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

<table>
<thead>
<tr>
<th>Relational database</th>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Collection</td>
</tr>
<tr>
<td>Record</td>
<td>Document</td>
</tr>
<tr>
<td>Column</td>
<td>Field</td>
</tr>
</tbody>
</table>
High-level, business-ready representation of the data

- Records are stored into Documents
  - field-value pairs
  - similar to JSON objects
  - may be nested

```json
{
  _id: <ObjectId>,
  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

Querying data using operators
Most of the operations available in SQL language can be expressed in MongoDB language.

<table>
<thead>
<tr>
<th>MySQL clause</th>
<th>MongoDB operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>find()</td>
</tr>
<tr>
<td>SELECT * FROM people</td>
<td>db.people.find()</td>
</tr>
</tbody>
</table>
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
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
E.g.,

```javascript
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

```javascript
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

<table>
<thead>
<tr>
<th>MySQL clause</th>
<th>MongoDB operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>find()</td>
</tr>
</tbody>
</table>

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

```javascript
db.people.find(
  { },
  { user_id: 1, status: 1 }
)
```
### MongoDB: find() operator

<table>
<thead>
<tr>
<th>MySQL clause</th>
<th>MongoDB operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>find()</td>
</tr>
</tbody>
</table>

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

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

**Where Condition**

**Select fields**
### MongoDB: `find()` operator

<table>
<thead>
<tr>
<th>MySQL clause</th>
<th>MongoDB operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td><code>find()</code></td>
</tr>
<tr>
<td>WHERE</td>
<td><code>find({&lt;WHERE CONDITIONS&gt;})</code></td>
</tr>
</tbody>
</table>

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

**Where Condition**
MongoDB: find() operator

<table>
<thead>
<tr>
<th>MySQL clause</th>
<th>MongoDB operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>find()</td>
</tr>
<tr>
<td>WHERE</td>
<td>find({&lt;WHERE CONDITIONS&gt;})</td>
</tr>
</tbody>
</table>

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

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

By default, the _id field is shown. To remove it from visualization use: _id: 0
**MongoDB: find() operator**

<table>
<thead>
<tr>
<th>MySQL clause</th>
<th>MongoDB operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>find()</td>
</tr>
<tr>
<td>WHERE</td>
<td>find({&lt;WHERE CONDITIONS&gt;})</td>
</tr>
</tbody>
</table>

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

```javascript
{
  _id: "A",
  address: {
    street: "Via Torino",
    number: "123/B",
    city: "Rome",
    code: "00184"
  }
}
```
MongoDB: Read data from documents

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

- **Age greater than 25 and less than or equal to 50**
  - Returns all documents having \texttt{age > 25 and age \leq 50}

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

- **Status = “A” or age = 55**
  - Returns all documents having \texttt{status=“A” or age=55}

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

- **Status = “A” or status = B**
  - Returns all documents where the \texttt{status} field value is either \texttt{“A” or “B”}
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.
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.

MongoDB: (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/
In SQL language, comparison operators are essential to express conditions on data.

In Mongo query language they are available with a different syntax.

<table>
<thead>
<tr>
<th>MySQL</th>
<th>MongoDB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>$gt</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>$gte</td>
<td>greater equal then</td>
</tr>
<tr>
<td>&lt;</td>
<td>$lt</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>$lte</td>
<td>less equal then</td>
</tr>
<tr>
<td>=</td>
<td>$eq</td>
<td>equal to</td>
</tr>
<tr>
<td>!=</td>
<td>$neq</td>
<td>not equal to</td>
</tr>
</tbody>
</table>
# MongoDB: Comparison query operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$eq  or $</td>
<td>Matches values that are equal to a specified value</td>
</tr>
<tr>
<td>$gt</td>
<td>Matches values that are greater than a specified value</td>
</tr>
<tr>
<td>$gte</td>
<td>Matches values that are greater than or equal to a specified value</td>
</tr>
<tr>
<td>$in</td>
<td>Matches any of the values specified in an array</td>
</tr>
<tr>
<td>$lt</td>
<td>Matches values that are less than a specified value</td>
</tr>
<tr>
<td>$lte</td>
<td>Matches values that are less than or equal to a specified value</td>
</tr>
<tr>
<td>$ne</td>
<td>Matches all values that are not equal to a specified value</td>
</tr>
<tr>
<td>$nin</td>
<td>Matches none of the values specified in an array</td>
</tr>
</tbody>
</table>
### MongoDB: comparison operators (>)

<table>
<thead>
<tr>
<th>MySQL</th>
<th>MongoDB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>$gt</td>
<td>greater than</td>
</tr>
</tbody>
</table>

**MySQL Example:**
```
SELECT * FROM people WHERE age > 25
```

**MongoDB Example:**
```
db.people.find(
    { age: { $gt: 25 } }
)
```
MongoDB: comparison operators (\(\geq\))

<table>
<thead>
<tr>
<th>MySQL</th>
<th>MongoDB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>$gt</td>
<td>greater than</td>
</tr>
<tr>
<td>(\geq)</td>
<td>$gte</td>
<td>greater equal than</td>
</tr>
</tbody>
</table>

MySQL

