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

Relational database	MongoDB
Table	Collection
Record	Document
Column	Field

MongoDB: Document Data Design

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

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

MySQL clause	MongoDB operator
SELECT	find()

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

MongoDB: Read data from documents

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

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

  - Returns all documents contained in the people collection

MongoDB: Read data from documents

- Select documents
  - `db.<collection name>.find( {<conditions>}, {<fields of interest>} );`
- Select the documents satisfying the specified conditions and specifically only the fields specified in fields of interest
  - `<conditions>` are optional
    - conditions take a document with the form:
      `{field1 : <value>, field2 : <value> ... }`
    - Conditions may specify a value or a regular expression

MongoDB: Read data from documents

- Select documents
  - `db.<collection name>.find( {<conditions>}, {<fields of interest>} );`
- Select the documents satisfying the specified conditions and specifically only the fields specified in fields of interest
  - `<fields of interest>` are optional
    - projections take a document with the form:
      `{field1 : <value>, field2 : <value> ... }`
    - 1/true to include the field, 0/false to exclude the field

MongoDB: Read data from documents

- E.g.,

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

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

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

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

MongoDB: Read data from documents

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

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

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

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

MongoDB: find() operator

MySQL clause	MongoDB operator
SELECT	find()

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

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

MongoDB: find() operator

MySQL clause	MongoDB operator
SELECT	find()

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

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

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" })

MongoDB: find() operator

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

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

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

Where Condition

Selection fields

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

MongoDB: find() operator

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

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

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

nested document

MongoDB: Read data from documents

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

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

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

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

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

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

MongoDB: Read data from documents

- Select a single document
  - `db.<collection name>.findOne( {<conditions>}, {<fields of interest>} );`
- Select one document that satisfies the specified query criteria.
  - If multiple documents satisfy the query, it returns the first one according to the natural order which reflects the order of documents on the disk.

MongoDB: (no) joins

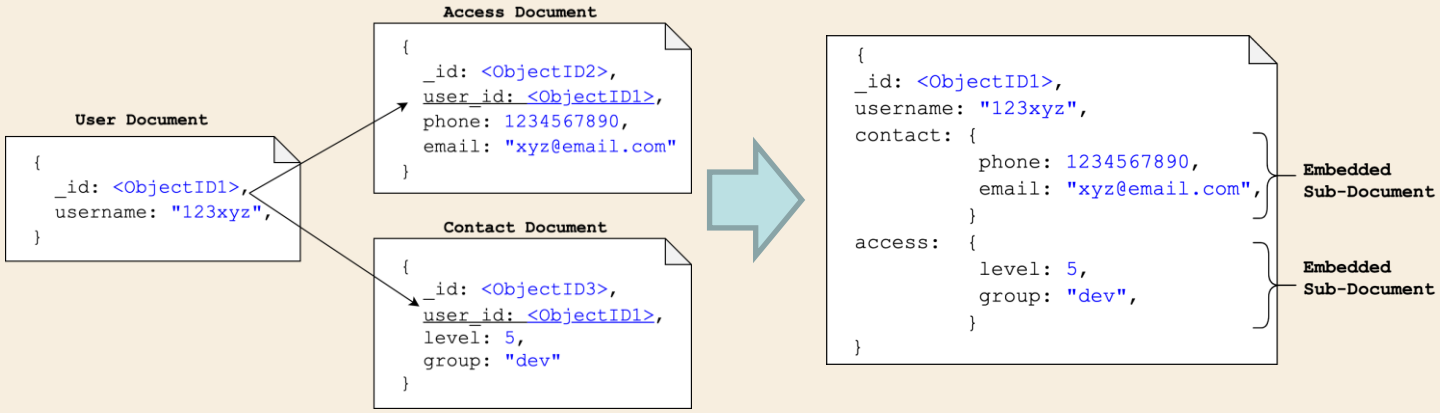
- There are other operators for selecting data from MongoDB collections
- However, no join operator exists (but `$lookup`)
  - You must write a program that
    - Selects the documents of the first collection you are interested in
    - Iterates over the documents returned by the first step, by using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

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

MongoDB: (no) joins

(no) joins

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



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

MongoDB: comparison operators

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

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

MongoDB: Comparison query operators

Name	Description
`$eq or :`	Matches values that are equal to a specified value
`$gt`	Matches values that are greater than a specified value
`$gte`	Matches values that are greater than or equal to a specified value
`$in`	Matches any of the values specified in an array
`$lt`	Matches values that are less than a specified value
`$lte`	Matches values that are less than or equal to a specified value
`$ne`	Matches all values that are not equal to a specified value
`$nin`	Matches none of the values specified in an array

MongoDB: comparison operators (>)

MySQL	MongoDB	Description
>	$gt	greater than

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

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

MongoDB: comparison operators (>=)

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

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

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

MongoDB: comparison operators (<)

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

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

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

MongoDB: comparison operators (<=)

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

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

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

MongoDB: comparison operators (=)

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

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

MongoDB: comparison operators (!=)

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

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

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

MongoDB: conditional operators

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

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

MongoDB: conditional operators (AND)

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

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

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

MongoDB: conditional operators (OR)

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

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

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

MongoDB: Cursor

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

MongoDB: Cursor

- Cursor examples:

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

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

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

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

MongoDB: sorting data

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

MongoDB: sorting data

- E.g.,

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

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

MongoDB: sorting data

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

MySQL clause	MongoDB operator
ORDER BY	sort()

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

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

MongoDB: sorting data

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

MySQL clause	MongoDB operator
ORDER BY	sort()

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

MongoDB: counting

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

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

MongoDB: counting

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

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

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

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

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
//LIMIT	$limit
SUM	$sum
COUNT	$sum

