MongoDB Notes

Asaad Cheema

January 16, 2025

1 Introduction to Basic Database Operations

1.1 Listing Databases

To list all databases on a MongoDB server, use the command:

```
show dbs
```

This command will output the existing databases and their sizes, for example:

```
admin   40.00 KiB
config  60.00 KiB
local   72.00 KiB
```

1.2 Switching and Using Databases

To switch to a specific database, use the command:

```
use <database_name>
```

For instance, to switch to the *admin* database:

```
use admin
```

This will output:

```
switched to db admin
```

1.3 Creating a Collection

After selecting a database, you can create a new collection using:

```
db.createCollection("collection_name")
```

For example, to create a collection named *students*:

```
db.createCollection("students")
```

This will output:

```
{ ok: 1 }
```

1.4 Dropping a Database

To delete a database, first switch to that database, then use:

```
db.dropDatabase()
```

This will output:

```
{ ok: 1, dropped: 'database_name' }
```

1.5 Using a Specific Database

To confirm you're using a specific database, or to switch to it, use the command:

```
use school
```

This will output, if you're already using the *school* database:

```
already on db school
```

1.6 Inserting a Single Document into a Collection

To insert a single document into a collection, use:

```
db.students.insertOne({name: "Bob", age: 30, gpa: 3.2})
```

This will output:

```
{
  acknowledged: true,
  insertedId: ObjectId('6787ccfd790443464584d147')
}
```

1.7 Finding Documents in a Collection

To find and display all documents within a collection, use:

```
db.students.find()
```

This will display documents like:

```
{
  _id: ObjectId('6787ccfd790443464584d147'),
  name: 'Bob',
  age: 30,
  gpa: 3.2
}
```

1.8 Inserting Multiple Documents

To insert multiple documents at once into a collection, use:

```
db.students.insertMany([
    {name: "Pattric", age: 38, gpa: 1.5},
    {name: "Asaad", age: 28, gpa: 4.0},
    {name: "Gerry", age: 18, gpa: 2.5}
])
```

This will output:

```
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId('6787cd8c790443464584d148'),
    '1': ObjectId('6787cd8c790443464584d149'),
    '2': ObjectId('6787cd8c790443464584d14a')
  }
}
```

1.9 Data Types in MongoDB

MongoDB supports a wide array of data types, each suited for different uses within documents. Here are some of the most common data types:

- **String**: Stores text as UTF-8 encoded characters.
- **Integer**: Available as 32-bit or 64-bit numbers, suitable for numerical data.
- **Boolean**: Represents 'true' or 'false' values.
- **Double**: Stores floating point numbers, for precision calculations.
- **ObjectId**: A unique 12-byte identifier for documents.
- **Array**: An ordered list of values, which can include various data types.
- **Embedded documents**: Allows storing a document as a field within another document.
- **Null**: Represents the absence of a value.
- **Date**: Stores date and time as the number of milliseconds since the Unix epoch (January 1, 1970). Useful for recording timestamps and performing time-based queries.

  `Example: { eventDate: ISODate("2023-10-05T14:00:00Z") }`

- **Binary Data**: For storing data such as images, audio files, etc., in binary format.
- **Code**: To store JavaScript code that can be executed within the database.
- **Regular expression**: Useful for pattern matching within queries.

1.9.1 Working with the Date Data Type

The **Date** data type in MongoDB is particularly powerful for applications that require recording events or entities with a temporal dimension. It stores the date and time as a 64-bit integer, which represents the number of milliseconds since the Unix epoch. You can manipulate and query dates using MongoDB's built-in methods and operators:

```
// Insert a document with a specific date
db.events.insertOne({event: "Webinar", date: ISODate("2023-10-05T14:00:00Z")})

// Query documents for events after a specific date
db.events.find({date: {$gt: ISODate("2023-10-01")}})
```

These capabilities make it easy to handle and query date and time information effectively.

1.10 Sorting and Limiting Results in Queries

MongoDB provides powerful methods to sort and limit the results returned by queries. These are crucial for managing large datasets and returning only the most relevant documents.

1.10.1 Sorting Documents

To sort documents by a specific field, use the `sort()` method. The sort method takes a document containing the field name(s) and the direction of the sort (1 for ascending order, -1 for descending order). For example, to sort students by their GPA in descending order:

```
db.students.find().sort({gpa: -1})
```

This will return students sorted from the highest to the lowest GPA.

