

# 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

<b>Relational database</b>	<b>MongoDB</b>
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",
  }
}
```

Embedded Sub-Document

Embedded Sub-Document

# 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	<code>find()</code>

<b>SELECT</b> * FROM people	<code>db.people.find()</code>
--------------------------------	-------------------------------

# 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()

<pre>SELECT id,        user_id,        status FROM people</pre>	<pre>db.people.find(     { },     { user_id: 1,       status: 1     } )</pre>
-----------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------

# MongoDB: find() operator

MySQL clause	MongoDB operator
SELECT	find()

<pre>SELECT id,       user_id,       status FROM people</pre>	<pre>db.people.find(   { },   { user_id: 1,     status: 1   } )</pre>
---------------------------------------------------------------------------	-------------------------------------------------------------------------------------------

Where Condition

Select fields

# MongoDB: find() operator

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

<pre>SELECT * FROM people WHERE status = "A"</pre>	<pre>db.people.find(     { status: "A" } )</pre>
------------------------------------------------------------	----------------------------------------------------------



Where Condition

# MongoDB: find() operator

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

Where Condition

<pre>SELECT user_id, status FROM people WHERE status = "A"</pre>	<pre>db.people.find(   { status: "A" },   { user_id: 1,     status: 1,     _id: 0   } )</pre>
------------------------------------------------------------------	-----------------------------------------------------------------------------------------------

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

Selection fields

# MongoDB: find() operator

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

	<pre>db.people.find(   {"address.city": "Rome" } )</pre>
--	------------------------------------------------------------------

```
{ _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

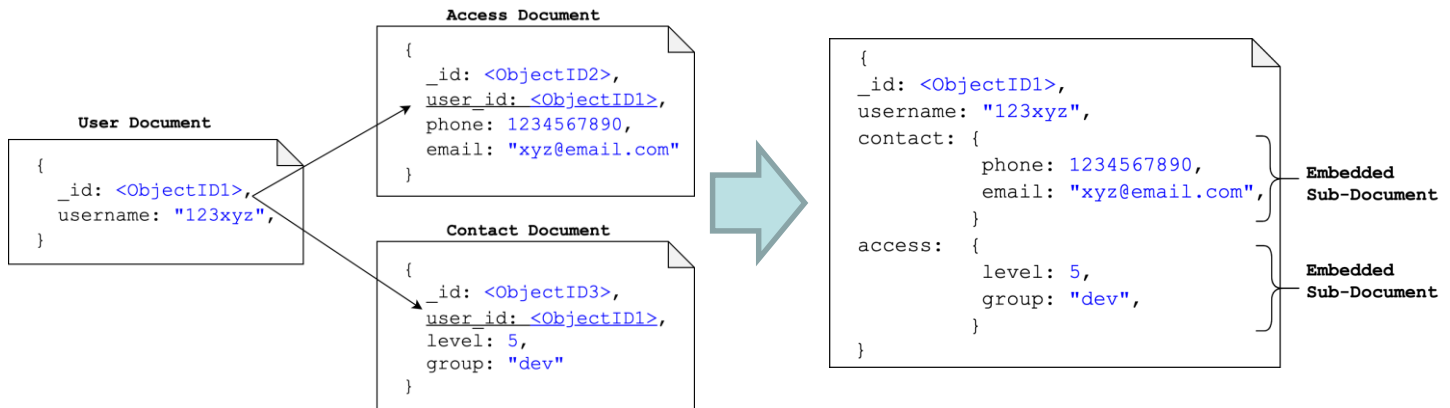
- There are other operators for selecting data from MongoDB collections
- However, no join operator exists (but `$lookup`)
  - You must write a program that
    - Selects the documents of the first collection you are interested in
    - Iterates over the documents returned by the first step, by using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



