Complex Filter Object

Performing complex updates in MongoDB often involves using advanced operators like $set, $push, $addToSet, $inc, $unset, and array manipulation operators like $[<identifier>] for array filtering. Here's a breakdown of how to handle complex updates.

Example Use Case

Suppose you have a collection users where each document looks like this:

```
{
  "_id": ObjectId("1234567890abcdef"),
  "name": "John Doe",
  "age": 30,
  "address": {
    "city": "New York",
    "zip": "10001"
  },
  "hobbies": ["reading", "traveling"],
  "orders": [
    { "order_id": 1, "item": "book", "quantity": 1 },
    { "order_id": 2, "item": "pen", "quantity": 3 }
  ]
}
```

Complex Update Scenarios

1. Update Nested Fields

   To update a nested field (e.g., address.city):

   ```
   db.users.updateOne(
     { _id: ObjectId("1234567890abcdef") },
     { $set: { "address.city": "Los Angeles" } }
   );
   ```

2. Increment a Value

To increment the age field:

```
db.users.updateOne(
  { _id: ObjectId("1234567890abcdef") },
  { $inc: { age: 1 } }
);
```

3. Add an Element to an Array

To add a new hobby:

```
db.users.updateOne(
  { _id: ObjectId("1234567890abcdef") },
  { $push: { hobbies: "coding" } }
);
```

4. Add an Element to an Array if It Doesn't Exist

To ensure no duplicates:

```
db.users.updateOne(
  { _id: ObjectId("1234567890abcdef") },
  { $addToSet: { hobbies: "coding" } }
);
```

5. Update an Element in an Array

To update the quantity of the order with order_id: 2

```
db.users.updateOne(
  { _id: ObjectId("1234567890abcdef"), "orders.order_id": 2 },
  { $set: { "orders.$.quantity": 5 } }
);
```

6. Update Multiple Elements in an Array

To update all orders with item: "pen":

```
db.users.updateMany(
  { "orders.item": "pen" },
  { $set: { "orders.$[elem].quantity": 10 } },
  { arrayFilters: [{ "elem.item": "pen" }] }
);
```

7. Remove an Element from an Array :

To remove a specific hobby:

```
db.users.updateOne(
  { _id: ObjectId("1234567890abcdef") },
  { $pull: { hobbies: "traveling" } }
);
```

8. Unset a Field

To remove the zip field from address:

```
db.users.updateOne(
  { _id: ObjectId("1234567890abcdef") },
  { $unset: { "address.zip": "" } } );
```

9. Combine Operations

To update multiple fields and add/remove array elements in one operation:

```
db.users.updateOne(
  { _id: ObjectId("1234567890abcdef") },
  {
    $set: { "address.city": "San Francisco", age: 35 },
    $push: { hobbies: "photography" },
    $pull: { hobbies: "traveling" }
  }
);
```

10. Upsert

To update or insert a document if it doesn't exist:

```
db.users.updateOne(
  { name: "Jane Doe" },
  {
    $set: { age: 28, "address.city": "Boston" },
    $setOnInsert: { hobbies: ["painting"] }
  },
  { upsert: true }
);
```

11. Rename

If you want to **only rename** fields in MongoDB, you can use the $rename operator.

```
db.users.updateOne(
 { _id: ObjectId("1234567890abcdef") }, // Filter condition
 {
  $rename: {
   "age": "user_age",       // Rename "age" to "user_age"
   "address.zip": "address.postal_code" // Rename nested field "zip" to "postal_code"
  }
 }
);
```

Read Modifiers

Read modifiers in MongoDB allow you to customize how data is retrieved from the database, providing greater control over the output. They include **projection modifiers**, **cursor modifiers**, **query modifiers**, and **read preferences**. Below is a detailed breakdown of each category.

Projection Modifiers

Projection modifiers control the fields included or excluded in the query result. They are specified in the projection document, which is the second argument of the find() method.

1. Inclusion/Exclusion of Fields
    a. Specify fields to include (1) or exclude (0).
    b. _id is included by default unless explicitly excluded.
    c. You cannot mix inclusion and exclusion, except for _id.

```
db.users.find(
  { age: { $gte: 18 } },
  { name: 1, email: 1, _id: 0 } // Include `name` and `email`, exclude `_id`
);
```

2. $slice

The $slice modifier limits the number of elements returned from an array field.

```
{ arrayField: { $slice: <number> } }
```

- A positive number returns the first N elements.
- A negative number returns the last N elements.
- You can also use [<skip>, <limit>] to skip and limit array elements.

```
db.users.find(
  { name: "John" },
  { comments: { $slice: 3 } } // First 3 elements of the `comments` array
);


db.users.find(
  { name: "John" },
  { comments: { $slice: [-3, 3] } } // Last 3 elements
);
```

3. $elemMatch

The $elemMatch modifier returns the first matching element in an array that satisfies the query.

```
db.orders.find(
  { "items.item": "book" },
  { items: { $elemMatch: { item: "book" } } } // Only the matching element is returned
);
```

4. $meta

The `$meta` projection is used to include metadata, such as text search scores, in the results.

```
db.articles.find(
  { $text: { $search: "MongoDB" } },
  { score: { $meta: "textScore" } } // Include `score` field in results
);
```

Cursor Modifiers

Cursor modifiers allow fine-tuning of how query results are retrieved.

1. Limit
   Limits the number of documents returned.

```
db.users.find({}).limit(5); // Return only the first 5 documents
```

2. Skip
   Skips a specified number of documents in the result set.

```
db.users.find({}).skip(5); // Skip the first 5 documents
```

3. Sort
   Sorts the query results by one or more fields in ascending (1) or descending (-1) order.

```
db.users.find({}).sort({ age: -1, name: 1 }); // Sort by age descending and name ascending
```

4. Batchsize
   Specifies the number of documents to return in each batch.

```
db.users.find({}).batchSize(100); // Retrieve results in batches of 100
```

Query Modifiers

Query modifiers customize how MongoDB retrieves documents from the collection.

1. Hint
   Forces the query to use a specific index, useful for query optimization.

```
db.users.find({ age: { $gte: 30 } }).hint({ age: 1 });
```

2. min and max
   Defines bounds for index scanning, returning only documents within the specified range.

```
db.users.find({}).min({ age: 25 }).max({ age: 35 }); // Retrieve users aged 25-35
```

3. returnKey
   Returns only the indexed fields in the results, not the entire document.

```
db.users.find({ name: "John" }).returnKey();
```

Read Preferences

Read preferences determine which members of a replica set handle the query. This is crucial in distributed systems where read performance and consistency are important.

Modes

- **primary**: Default. Reads from the primary node.
- **secondary**: Reads from a secondary node.
- **nearest**: Reads from the nearest node based on latency.
- **primaryPreferred**: Reads from the primary if available; otherwise, from a secondary.
- **secondaryPreferred**: Reads from a secondary if available; otherwise, from the primary.

```
db.users.find({ status: "active" }).readPref("secondary");
```

Read Concern

Read concern controls the level of isolation and guarantees provided for read operations.

Levels:

- local: Default. Returns the most recent data available on the node.
- majority: Returns data acknowledged by the majority of replica set members.
- linearizable: Ensures data is the most recent globally.
- snapshot: Provides a point-in-time snapshot (used with transactions).

```
db.users.find({ status: "active" }).readConcern("majority");
```