1.10.2 Limiting the Number of Results

The `limit()` method is used to restrict the number of documents returned by a query. This is especially useful when you need only a certain number of results from a query. For instance, to find the student with the highest GPA:

```
db.students.find().sort({gpa: -1}).limit(1)
```

This query will sort the students by GPA in descending order and then return only the top student.

Combining `sort()` and `limit()` is particularly effective in scenarios where you need to retrieve the top or bottom records based on a certain criterion.

1.11 Querying and Projecting in MongoDB

1.11.1 Querying Based on Specific Criteria

MongoDB allows querying documents based on specific criteria using field-value pairs. This is useful when you need to retrieve documents that match certain conditions. For example, to find a student named "Asaad":

```
db.students.find({name: "Asaad"})
```

This query will return all documents within the *students* collection where the name field is exactly "Asaad".

1.11.2 Projecting Specific Fields from Documents

Projection in MongoDB is used to specify which fields should be included or excluded from the results of a query. This can help reduce the amount of data transferred from the database server to the client by omitting unnecessary fields. For instance, to return only the names of all students without their unique identifiers:

```
db.students.find({}, {_id: 0, name: 1})
```

This query specifies that the _id field should not be returned (_id: 0) and only the name field should be included (name: 1) for each document in the results.

Note: In projection, setting a field to '1' includes it in the results, while setting it to '0' excludes it. By default, the _id field is always included unless explicitly excluded.

These querying and projecting techniques are fundamental for manipulating and accessing data efficiently in MongoDB, allowing for more targeted and streamlined data retrieval.

1.12 Updating Documents in MongoDB

1.12.1 Updating a Single Document

MongoDB allows updating individual documents using the `updateOne()` method. This method updates the first document that matches the query criteria. For example, to update the *fullTime* status for a student named "Asaad":

```
db.students.updateOne(
    {name: "Asaad"},
    {$set: {fullTime: false}}
)
```

This command sets the *fullTime* field to *false* for the first document where *name* equals "Asaad".

To remove a field from a document, use the `$unset` operator. For example, to remove the *fullTime* field from a specific student using their _id:

```
db.students.updateOne(
    {_id: ObjectId("6787cd8c790443464584d149")},
    {$unset: {fullTime: ""}}
)
```

This command will remove the *fullTime* field from the document with the specified _id.

1.12.2 Updating Multiple Documents

The `updateMany()` method is used to update all documents that match the query criteria. For example, to set *fullTime* to *true* for all students who have a GPA greater than 3.0:

```
db.students.updateMany(
    {gpa: {$gt: 3.0}},
    {$set: {fullTime: true}}
)
```

This will update the *fullTime* field for every student whose GPA exceeds 3.0, setting it to *true*.

Note: The `$set` and `$unset` operators are commonly used in updates to modify the fields of a document. The `$set` operator adds or modifies the specified field with a new value, while the `$unset` operator removes the specified field.

These update methods provide flexibility and power in modifying data stored in MongoDB, allowing precise control over document fields and their values.

1.12.3 Updating Multiple Documents Based on Field Existence

The `updateMany()` method can also be used to update documents based on the existence of a field within the documents. This is useful when you want to modify documents that include a certain field, regardless of the field's value. For example, to add a new field *status* with the value "reviewed" to all students who currently have a *fullTime* field:

```
db.students.updateMany(
    {fullTime: {$exists: true}},
    {$set: {status: "reviewed"}}
)
```

This command will update every student document where the *fullTime* field exists, adding a new field called *status* with the value "reviewed".

Note: The `$exists` operator in MongoDB is used to find documents that have (or do not have) a particular field, specified by true or false. This makes it highly effective for updates based on the presence or absence of data elements within your documents.

1.13 Deleting Documents in MongoDB

1.13.1 Deleting a Single Document

To delete a single document from a MongoDB collection, you can use the `deleteOne()` method. This method removes the first document that matches the query criteria. For example, to delete a student named "Asaad":

```
db.students.deleteOne({name: "Asaad"})
```

This command will delete the first document where the *name* equals "Asaad". It is important to ensure that the query criteria accurately target the intended document to avoid unintentional deletions.