# 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
<code>\$eq</code> or <code>:</code>	Matches values that are equal to a specified value
<code>\$gt</code>	Matches values that are greater than a specified value
<code>\$gte</code>	Matches values that are greater than or equal to a specified value
<code>\$in</code>	Matches any of the values specified in an array
<code>\$lt</code>	Matches values that are less than a specified value
<code>\$lte</code>	Matches values that are less than or equal to a specified value
<code>\$ne</code>	Matches all values that are not equal to a specified value
<code>\$nin</code>	Matches none of the values specified in an array

# MongoDB: comparison operators (>)

MySQL	MongoDB	Description
>	\$gt	greater than

<pre>SELECT * FROM people WHERE age &gt; 25</pre>	<pre>db.people.find(     { age: { \$gt: 25 } } )</pre>
-----------------------------------------------------------	----------------------------------------------------------------

# MongoDB: comparison operators ( $\geq$ )

MySQL	MongoDB	Description
$>$	<code>\$gt</code>	greater than
$\geq$	<code>\$gte</code>	greater equal then

```
SELECT *  
FROM people  
WHERE age  $\geq$  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	<b>equal to</b> The \$eq expression is equivalent to { field: <value> }.

<pre>SELECT * FROM people WHERE <b>age</b> = 25</pre>	<pre>db.people.find(     { <b>age</b>: { \$eq: 25 } } )</pre>
---------------------------------------------------------------	-----------------------------------------------------------------------

# MongoDB: comparison operators (!=)

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

```
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

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

# MongoDB: conditional operators (OR)

MySQL	MongoDB	Description
AND	,	Both verified
OR	<code>\$or</code>	At least one verified

<pre>SELECT * FROM people WHERE status = "A" OR age = 50</pre>	<pre>db.people.find(   { <b>\$or</b>:     [ { status: "A" } ,       { age: 50 }     ]   } )</pre>
----------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------

# 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 field 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	<code>sort()</code>

<pre>SELECT * FROM people WHERE status = "A" ORDER BY user_id ASC</pre>	<pre>db.people.find(   { status: "A" } )<b>.sort( { user_id: 1 } )</b></pre>
-------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------

# MongoDB: sorting data

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

MySQL clause	MongoDB operator
ORDER BY	<code>sort()</code>

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

# MongoDB: counting

MySQL clause	MongoDB operator
COUNT	<code>count()</code> or <code>find().count()</code>

<pre>SELECT COUNT(*) FROM people</pre>	<pre>db.people.count() or db.people.find().count()</pre>
--------------------------------------------	------------------------------------------------------------------

# MongoDB: counting

MySQL clause	MongoDB operator
COUNT	<code>count()</code> or <code>find().count()</code>

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

<pre>SELECT COUNT(*) FROM people WHERE <b>age</b> &gt; 30</pre>	<pre>db.people.count(   { <b>age</b>: { \$gt: 30 } } )</pre>
-------------------------------------------------------------------------	----------------------------------------------------------------------

# 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
<u>//LIMIT</u>	<u>\$limit</u>
<u>SUM</u>	<u>\$sum</u>
<u>COUNT</u>	<u>\$sum</u>

