents

```
db.people.find({ age: { $gt: 25, $lte: 50 } })
```

- Age greater than 25 and less than or equal to 50
  - Returns all documents having **age > 25 and age <= 50**

```
db.people.find({$or:[{status: "A"},{age: 55}]})
```

- Status = “A” or age = 55
  - Returns all documents having **status=“A” or age=55**

```
db.people.find({ status: {$in:["A", "B"]}})
```

- Status = “A” or status = B
  - Returns all documents where the **status** field value is **either “A” or “B”**

MongoDB: Read data from documents

- 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.

MongoDB: (no) joins

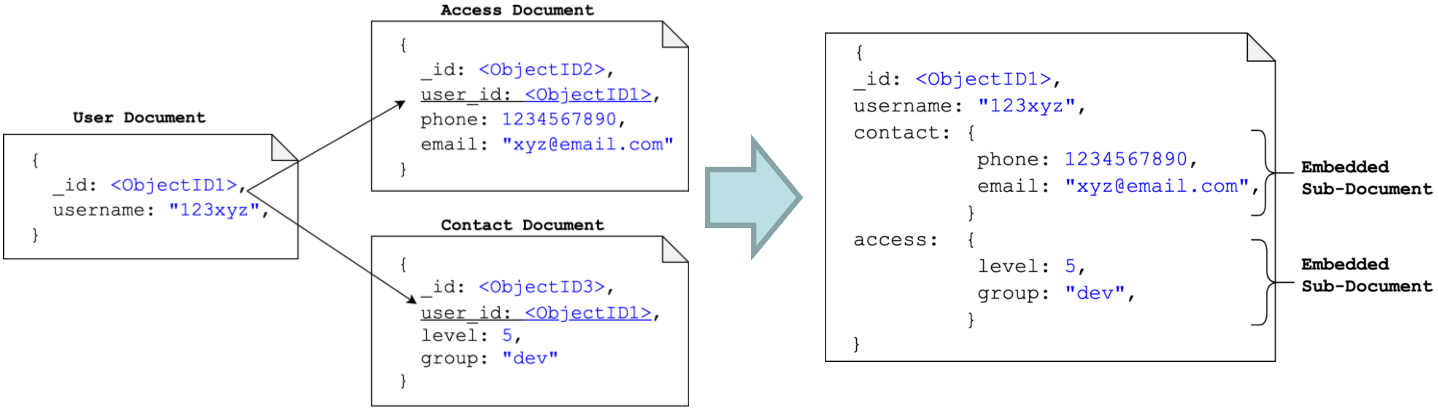
- There are other operators for selecting data from MongoDB collections
- However, no join operator exists (but `$lookup`)
  - You must write a program that
    - Selects the documents of the first collection you are interested in
    - Iterates over the documents returned by the first step, by using 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

MongoDB: (no) joins

(no) joins

- Relations among documents/records are provided by
  - Object(ID) reference, with **no native join**
  - **DBRef**, across collections and databases



User Document

```
{
  _id: <ObjectID1>,
  username: "123xyz",
}
```

Access Document

```
{
  _id: <ObjectID2>,
  user_id: <ObjectID1>,
  phone: 1234567890,
  email: "xyz@email.com"
}
```

Contact Document

```
{
  _id: <ObjectID3>,
  user_id: <ObjectID1>,
  level: 5,
  group: "dev"
}
```

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

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

MongoDB: comparison operators

- In SQL language, comparison operators are essential to express conditions on data.
- In Mongo query language they are available with a different syntax.

MySQL	MongoDB	Description
>	$gt	greater than
>=	$gte	greater equal then
<	$lt	less than
<=	$lte	less equal then
=	$eq	equal to
!=	$neq	not equal to

MongoDB: 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
`$ne`	Matches all values that are not equal to a specified value
`$nin`	Matches none of the values specified in an array

MongoDB: comparison operators (>)

MySQL	MongoDB	Description
>	$gt	greater than

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

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

MongoDB: comparison operators (>=)

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

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

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

MongoDB: 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 }** }
	)

MongoDB: 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 } }
)
```

MongoDB: 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 }** }
	)

MongoDB: comparison operators (!=)

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

SELECT *
FROM people
WHERE **age != 25**
	db.people.find(
	    { **age: { $neq: 25 }** }
	)

MongoDB: 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

MongoDB: 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 }
)
```

MongoDB: 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 }
   ]
}
)
```

MongoDB: 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()`

MongoDB: Cursor

- Cursor examples:

```
db.people.find({ status: "A"}).count()
```

- Select documents with status="A" and count them.

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

- `forEach` applies a JavaScript function to apply to each document from the cursor.
  - Select documents with status="A" and print the document name.

MongoDB: 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

MongoDB: sorting data

- E.g.,

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

  - Select documents with status="A" and sort them in ascending order based on the age value
    - Returns all documents having status="A". The result is sorted in ascending age order

MongoDB: sorting data

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

MySQL clause	MongoDB operator
ORDER BY	sort()

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

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

MongoDB: sorting data

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

MySQL clause	MongoDB operator
ORDER BY	sort()

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

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

MongoDB: counting

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

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

MongoDB: counting

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

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

SELECT COUNT(*)
FROM people
WHERE **age > 30**
	db.people.count(
	{ age: { $gt: 30 } }
	)

Introduction to data aggregation

Aggregation in MongoDB

- Aggregation operations process data records and return computed results.
- Documents enter a multi-stage pipeline that transforms the documents into an aggregated result.

MongoDB: Aggregation Framework

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

MongoDB: Aggregation

- 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

MongoDB: Aggregation

```
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”

MongoDB: Aggregation

```
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

MongoDB: Aggregation

```
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”

MongoDB: Aggregation

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

- Creates one group of documents for each value of status and counts the number of documents per group
  - Returns one value for each group containing the value of the grouping field and an integer representing the number of documents

MongoDB: Aggregation

```
db.people.aggregate( [
   { $group: { _id: "$status",
            count: { $sum: 1 }
         }
  },
  { $match: { count: { $gte: 3 } } }
] )
```

- Creates one group of documents for each value of status and counts the number of documents per group. Returns only the groups with at least 3 documents

MongoDB: Aggregation

```
db.people.aggregate( [
   { $group: { _id: "$status",
               count: { $sum: 1 }
            }
  },
  { $match: { count: { $gte: 3 } } }
] )
```

Having condition

- Creates one group of documents for each value of status and counts the number of documents per group. Returns only the groups with at least 3 documents

MongoDB: Aggregation Framework

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

Aggregation in MongoDB: 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 MongoDB: 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 MongoDB: 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 MongoDB: Group By

