



# NoSQL databases

## 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

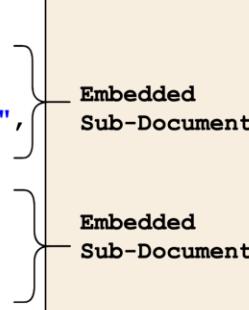
| <b>Relational database</b> | <b>MongoDB</b> |
|----------------------------|----------------|
| Table                      | Collection     |
| Record                     | Document       |
| Column                     | Field          |

# 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",  
  }  
}
```



The diagram illustrates the structure of a MongoDB document. It shows a main object with several fields: '\_id', 'username', 'contact', 'access', and a final closing brace. The 'contact' field contains an object with 'phone' and 'email' properties. The 'access' field also contains an object with 'level' and 'group' properties. Two curly braces on the right side of the document point to these nested objects, with the label 'Embedded Sub-Document' placed next to each brace.

# 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

**Querying data using operators**

# MongoDB: query language

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

| <b>MySQL clause</b> | <b>MongoDB operator</b> |
|---------------------|-------------------------|
| SELECT              | find()                  |

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

# 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 <b>id</b> ,<br><b>user_id</b> ,<br><b>status</b><br>FROM people | db.people.find(<br>{ },<br>{ <b>user_id: 1</b> ,<br><b>status: 1</b><br>}<br>) |
|--|--|

Where Condition

Select fields

# 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>}) |

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

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

Where Condition

Selection fields

# 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



<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.

| <b>MySQL</b> | <b>MongoDB</b> | <b>Description</b> |
|--------------|----------------|--------------------|
| >            | \$gt           | greater than       |
| >=           | \$gte          | greater equal than |
| <            | \$lt           | less than          |
| <=           | \$lte          | less equal than    |
| =            | \$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 |

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

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

# MongoDB: comparison operators ( $\geq$ )

| MySQL                          | MongoDB            | Description               |
|--------------------------------|--------------------|---------------------------|
| <code>&gt;</code>              | <code>\$gt</code>  | greater than              |
| <code><math>\geq</math></code> | <code>\$gte</code> | <b>greater equal than</b> |

```
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 than |
| <     | \$lt    | <b>less than</b>   |

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

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

# MongoDB: comparison operators (<=)

| MySQL | MongoDB      | Description            |
|-------|--------------|------------------------|
| >     | \$gt         | greater than           |
| >=    | \$gte        | greater equal than     |
| <     | \$lt         | less than              |
| <=    | <b>\$lte</b> | <b>less equal than</b> |

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

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

# MongoDB: comparison operators (=)

| MySQL | MongoDB     | Description   |
|-------|-------------|---|
| >     | \$gt        | greater than  |
| >=    | \$gte       | greater equal than  |
| <     | \$lt        | less than   |
| <=    | \$lte       | less equal than   |
| =     | <b>\$eq</b> | <b>equal to</b><br>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 than  |
| <     | \$lt         | less than           |
| <=    | \$lte        | less equal than     |
| =     | \$eq         | equal to            |
| !=    | <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          |
|------------|---------|----------------------|
| <b>AND</b> | ,       | <b>Both verified</b> |

```
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    | <b>\$or</b> | <b>At least one verified</b> |

```
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     | <code>sort()</code> |

```
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    |
|-----------------------|---------------------|
| <code>ORDER BY</code> | <code>sort()</code> |

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

# MongoDB: counting

| <b>MySQL clause</b> | <b>MongoDB operator</b>   |
|---------------------|---------------------------|
| 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 } }  
)
```



# MongoDB

**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         |
| <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) |