# 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 }



# 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.



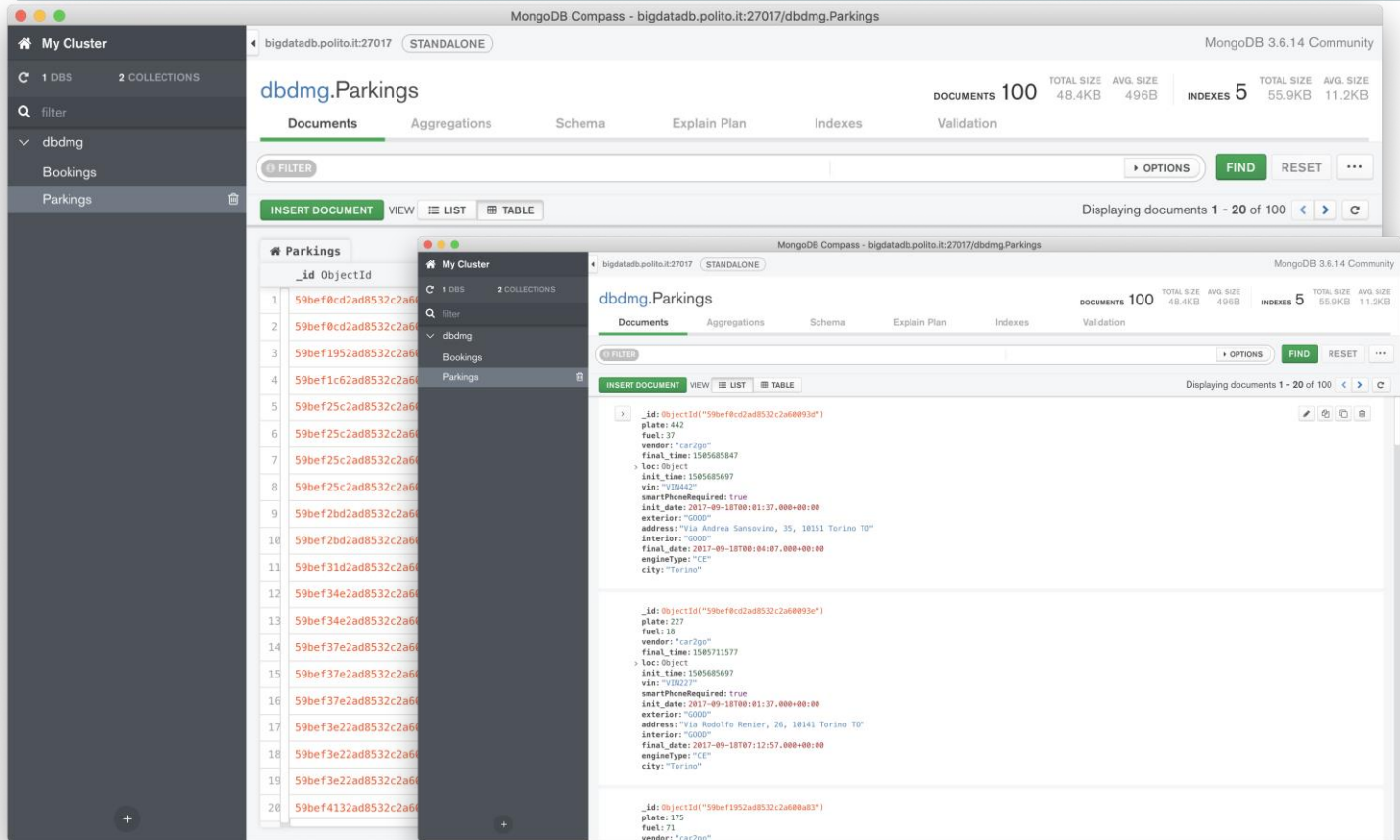
# MongoDB Compass

The screenshot shows the 'Connect to Host' window in MongoDB Compass. The window has a title bar 'MongoDB Compass - Connect'. On the left is a sidebar with options: 'CREATE FREE ATLAS CLUSTER' (with a link to 'Learn more'), 'New Connection', 'Favorites', and 'RECENTS'. The 'RECENTS' list shows several entries for 'bigdatadb.polito.it:27017' with timestamps. The main area is titled 'Connect to Host' and contains several configuration fields:

- Hostname:** bigdatadb.polito.it
- Port:** 27017
- SRV Record:** A toggle switch that is currently turned off.
- Authentication:** A dropdown menu set to 'Username / Password'.
- Username:** Gestionali
- Password:** A field with masked characters (dots).
- Authentication Database:** dbdmg
- Replica Set Name:** An empty text field.
- Read Preference:** Primary
- SSL:** A dropdown menu set to 'Unvalidated (insecure)'.
- SSH Tunnel:** A dropdown menu set to 'None'.
- Favorite Name:** A field with the text 'e.g. Shared Dev, QA Box, PRODUCTION'.