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 MongoDB: Group By

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 MongoDB: Group By

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

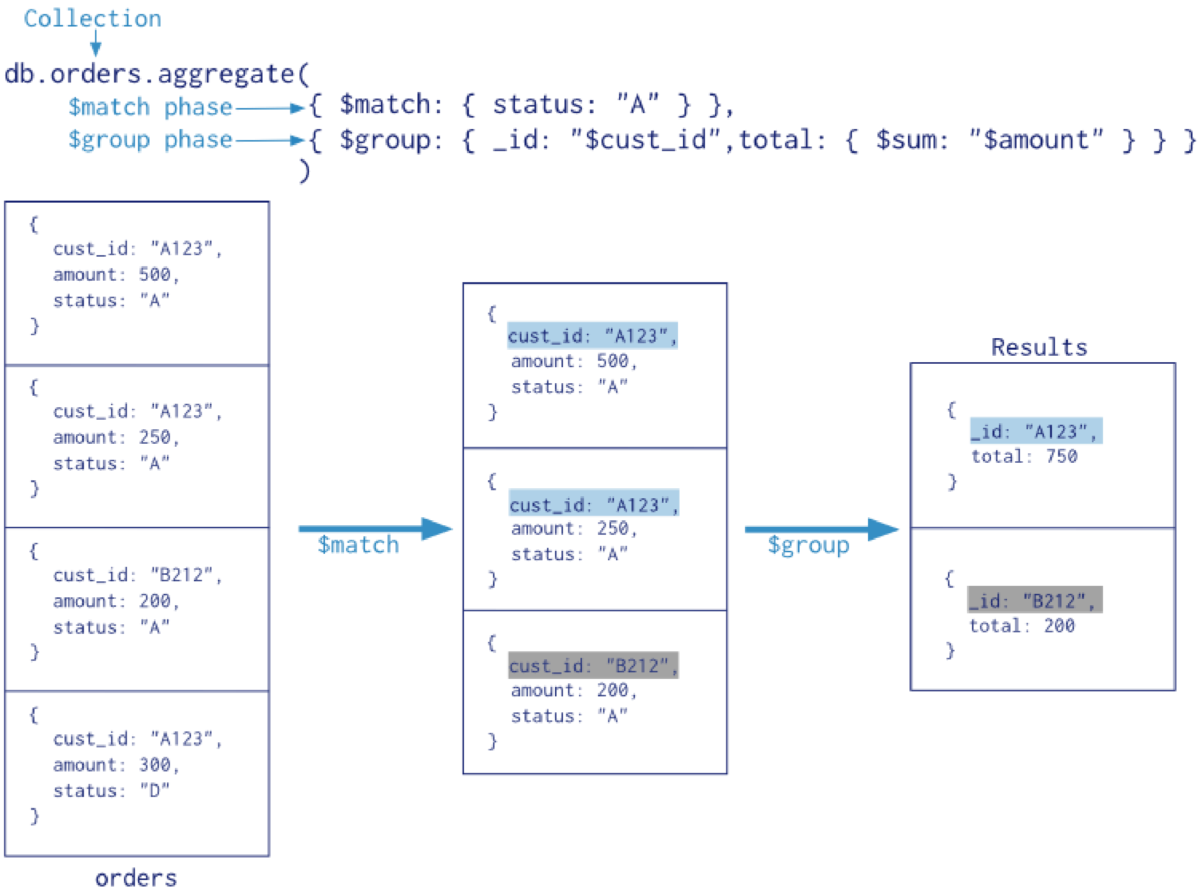
```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

