NoSQL Databases

Introduction to MongoDB

DANIELE APILETTI

POLITECNICO DI TORINO

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

Document Data Design

- High-level, business-ready representation of the data
  - Records are stored into BSON Documents
    - BSON is a binary representation of JSON documents
    - field-value pairs
    - may be nested

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

Embedded Sub-Document

Embedded Sub-Document

```
{
   _id: ObjectId("5099803df3f4948bd2f98391"),
   name: { first: "Alan", last: "Turing" },
   birth: new Date('Jun 23, 1912'),
   death: new Date('Jun 07, 1954'),
   contribs: [ "Turing machine", "Turing test", "Turingery" ],
   views : NumberLong(1250000)
}
```

Document Data Design

- High-level, business-ready representation of the data
- Mapping into developer-language objects
  - date, timestamp, array, sub-documents, etc.
- Field names
  - The field name **_id** is reserved for use as a **primary key**; its value must be unique in the collection, is **immutable**, possibly autogenerated, and may be of any type other than an array.
  - Field names **cannot** contain the **null** character.
  - The server **permits** storage of field names that contain dots (.) and dollar signs ($)
  - BSON documents may have more than one field with **the same name**. Most MongoDB interfaces, however, represent MongoDB with a structure (e.g., a hash table) that does not support duplicate field names.
  - The maximum BSON document size is **16 megabytes**. To store documents larger than the maximum size, MongoDB provides GridFS.
  - Unlike JavaScript objects, the fields in a BSON document are **ordered.**

Databases and collections.

Create and delete operations (1)

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
  - Starting in MongoDB 3.2, you can enforce **document validation** rules for a collection during update and insert operations.

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`

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();
```

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

Databases and Collections

- Show collections

```
show collections
```

- Drop collections

```
db.<collection_name>.drop()
```

- E.g.
  - `db.authors.drop()`

C.R.U.D. Operations

- **Create**

```
db.users.insertOne(
  {
    name: "sue",
    age: 26,
    status: "pending"
  }
)
```

collection
field: value
field: value
field: value
document

- **Read**

```
db.users.find(
  { age: { $gt: 18 } },
  { name: 1, address: 1 }
).limit(5)
```

collection
query criteria
projection
cursor modifier

- **Update**

```
db.users.updateMany(
  { age: { $lt: 18 } },
  { $set: { status: "reject" } }
)
```

collection
update filter
update action

- **Delete**

```
db.users.deleteMany(
  { status: "reject" }
)
```

collection
delete filter

Create: insert **one** document

- 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"
    } );
```

Create: insert **one** document

- 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

Create: insert **one** document

- 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

Create: insert **one** document

- 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"
```

Create: insert **one** document

- 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"`

Create: insert **one** document

- 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

Create: insert **many** documents

- Insert multiple documents in a single statement:

```
db.<collection name>.insertMany([ <comma separated list of documents> ]);
```

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

Create: insert **many** 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"]}
] );
```

