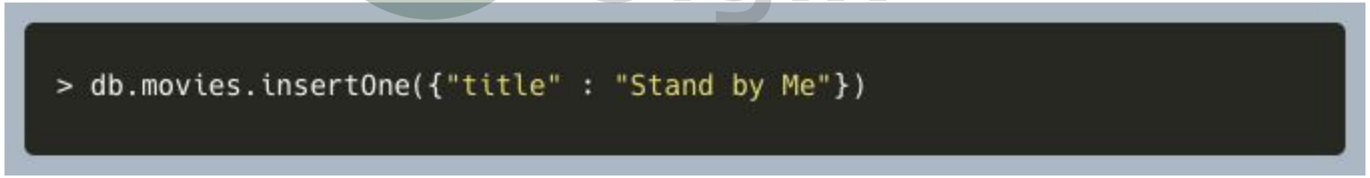
Full Stack Engineer

Module 3.2: MongoDB Basics

Inserting Documents (1)

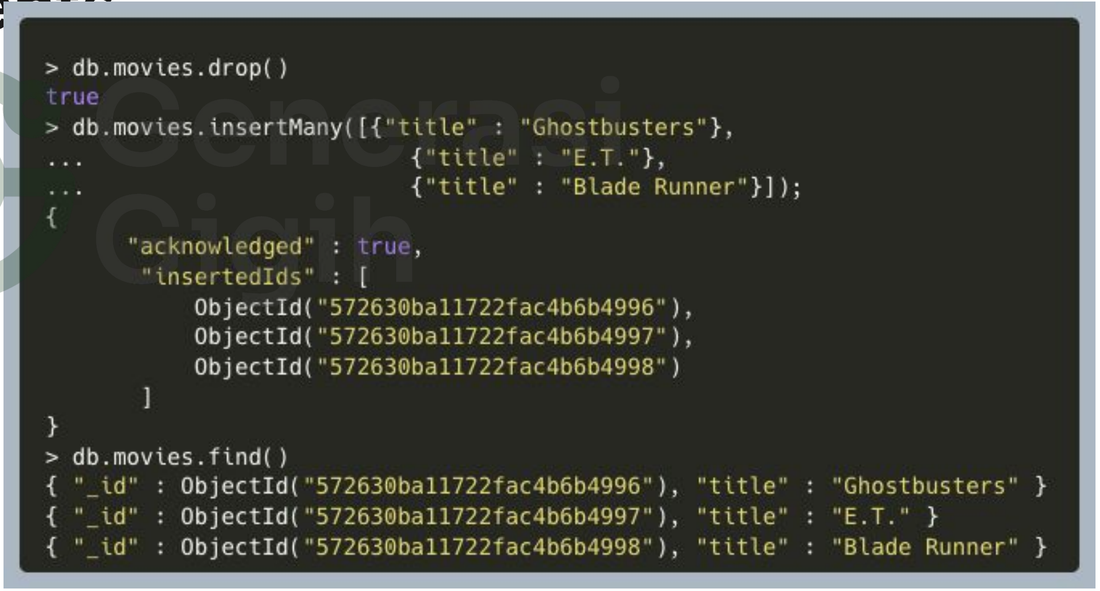
We have covered this a little bit in our Intro to MongoDB, but let's revisit. To insert a single document, use insertOne method:



```
> db.movies.insertOne({"title" : "Stand by Me"})
```

Inserting Documents (2)

If we want to insert multiple documents into a collection, we can use insertMany method.



```
> db.movies.drop()
true
> db.movies.insertMany([{"title" : "Ghostbusters"},
...                     {"title" : "E.T."},
...                     {"title" : "Blade Runner"}]);
{
      "acknowledged" : true,
      "insertedIds" : [
            ObjectId("572630ba11722fac4b6b4996"),
            ObjectId("572630ba11722fac4b6b4997"),
            ObjectId("572630ba11722fac4b6b4998")
      ]
}
> db.movies.find()
{ "_id" : ObjectId("572630ba11722fac4b6b4996"), "title" : "Ghostbusters" }
{ "_id" : ObjectId("572630ba11722fac4b6b4997"), "title" : "E.T." }
{ "_id" : ObjectId("572630ba11722fac4b6b4998"), "title" : "Blade Runner" }
```

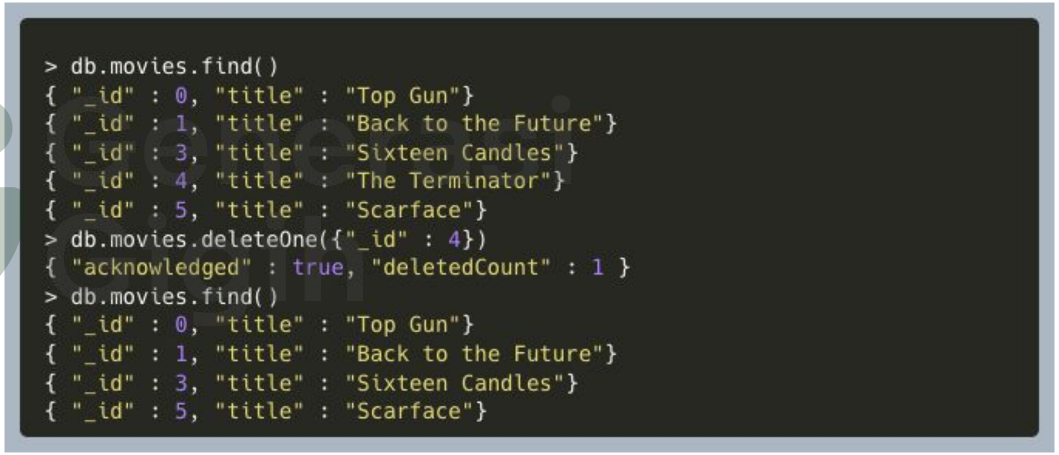
Inserting Documents (3)

MongoDB does minimal checks on data being inserted: it checks the document’s basic structure and adds an "_id" field if one does not exist.

One of the basic structure checks is size: all documents must be smaller than 16 MB. This is a somewhat arbitrary limit (and may be raised in the future); it is mostly intended to prevent bad schema design and ensure consistent performance. To see the Binary JSON (BSON) size, in bytes, of the document doc, run Object.bsonsize(doc) from the shell.