```
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 in MongoDB



Collection

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

```
{
  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

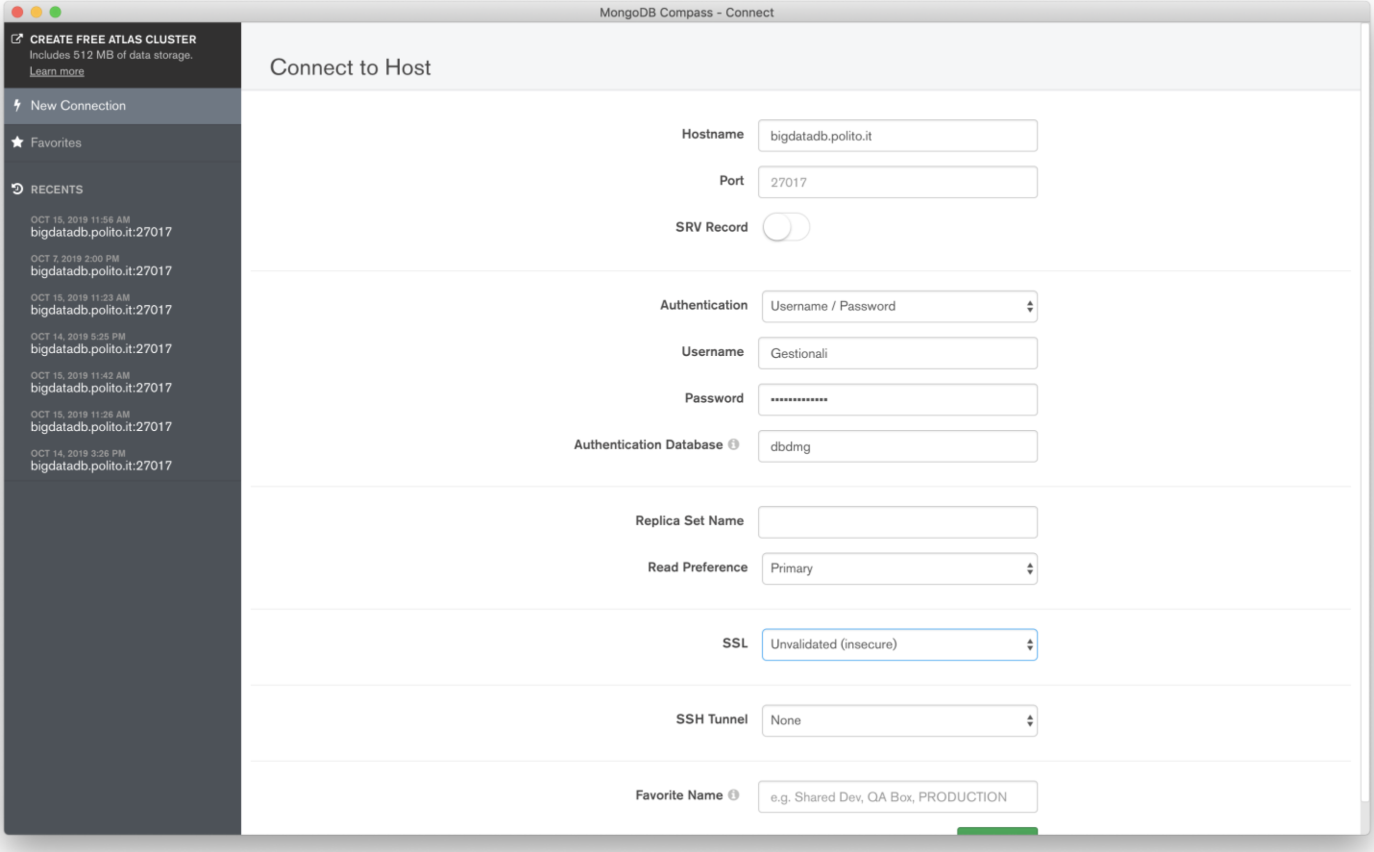
```
{
  _id: "A123",
  total: 750
}
{
  _id: "B212",
  total: 200
}
```

GUI for Mongo DB

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



MongoDB Compass - Connect
CREATE FREE ATLAS CLUSTER
Includes 512 MB of data storage.
Learn more
New Connection
Favorites
RECENTS

OCT 15, 2019 11:56 AM
bigdatadb.polito.it:27017

OCT 7, 2019 2:00 PM
bigdatadb.polito.it:27017

OCT 15, 2019 11:23 AM
bigdatadb.polito.it:27017

OCT 14, 2019 5:25 PM
bigdatadb.polito.it:27017

OCT 15, 2019 11:42 AM
bigdatadb.polito.it:27017

OCT 15, 2019 11:26 AM
bigdatadb.polito.it:27017

OCT 14, 2019 3:26 PM
bigdatadb.polito.it:27017

Connect to Host

Hostname: bigdatadb.polito.it

Port: 27017

SRV Record: ○

Authentication: Username / Password

Username: Gestionali

Password: ••••••••••••••

Authentication Database: dbdmg

Replica Set Name:

Read Preference: Primary