1.13.2 Deleting Multiple Documents

The `deleteMany()` method allows for the deletion of all documents that match the specified query criteria. For instance, if you want to delete all students who are not full-time:

```
db.students.deleteMany({fullTime: false})
```

This will remove all documents from the *students* collection where the *fullTime* field is *false*.

Note: Care should be taken when using `deleteMany()` because it can potentially remove a large number of documents. Always double-check the query criteria before executing a deletion to ensure that only the intended documents are affected.

These deletion methods are powerful tools for managing the contents of your database, allowing for precise control over the data being stored. Proper caution should always be exercised to prevent data loss.

1.14 Using Comparison Operators in MongoDB Queries

MongoDB provides several comparison operators that can be used to perform more specific queries. Here are examples using different operators:

1.14.1 Using $ne (not equal)

To find documents where the value of a field is not equal to a specified value:

```
db.students.find({name: {$ne: "Asaad"}})
```

This query returns all documents where the *name* is not "Asaad".

1.14.2 Using $lt (less than) and $lte (less than or equal to)

To find documents where the value of a field is less than a specified value:

```
db.students.find({age: {$lt: 20}})
```

And to find documents where the age is less than or equal to 28:

```
db.students.find({age: {$lte: 28}})
```

1.14.3 Using $gte (greater than or equal to)

To find documents where the age is greater than or equal to 28, and sort them by name in descending order:

```
db.students.find({age: {$gte: 28}}).sort({name: -1})
```

1.14.4 Using $in and $nin (in and not in)

To find documents where the name is either "Pattric" or "Asaad":

```
db.students.find({name: {$in: ["Pattric", "Asaad"]}})
```

And to find documents where the name is neither "Pattric" nor "Asaad":

```
db.students.find({name: {$nin: ["Pattric", "Asaad"]}})
```

Additional Comparison Operators:

- **$gt (greater than)**: Finds documents where the value of the specified field is greater than the specified value.
- **$eq (equal)**: Finds documents where the value of the specified field is equal to the specified value.

These operators allow for flexible and powerful querying capabilities, enabling precise data retrieval based on specific conditions.

1.15 Using Logical Operators in MongoDB Queries

MongoDB supports logical operators that allow for complex queries involving multiple conditions. Here are examples demonstrating the use of '*and*', 'or', and '*nor*' :

1.15.1 Using $and

The $and operator performs a logical AND operation on an array of two or more expressions and selects the documents that satisfy all the conditions.

- To find students who are full-time and aged 22 or younger:

```
db.students.find({$and: [{fulltime: true}, {age: {$lte: 22}}]})
```

- To find students who have a GPA of at least 2.5 and are aged 22 or younger:

  ```
  db.students.find({$and: [{gpa: {$gte: 2.5}}, {age: {$lte: 22}}]})
  ```

1.15.2 Using $or

The $or operator performs a logical OR operation on an array of two or more conditions and selects the documents that satisfy at least one of the conditions.

```
db.students.find({$or: [{gpa: {$gte: 2.5}}, {age: {$lte: 22}}]})
```

This query returns students who either have a GPA of at least 2.5 or are aged 22 or younger.

1.15.3 Using $nor

The $nor operator performs a logical NOR operation and selects the documents that fail all of the conditions specified.

- To find students who neither have a GPA of at least 2.5 nor are aged 22 or younger:

  ```
  db.students.find({$nor: [{gpa: {$gte: 2.5}}, {age: {$lte: 22}}]})
  ```

- To find students who are not aged 22 or older:

  ```
  db.students.find({$nor: [{age: {$gte: 22}}]})
  ```

These logical operators provide powerful tools for querying documents based on multiple criteria, enabling more precise control over the data retrieved.

Note: When using logical operators, ensure that each condition within the operator is a document itself. This structure allows MongoDB to evaluate each condition separately before applying the logical operator.

1.16 Using Logical Operators in MongoDB Queries

MongoDB supports logical operators that allow for complex queries involving multiple conditions. Here are examples demonstrating the use of `$and`, `$or`, and `$nor`:

1.16.1 Using `$and`

The `$and` operator performs a logical AND operation on an array of two or more expressions and selects the documents that satisfy all the conditions.