Removing Documents (1)

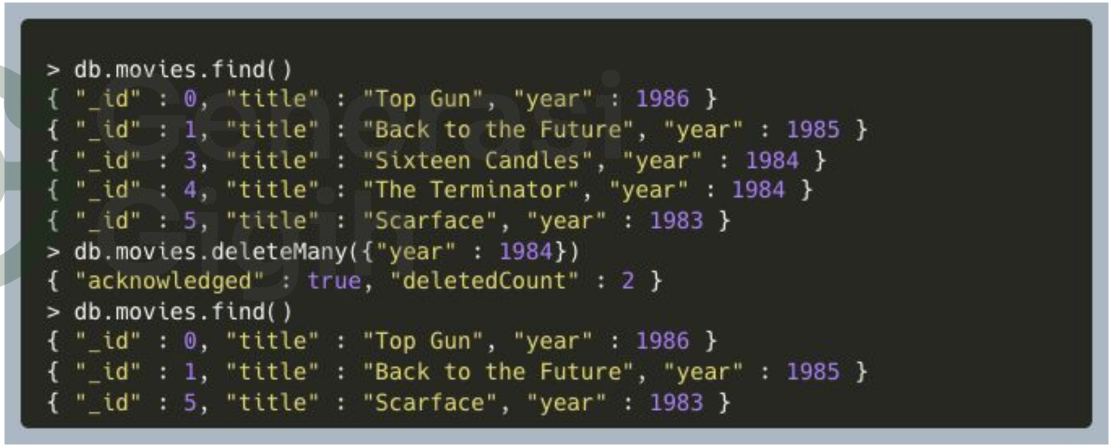
We also have touched this a little bit before, but to remove one document, we can use deleteOne.



```
> db.movies.find()
{ "_id" : 0, "title" : "Top Gun"}
{ "_id" : 1, "title" : "Back to the Future"}
{ "_id" : 3, "title" : "Sixteen Candles"}
{ "_id" : 4, "title" : "The Terminator"}
{ "_id" : 5, "title" : "Scarface"}
> db.movies.deleteOne({"_id" : 4})
{ "acknowledged" : true, "deletedCount" : 1 }
> db.movies.find()
{ "_id" : 0, "title" : "Top Gun"}
{ "_id" : 1, "title" : "Back to the Future"}
{ "_id" : 3, "title" : "Sixteen Candles"}
{ "_id" : 5, "title" : "Scarface"}
```

Removing Documents (2)

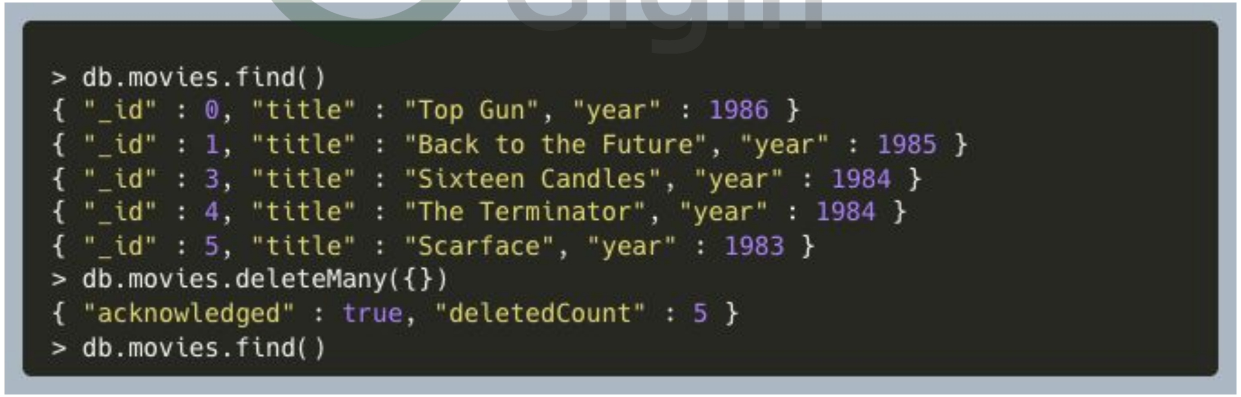
Just as with inserting documents, we can remove many documents at once using deleteMany method.



```
> db.movies.find( )
{ "_id" : 0, "title" : "Top Gun", "year" : 1986 }
{ "_id" : 1, "title" : "Back to the Future", "year" : 1985 }
{ "_id" : 3, "title" : "Sixteen Candles", "year" : 1984 }
{ "_id" : 4, "title" : "The Terminator", "year" : 1984 }
{ "_id" : 5, "title" : "Scarface", "year" : 1983 }
> db.movies.deleteMany({"year" : 1984})
{ "acknowledged" : true, "deletedCount" : 2 }
> db.movies.find( )
{ "_id" : 0, "title" : "Top Gun", "year" : 1986 }
{ "_id" : 1, "title" : "Back to the Future", "year" : 1985 }
{ "_id" : 5, "title" : "Scarface", "year" : 1983 }
```

Removing Documents (3)

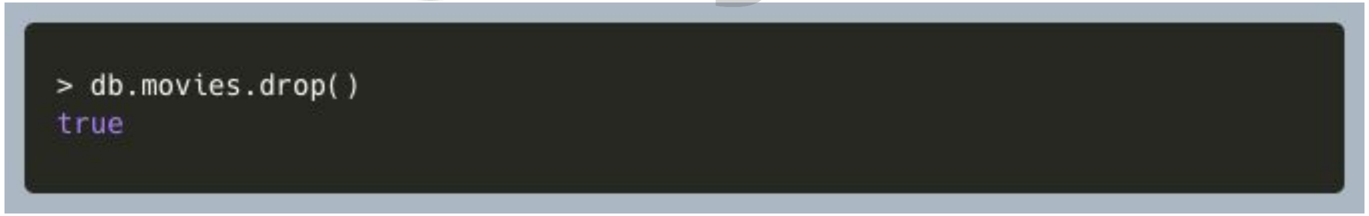
We can also use deleteMany to remove all documents in a collection:



```
> db.movies.find()
{ "_id" : 0, "title" : "Top Gun", "year" : 1986 }
{ "_id" : 1, "title" : "Back to the Future", "year" : 1985 }
{ "_id" : 3, "title" : "Sixteen Candles", "year" : 1984 }
{ "_id" : 4, "title" : "The Terminator", "year" : 1984 }
{ "_id" : 5, "title" : "Scarface", "year" : 1983 }
> db.movies.deleteMany({})
{ "acknowledged" : true, "deletedCount" : 5 }
> db.movies.find()
```

Removing Documents (4)

Another way to remove all documents in a collection is we just simply empty the collection with drop method.



```
> db.movies.drop()
true
```

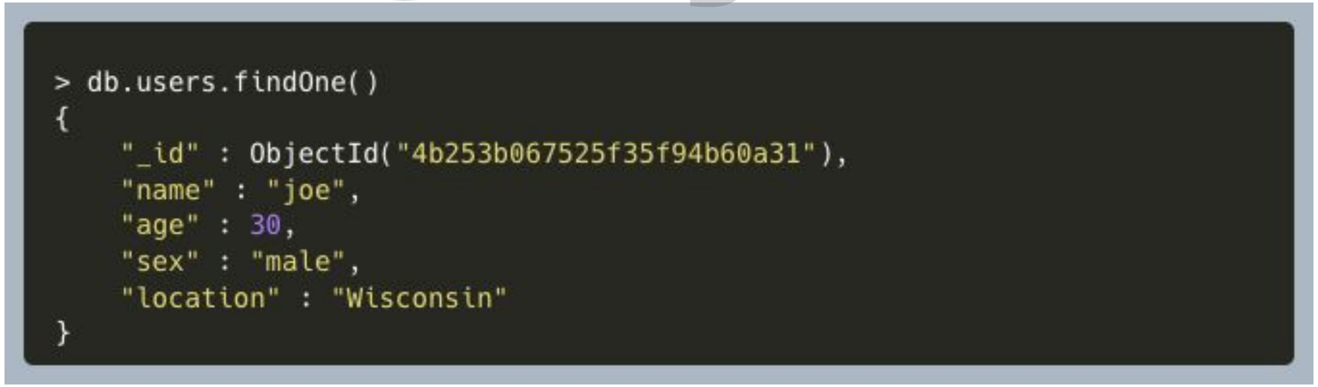
Updating Documents (1)

Once a document is stored in the database, it can be changed using one of several update methods: updateOne, updateMany, and replaceOne. updateOne and updateMany each take a filter document as their first parameter and a modifier document, which describes changes to make, as the second parameter.