Delete

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

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

Delete

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

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

Delete

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

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

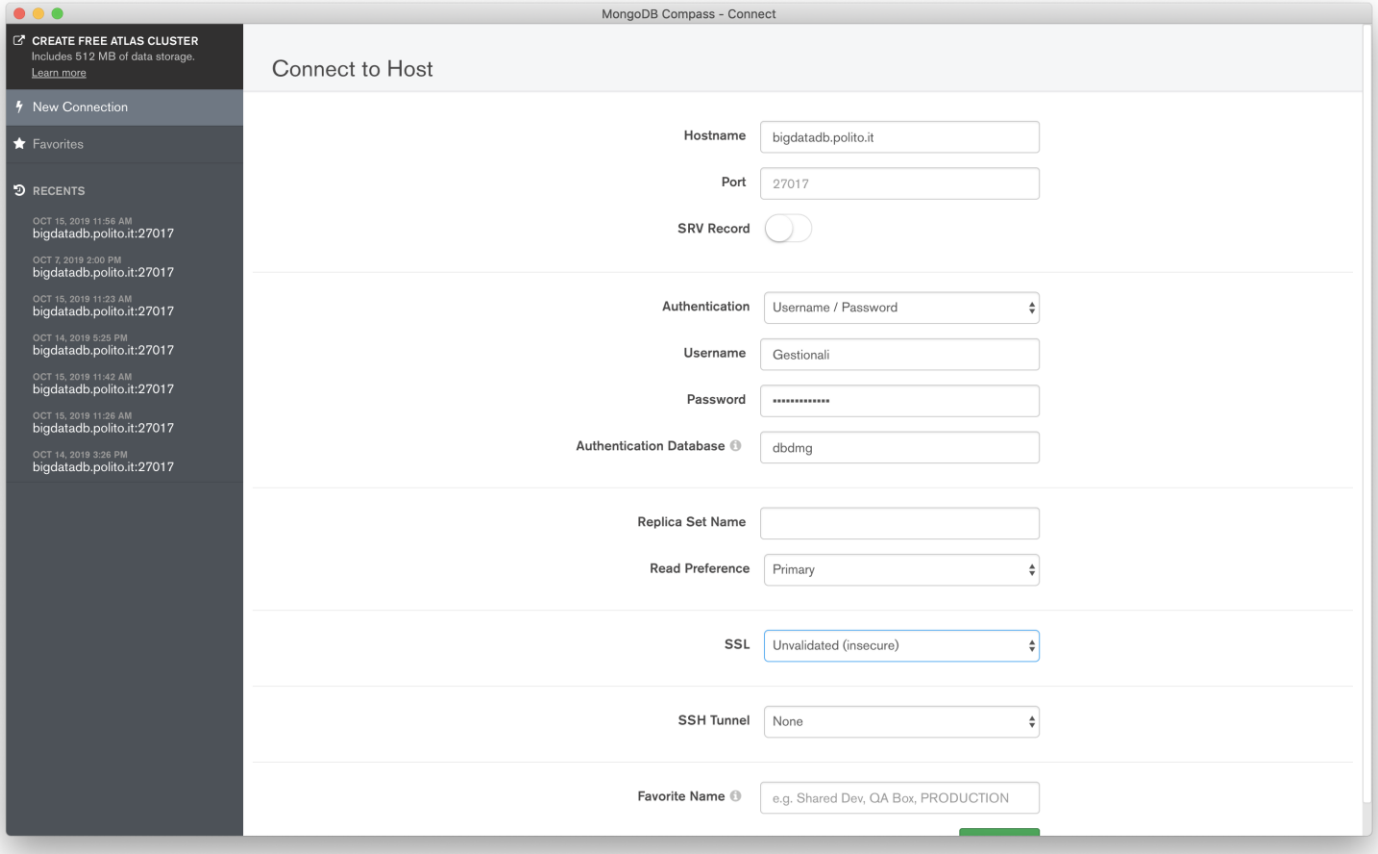
`DELETE FROM people`	`db.people.deleteMany({})`