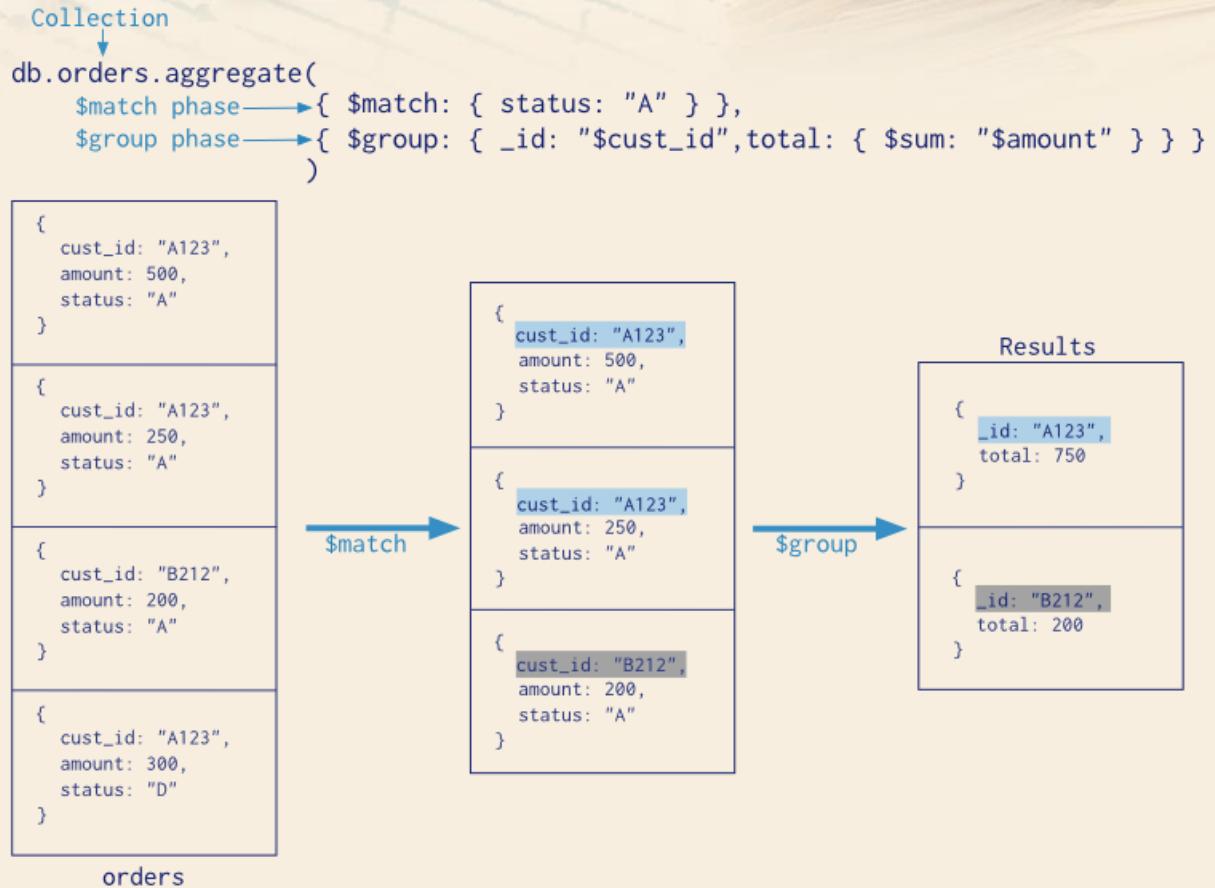
```
SELECT status,  
       SUM(age) AS total  
  FROM people  
 GROUP BY status  
 HAVING total > 1000
```

```
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





# MongoDB Compass

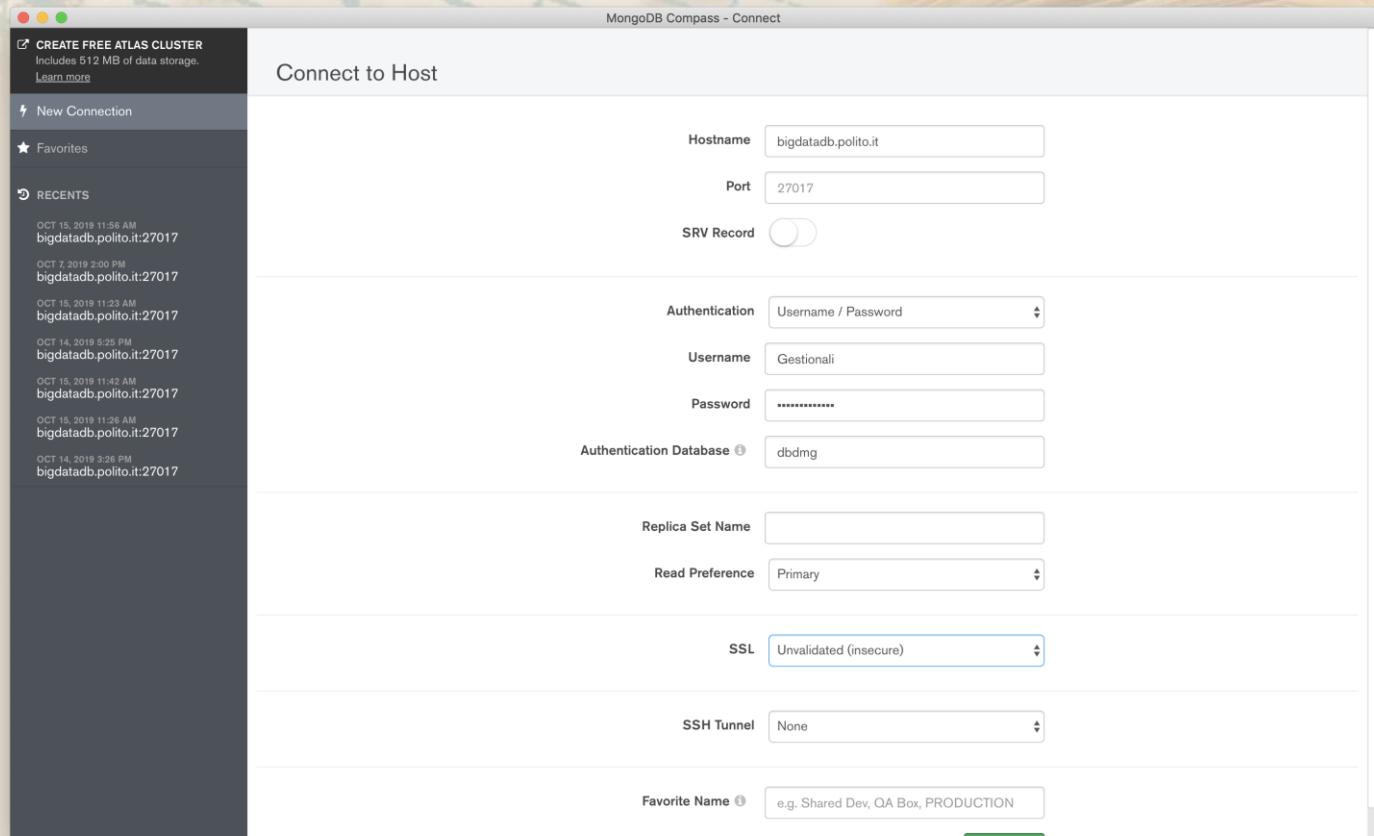
**GUI for Mongo DB**

# 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



The screenshot shows the MongoDB Compass 'Connect' dialog. On the left is a sidebar with a 'CREATE FREE ATLAS CLUSTER' button, a 'New Connection' button, a 'Favorites' section, and a 'RECENTS' section listing previous connections. The main area is titled 'Connect to Host' and contains fields for 'Hostname' (bigdatadb.polito.it), 'Port' (27017), and an 'SRV Record' toggle switch. Below these are sections for 'Authentication' (Username / Password, Username: Gestionali, Password: redacted, Authentication Database: dbdmg), 'Replica Set Name' (empty), 'Read Preference' (Primary), 'SSL' (Unvalidated (insecure)), 'SSH Tunnel' (None), and a 'Favorite Name' field (e.g. Shared Dev, QA Box, PRODUCTION). A green progress bar at the bottom indicates the connection process.

▷ Connect to local or remote instances of MongoDB.

# MongoDB Compass

The screenshot shows the MongoDB Compass interface running on a Mac OS X system. It displays two windows side-by-side, both connected to the same MongoDB instance at `bigdatadb.polito.it:27017`.

**Left Window:** This window is titled "My Cluster" and shows the "dbdmg" database selected. Under "dbdmg", the "Parkings" collection is listed. The "Documents" tab is active, showing 100 documents. The document list table has columns: #, \_id, ObjectID, and a truncated preview column. The preview shows fields like `plate`, `fuel`, `vendor`, `final_time`, `loc`, `initial_time`, `vin`, `smartPhoneRequired`, `init_date`, `exterior`, `address`, `interior`, `final_date`, `engineType`, and `city`. The first few documents are:

| #  | _id                  | ObjectID | Preview  |
|----|----------------------|----------|--|
| 1  | 59bef0cd2ad8532c2a60 |          | ...<br>plate: 442<br>fuel: 37<br>vendor: "car2go"<br>final_time: 1565685647<br>> loc: 0000000000000000<br>init_time: 1565685697<br>vin: "VIN42"<br>smartPhoneRequired: true<br>init_date: 2017-09-18T00:01:37.000+00:00<br>exterior: "G000P"<br>address: "Via Andrea Sanseverino, 35, 10151 Torino TO"<br>interior: "G000P"<br>final_date: 2017-09-18T00:04:07.000+00:00<br>engineType: "CE"<br>city: "Torino" |