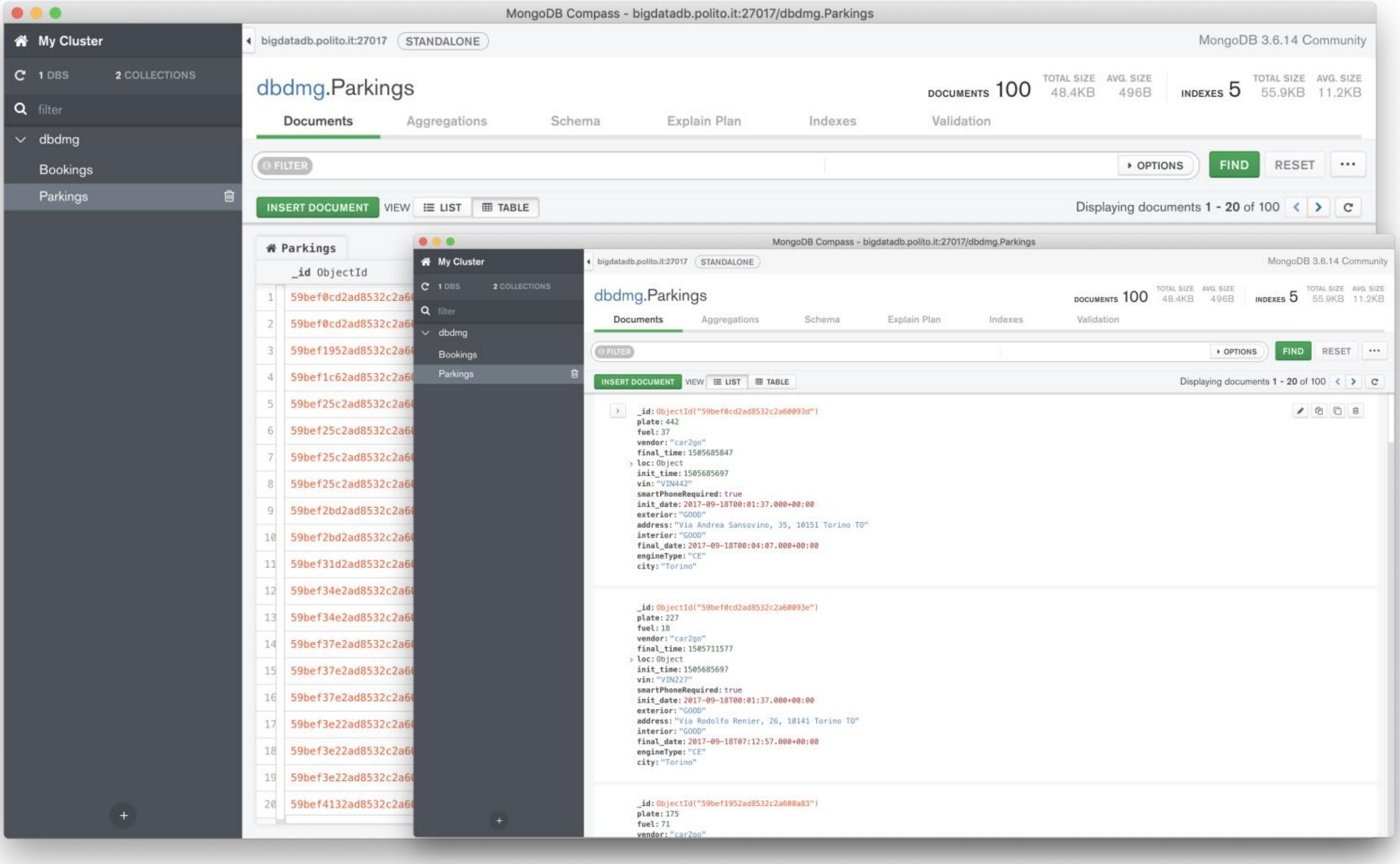
SSL: Unvalidated (insecure)

SSH Tunnel: None

Favorite Name: e.g. Shared Dev, QA Box, PRODUCTION

- Connect to local or remote instances of MongoDB.

MongoDB Compass



	_id ObjectId
1	59bef0cd2ad8532c2a6
2	59bef0cd2ad8532c2a6
3	59bef1952ad8532c2a6
4	59bef1c62ad8532c2a6
5	59bef25c2ad8532c2a6
6	59bef25c2ad8532c2a6
7	59bef25c2ad8532c2a6
8	59bef25c2ad8532c2a6
9	59bef2bd2ad8532c2a6
10	59bef2bd2ad8532c2a6
11	59bef31d2ad8532c2a6
12	59bef34e2ad8532c2a6
13	59bef34e2ad8532c2a6
14	59bef37e2ad8532c2a6
15	59bef37e2ad8532c2a6
16	59bef37e2ad8532c2a6
17	59bef3e22ad8532c2a6
18	59bef3e22ad8532c2a6
19	59bef3e22ad8532c2a6
20	59bef4132ad8532c2a6

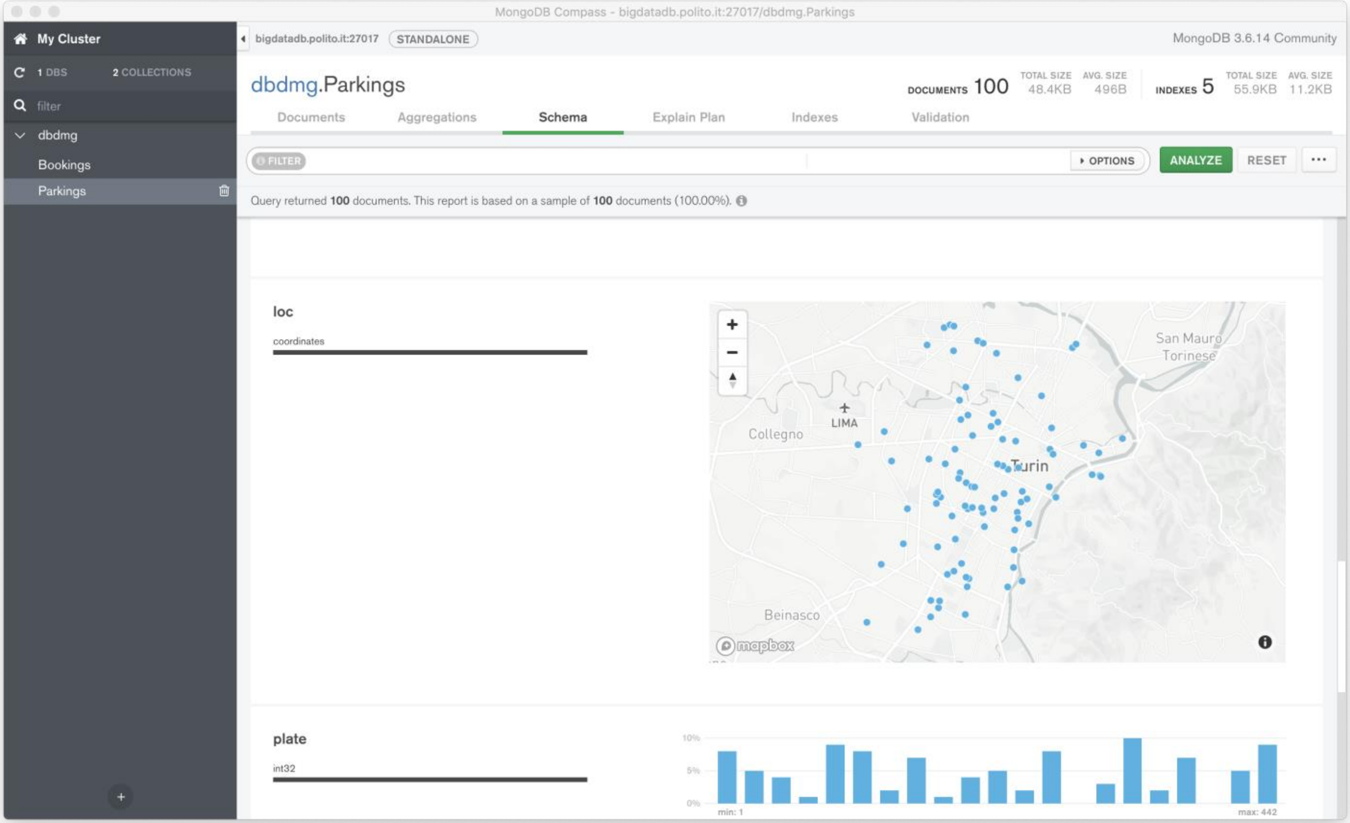
```
_id: ObjectId("59bef0cd2ad8532c2a60093d")
plate: 442
fuel: 37
vendor: "car2go"
final_time: 1505685847
loc: Object
init_time: 1505685697
vin: "VIN442"
smartPhoneRequired: true
init_date: 2017-09-18T00:01:37.000+00:00
exterior: "GOOD"
address: "Via Andrea Sansovino, 35, 10151 Torino TO"
interior: "GOOD"
final_date: 2017-09-18T00:04:07.000+00:00
engineType: "CE"
city: "Torino"