MongoDB: Aggregation

- Aggregate functions can be applied to collections to group documents

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

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

MongoDB: Aggregation

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

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

MongoDB: Aggregation

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

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

MongoDB: Aggregation

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

Where conditions

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

MongoDB: Aggregation

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

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

MongoDB: Aggregation

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

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

MongoDB: Aggregation

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

Having condition

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

MongoDB: Aggregation Framework

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

Aggregation in MongoDB: Group By

MySQL clause	MongoDB operator
GROUP BY	aggregate($group)

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

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

Aggregation in MongoDB: Group By

MySQL clause	MongoDB operator
GROUP BY	aggregate($group)

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

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

Group field

Aggregation in MongoDB: Group By

MySQL clause	MongoDB operator
GROUP BY	aggregate($group)

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

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

Group field

Aggregation function

Aggregation in MongoDB: Group By

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

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

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

Aggregation in MongoDB: Group By

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

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

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

Group stage: Specify the aggregation field and the aggregation function

Aggregation in MongoDB: Group By

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

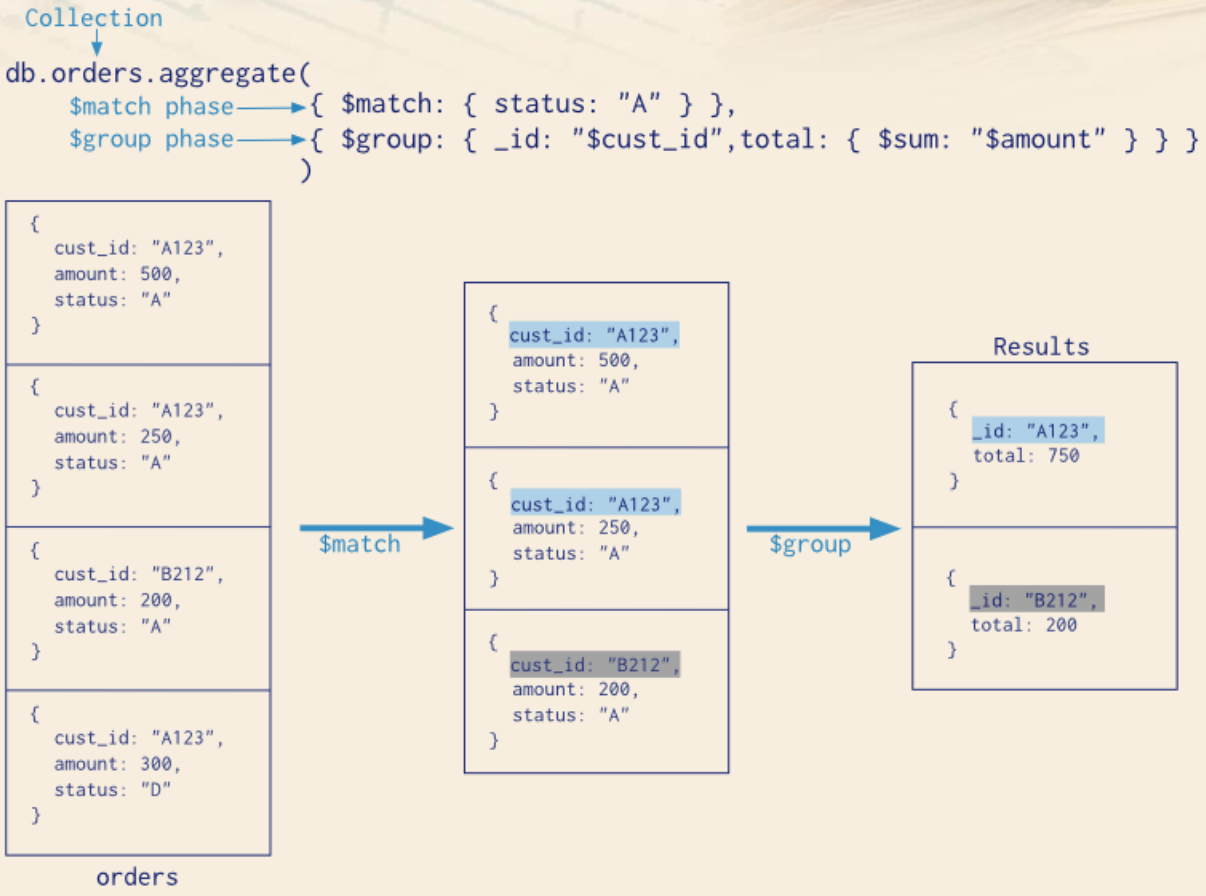
```
SELECT status,
       SUM(age) AS total
FROM people
GROUP BY status
HAVING total > 1000
```