GUI for MongoDB

MongoDB Compass

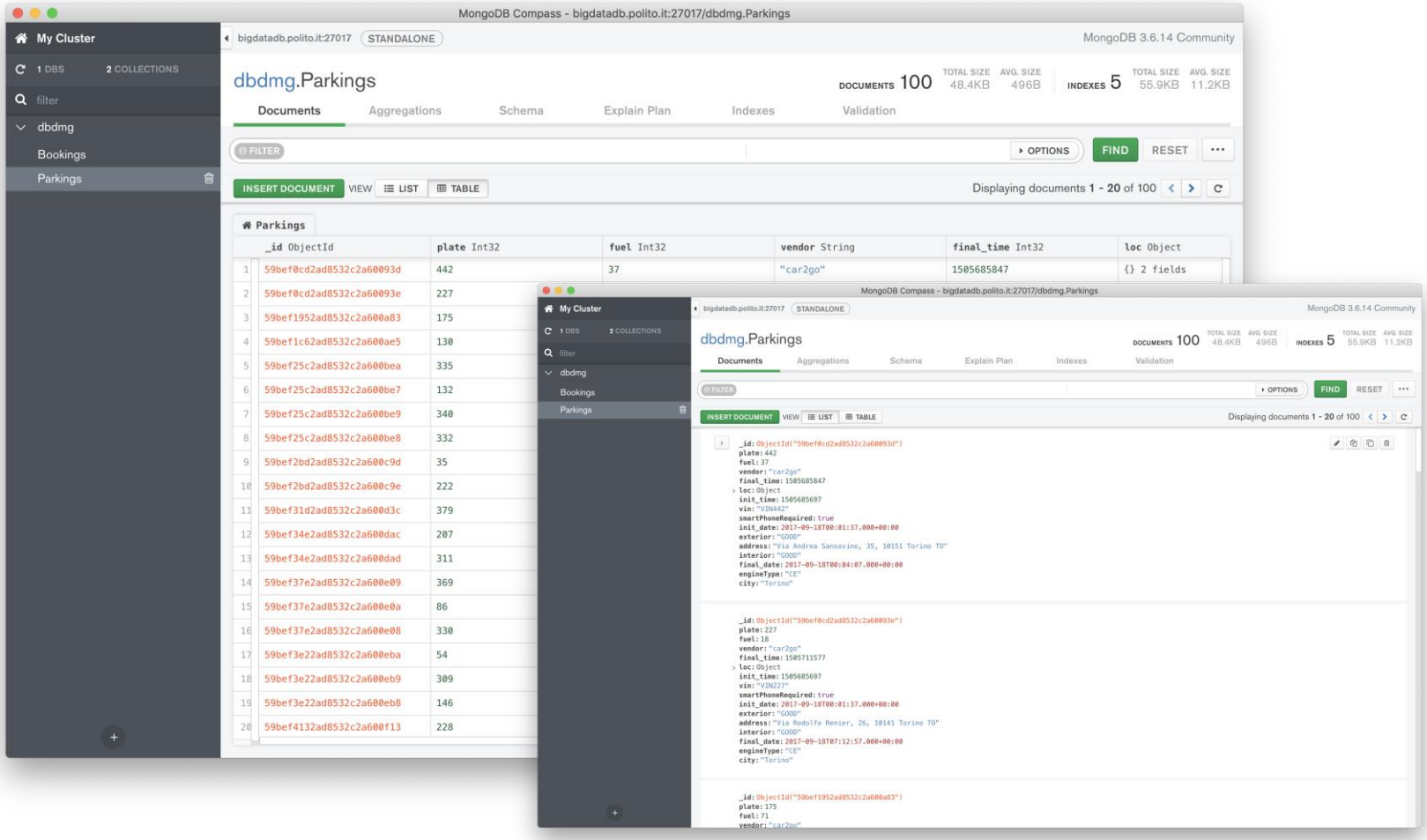
- Visually explore data.
- Available on Linux, Mac, or Windows.
- MongoDB Compass analyzes documents and displays rich structures within collections.
- Visualize, understand, and work with your geospatial data.