Updating a document is atomic: if two updates happen at the same time, whichever one reaches the server first will be applied, and then the next one will be applied. Thus, conflicting updates can safely be sent in rapid-fire succession without any documents being corrupted: the last update will “win.”

Updating Documents - Set (1)

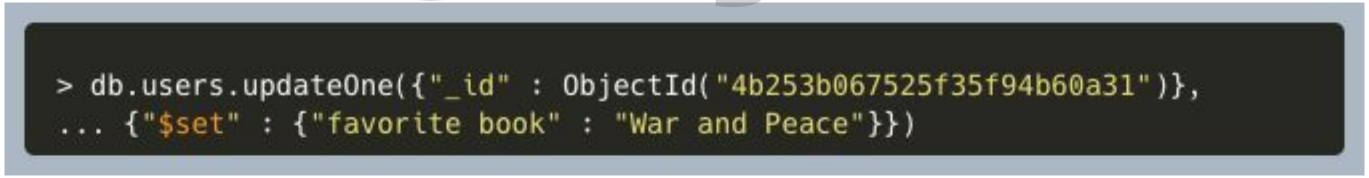
"$set" sets the value of a field. If the field does not yet exist, it will be created. This can be handy for updating schemas or adding user-defined keys. For example, suppose you have a simple user profile stored as a document that looks something like the following:



```
> db.users.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "name" : "joe",
    "age" : 30,
    "sex" : "male",
    "location" : "Wisconsin"
}
```

Updating Documents - Set (2)

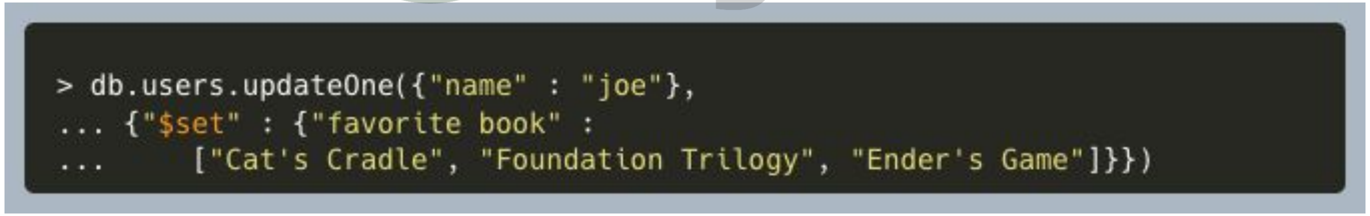
This is a pretty bare-bones user profile. If the user wanted to store his favorite book in his profile, he could add it using "$set":



```
> db.users.updateOne({"_id" : ObjectId("4b253b067525f35f94b60a31")},
... {"$set" : {"favorite book" : "War and Peace"}})
```

Updating Documents - Set (3)

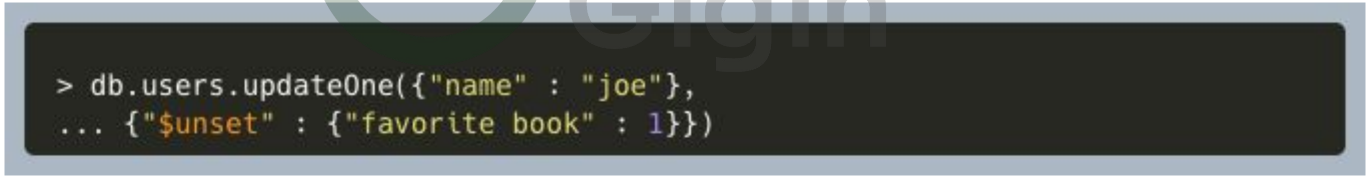
"$set" can even change the type of the key it modifies. For instance, if our fickle user decides that he actually likes quite a few books, he can change the value of the "favorite book" key into an array:



```
> db.users.updateOne({"name" : "joe"},
... {"$set" : {"favorite book" :
...     ["Cat's Cradle", "Foundation Trilogy", "Ender's Game"]}})
```

Updating Documents - Unset

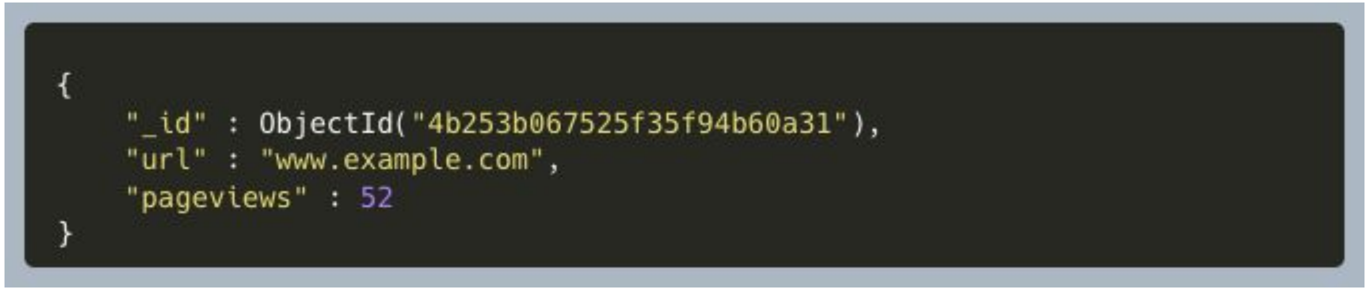
"$unset" can remove a key from a document:



```
> db.users.updateOne({"name" : "joe"},
... {"$unset" : {"favorite book" : 1}})
```

Updating Documents - Increment (1)

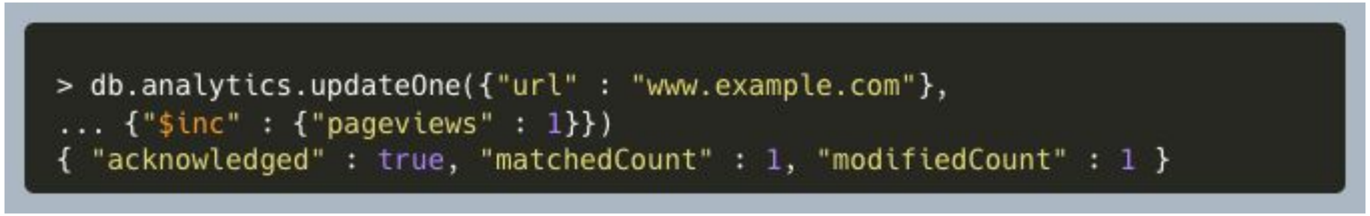
Suppose we're keeping website analytics in a collection and want to increment a counter each time someone visits a page. We can use update operators to do this increment atomically. Each URL and its number of page views is stored in a document that looks like this:



```
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "url" : "www.example.com",
    "pageviews" : 52
}
```

Updating Documents - Increment (2)