| 2  | 59bef0cd2ad8532c2a60 |          | ...<br>plate: 227<br>fuel: 1<br>vendor: "car2go"<br>final_time: 1565711577<br>> loc: 0000000000000000<br>init_time: 1565685697<br>vin: "VIN227"<br>smartPhoneRequired: true<br>init_date: 2017-09-18T00:01:37.000+00:00<br>exterior: "G000P"<br>address: "Via Rodolfo Renier, 26, 10141 Torino TO"<br>interior: "G000P"<br>final_date: 2017-09-18T07:12:57.000+00:00<br>engineType: "CE"<br>city: "Torino"     |
| 3  | 59bef1952ad8532c2a60 |          | ...<br>plate: 175<br>fuel: 71<br>vendor: "car2go"  |
| 4  | 59bef1c12ad8532c2a60 |          | ...  |
| 5  | 59bef25c2ad8532c2a60 |          | ...  |
| 6  | 59bef25c2ad8532c2a60 |          | ...  |
| 7  | 59bef25c2ad8532c2a60 |          | ...  |
| 8  | 59bef25c2ad8532c2a60 |          | ...  |
| 9  | 59bef2bd2ad8532c2a60 |          | ...  |
| 10 | 59bef2bd2ad8532c2a60 |          | ...  |
| 11 | 59bef31d2ad8532c2a60 |          | ...  |
| 12 | 59bef34e2ad8532c2a60 |          | ...  |
| 13 | 59bef34e2ad8532c2a60 |          | ...  |
| 14 | 59bef37e2ad8532c2a60 |          | ...  |
| 15 | 59bef37e2ad8532c2a60 |          | ...  |
| 16 | 59bef37e2ad8532c2a60 |          | ...  |
| 17 | 59bef3e22ad8532c2a60 |          | ...  |
| 18 | 59bef3e22ad8532c2a60 |          | ...  |
| 19 | 59bef3e22ad8532c2a60 |          | ...  |
| 20 | 59bef4132ad8532c2a60 |          | ...  |

**Right Window:** This window also displays the "dbdmg" database and the "Parkings" collection. It shows the same 100 documents and provides a detailed view of the first few documents in the preview pane.

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

# MongoDB Compass

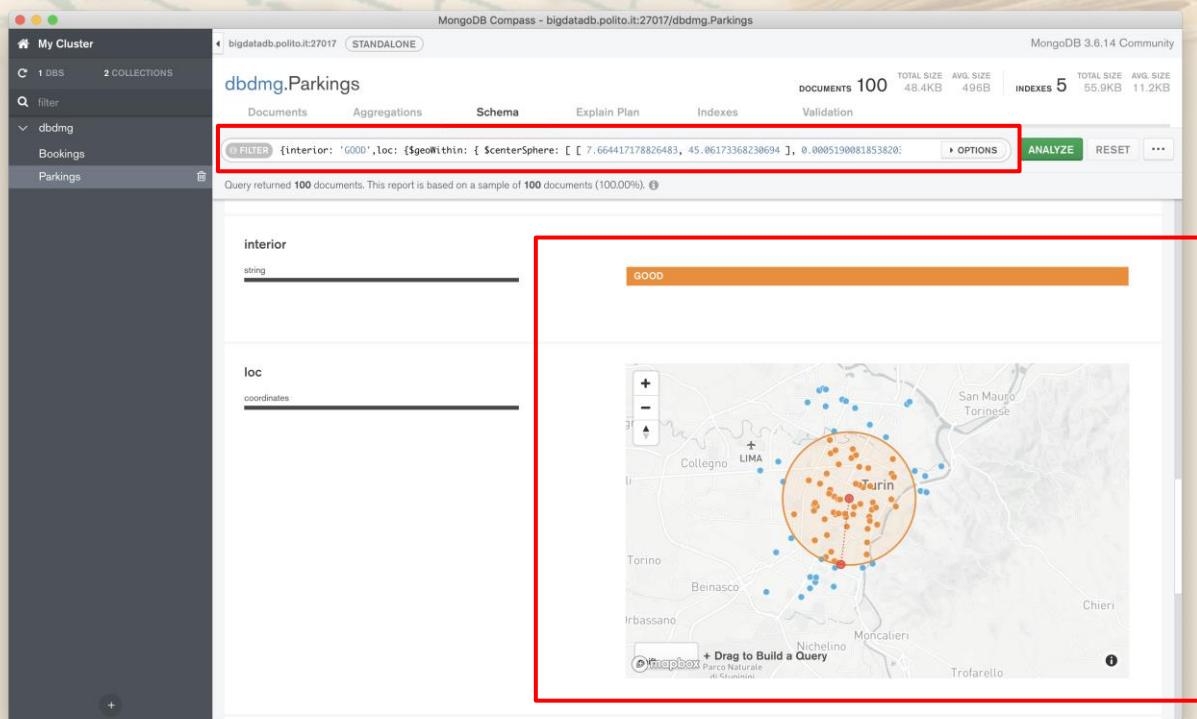
The screenshot shows the MongoDB Compass interface connected to a cluster named 'My Cluster' at 'bigdatadb.polito.it:27017'. The database 'dbdmg' contains two collections: 'Bookings' and 'Parkings'. The 'Parkings' collection is selected, displaying 100 documents with a total size of 48.4KB and average size of 496B. There are 5 indexes with a total size of 55.9KB and average size of 11.2KB.

The 'Schema' tab is active, showing the field 'loc' with a 'coordinates' input field and a map view. The map displays numerous blue dots representing parking locations across the city of Turin, Italy, with labels for 'LIMA', 'Collegno', 'Turin', 'San Mauro Torinese', and 'Beinasco'. A mepbox logo is visible in the bottom left corner of the map.

The 'plate' field is also shown with an 'int32' input field and a histogram chart. The histogram shows the distribution of plate values, with the x-axis ranging from 'min: 1' to 'max: 442' and the y-axis showing percentages from 0% to 10%. The distribution is highly skewed, with most values falling between 1 and 100.