MongoDB Compass



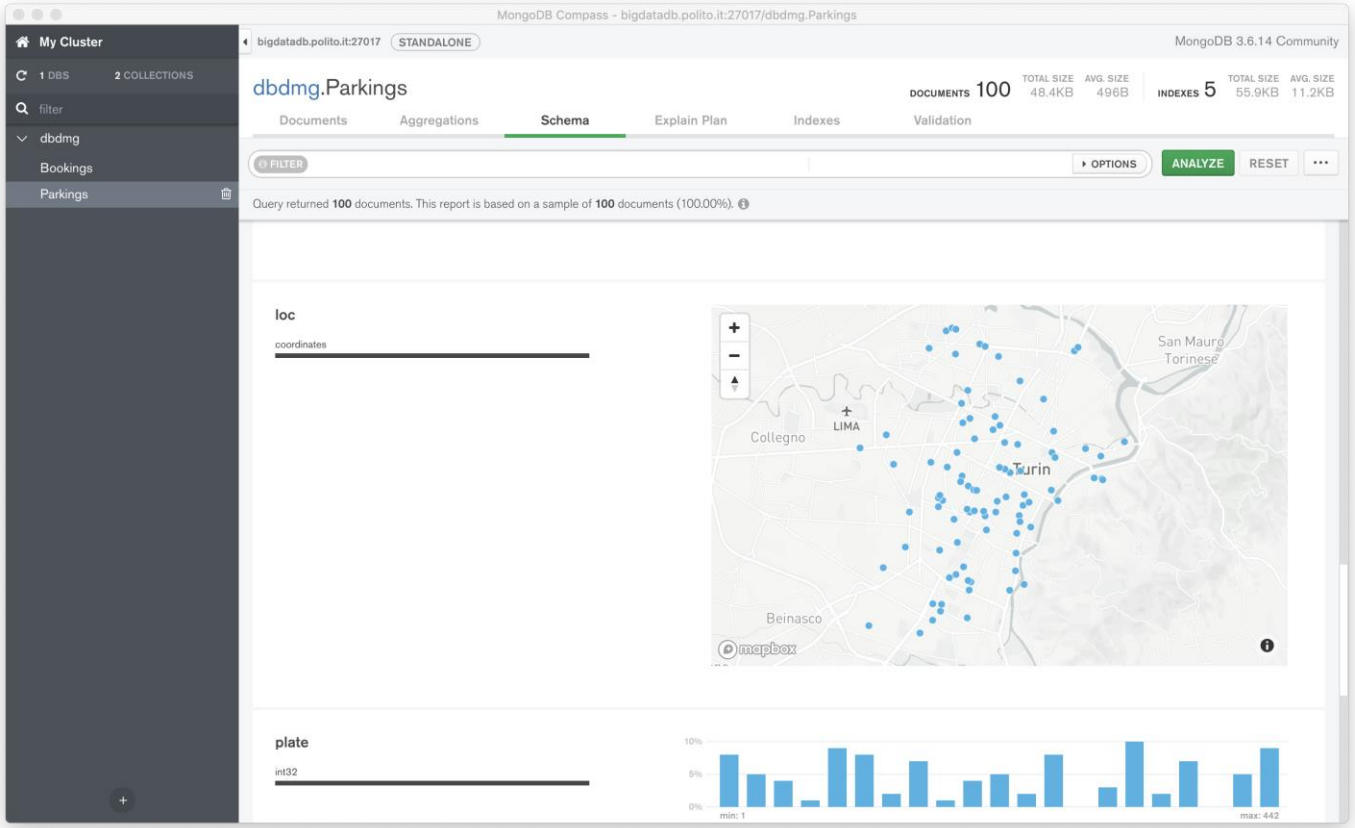
- Connect to local or remote instances of MongoDB.