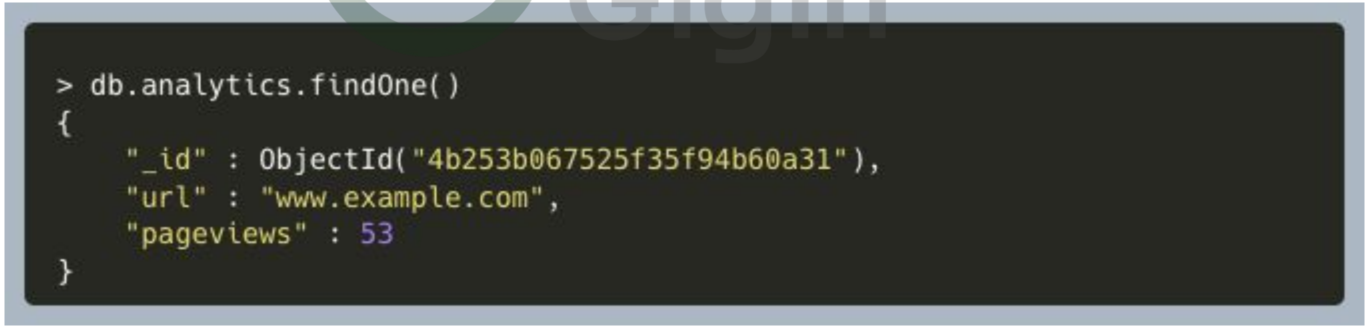
Every time someone visits a page, we can find the page by its URL and use the "$inc" modifier to increment the value of the "pageviews" key:



```
> db.analytics.updateOne({"url" : "www.example.com"},
... {"$inc" : {"pageviews" : 1}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

Updating Documents - Increment (3)

Now, if we do a findOne, we see that "pageviews" has increased by one:



```
> db.analytics.findOne()
{
    "_id" : ObjectId("4b253b067525f35f94b60a31"),
    "url" : "www.example.com",
    "pageviews" : 53
}
```

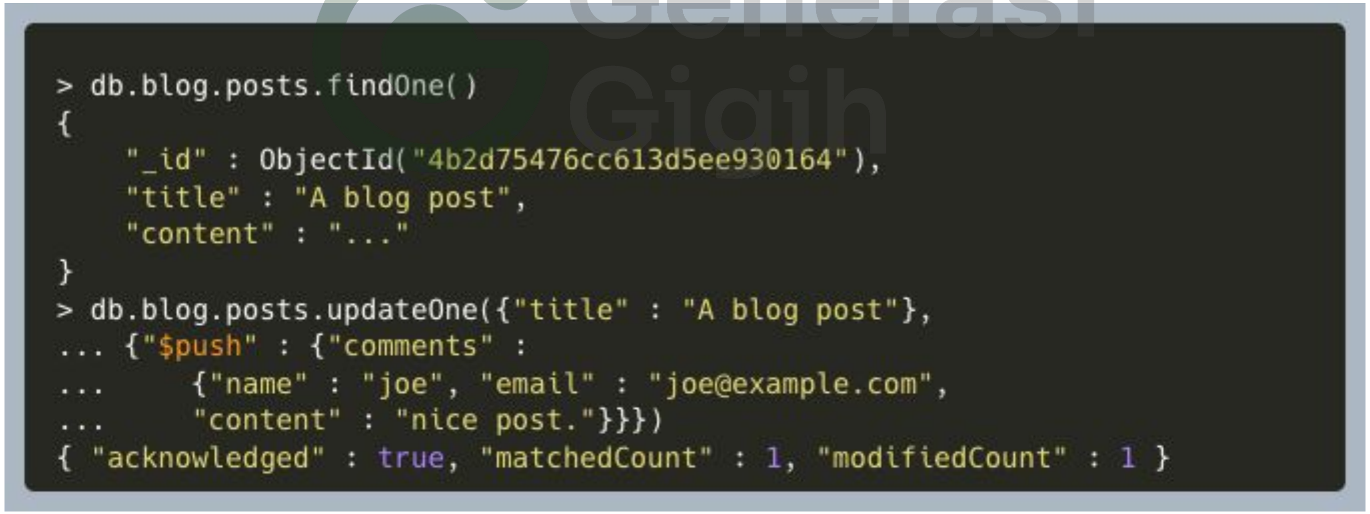
Array Operators

An extensive class of update operators exists for manipulating arrays. Arrays are common and powerful data structures: not only are they lists that can be referenced by index, but they can also double as sets.

Array Operators - Push (1)

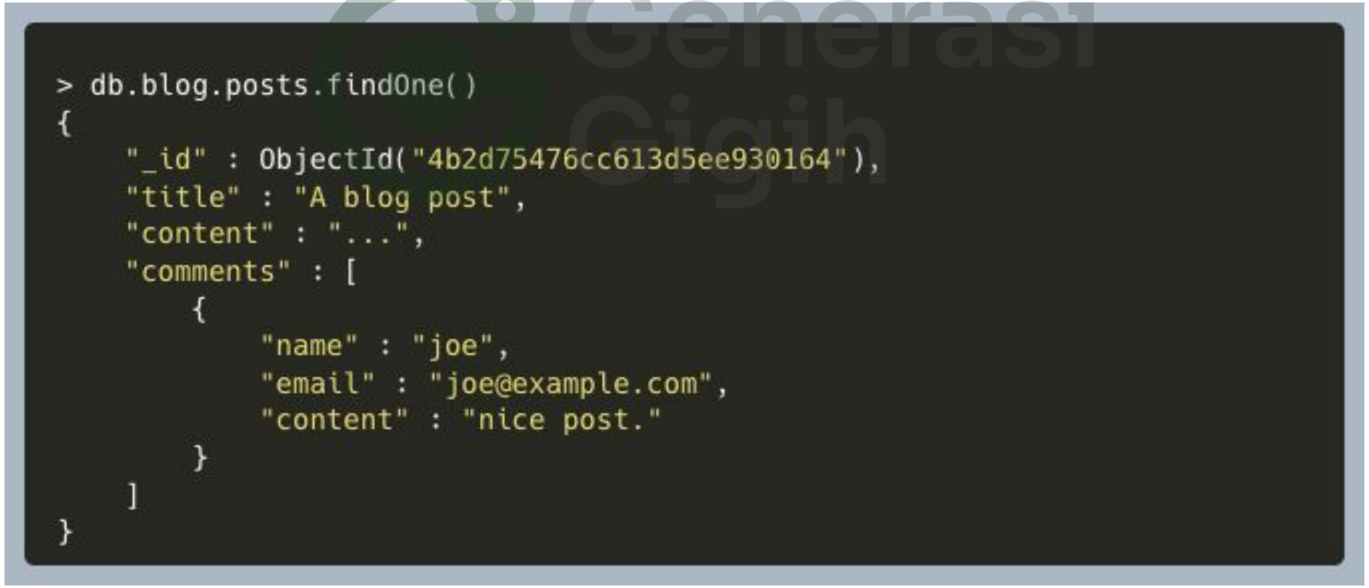
"$push" adds elements to the end of an array if the array exists and creates a new array if it does not. For example, suppose that we are storing blog posts and want to add a "comments" key containing an array. We can push a comment onto the nonexistent "comments" array, which will create the array and add the comment:

Array Operators - Push (2)



```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "..."
}
> db.blog.posts.updateOne({"title" : "A blog post"},
... {"$push" : {"comments" :
...     {"name" : "joe", "email" : "joe@example.com",
...     "content" : "nice post."}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