```sql
SELECT * 
FROM people 
WHERE age \(\geq\) 25
```

MongoDB

```js
db.people.find(
    { age: { $gte: 25 } }
)
```
## MongoDB: comparison operators (<>)

<table>
<thead>
<tr>
<th>MySQL</th>
<th>MongoDB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>$gt</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>$gte</td>
<td>greater equal then</td>
</tr>
<tr>
<td>&lt;</td>
<td>$lt</td>
<td>less than</td>
</tr>
</tbody>
</table>

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

```javascript
db.people.find(
    { age: { $lt: 25 } }
)
```
**MongoDB: comparison operators (<=)**

<table>
<thead>
<tr>
<th>MySQL</th>
<th>MongoDB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>$gt</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>$gte</td>
<td>greater equal then</td>
</tr>
<tr>
<td>&lt;</td>
<td>$lt</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>$lte</td>
<td>less equal then</td>
</tr>
</tbody>
</table>

**Example:**

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

```javascript
db.people.find(
  { age: { $lte: 25 } }
)
```
# MongoDB: comparison operators (=)

<table>
<thead>
<tr>
<th>MySQL</th>
<th>MongoDB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>$gt</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>$gte</td>
<td>greater equal then</td>
</tr>
<tr>
<td>&lt;</td>
<td>$lt</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>$lte</td>
<td>less equal then</td>
</tr>
<tr>
<td>=</td>
<td>$eq</td>
<td>equal to</td>
</tr>
</tbody>
</table>

The $eq expression is equivalent to `{ field: <value> }.

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

```javascript
db.people.find({ age: { $eq: 25 } })
```
# MongoDB: comparison operators (!=)

<table>
<thead>
<tr>
<th>MySQL</th>
<th>MongoDB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>$gt</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>$gte</td>
<td>greater equal then</td>
</tr>
<tr>
<td>&lt;</td>
<td>$lt</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>$lte</td>
<td>less equal then</td>
</tr>
<tr>
<td>=</td>
<td>$eq</td>
<td>equal to</td>
</tr>
<tr>
<td>!=</td>
<td>$neq</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>

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

```javascript
db.people.find(
  { age: { $neq: 25 } }
)
```
To specify multiple conditions, **conditional operators** are used.

MongoDB offers the same functionalities of MySQL with a different syntax.

<table>
<thead>
<tr>
<th>MySQL</th>
<th>MongoDB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>,</td>
<td>Both verified</td>
</tr>
<tr>
<td>OR</td>
<td>$or</td>
<td>At least one verified</td>
</tr>
</tbody>
</table>
## MongoDB: conditional operators (AND)

<table>
<thead>
<tr>
<th>MySQL</th>
<th>MongoDB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>,</td>
<td>Both verified</td>
</tr>
</tbody>
</table>

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