MongoDB Compass



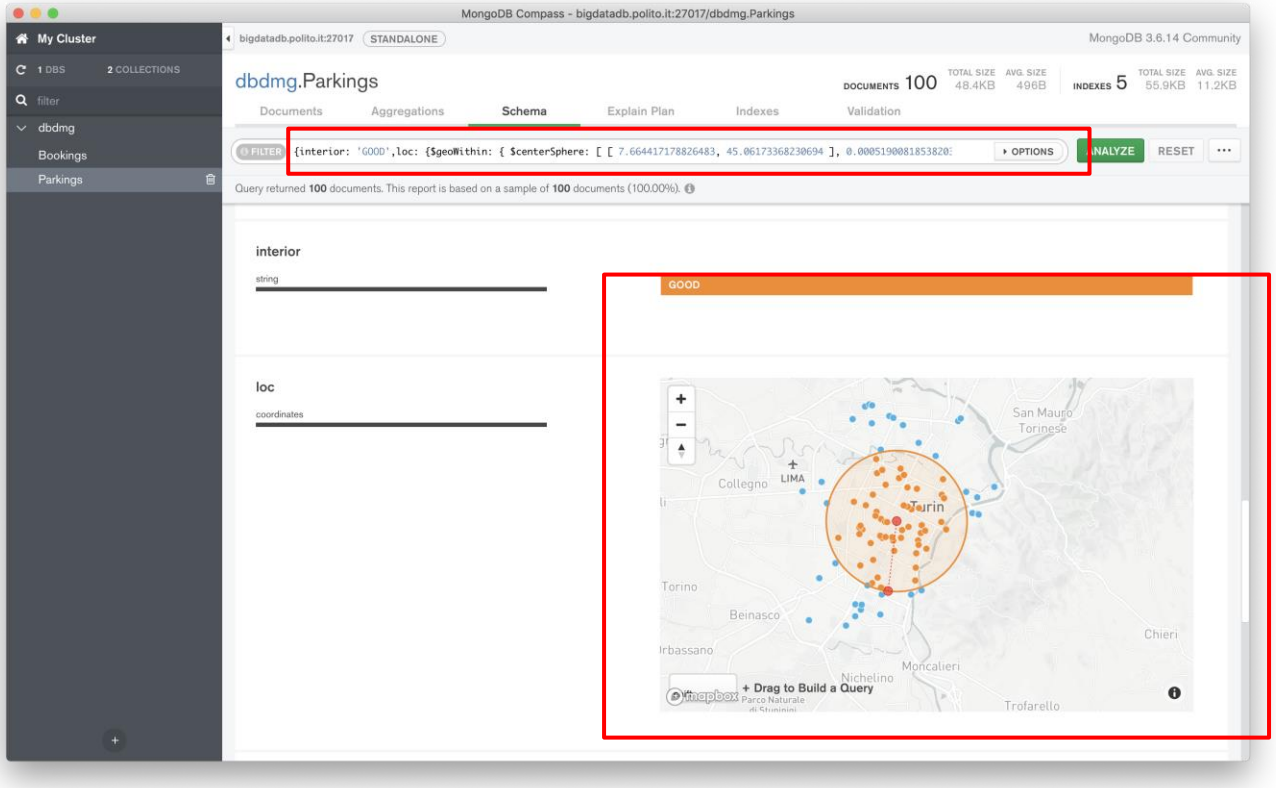
- Get an overview of the data in list or table format.

MongoDB Compass



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

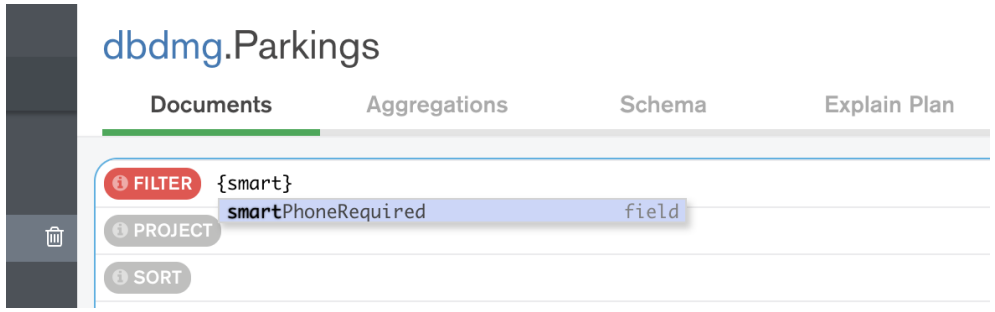
MongoDB Compass



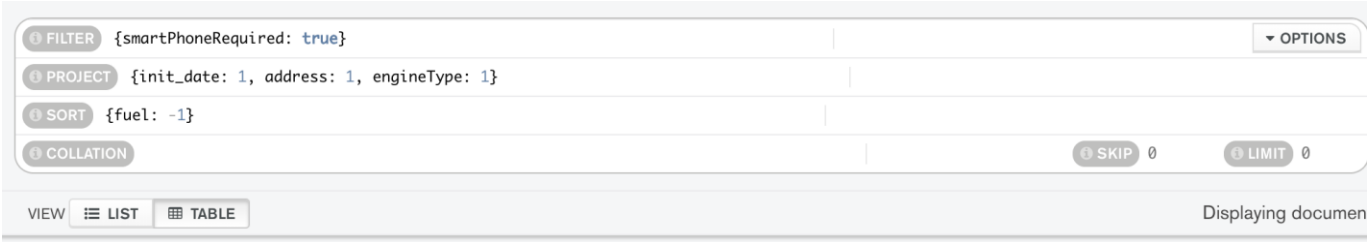
- Visually build the query conditioning on analyzed fields.