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

# MongoDB Compass



▷ Visually build the query conditioning on analyzed fields.

# MongoDB Compass

dbdmg.Parkings

Documents Aggregations Schema Explain Plan

FILTER {smartPhoneRequired: true}

PROJECT smartPhoneRequired

SORT

field

▷ Autocomplete enabled by default.

FILTER {smartPhoneRequired: true}

PROJECT {init\_date: 1, address: 1, engineType: 1}

SORT {fuel: -1}

COLLATION

OPTIONS

Skip 0 Limit 0

VIEW LIST TABLE Displaying document

▷ Construct the query step by step.

# MongoDB Compass

The screenshot shows the MongoDB Compass interface for a cluster named "My Cluster". The left sidebar lists databases (1 DBS) and collections (2 COLLECTIONS), with "dbdmg" expanded to show "Bookings" and "Parkings". The main area displays the "dbdmg.Parkings" collection with 100 documents, totaling 48.4KB with an average size of 496B. The "Indexes" section shows 5 indexes, totaling 55.9KB with an average size of 11.2KB.

The "Explain Plan" tab is selected, showing the execution plan for a query:

- Filter:** `{interior: 'GOOD', loc: {$geoWithin: { $centerSphere: [ [ 7.664417178826483, 45.06173368230694 ], 0.0005190081853820 ] }}`
- Project:** (No details shown)
- Sort:** (No details shown)
- Collation:** (No details shown)

Execution parameters: `SKIP 0`, `LIMIT 0`. Buttons: EXPLAIN, RESET, ...

View Details As: VISUAL TREE, RAW JSON

**Query Performance Summary**

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

**PROJECTION**  
nReturned: 97 Execution Time: 0 ms  
Transform by:  
`{"init_date":1,"address":1,"engineType":1}`  
DETAILS

**SORT**  
nReturned: 97 Execution Time: 0 ms  
DETAILS

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

# MongoDB Compass

The screenshot shows the MongoDB Compass interface for a database named 'bigdataadb.polito.it:27017'. The 'STANDALONE' tab is selected. On the left, the 'My Cluster' sidebar shows 1 DB and 2 Collections: 'dbdmg' and 'Bookings'. The 'dbdmg' collection is expanded, showing 'Parkings'.

The main area displays the 'dbdmg.Parkings' collection with 100 documents, totaling 48.4KB with an average size of 496B. There are 5 indexes, totaling 55.9KB with an average size of 11.2KB.

The 'Validation' tab is active. It shows validation settings: Action is set to 'ERROR', Level is 'STRICT'. The validation 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 }
```