Array Operators - Push (3)



```
> db.blog.posts.findOne()
{
    "_id" : ObjectId("4b2d75476cc613d5ee930164"),
    "title" : "A blog post",
    "content" : "...",
    "comments" : [
        {
            "name" : "joe",
            "email" : "joe@example.com",
            "content" : "nice post."
        }
    ]
}
```

Indexing

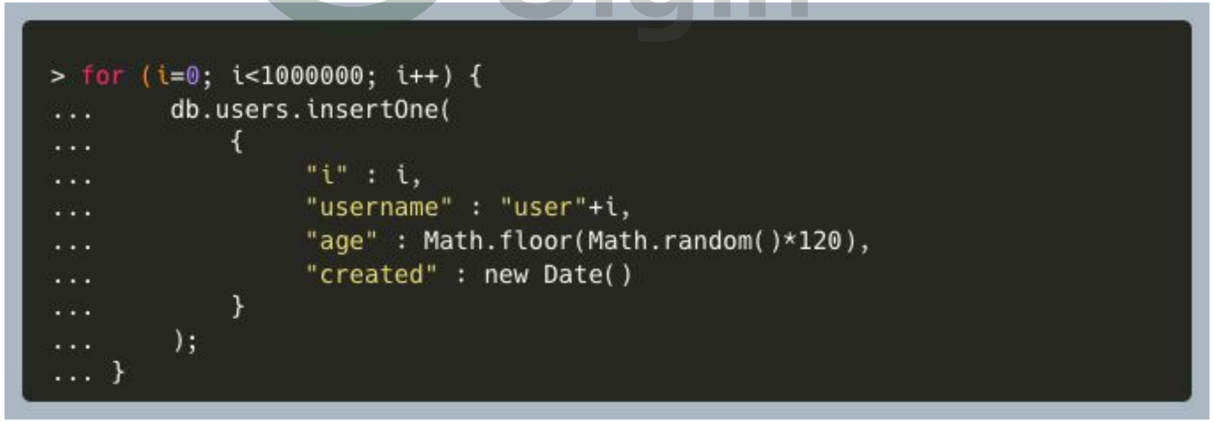
Index (1)

A database index is similar to a book's index. Instead of looking through the whole book, the database takes a shortcut and just looks at an ordered list with references to the content. This allows MongoDB to query orders of magnitude faster.

A query that does not use an index is called a collection scan, which means that the server has to "look through the whole book" to find a query's results. This process is basically what you'd do if you were looking for information in a book without an index: you'd start at page 1 and read through the whole thing. In general, you want to avoid making the server do collection scans because the process is very slow for large collections.

Index (2)

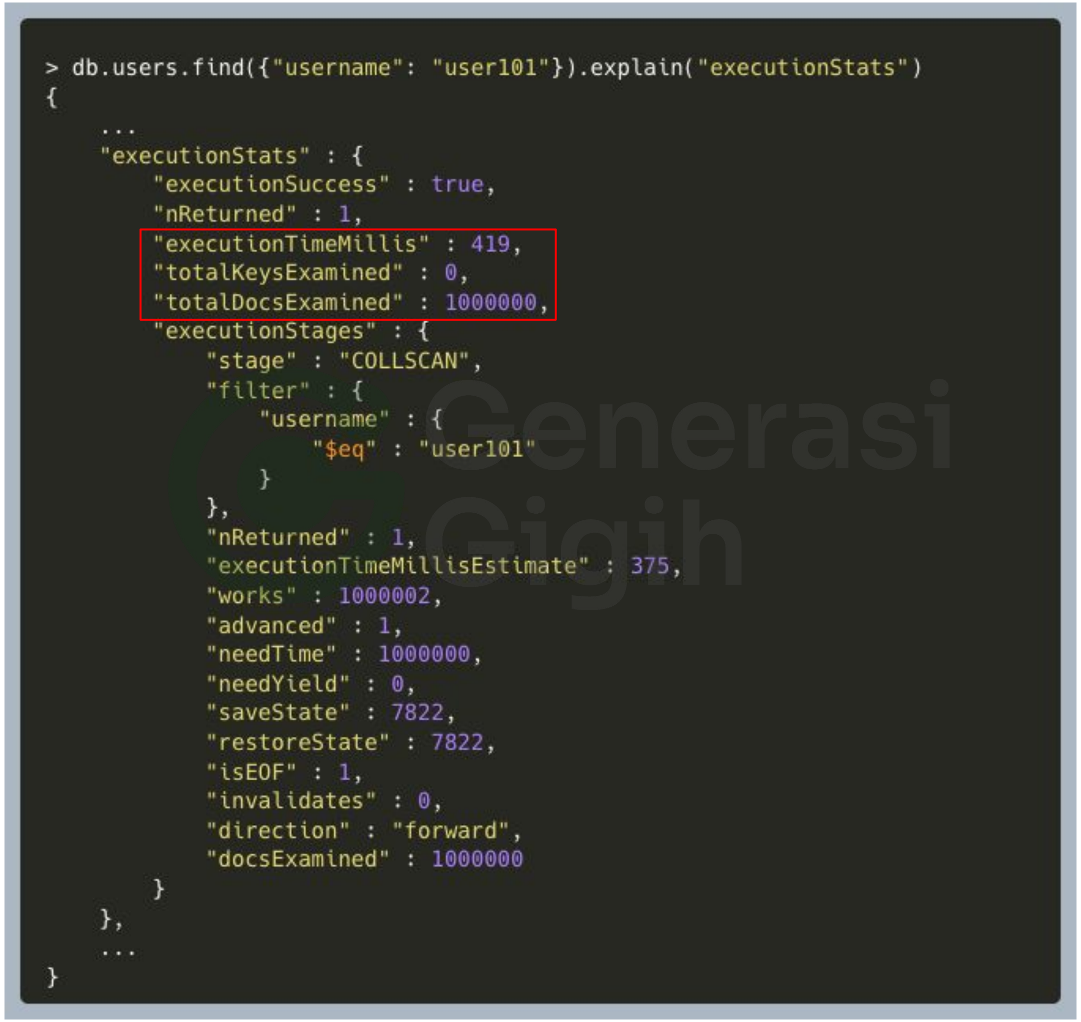
Let’s look at an example. To get started, we’ll create a collection with 1 million documents in it (or 10 million or 100 million, if you have the patience):



```
> for (i=0; i<1000000; i++) {
...     db.users.insertOne(
...         {
...              "i" : i,
...              "username" : "user"+i,
...              "age" : Math.floor(Math.random()*120),
...              "created" : new Date()
...         }
...     );
... }
```

Index (3)

If we do a query on this collection, we can use the explain command to see what MongoDB is doing when it executes the query. The preferred way to use the explain command is through the cursor helper method that wraps this command. The explain cursor method provides information on the execution of a variety of CRUD operations. This method may be run in several verbosity modes. We'll look at executionStats mode since this helps us understand the effect of using an index to satisfy queries. Try querying on a specific username to see an example:



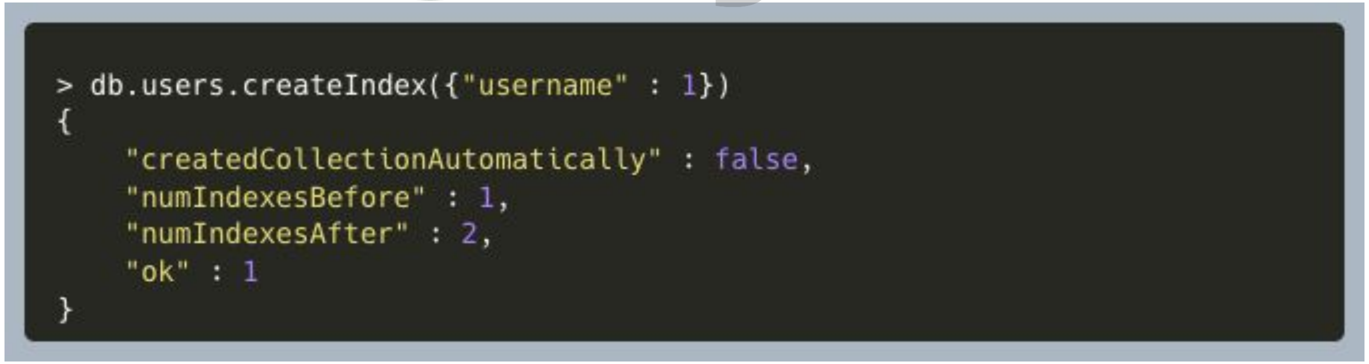
```
> db.users.find({"username": "user101"}).explain("executionStats")
{
    ...
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 1,
        "executionTimeMillis" : 419,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 1000000,
        "executionStages" : {
            "stage" : "COLLSCAN",
            "filter" : {
                "username" : {
                    "$eq" : "user101"
                }
            },
            "nReturned" : 1,
            "executionTimeMillisEstimate" : 375,
            "works" : 1000002,
            "advanced" : 1,
            "needTime" : 1000000,
            "needYield" : 0,
            "saveState" : 7822,
            "restoreState" : 7822,
            "isEOF" : 1,
            "invalidates" : 0,
            "direction" : "forward",
            "docsExamined" : 1000000
        }
    },
    ...
}
```

Index (4)

To enable MongoDB to respond to queries efficiently, all query patterns in your application should be supported by an index. By query patterns, we simply mean the different types of questions your application asks of the database. In this example, we queried the users collection by username. That is an example of a specific query pattern. In many applications, a single index will support several query patterns. We will discuss tailoring indexes to query patterns in a later section.

Creating an Index

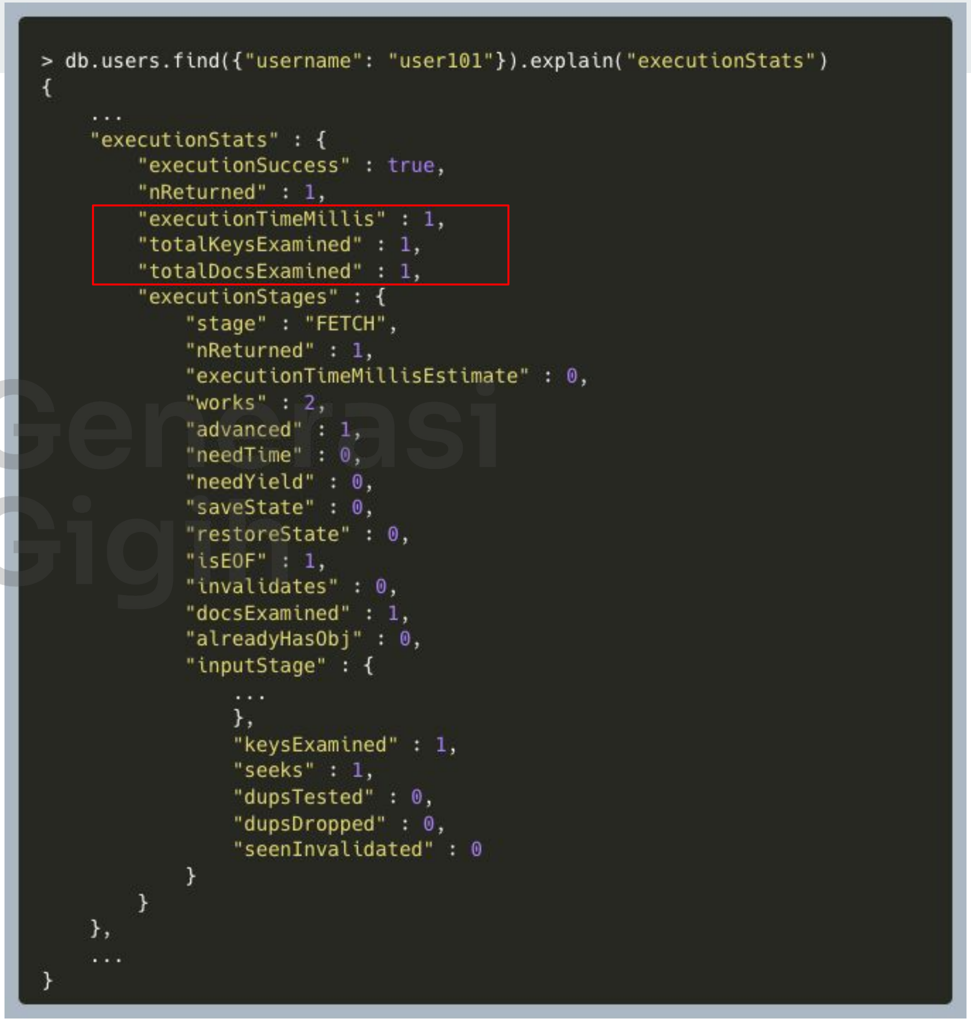
Now let’s try creating an index on the "username" field. To create an index, we’ll use the createIndex collection method:



```
> db.users.createIndex({"username" : 1})
{
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 1,
    "numIndexesAfter" : 2,
    "ok" : 1
}
```

Impact on Execution Stat (1)

Once the index build is complete, try repeating the original query:



```
> db.users.find({"username": "user101"}).explain("executionStats")
{
    ...
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 1,
        "executionTimeMillis" : 1,
        "totalKeysExamined" : 1,
        "totalDocsExamined" : 1,
        "executionStages" : {
            "stage" : "FETCH",
            "nReturned" : 1,
            "executionTimeMillisEstimate" : 0,
            "works" : 2,
            "advanced" : 1,
            "needTime" : 0,
            "needYield" : 0,
            "saveState" : 0,
            "restoreState" : 0,
            "isEOF" : 1,
            "invalidates" : 0,
            "docsExamined" : 1,
            "alreadyHasObj" : 0,
            "inputStage" : {
                ...
                },
                "keysExamined" : 1,
                "seeks" : 1,
                "dupsTested" : 0,
                "dupsDropped" : 0,
                "seenInvalidated" : 0
            }
        }
    },
    ...
}
```

Impact on Execution Stat (2)

An index can make a dramatic difference in query times.

However, indexes have their price: write operations (inserts, updates, and deletes) that modify an indexed field will take longer. This is because in addition to updating the document, MongoDB has to update indexes when your data changes. Typically, the tradeoff is worth it.

The tricky part becomes figuring out which fields to index.

Which Fields to Index?

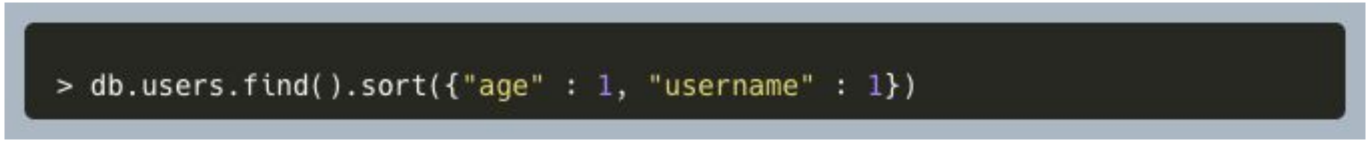
To choose which fields to create indexes for, look through your frequent queries and queries that need to be fast and try to find a common set of keys from those.