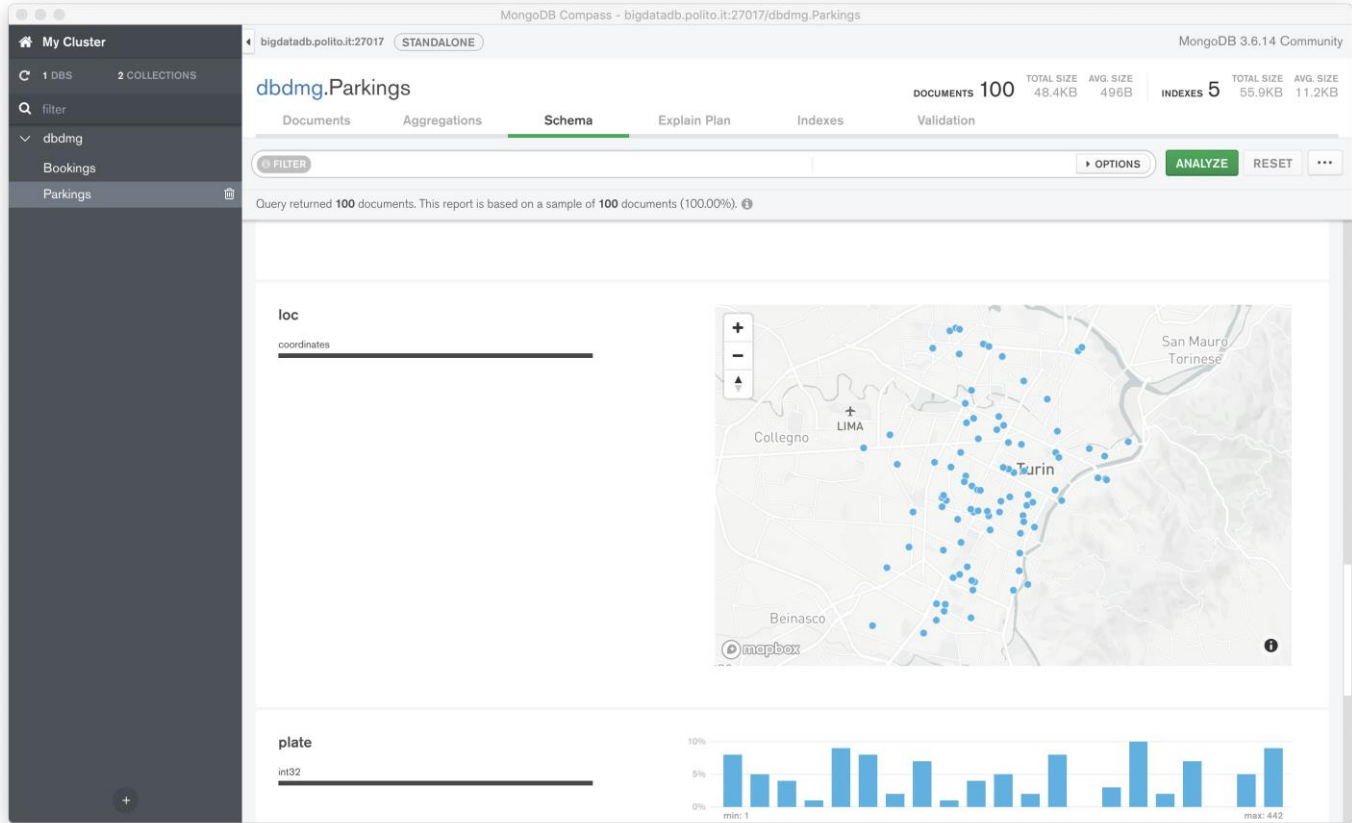
➤ Connect to local or remote instances of MongoDB.