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

Group stage: Specify the aggregation field and the aggregation function

Match Stage: specify the condition as in HAVING

Aggregation in MongoDB



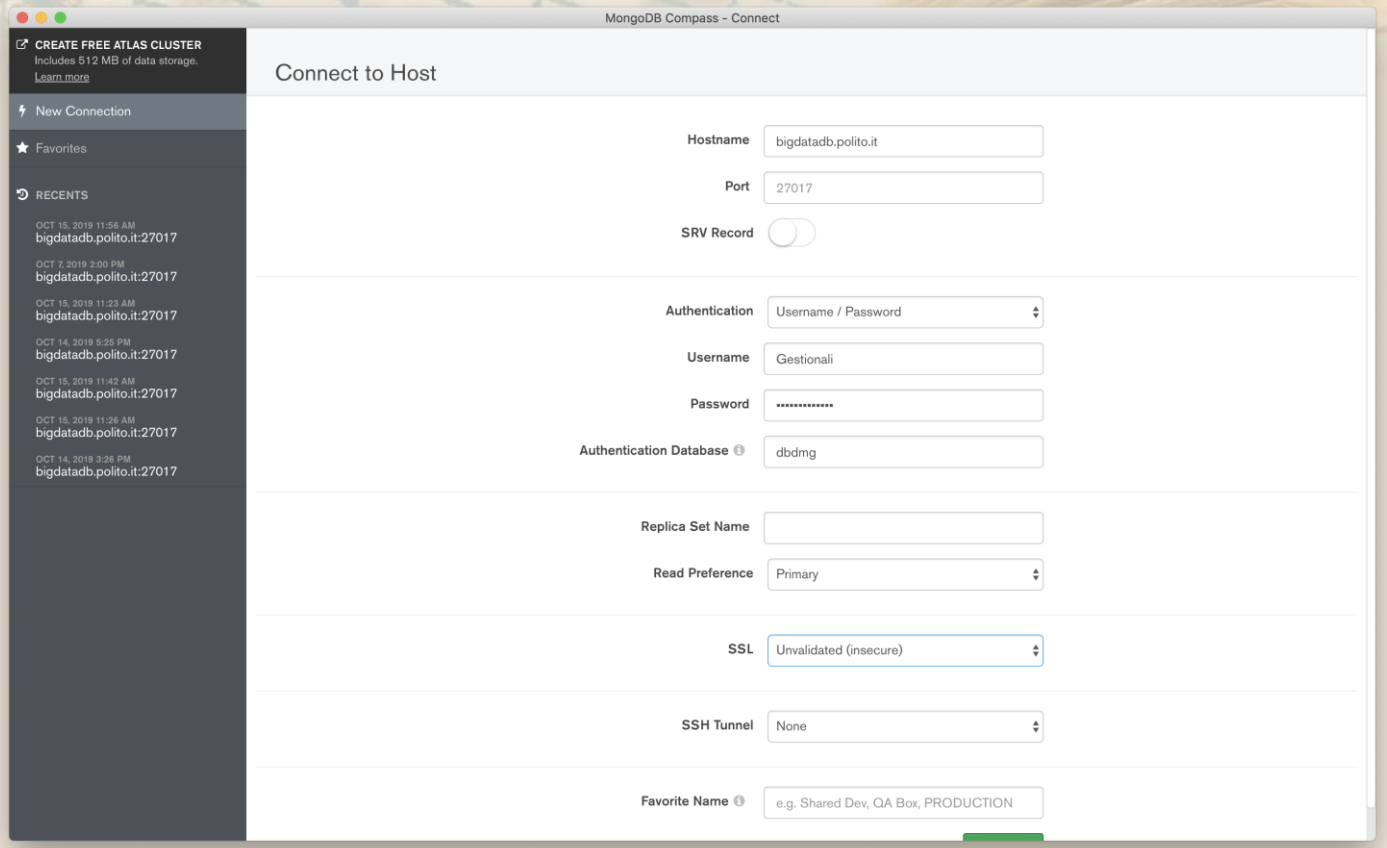
MongoDB Compass

GUI for Mongo DB

MongoDB Compass

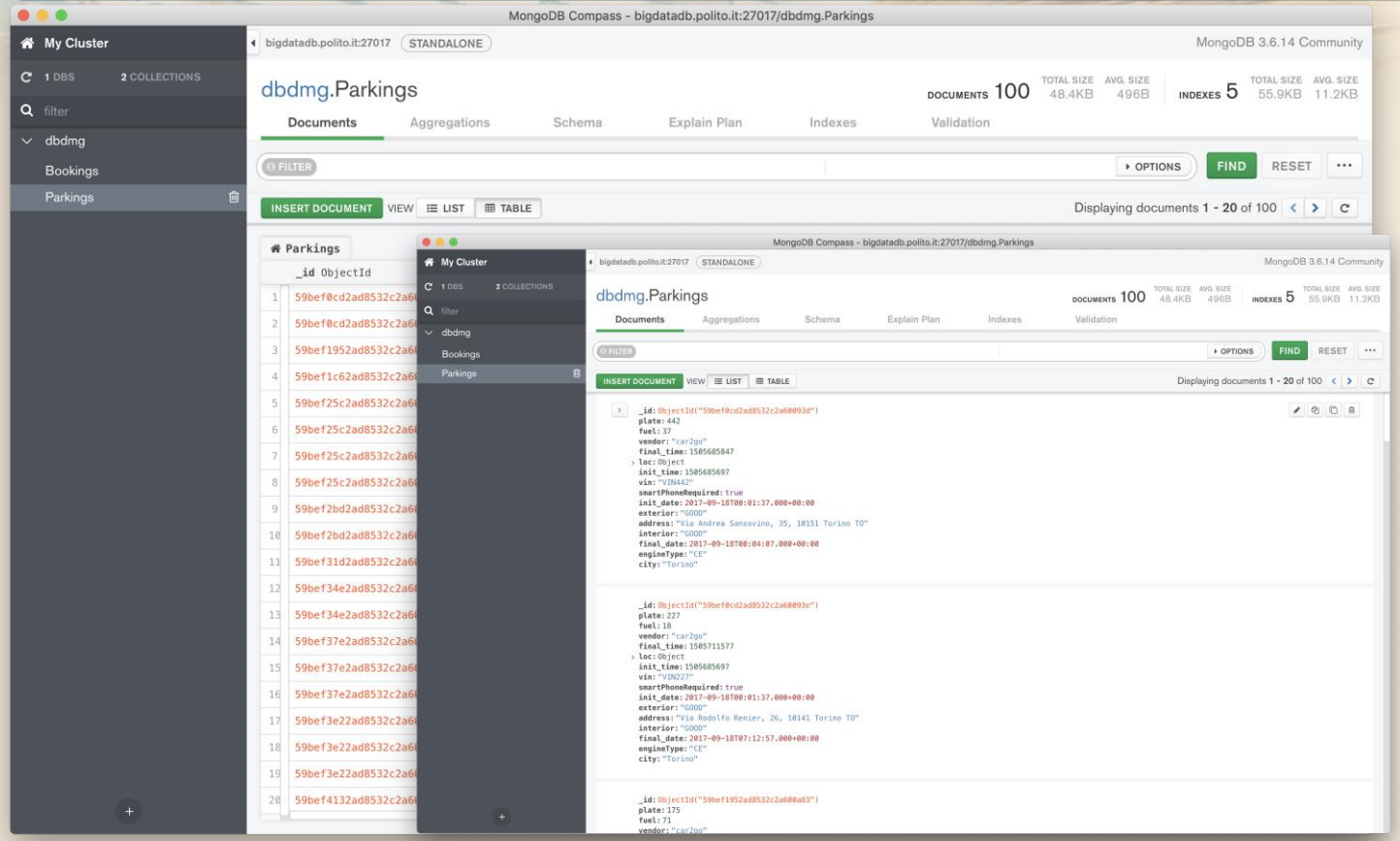
- Visually explore data.
- Available on Linux, Mac, or Windows.
- MongoDB Compass analyzes documents and displays rich structures within collections.
- Visualize, understand, and work with your geospatial data.