For instance, in the preceding example, we were querying on "username". If that were a particularly common query or were becoming a bottleneck, indexing "username" would be a good choice.

However, if this were an unusual query or one that’s only done by administrators who don’t care how long it takes, it would not be a good choice for indexing.

Compound Indexes (1)

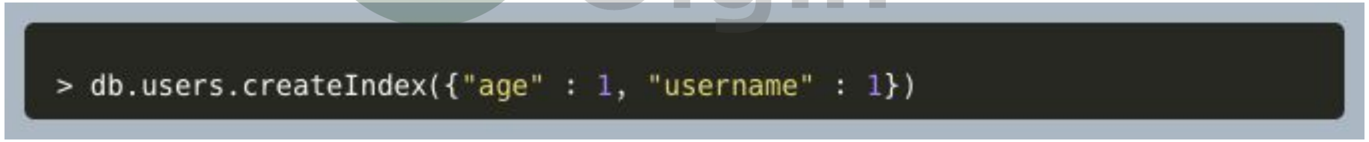
The purpose of an index is to make your queries as efficient as possible. For many query patterns it is necessary to build indexes based on two or more keys. For example, an index keeps all of its values in a sorted order, so it makes sorting documents by the indexed key much faster. However, an index can only help with sorting if it is a prefix of the sort. For example, the index on "username" wouldn't help much for this sort:



```
> db.users.find().sort({"age" : 1, "username" : 1})
```

Compound Indexes (2)

This sorts by "age" and then "username", so a strict sorting by "username" isn't terribly helpful. To optimize this sort, you could make an index on "age" and "username":



```
> db.users.createIndex({"age" : 1, "username" : 1})
```

This is called a compound index and is useful if your query has multiple sort directions or multiple keys in the criteria. A compound index is an index on more than one field.

Aggregation

The Aggregation Framework

The aggregation framework is a set of analytics tools within MongoDB that allow you to do analytics on documents in one or more collections.

The aggregation framework is based on the concept of a pipeline. With an aggregation pipeline we take input from a MongoDB collection and pass the documents from that collection through one or more stages, each of which performs a different operation on its inputs.

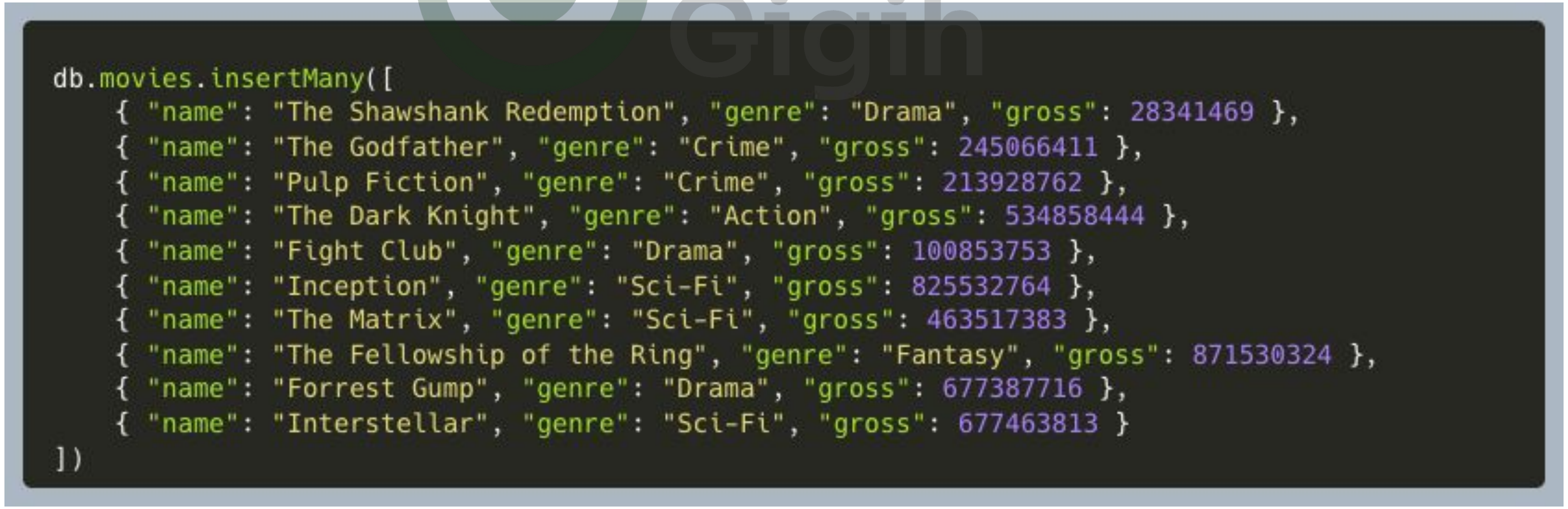
The Aggregation Pipeline

Stages can perform operations on data such as:

- filtering: narrow down the list of documents through a set of criteria
- sorting: reorder documents based on a chosen field
- transforming: change the structure of documents
- grouping: process multiple documents together to form a summarized result

Preparation

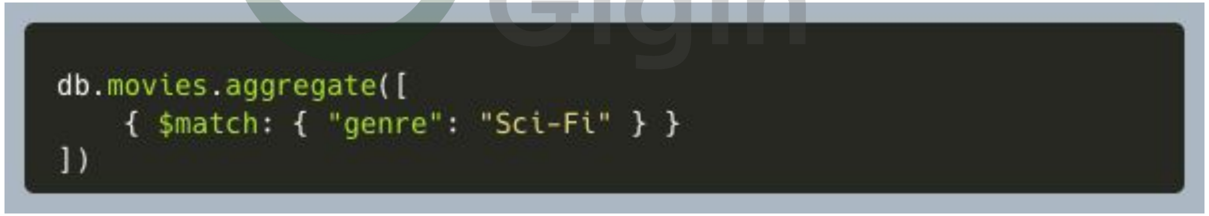
Create a new collection and populate it with the following data:



```
db.movies.insertMany([
    { "name": "The Shawshank Redemption", "genre": "Drama", "gross": 28341469 },
    { "name": "The Godfather", "genre": "Crime", "gross": 245066411 },
    { "name": "Pulp Fiction", "genre": "Crime", "gross": 213928762 },
    { "name": "The Dark Knight", "genre": "Action", "gross": 534858444 },
    { "name": "Fight Club", "genre": "Drama", "gross": 100853753 },
    { "name": "Inception", "genre": "Sci-Fi", "gross": 825532764 },
    { "name": "The Matrix", "genre": "Sci-Fi", "gross": 463517383 },
    { "name": "The Fellowship of the Ring", "genre": "Fantasy", "gross": 871530324 },
    { "name": "Forrest Gump", "genre": "Drama", "gross": 677387716 },
    { "name": "Interstellar", "genre": "Sci-Fi", "gross": 677463813 }
])
```

Match (1)