# MongoDB Compass



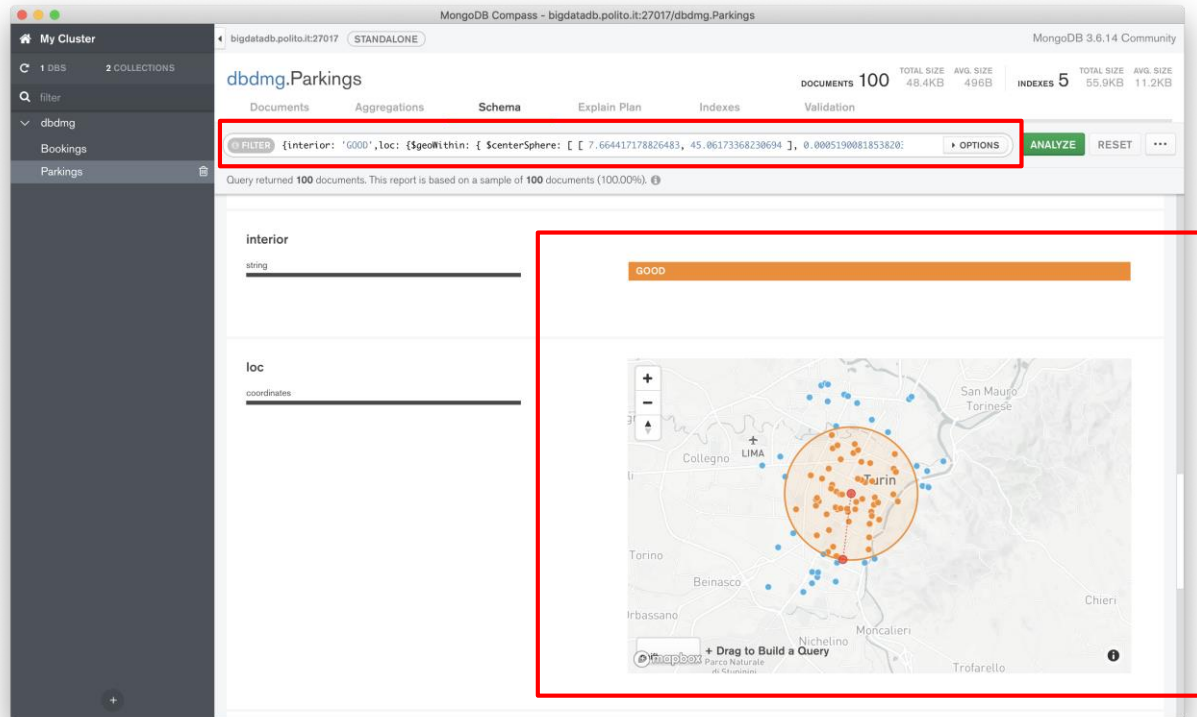
➤ Get an overview of the data in list or table format.

# MongoDB Compass



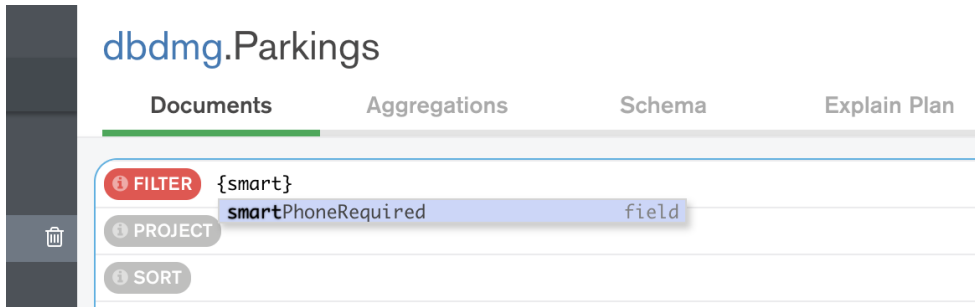
- Analyze the documents and their fields.
- Native support for geospatial coordinates.