MongoDB Compass



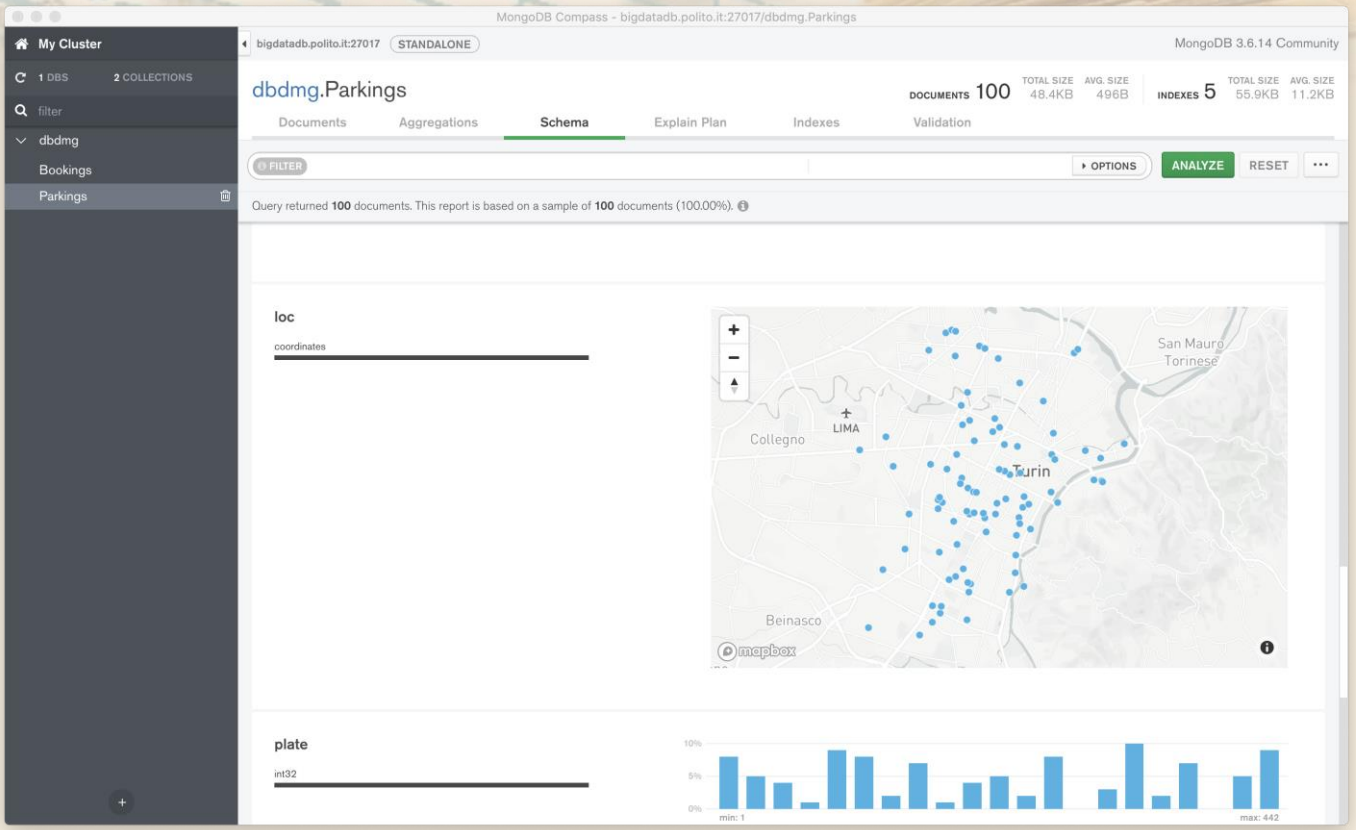
- Connect to local or remote instances of MongoDB.

MongoDB Compass



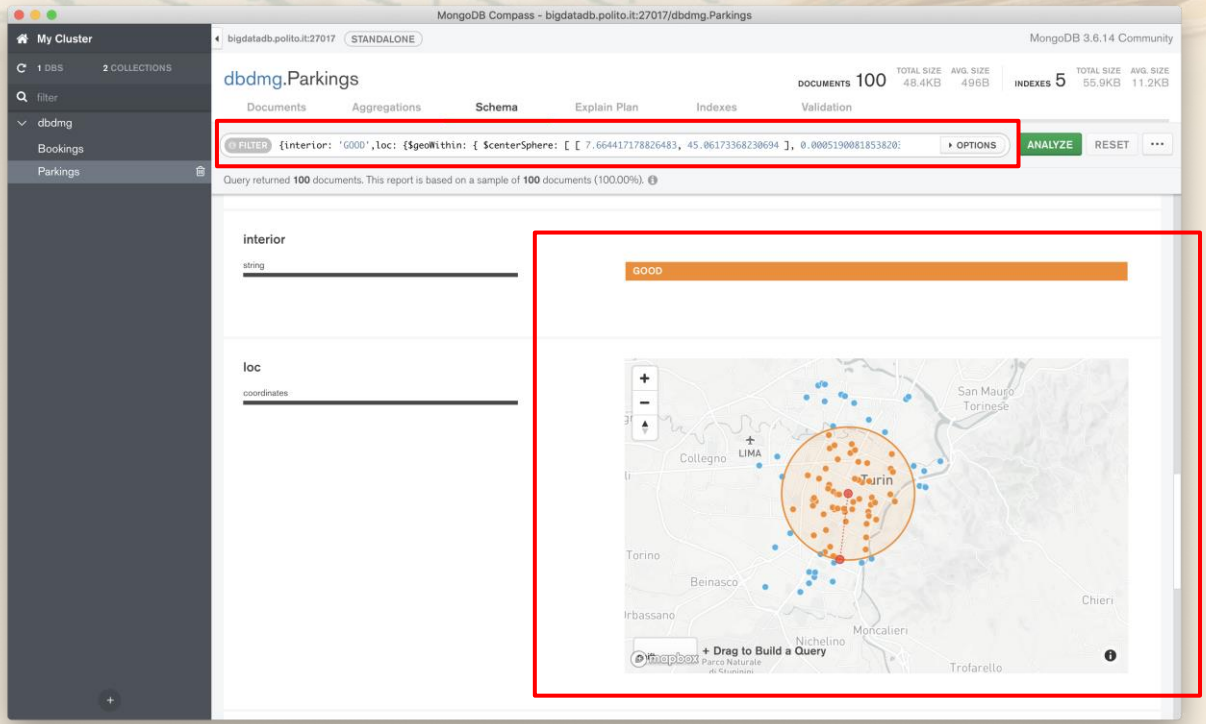
- Get an overview of the data in list or table format.