_id: ObjectId("59bef0cd2ad8532c2a60093e")
plate: 227
fuel: 18
vendor: "car2go"
final_time: 1505711577
loc: Object
init_time: 1505685697
vin: "VIN227"
smartPhoneRequired: true
init_date: 2017-09-18T00:01:37.000+00:00
exterior: "GOOD"
address: "Via Rodolfo Renier, 26, 10141 Torino TO"
interior: "GOOD"
final_date: 2017-09-18T07:12:57.000+00:00
engineType: "CE"
city: "Torino"

_id: ObjectId("59bef1952ad8532c2a600a83")
plate: 175
fuel: 71
vendor: "car2go"
```

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

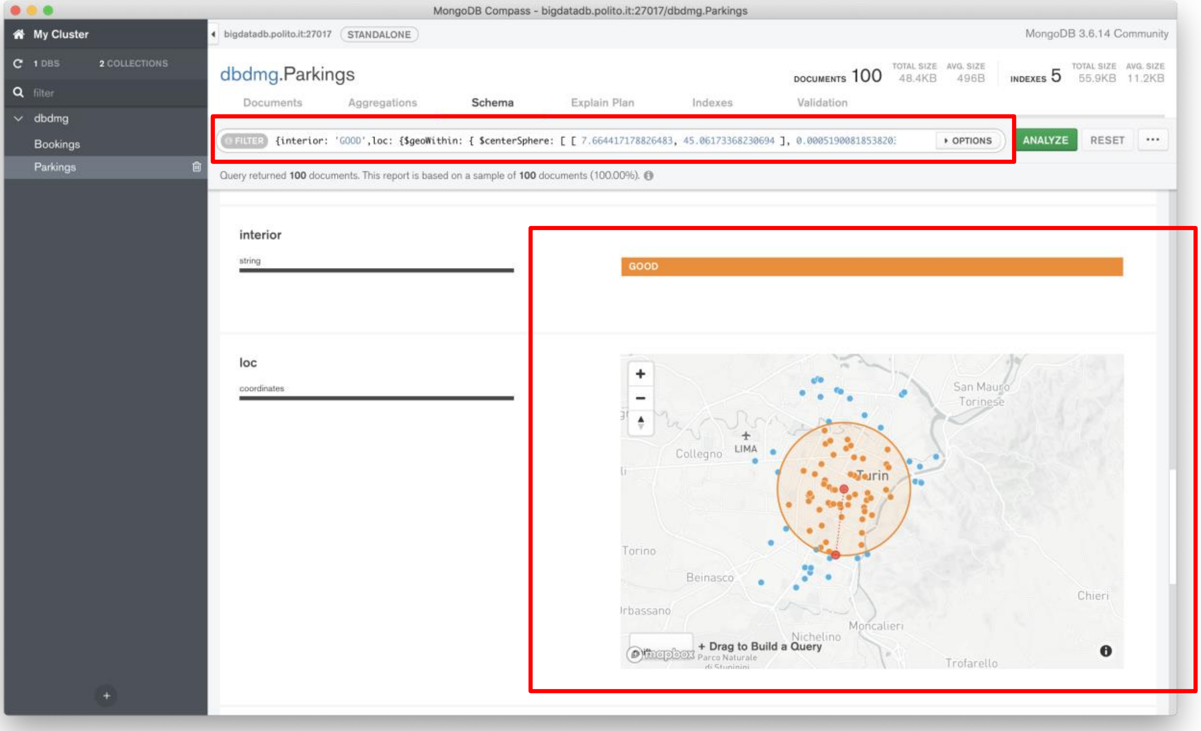
MongoDB Compass



Query returned 100 documents. This report is based on a sample of 100 documents (100.00%).

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

MongoDB Compass

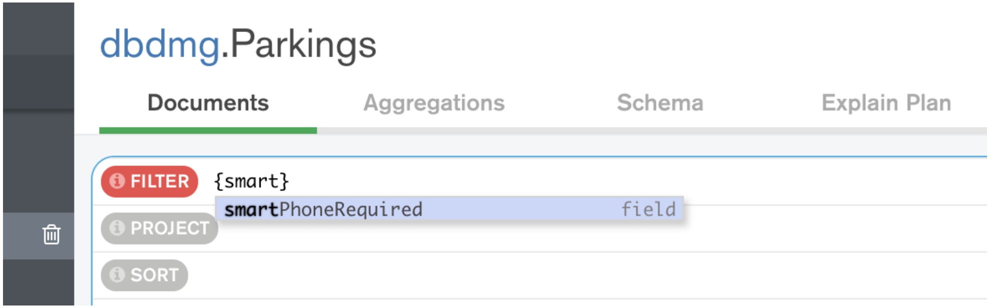


```
{interior: 'GOOD',loc: {$geoWithin: { $centerSphere: [ [ 7.664417178826483, 45.06173368230694 ], 0.0005190081853820
```

Query returned **100** documents. This report is based on a sample of **100** documents (100.00%).

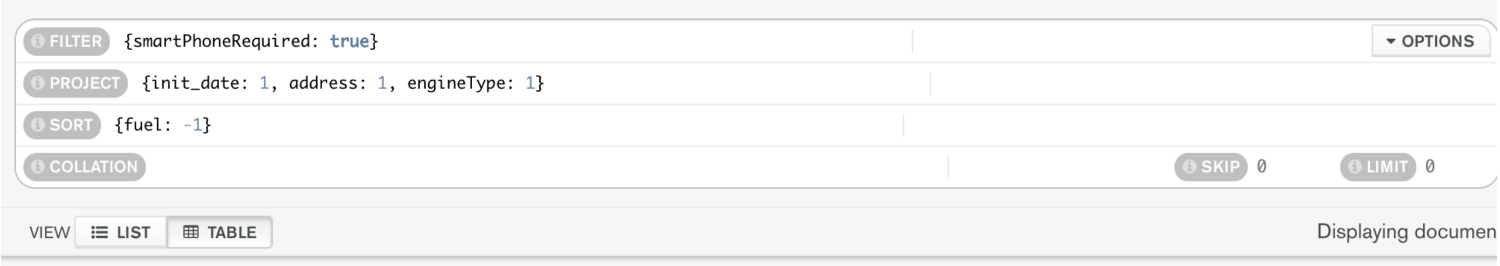
- Visually build the query conditioning on analyzed fields.

MongoDB Compass



```
{smart}
smartPhoneRequired          field
```

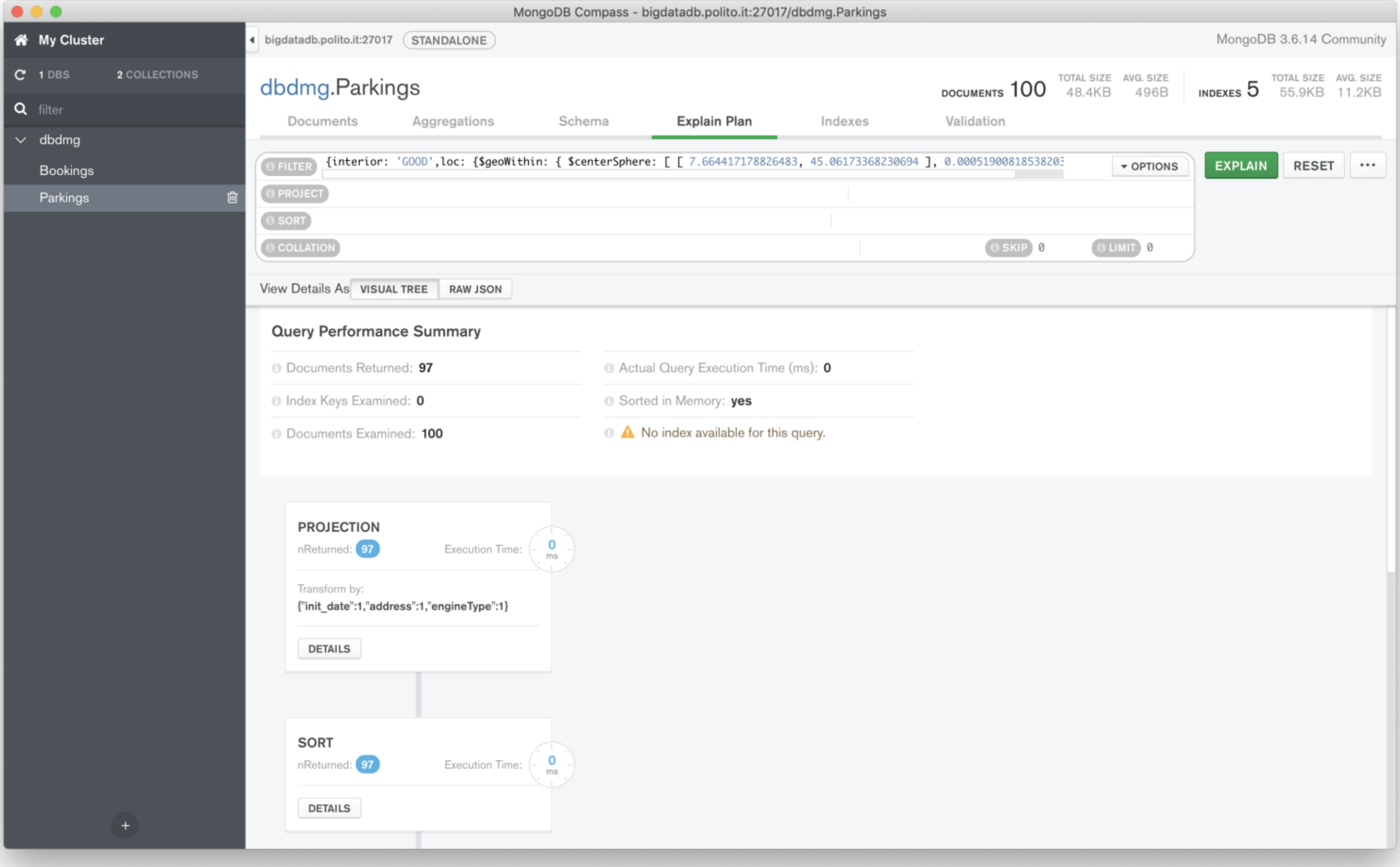
- Autocomplete enabled by default.