MongoDB Compass

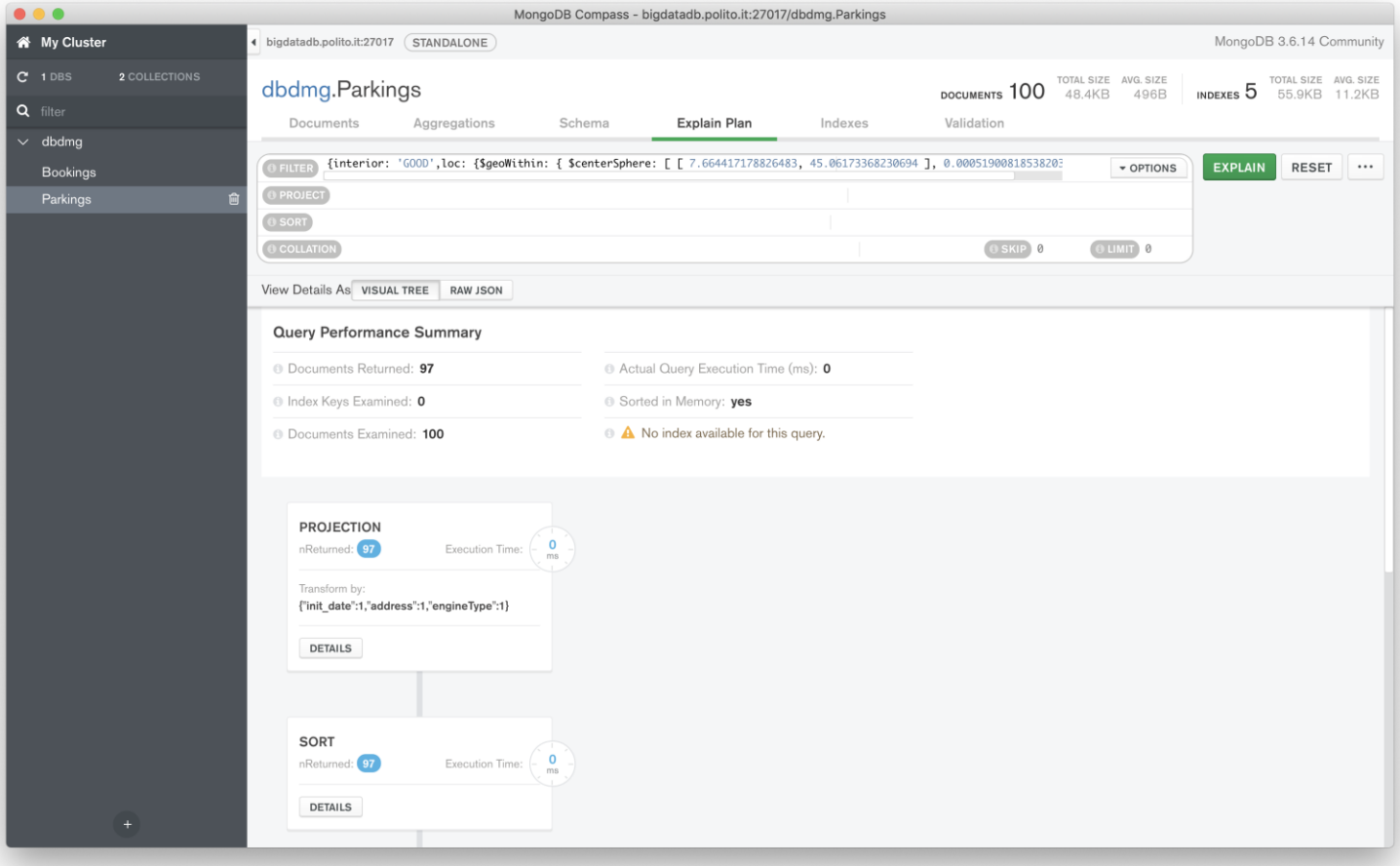
- Autcomplete enabled by default



- Construct the query step by step.