MongoDB Compass



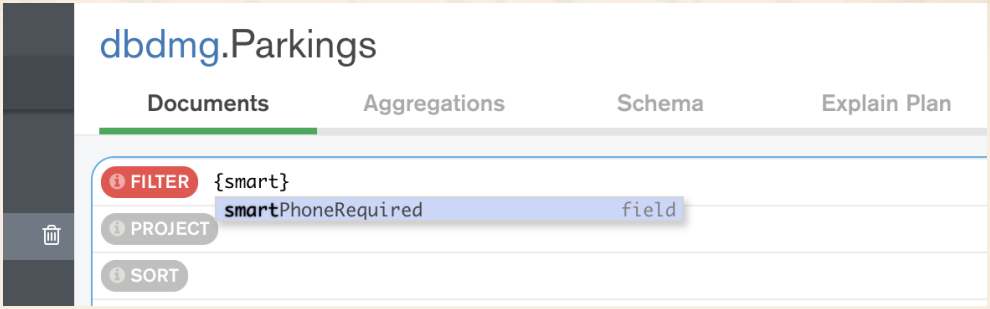
- Analyze the documents and their fields.
- Native support for geospatial coordinates.

MongoDB Compass

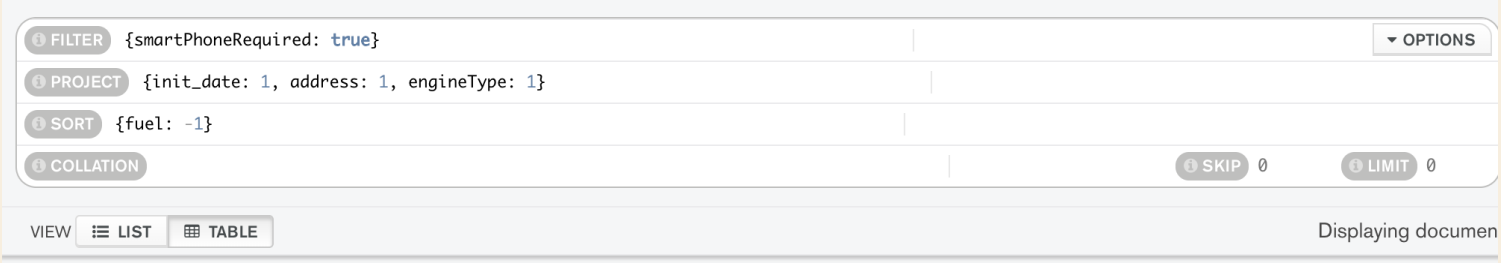


- Visually build the query conditioning on analyzed fields.