A message 'Validation modified' is shown at the bottom left, with 'CANCEL' and 'UPDATE' buttons.

Below, two sample documents are shown:

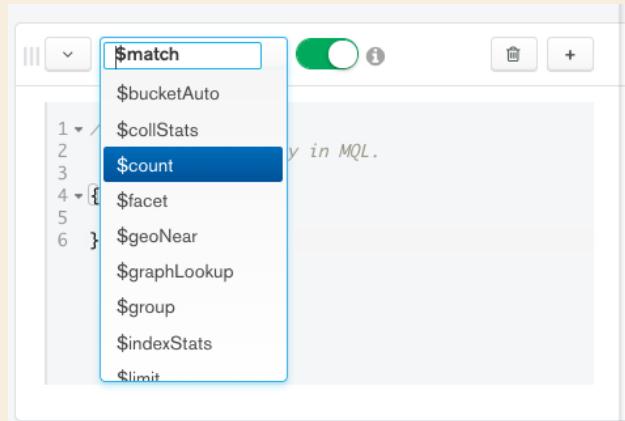
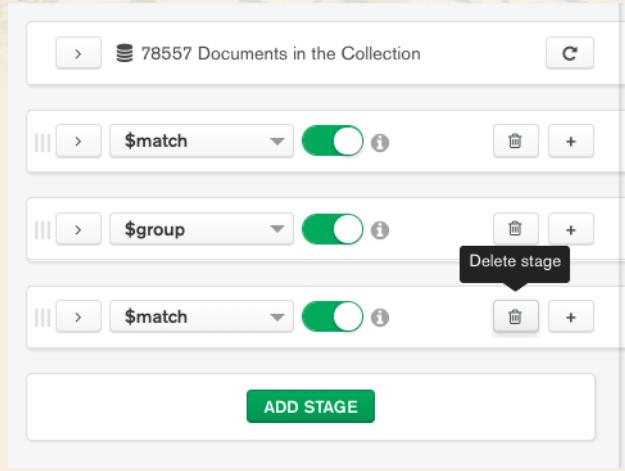
- Sample Document That Passed Validation** (green checkmark):

```
_id:ObjectId("59bef0cd2ad8532c2a60093d")  
plate:442  
fuel:37  
vendor:"car2go"  
final_time:1505685847  
loc:Object  
init_time:1505685697  
vin:"VINA42"  
smartPhoneRequired:true
```
- Sample Document That Failed Validation** (red X):