```javascript
db.people.find(  
    { status: "A",  
      age: 50  
    })
```
## MongoDB: conditional operators (OR)

<table>
<thead>
<tr>
<th>MySQL</th>
<th>MongoDB</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>,</td>
<td>Both verified</td>
</tr>
<tr>
<td>OR</td>
<td>$or</td>
<td>At least one verified</td>
</tr>
</tbody>
</table>

### MySQL

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

### MongoDB

```javascript
db.people.find(
  { $or: 
    [ { status: "A" }, 
      { age: 50 } 
    ]
  }
)
```
db.collection.find() gives back a cursor. It can be used to iterate over the result or as input for next operations.

E.g.,

- cursor.sort()
- cursor.count()
- cursor.forEach() //shell method
- cursor.limit()
- cursor.max()
- cursor.min()
- cursor.pretty()
Cursor examples:

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

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

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

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

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

```javascript
db.people.find({ status: "A" }).forEach(function(myDoc) {
  print( "user: " + myDoc.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
E.g.,

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

<table>
<thead>
<tr>
<th>MySQL clause</th>
<th>MongoDB operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDER BY</td>
<td>sort()</td>
</tr>
</tbody>
</table>

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

```javascript
db.people.find({ status: "A" }).sort({ user_id: 1 })
```
## MongoDB: sorting data

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

<table>
<thead>
<tr>
<th>MySQL clause</th>
<th>MongoDB operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDER BY</td>
<td><code>sort()</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM people WHERE status = &quot;A&quot; ORDER BY user_id ASC</td>
<td></td>
</tr>
<tr>
<td>db.people.find( { status: &quot;A&quot; } ).sort( { user_id: 1 } )</td>
<td></td>
</tr>
<tr>
<td>SELECT * FROM people WHERE status = &quot;A&quot; ORDER BY user_id DESC</td>
<td></td>
</tr>
<tr>
<td>db.people.find( { status: &quot;A&quot; } ).sort( { user_id: -1 } )</td>
<td></td>
</tr>
</tbody>
</table>
### MongoDB: counting

<table>
<thead>
<tr>
<th><strong>MySQL clause</strong></th>
<th><strong>MongoDB operator</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>count() or find().count()</td>
</tr>
</tbody>
</table>

### SQL vs. MongoDB

<table>
<thead>
<tr>
<th>SQL Query</th>
<th>MongoDB Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT COUNT(*) FROM people</td>
<td>db.people.count() or db.people.find().count()</td>
</tr>
</tbody>
</table>
MongoDB: counting

<table>
<thead>
<tr>
<th>MySQL clause</th>
<th>MongoDB operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>count() or find().count()</td>
</tr>
</tbody>
</table>

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

```sql
SELECT COUNT(*)
FROM people
WHERE age > 30
```

```javascript
db.people.count(
{ age: { $gt: 30 } }
)
```
MongoDB

Introduction to data aggregation
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

<table>
<thead>
<tr>
<th>SQL</th>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>$match</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>$group</td>
</tr>
<tr>
<td>HAVING</td>
<td>$match</td>
</tr>
<tr>
<td>SELECT</td>
<td>$project</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>$sort</td>
</tr>
<tr>
<td>//LIMIT</td>
<td>$limit</td>
</tr>
<tr>
<td>SUM</td>
<td>$sum</td>
</tr>
<tr>
<td>COUNT</td>
<td>$sum</td>
</tr>
</tbody>
</table>
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
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”
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
```javascript
db.people.aggregate([{
  $match: {status: "A"}
},
{
  $group: {
    _id: null,
    count: { $sum: 1 }
  }
}
])
```