MongoDB Compass

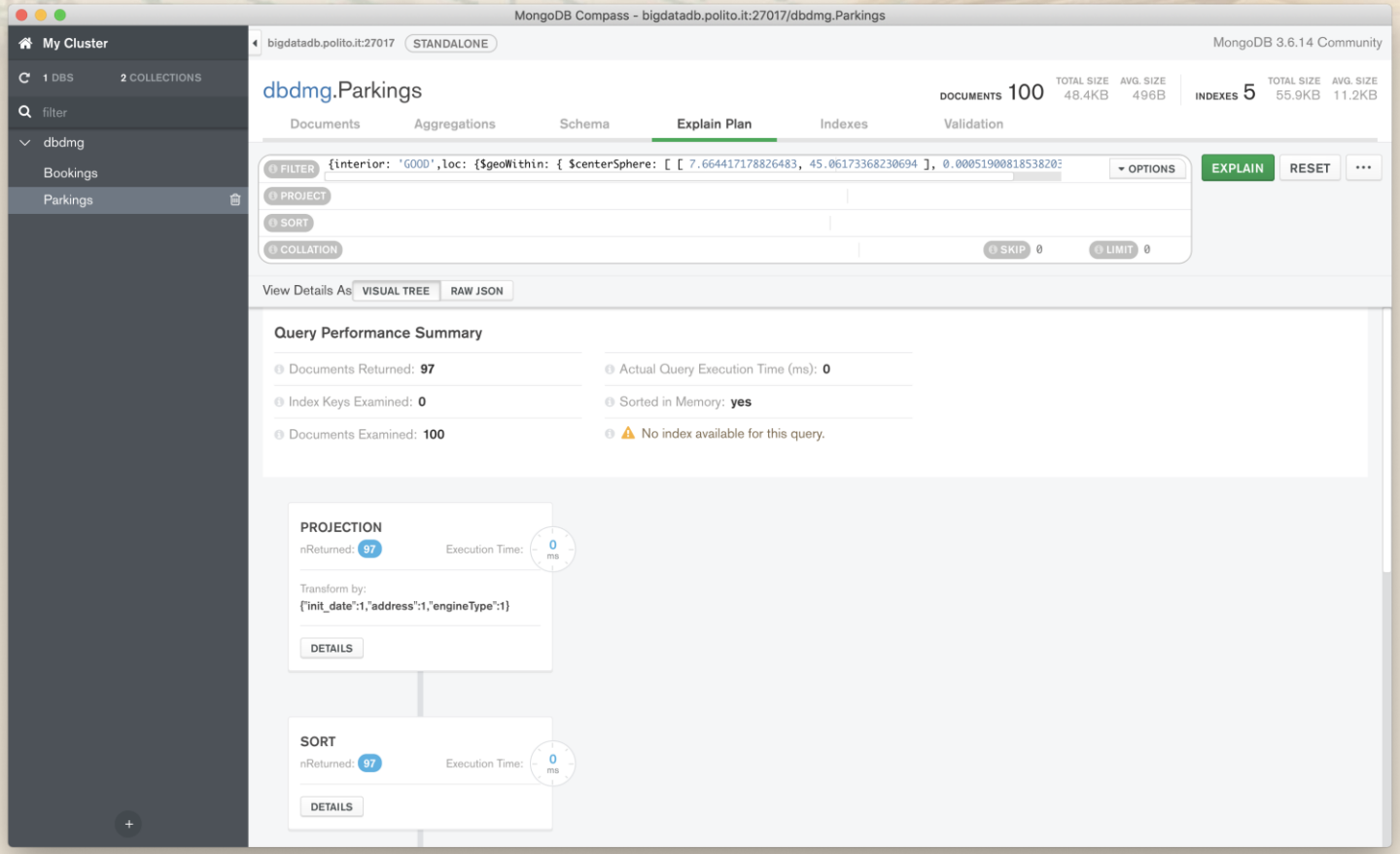


- Autocomplete enabled by default.



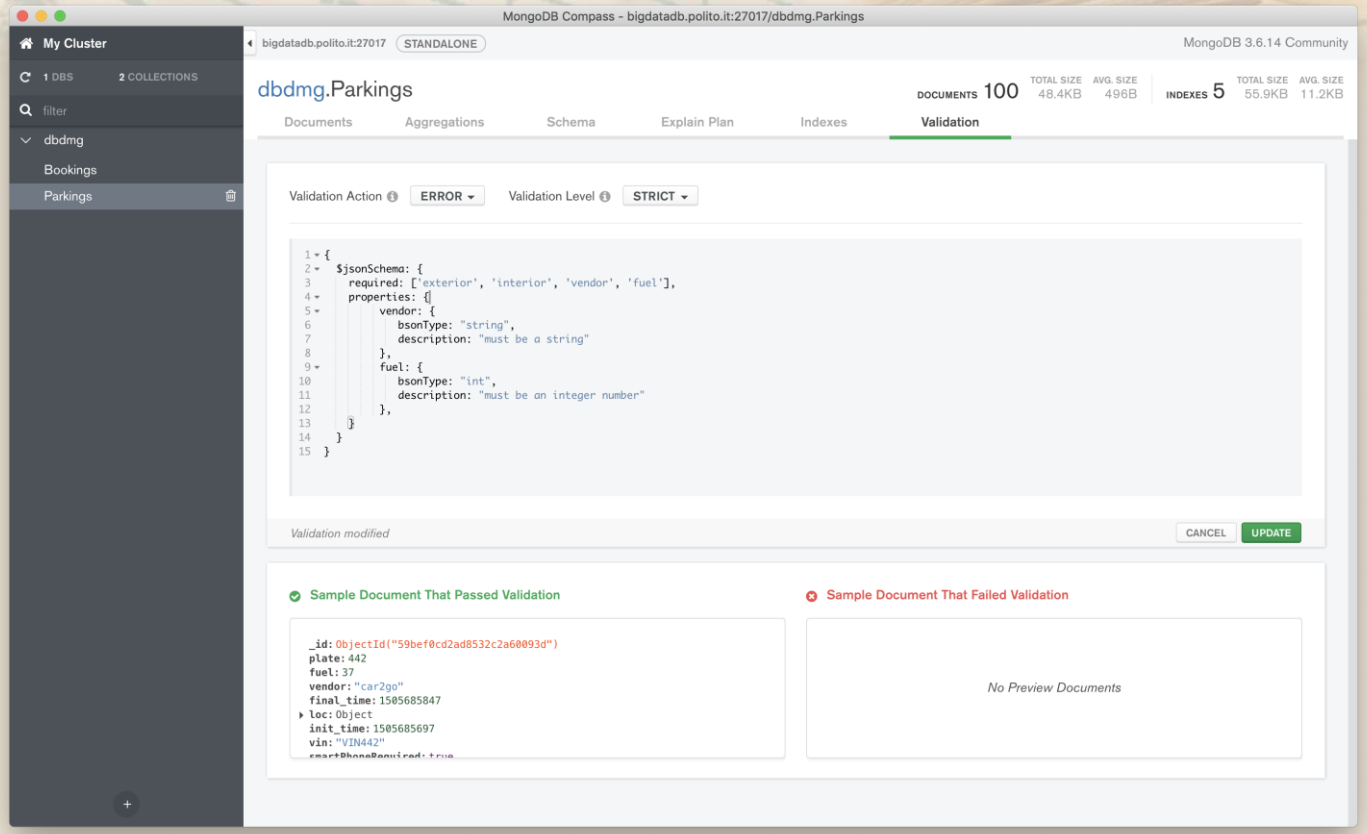
- Construct the query step by step.