No Preview Documents

- Specify constraints to validate data
- Find inconsistent documents.

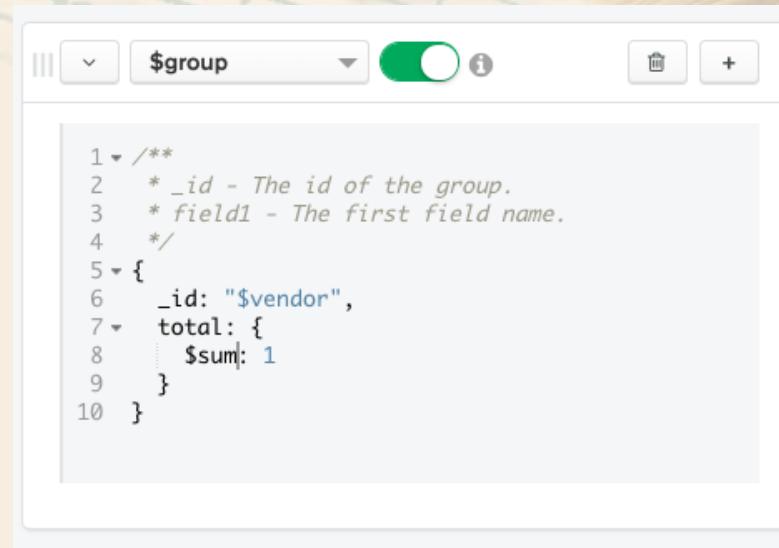
# MongoDB Compass: Aggregation



➤ Build a pipeline consisting of multiple aggregation stages.