- To find students who are full-time and aged 22 or younger:

  ```
  db.students.find({$and: [{fulltime: true}, {age: {$lte: 22}}]})
  ```

- To find students who have a GPA of at least 2.5 and are aged 22 or younger:

  ```
  db.students.find({$and: [{gpa: {$gte: 2.5}}, {age: {$lte: 22}}]})
  ```

1.16.2 Using `$or`

The `$or` operator performs a logical OR operation on an array of two or more conditions and selects the documents that satisfy at least one of the conditions.

```
db.students.find({$or: [{gpa: {$gte: 2.5}}, {age: {$lte: 22}}]})
```

This query returns students who either have a GPA of at least 2.5 or are aged 22 or younger.

1.16.3 Using `$nor`

The `$nor` operator performs a logical NOR operation and selects the documents that fail all of the conditions specified.

- To find students who neither have a GPA of at least 2.5 nor are aged 22 or younger:

  ```
  db.students.find({$nor: [{gpa: {$gte: 2.5}}, {age: {$lte: 22}}]})
  ```

- To find students who are not aged 22 or older:

  ```
  db.students.find({$nor: [{age: {$gte: 22}}]})
  ```

These logical operators provide powerful tools for querying documents based on multiple criteria, enabling more precise control over the data retrieved.

Note: When using logical operators, ensure that each condition within the operator is a document itself. This structure allows MongoDB to evaluate each condition separately before applying the logical operator.

1.17 Performance Analysis and Index Management in MongoDB

1.17.1 Analyzing Query Performance

To analyze the performance of a query, MongoDB provides the `explain()` method, which can be used with different verbosity levels. The `"executionStats"` mode provides detailed statistics about the query execution process, including the number of documents examined and the time taken. For example, to analyze how a query searching for a student named "Asaad" is executed:

```
db.students.find({name: "Asaad"}).explain("executionStats")
```

This will return execution details that help in understanding the efficiency of the query.

1.17.2 Creating Indexes

Indexes are special data structures that store a small portion of the collection's data in an easy-to-traverse form. Creating an index can drastically improve the query performance. To create an index on the *name* field:

```
db.students.createIndex({name: 1})
```

This command creates an ascending index on the *name* field.

1.17.3 Listing All Indexes

To list all indexes that have been created on a collection:

```
db.students.getIndexes()
```

This will display all indexes, providing insights into the indexed fields and their types.

1.17.4 Dropping Indexes

To remove an index, you can use the `dropIndex()` method by specifying the index name. For example, to drop the index named "name_1":

```
db.students.dropIndexes("name_1")
```

This command removes the specified index, which could be necessary if the index is no longer needed or if it is negatively impacting write performance.

Note: Proper management of indexes is crucial as it affects both the performance of queries and the storage overhead of the database. Always ensure that indexes are created based on the query patterns to optimize performance effectively.

1.18 Creating Collections with Specific Options

MongoDB allows for detailed specification of options when creating collections, which can be useful for tailoring database behavior to specific requirements.

1.18.1 Creating a Capped Collection

Capped collections are fixed-size collections that maintain insertion order and are particularly useful for logging and caching where old data is automatically purged as new data comes in. To create a capped collection named *teachers* with a maximum size and document count:

```
db.createCollection("teachers", {
    capped: true,
    size: 1000000,
    max: 100
})
```

This command creates a capped collection named *teachers* with a size limit of 1,000,000 bytes and a maximum of 100 documents.

1.18.2 Disabling Auto-Indexing on the _id Field

By default, MongoDB automatically creates an index on the _id field for new collections. However, this behavior can be disabled:

```
db.createCollection("teachers", {
    capped: true,
    size: 1000000,
    max: 100
}, {
    autoIndexId: false
})
```

In this example, the collection *teachers* is created as a capped collection without an automatic index on the _id field. This might be useful in scenarios where custom indexing strategies are preferred or when the _id field is not needed for querying.

Note: When creating collections with specific options, it is important to understand the implications of each option, such as the impact on performance and data retrieval. Capped collections, for instance, are excellent for high-performance reads and writes but come with limitations on updates and deletions.

These detailed options allow MongoDB to be highly customizable and optimized for specific use cases, providing developers with powerful tools to manage data efficiently.