MongoDB Compass



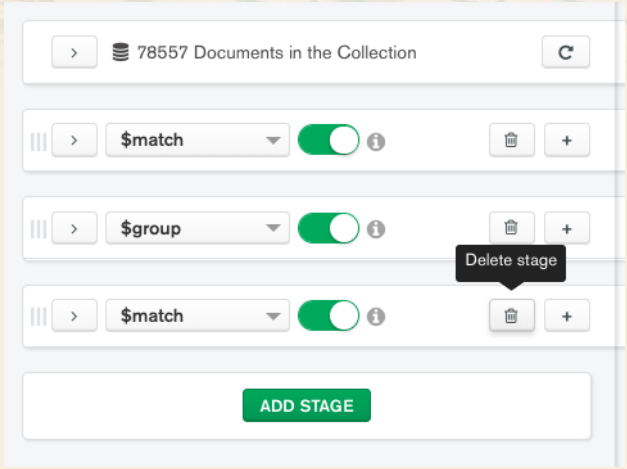
- Analyze query performance and get hints to speed it up.

MongoDB Compass

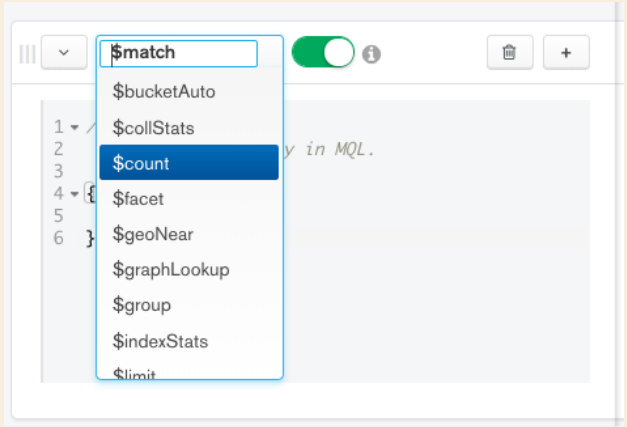


- Specify contraints to validate data
- Find unconsistent documents.

MongoDB Compass: Aggregation

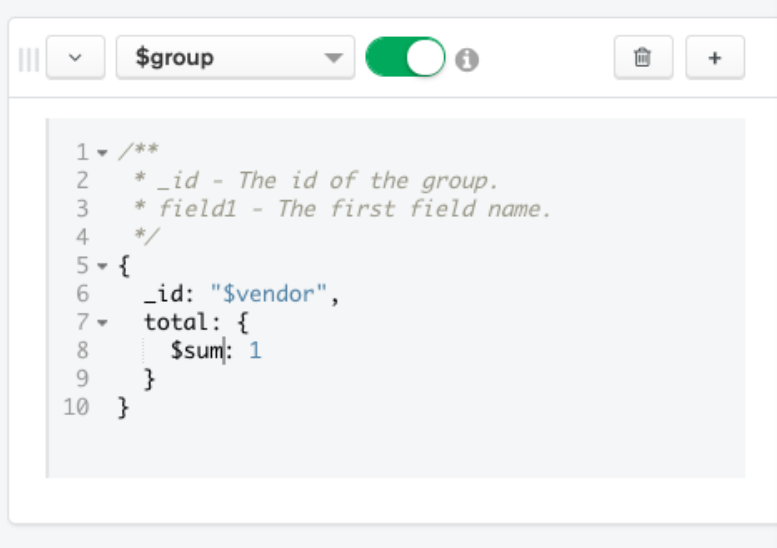


- Build a pipeline consisting of multiple aggregation stages.

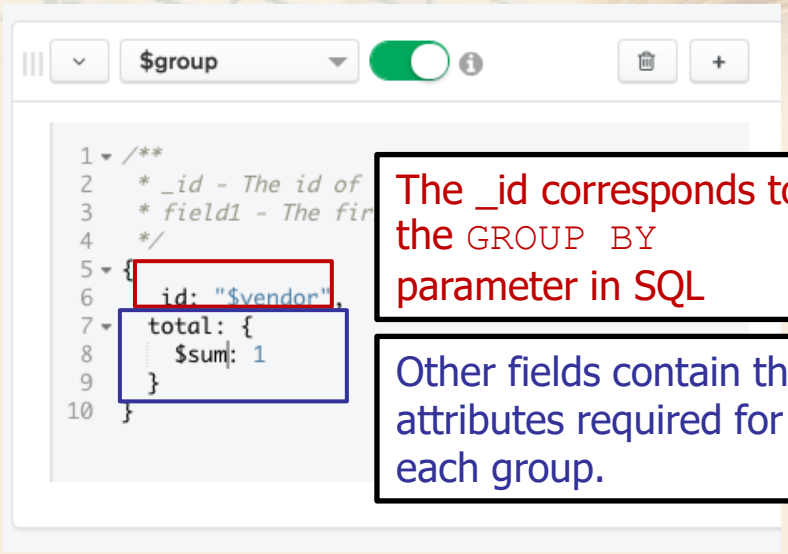


- Define the filter and aggregation attributes for each operator.