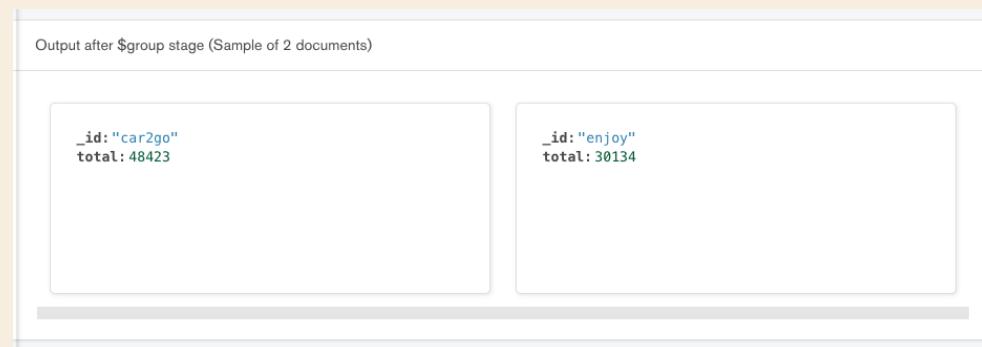
➤ Define the filter and aggregation attributes for each operator.

# MongoDB Compass: Aggregation stages



The screenshot shows the MongoDB Compass interface for defining an aggregation pipeline. At the top, there's a toolbar with a dropdown menu, a search bar containing '\$group', a green toggle switch, and several icons for trash, plus, and info. Below the toolbar is a code editor area displaying the following aggregation stage:

```
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 }
```



The screenshot shows the results of the aggregation pipeline after the '\$group' stage. The title indicates 'Output after \$group stage (Sample of 2 documents)'. There are two documents displayed in separate boxes:

- Document 1: '\_id: "car2go" total: 48423'
- Document 2: '\_id: "enjoy" total: 30134'

# MongoDB Compass: Aggregation stages

The screenshot shows the MongoDB Compass interface with an aggregation pipeline editor. The top bar has a dropdown set to '\$group' and a green toggle switch. Below is the pipeline code:

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

Annotations explain parts of the code:

- A red box highlights the `id: "$vendor"` field.
- A blue box highlights the `total: { $sum: 1 }` object.
- A red callout box states: "The `_id` corresponds to the GROUP BY parameter in SQL".
- A blue callout box states: "Other fields contain the attributes required for each group."

| Output after \$group stage (Sample of 2 documents) |                           |
|--|---------------------------|
| <code>_id: "car2go"</code>                         | <code>total: 48423</code> |
| <code>_id: "enjoy"</code>                          | <code>total: 30134</code> |

One group for each "vendor".

# MongoDB Compass: Pipelines

The screenshot shows the MongoDB Compass interface with a pipeline editor. The top stage is a `$group` stage. The code is:1 `/**`  
2 `* _id - The id of the group.`  
3 `* field1 - The first field name.`  
4 `*/`  
5 `{`  
6 `_id: "$vendor",`  
7 `total: { $sum: 1 },`  
8 `avg_fuel : {$avg : "$fuel"}`  
9 `}`  
10A red box highlights the text "1<sup>st</sup> stage: grouping by vendor".

The output after the `$group` stage shows two documents:

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

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

An arrow points from the bottom of the previous screenshot to the top of this screenshot, indicating the flow of the pipeline.

The second stage is a `$match` stage. The code is:1 `/**`  
2 `* query - The query in MQL.`  
3 `*/`  
4 `{`  
5 `avg_fuel: {$gt: 63},`  
6 `total : {$gt : 35000}`  
7 `}`A red box highlights the text "2<sup>nd</sup> stage: condition over fields created in the previous stage (avg\_fuel, total)."