# MongoDB Compass

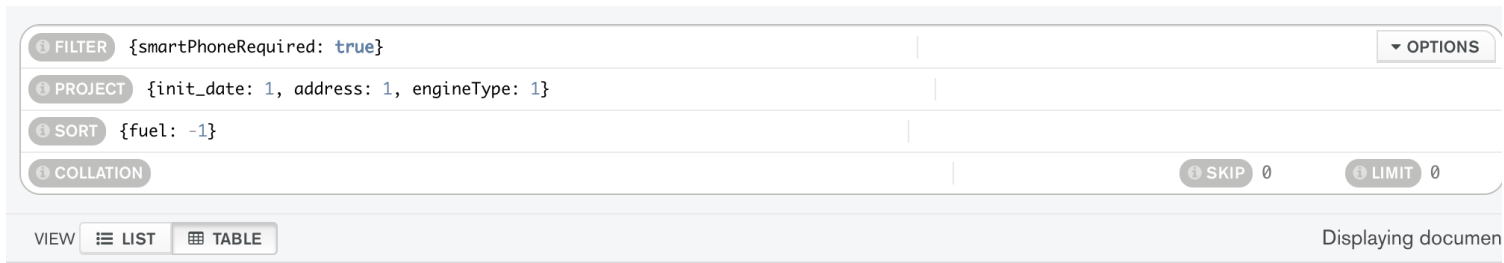


➤ Visually build the query conditioning on analyzed fields.

# MongoDB Compass



➤ Autocomplete enabled by default.



➤ Construct the query step by step.

# MongoDB Compass

The screenshot displays the MongoDB Compass interface for a cluster named 'bigdatadb.polito.it:27017'. The selected database is 'dbdmg' and the collection is 'Parkings'. The 'Explain Plan' tab is active, showing a query with a filter on 'interior' and a location-based query using '\$geoWithin' and '\$centerSphere'. The query returns 97 documents. The 'Query Performance Summary' section provides key metrics: Documents Returned (97), Index Keys Examined (0), Documents Examined (100), Actual Query Execution Time (0 ms), Sorted in Memory (yes), and a warning that no index is available for this query. Below the summary, the 'PROJECTION' and 'SORT' stages are detailed, showing the number of documents returned (97) and execution time (0 ms) for each stage.

**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**

nReturned: 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

The screenshot shows the MongoDB Compass interface for a cluster named 'My Cluster'. The left sidebar shows the database 'dbdmg' and the collection 'Parkings'. The main panel is titled 'dbdmg.Parkings' and shows the 'Validation' tab. The 'Validation' tab displays the following information:

- Documents: 100
- Total Size: 48.4KB
- Avg. Size: 496B
- Indexes: 5
- Total Size: 55.9KB
- Avg. Size: 11.2KB

The 'Validation' tab is active, showing the 'Validation Action' set to 'ERROR' and the 'Validation Level' set to 'STRICT'. Below this, a JSON schema is displayed:

```
1- {
2-   $jsonSchema: {
3-     required: ['exterior', 'interior', 'vendor', 'fuel'],
4-     properties: {
5-       vendor: {
6-         bsonType: "string",
7-         description: "must be a string"
8-       },
9-       fuel: {
10-        bsonType: "int",
11-        description: "must be an integer number"
12-      },
13-    }
14-  }
15- }
```

Below the schema, there are two sections:

- Sample Document That Passed Validation**: A document with fields: `_id` (ObjectId), `plate` (442), `fuel` (37), `vendor` ("car2go"), `final_time` (1505685847), `loc` (Object), `init_time` (1505685697), `vin` ("VIN442"), and `smartPhoneReserved` (true).
- Sample Document That Failed Validation**: A section with the text "No Preview Documents".