Counts the number of documents in people with status equal to "A"
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
db.peopleaggregate([  
  { $group: { _id: "$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.
```
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 Framework

<table>
<thead>
<tr>
<th>SQL</th>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE</td>
<td>$match</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>$group</td>
</tr>
<tr>
<td>HAVING</td>
<td>$match</td>
</tr>
<tr>
<td>SELECT</td>
<td>$project</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>$sort</td>
</tr>
<tr>
<td>LIMIT</td>
<td>$limit</td>
</tr>
<tr>
<td>SUM</td>
<td>$sum</td>
</tr>
<tr>
<td>COUNT</td>
<td>$sum</td>
</tr>
</tbody>
</table>
### Aggregation in MongoDB: Group By

<table>
<thead>
<tr>
<th>MySQL clause</th>
<th>MongoDB operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP BY</td>
<td>aggregate($group)</td>
</tr>
</tbody>
</table>

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

```javascript
db.orders.aggregate([{
    $group: {
        _id: "$status",
        total: { $avg: "$age" }
    }
}])
```
Aggregation in MongoDB: Group By

<table>
<thead>
<tr>
<th>MySQL clause</th>
<th>MongoDB operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP BY</td>
<td>aggregate($group)</td>
</tr>
</tbody>
</table>

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

db.orders.aggregate( [ { $group: { _id: "$status", total: { $sum: "$age" } } } ] )
### Aggregation in MongoDB: Group By

<table>
<thead>
<tr>
<th>MySQL clause</th>
<th>MongoDB operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP BY</td>
<td>aggregate($group)</td>
</tr>
</tbody>
</table>

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

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

- **Group field**: `id: "$status"`
- **Aggregation function**: `total: { $sum: "$age" }`
Aggregation in MongoDB: Group By

<table>
<thead>
<tr>
<th><strong>MySQL clause</strong></th>
<th><strong>MongoDB operator</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>HAVING</td>
<td>aggregate($group, $match)</td>
</tr>
</tbody>
</table>

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

```javascript
db.orders.aggregate([ {
    $group: {
        _id: "$status",
        total: { $sum: "$age" }
    }
}, {
    $match: { total: { $gt: 1000 } } }
])
```
### Aggregation in MongoDB: Group By

<table>
<thead>
<tr>
<th>MySQL clause</th>
<th>MongoDB operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAVING</td>
<td><code>aggregate($group, $match)</code></td>
</tr>
</tbody>
</table>

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

```javascript
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

<table>
<thead>
<tr>
<th>MySQL clause</th>
<th>MongoDB operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAVING</td>
<td>aggregate($group, $match)</td>
</tr>
</tbody>
</table>

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

```javascript
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

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

Connect to local or remote instances of MongoDB.
Get an overview of the data in list or table format.
MongoDB Compass

- Analyze the documents and their fields.
- Native support for geospatial coordinates.
Visually build the query conditioning on analyzed fields.
Autocomplete enabled by default.

Construct the query step by step.
MongoDB Compass

MongoDB Compass - bigdatadb.polito.it:27017/dbdmg.Parkings

**dbdmg.Parkings**

*Documents Aggregations Schema Explain Plan Indexes Validation*

**FILTER**

{interior: 'GOOD', loc: {geoWithin: { $centerSphere: [ [ 7.64417178826483, 45.06173368230694 ], 0.0005190081853820 ]}}

**SORT**

**COLLATION**

**View Details As**: VISUAL TREE RAW JSON

**Query Performance Summary**

- Documents Returned: 97
- Index Keys Examined: 0
- Documents Examined: 100
- Actual Query Execution Time (ms): 0
- Sorted in Memory: yes

**PROJECTION**

- Projection: 97
- Execution Time: 0 ms

Transform by:

{"int_date":1,"address":1,"engineType":1}

**SORT**

- Sort: 97
- Execution Time: 0 ms

**No index available for this query.**

Analyze query performance and get hints to speed it up.
MongoDB Compass

- Specify contraints 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**

```javascript
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 `_id` corresponds to the `GROUP BY` parameter in SQL.

Other fields contain the attributes required for each group.

One group for each "vendor".
1st stage: grouping by `vendor`.

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

```javascript
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`, ..
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 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]`
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
$\textbf{$\text{near}$ syntax:}$

```json
{
    <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`
E.g.,

```javascript
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.
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"} )
```