The output after the `$match` stage shows one document:

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



# MongoDB

## 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" } )
```



## MongoDB operations

# MongoDB: 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

# MongoDB: 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;`

Note: shell commands vs GUI interface.

# MongoDB: 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();
```

# MongoDB: 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., e.g., to create a capped collection, or to create a new collection that uses document validation
- ▷ E.g.,  
`db.createCollection("authors", {capped: true});`

# MongoDB: Databases and Collections

## ▷ Show collections

```
show collections;
```

## ▷ Drop collections

```
db.<collection name>.drop();
```

## ▷ E.g.,

```
db.authors.drop();
```

# MongoDB: Read/Insert/Update data

| MongoDB   | Relational database  |
|---|--|
| <code>db.users.find()</code>  | <code>SELECT * FROM users</code>   |
| <code>db.users.insert({<br/>    user_id: 'bcd001',<br/>    age: 45,<br/>    status: 'A' })</code>                                     | <code>INSERT INTO<br/>users (user_id, age, status)<br/>VALUES ('bcd001', 45, 'A')</code> |
| <code>db.users.update(<br/>    { age: { \$gt: 25 } },<br/>    { \$set: { status:<br/>        'C' } },<br/>    { multi: true })</code> | <code>UPDATE users<br/>SET status = 'C'<br/>WHERE age &gt; 25</code>                     |

# MongoDB: Insert documents

## ▷ 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"  
} ) ;
```

# MongoDB: Insert documents

## ▷ 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"  
} ) ;
```

Field name

# MongoDB: Insert documents

## ▷ 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"  
} ) ;
```

Field value

# MongoDB: Insert documents

## ▷ 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"
```

# MongoDB: Insert documents

» E.g.,

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

Favorite\_colors  
is an array

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"

# MongoDB: Insert documents

▷ 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

# MongoDB: inserting data

- ▷ New data needs to be **inserted into** the database.
  - Each SQL tuple corresponds to a MongoDB document
- ▷ The primary key `_id` is automatically added if the `_id` field is not specified.

| MySQL clause             | MongoDB operator         |
|--------------------------|--------------------------|
| <code>INSERT INTO</code> | <code>insertOne()</code> |

# MongoDB: inserting data

| MySQL clause   | MongoDB operator |
|--|------------------|
| INSERT INTO people(user_id,<br>age,<br>status)<br>VALUES ("bcd001",<br>45,<br>"A") | insertOne()      |

|  |  |
|--|--|
| INSERT INTO people(user_id,<br>age,<br>status)<br>VALUES ("bcd001",<br>45,<br>"A") | db.people.insertOne(<br>{<br>user_id: "bcd001",<br>age: 45,<br>status: "A"<br>}<br>) |
|--|--|

# MongoDB: inserting data

- ▷ Insert multiple documents in a single statement:  
operator `insertMany()`

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

# MongoDB: Insert 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"] }  
] );
```

# MongoDB: Document update

▷ Documents can be updated by 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

# MongoDB: 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.

# MongoDB: updating data

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

| <b>MySQL clause</b>                                    | <b>MongoDB operator</b>  |
|--|--|
| UPDATE <table><br>SET <statement><br>WHERE <condition> | db.<table>.updateMany(<br>{ <condition> } ,<br>{ \$set: {<statement>} }<br>) |

# MongoDB: updating data

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

|  |  |
|--|--|
| UPDATE people<br>SET status = "C"<br>WHERE age > 25<br>) | db.people.updateMany(<br>{ age: { \$gt: 25 } },<br>{ \$set: { status: "C" } }<br>) |
|--|--|

# MongoDB: updating data

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

|   |  |
|---|--|
| UPDATE people<br>SET status = "C"<br>WHERE age > 25             | db.people.updateMany(<br>{ age: { \$gt: 25 } },<br>{ \$set: { status: "C" } }<br>) |
| UPDATE people<br>SET age = <b>age</b> + 3<br>WHERE status = "A" | db.people.updateMany(<br>{ status: "A" } ,<br>{ <b>\$inc: { age: 3 }</b> }<br>)    |

The **\$inc** operator increments a field by a specified value

# MongoDB: deleting data

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

| MySQL clause | MongoDB operator |
|--------------|------------------|
| DELETE FROM  | deleteMany()     |

# MongoDB: deleting data

| MySQL clause | MongoDB operator |
|--------------|------------------|
| DELETE FROM  | deleteMany()     |

|  |   |
|--|---|
| DELETE FROM people<br>WHERE status = "D" | db.people.deleteMany(<br>{ status: "D" }<br>) |
|--|---|

# MongoDB: deleting data

| MySQL clause | MongoDB operator |
|--------------|------------------|
| DELETE FROM  | deleteMany()     |

|  |   |
|--|---|
| DELETE FROM people<br>WHERE status = "D" | db.people.deleteMany(<br>{ status: "D" }<br>) |
| DELETE FROM people                       | db.people.deleteMany( {} )                    |