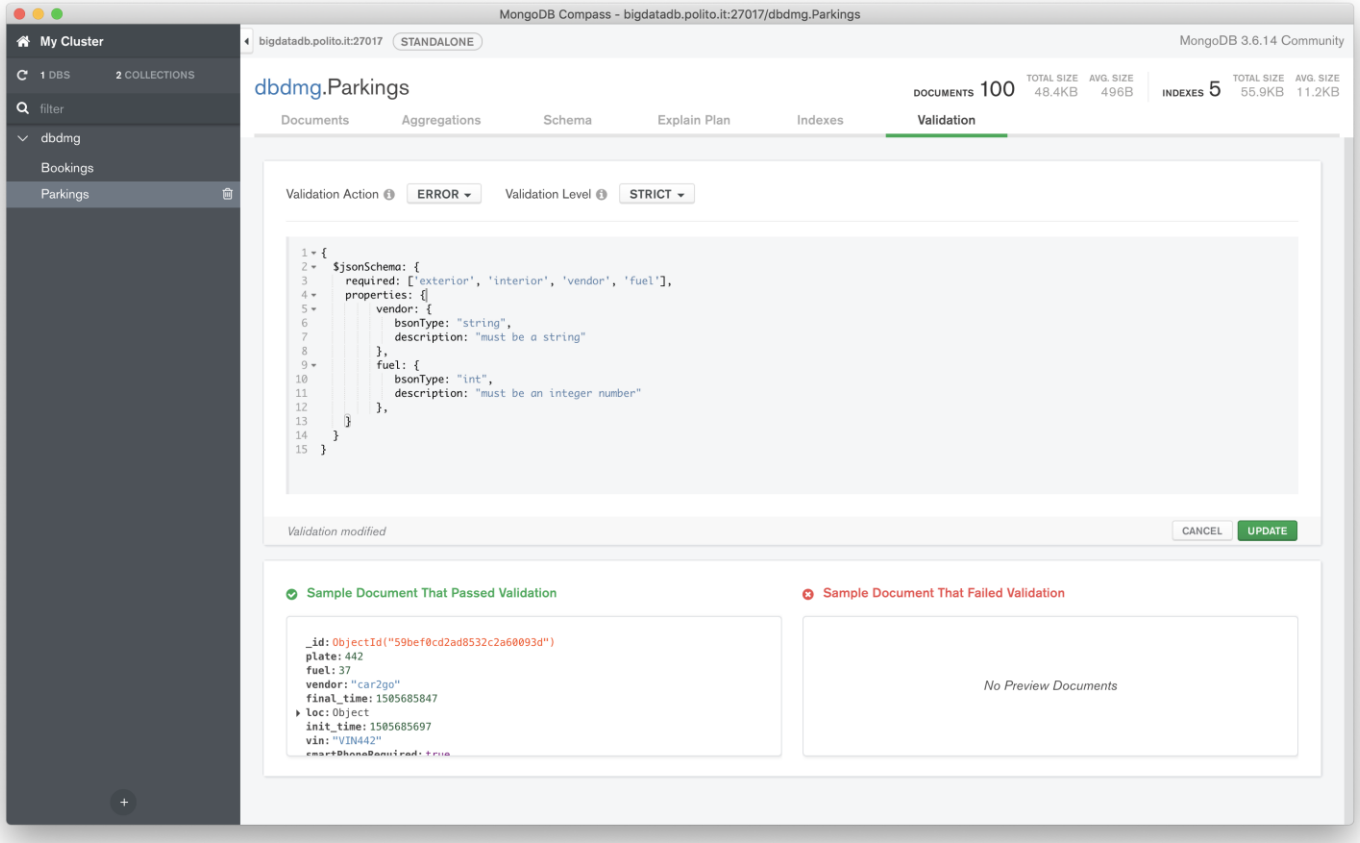


MongoDB Compass



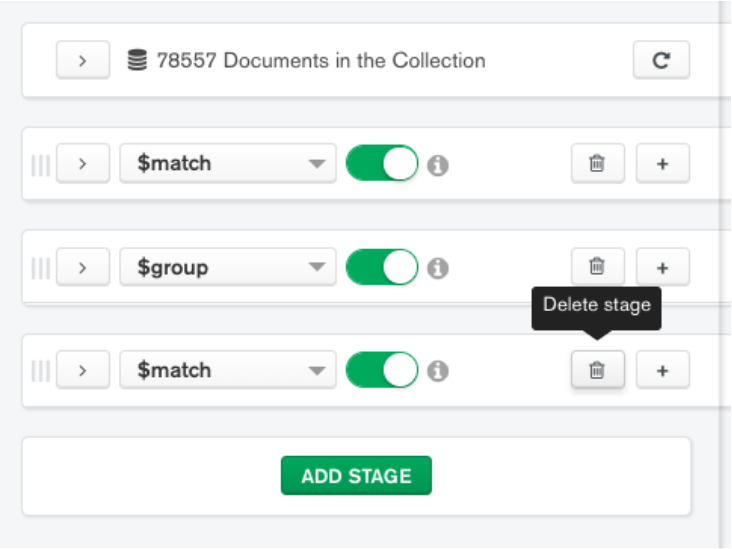
- Analyze query performance and get hints to speed it up.

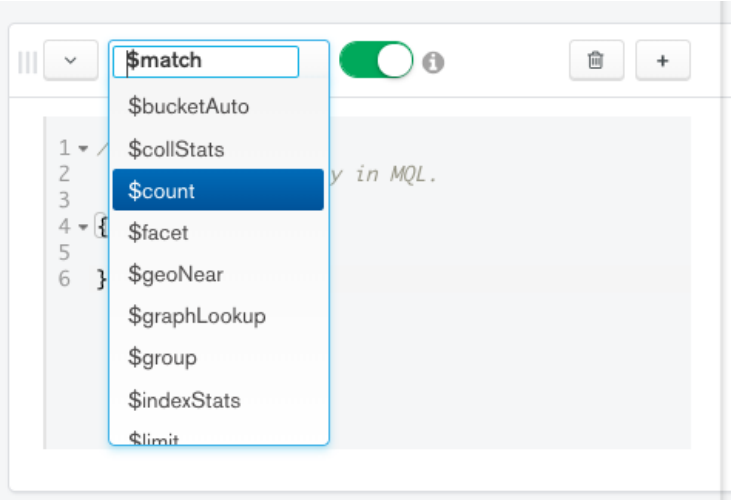
MongoDB Compass



- 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

$group

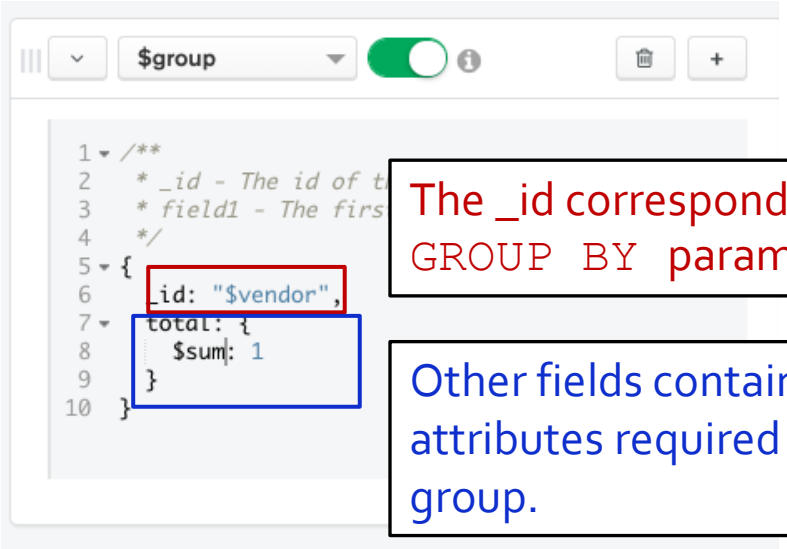
```
/**
 * _id - The id of the group.
 * field1 - The first field name.
 */
{
  _id: "$vendor",
  total: {
    $sum: 1
  }
}
```

Output after $group stage (Sample of 2 documents)

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

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

MongoDB Compass: Aggregation stages



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

Other fields contain the attributes required for each group.

One group for each "vendor".

MongoDB Compass: Pipelines

$group

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

Output after $group stage (Sample of 2 documents)

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

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

1st stage: grouping by `vendor`

$match

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

Output after $match stage (Sample of 1 document)

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

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