```
FILTER      {smartPhoneRequired: true}
PROJECT     {init_date: 1, address: 1, engineType: 1}
SORT        {fuel: -1}
SKIP 0      LIMIT 0
```

- Construct the query step by step.

MongoDB Compass



```
{interior: 'GOOD',loc: {$geoWithin: { $centerSphere: [ [ 7.664417178826483, 45.06173368230694 ], 0.00051900818538203
```

Query Performance Summary

- Documents Returned: **97**
- Index Keys Examined: **0**
- Documents Examined: **100**
- Actual Query Execution Time (ms): **0**
- Sorted in Memory: **yes**
- ⚠ No index available for this query.

PROJECTION

nRetured: 97 Execution Time: 0 ms

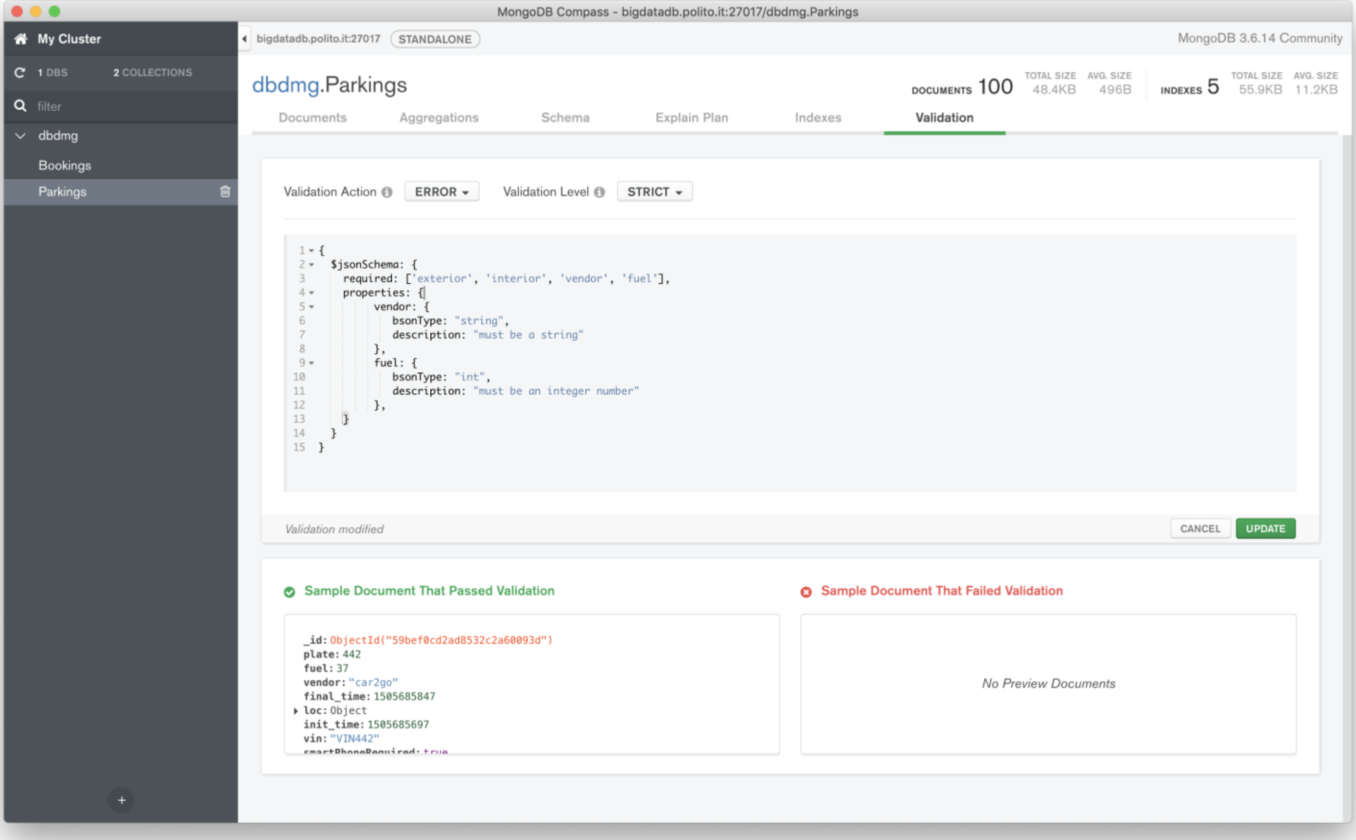
Transform by:
{"init_date":1,"address":1,"engineType":1}

SORT

nReturned: 97 Execution Time: 0 ms

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

MongoDB Compass



Validation Action: ERROR

Validation Level: STRICT