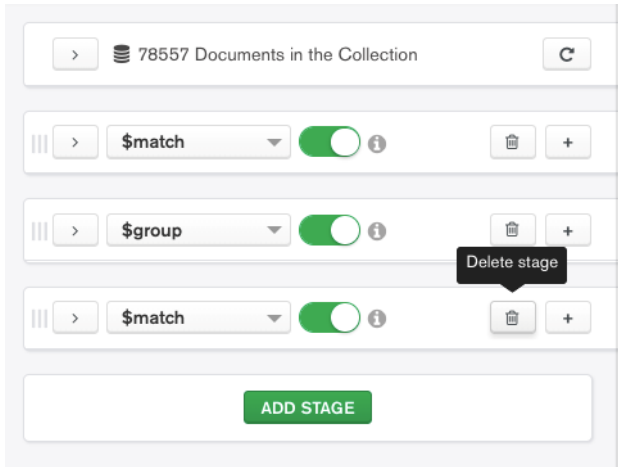
At the bottom of the interface, there are two orange arrow icons pointing right, followed by the text "Specify constraints to validate data" and "Find inconsistent documents."

➤ Specify constraints to validate data

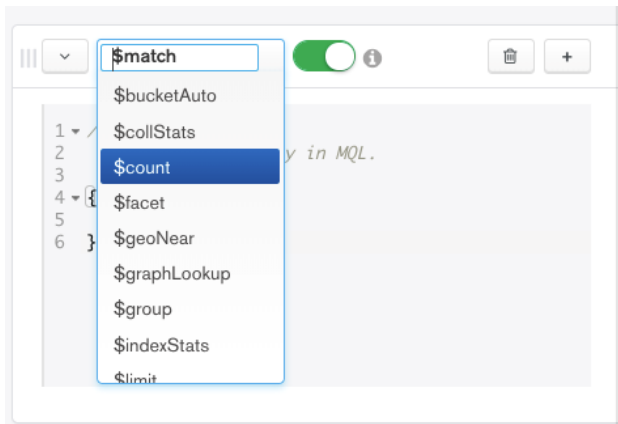
➤ Find inconsistent documents.



# MongoDB Compass: Aggregation

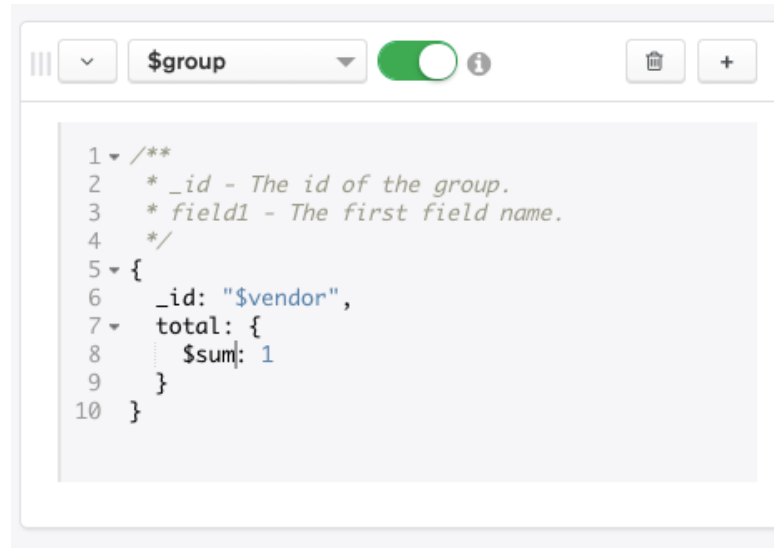


➤ Build a pipeline consisting of multiple aggregation stages.



➤ Define the filter and aggregation attributes for each operator.