MongoDB Compass: Aggregation stages

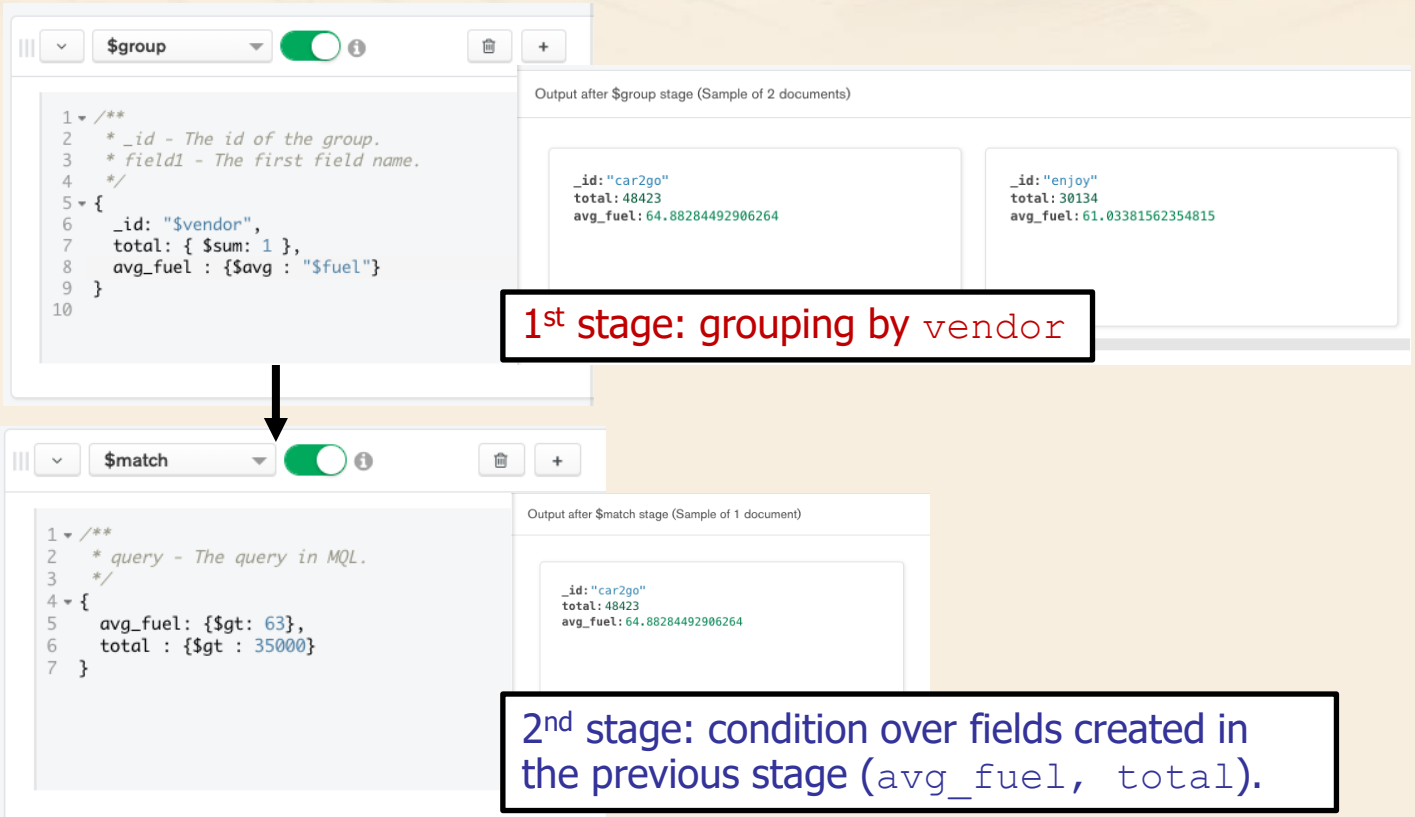


MongoDB Compass: Aggregation stages



One group for each “vendor”.

MongoDB Compass: Pipelines



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
`db.users.find()`	`SELECT * FROM users`
`db.users.insert({ user_id: 'bcd001', age: 45, status: 'A'})`	`INSERT INTO users (user_id, age, status) VALUES ('bcd001', 45, 'A')`
`db.users.update( { age: { $gt: 25 } }, { $set: { status: 'C'}}, { multi: true })`	`UPDATE users SET status = 'C' WHERE age > 25`

MongoDB: Insert documents

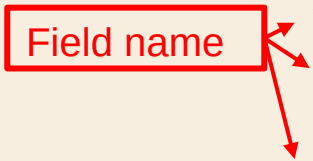
- Insert a single document in a collection
  - `db.<collection name>.insertOne( {<set of the field:value pairs of the new document>} );`
- E.g.,

```
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.,

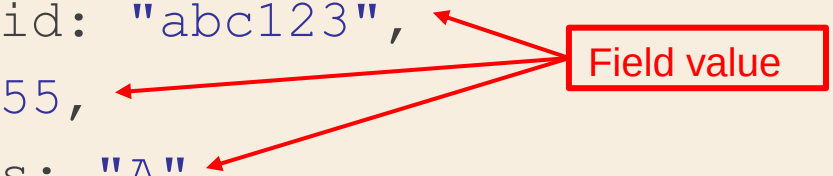
```
db.people.insertOne( {
        user_id: "abc123",
        age: 55,
        status: "A"
    } );
```



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>} );`

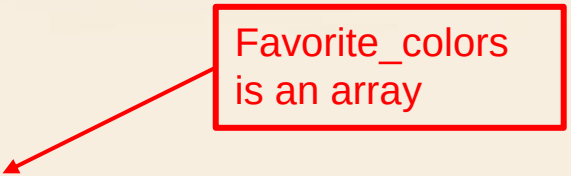
Now people contains a new document representing a user with:

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

MongoDB: Insert documents

- E.g.,

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



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
`INSERT INTO`	`insertOne()`

MongoDB: inserting data

MySQL clause	MongoDB operator
INSERT INTO	insertOne()

```
INSERT INTO
people(user_id,
       age,
       status)
VALUES ("bcd001",
        45,
        "A")
```

```
db.people.insertOne(
   {
     user_id: "bcd001",
     age: 45,
     status: "A"
   }
)
```

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

MySQL clause	MongoDB operator

`UPDATE <table>`
`SET <statement>`
`WHERE <condition>`
	`db.<table>.updateMany(`
	`  { <condition> },`
	`  { $set: {<statement>} }`
	`)`

MongoDB: updating data

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

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

MongoDB: updating data

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

UPDATE people SET status = "C" WHERE age > 25	db.people.updateMany({ age: { $gt: 25 } }, { $set: { status: "C" }})
UPDATE people SET age = **age + 3** WHERE status = "A"	db.people.updateMany({ status: "A" } , { **$inc: { age: 3 }** })

The $inc operator increments a field by a specified value

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

MongoDB: deleting data

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

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

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