```
{
  $jsonSchema: {
    required: ['exterior', 'interior', 'vendor', 'fuel'],
    properties: {
          vendor: {
             bsonType: "string",
             description: "must be a string"
          },
          fuel: {
             bsonType: "int",
             description: "must be an integer number"
          },
    }
  }
}
```

Validation modified

Sample Document That Passed Validation

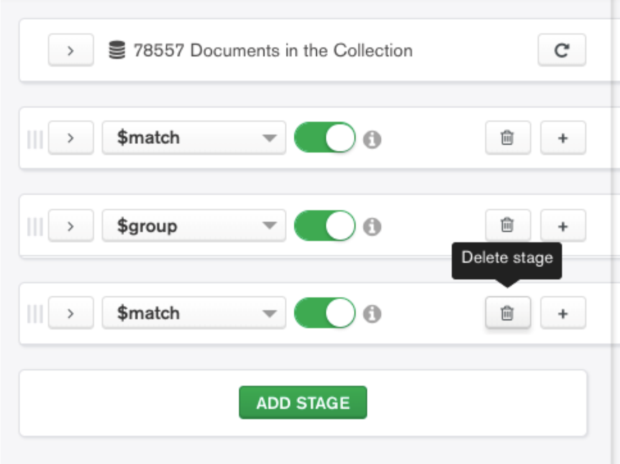
```
 _id: ObjectId("59bef0cd2ad8532c2a60093d")
 plate: 442
 fuel: 37
 vendor: "car2go"
 final_time: 1505685847
▸ loc: Object
 init_time: 1505685697
 vin: "VIN442"
 smartPhoneRequired: true
```

Sample Document That Failed Validation

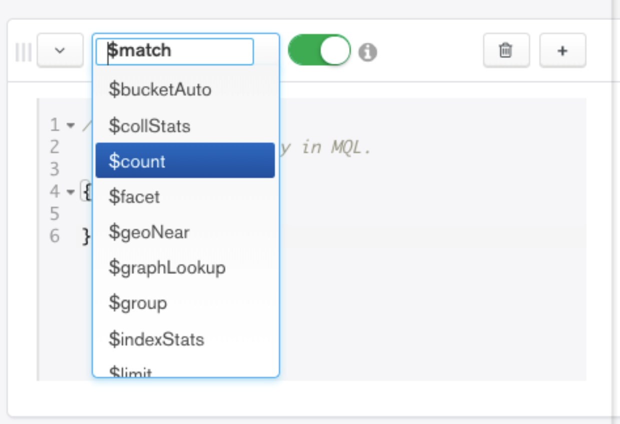
No Preview Documents

- Specify contraints to validate data
- Find unconsistent documents.

MongoDB Compass: Aggregation



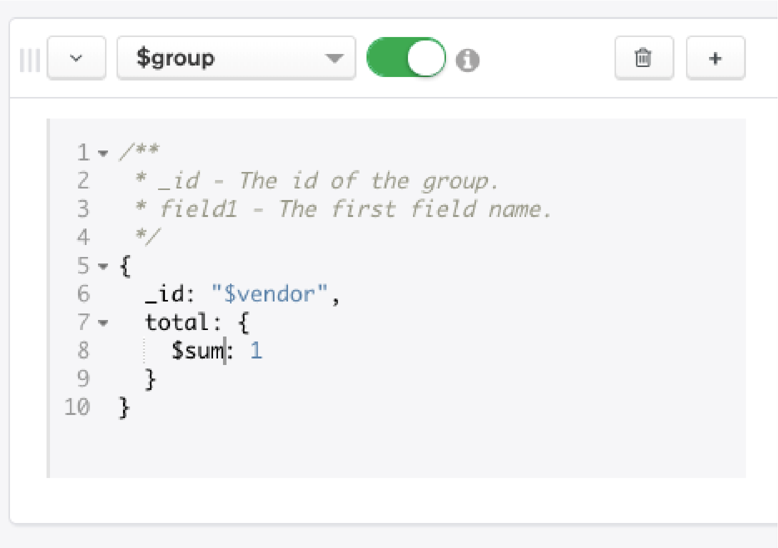
78557 Documents in the Collection



- Build a pipeline consisting of multiple aggregation stages.

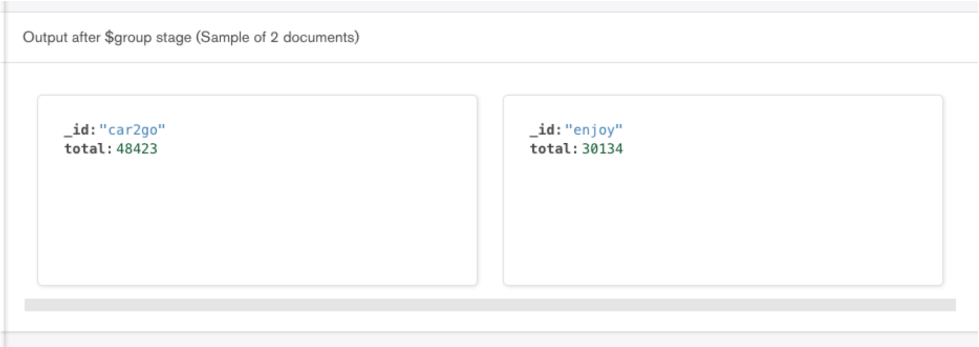
- Define the filter and aggregation attributes for each operator.

MongoDB Compass: Aggregation stages



$group

```
/**
 * _id - The id of the group.
 * field1 - The first field name.
 */
{
  _id: "$vendor",
  total: {
    $sum: 1
  }
}
```

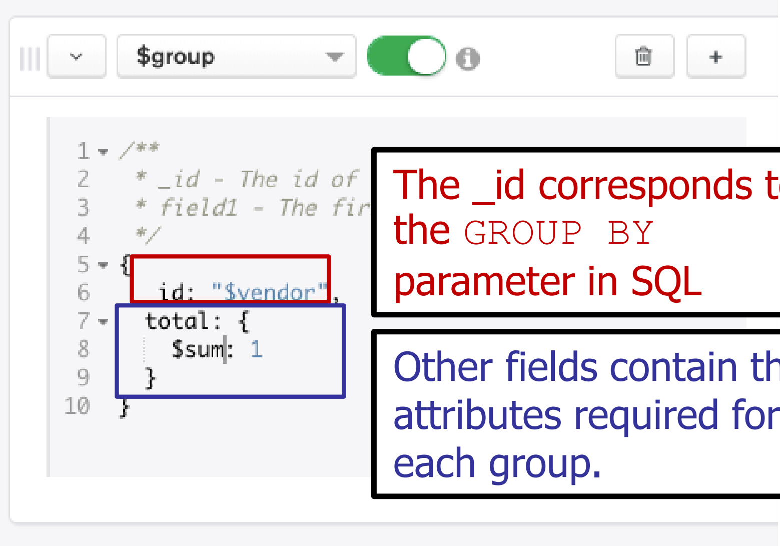


Output after $group stage (Sample of 2 documents)