# MongoDB Compass: Aggregation stages



The screenshot shows the MongoDB Compass interface for editing an aggregation pipeline. The top bar includes a menu icon, a dropdown set to '\$group', a toggle switch, and an information icon. To the right are delete and add stage buttons. The main area contains a code editor with the following pipeline:

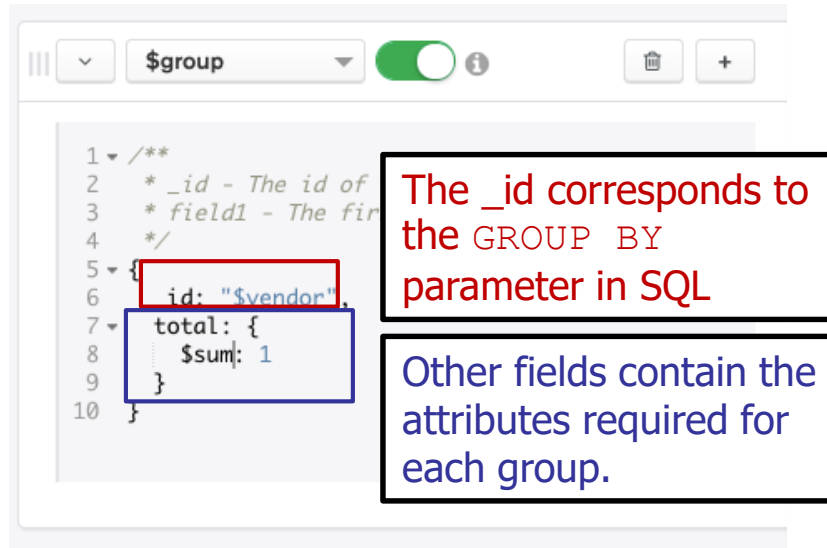
```
1 /**
2  * _id - The id of the group.
3  * field1 - The first field name.
4  */
5 {
6   _id: "$vendor",
7   total: {
8     $sum: 1
9   }
10 }
```

Output after \$group stage (Sample of 2 documents)

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

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

# MongoDB Compass: Aggregation stages



The screenshot shows the MongoDB Compass aggregation pipeline editor. The top bar includes a menu icon, a dropdown set to '\$group', a toggle switch, and icons for deleting and adding stages. The pipeline editor shows a single stage with the following code:

```
1 /**
2  * _id - The id of
3  * field1 - The fir
4  */
5 {
6   id: "$vendor",
7   total: {
8     $sum: 1
9   }
10 }
```

Annotations in the image include a red box around the `id: "$vendor"` line and a blue box around the `total: { $sum: 1 }` block. To the right of the code, two text boxes provide explanations:

- The `_id` corresponds to the GROUP BY parameter in SQL** (in red text)
- Other fields contain the attributes required for each group.** (in blue text)



The screenshot shows the output of the aggregation pipeline. The title is "Output after \$group stage (Sample of 2 documents)". Below the title, there are two document cards representing the grouped data:

Document 1	Document 2
<pre>{   "_id": "car2go",   "total": 48423 }</pre>	<pre>{   "_id": "enjoy",   "total": 30134 }</pre>

One group for each "vendor".

# MongoDB Compass: Pipelines

The screenshot displays the MongoDB Compass interface with a pipeline consisting of two stages. The first stage is a `$group` operation, and the second stage is a `$match` operation. An arrow points from the `$group` stage to the `$match` stage, indicating the flow of the pipeline.

**1<sup>st</sup> stage: grouping by vendor**

Query for the 1<sup>st</sup> stage:

```
1 /**
2  * _id - The id of the group.
3  * field1 - The first field name.
4  */
5 {
6   _id: "$vendor",
7   total: { $sum: 1 },
8   avg_fuel: { $avg: "$fuel" }
9 }
10
```

Output after \$group stage (Sample of 2 documents):

_id	total	avg_fuel
car2go	48423	64.88284492906264
enjoy	30134	61.03381562354815

**2<sup>nd</sup> stage: condition over fields created in the previous stage (avg\_fuel, total).**

Query for the 2<sup>nd</sup> stage:

```
1 /**
2  * query - The query in MQL.
3  */
4 {
5   avg_fuel: { $gt: 63 },
6   total: { $gt: 35000 }
7 }
```

Output after \$match stage (Sample of 1 document):

_id	total	avg_fuel
car2go	48423	64.88284492906264