```
_id:"car2go"
total:48423
```

```
_id:"enjoy"
total:30134
```

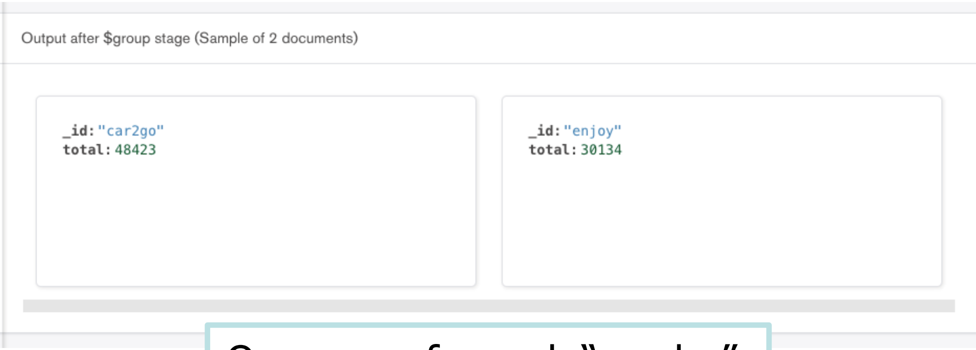
MongoDB Compass: Aggregation stages



```
/**
 * _id - The id of
 * field1 - The fir
 */
{
  _id: "$vendor",
  total: {
    $sum: 1
  }
}
```

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

Other fields contain the attributes required for each group.



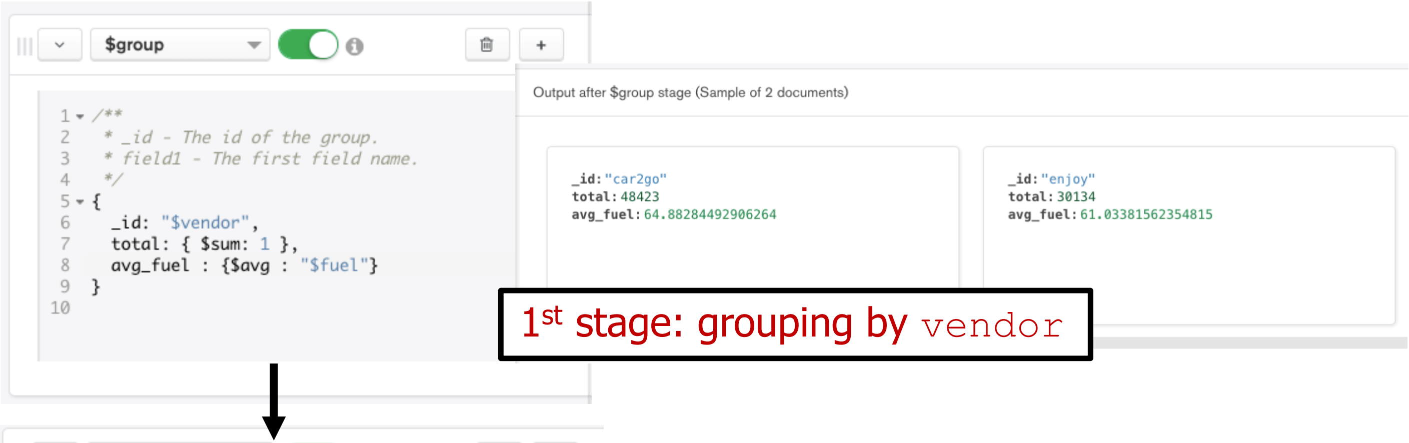
Output after $group stage (Sample of 2 documents)

```
_id:"car2go"
total:48423
```

```
_id:"enjoy"
total:30134
```

One group for each “vendor”.

MongoDB Compass: Pipelines



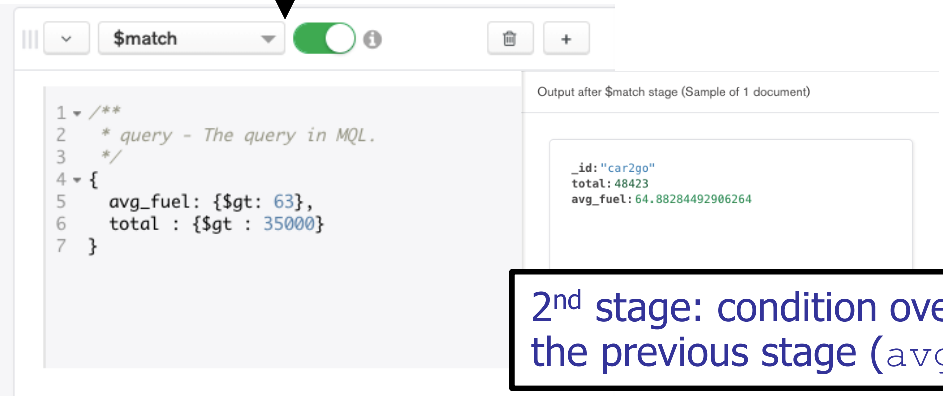
```
/**
 * _id - The id of the group.
 * field1 - The first field name.
 */
{
  _id: "$vendor",
  total: { $sum: 1 },
  avg_fuel : {$avg : "$fuel"}
}
```

Output after $group stage (Sample of 2 documents)

```
_id: "car2go"
total: 48423
avg_fuel: 64.88284492906264
```

```
_id: "enjoy"
total: 30134
avg_fuel: 61.03381562354815
```

1st stage: grouping by `vendor`



```
/**
  * query - The query in MQL.
  */
{
  avg_fuel: {$gt: 63},
  total : {$gt : 35000}
}
```

Output after $match stage (Sample of 1 document)

```
_id: "car2go"
total: 48423
avg_fuel: 64.88284492906264
```

2nd stage: condition over fields created in the previous stage (`avg_fuel, total`).

Indexing

MongoDB: Indexes

- 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.

MongoDB: Indexes

- MongoDB provides different data-type indexes
  - Single field indexes
  - Compound field indexes
  - Multikey indexes
  - Geospatial indexes
  - Text indexes
  - Hashed indexes

MongoDB: Create new indexes

- Creating an index

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

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

- Options include: `name`, `unique` (whether to accept or not insertion of documents with duplicate index keys), `background`, `dropDups`, ..

MongoDB: 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} )
    ```

MongoDB: 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]`

MongoDB: 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

MongoDB: Indexes

- `$near` syntax:

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

MongoDB: Indexes

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

MongoDB: 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

MongoDB: 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”} )
```