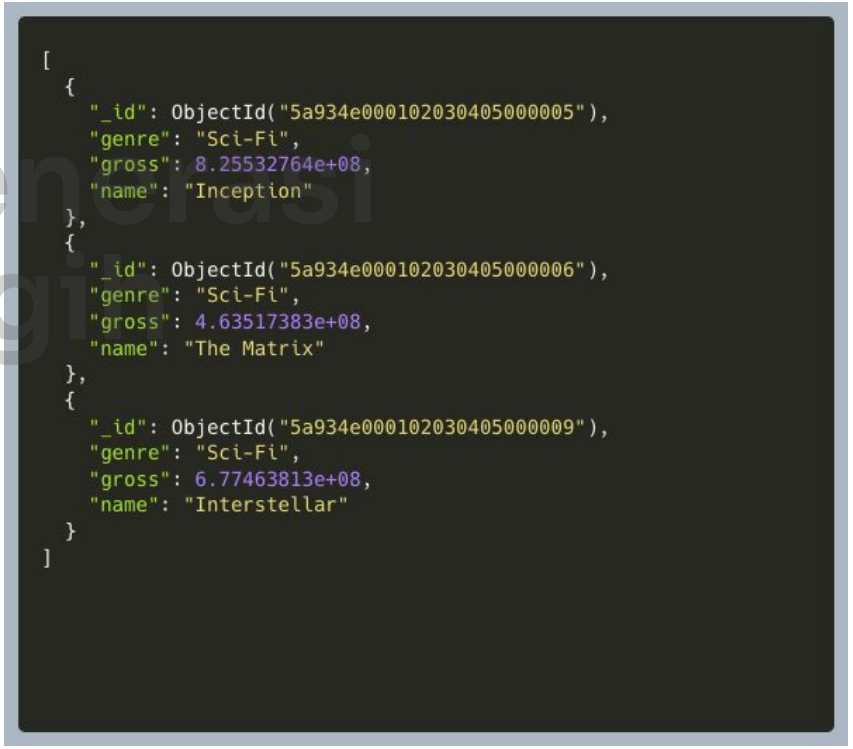
Run the following method:



```
db.movies.aggregate([
    { $match: { "genre": "Sci-Fi" } }
])
```

Match (2)

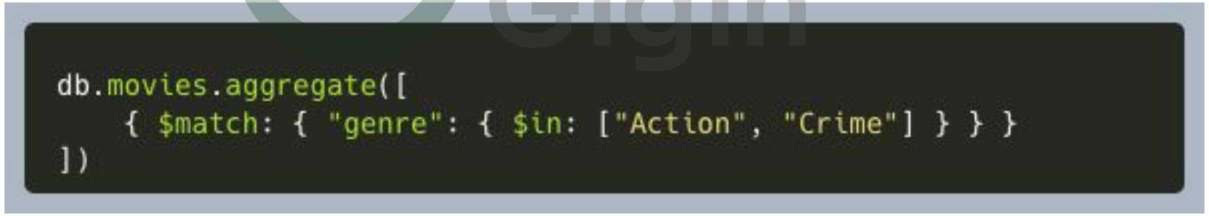
The result should look like:



```
[
  {
    "_id": ObjectId("5a934e000102030405000005"),
    "genre": "Sci-Fi",
    "gross": 8.25532764e+08,
    "name": "Inception"
  },
  {
    "_id": ObjectId("5a934e000102030405000006"),
    "genre": "Sci-Fi",
    "gross": 4.63517383e+08,
    "name": "The Matrix"
  },
  {
    "_id": ObjectId("5a934e000102030405000009"),
    "genre": "Sci-Fi",
    "gross": 6.77463813e+08,
    "name": "Interstellar"
  }
]
```

Match (3)

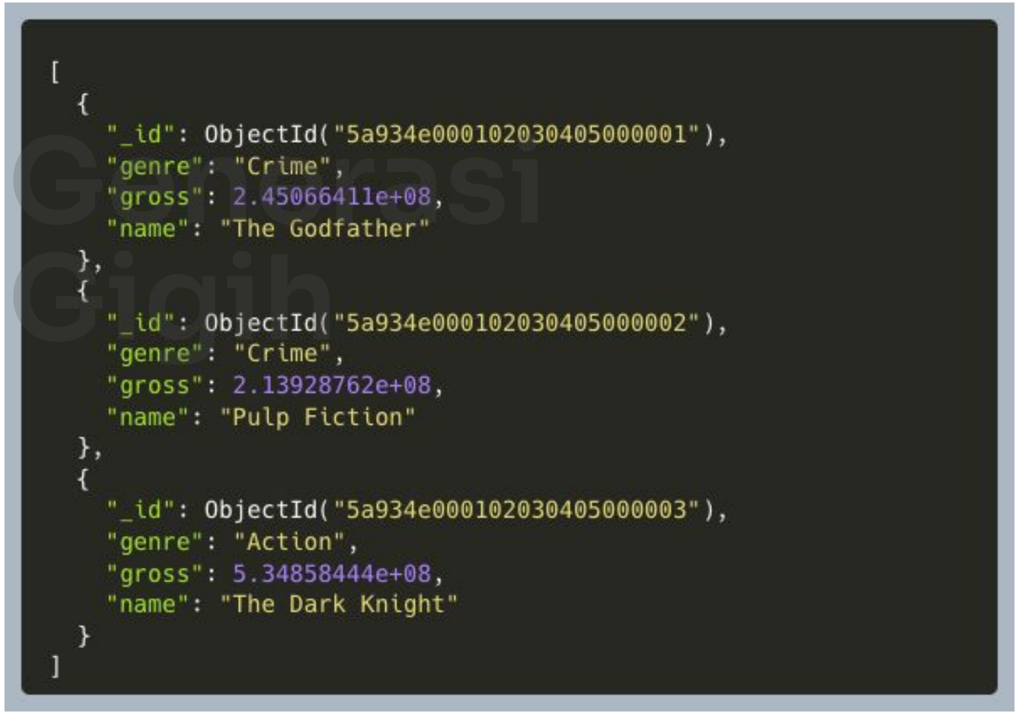
We can match based on several different values of a field:



```
db.movies.aggregate([
    { $match: { "genre": { $in: ["Action", "Crime"] } } }
])
```

Match (4)

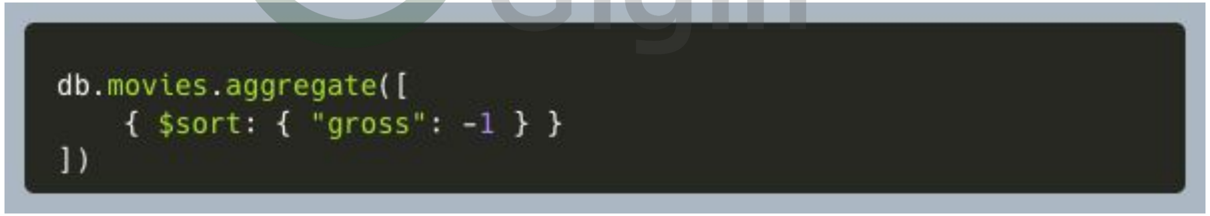
The result should look like:



```
[
  {
    "_id": ObjectId("5a934e000102030405000001"),
    "genre": "Crime",
    "gross": 2.45066411e+08,
    "name": "The Godfather"
  },
  {
    "_id": ObjectId("5a934e000102030405000002"),
    "genre": "Crime",
    "gross": 2.13928762e+08,
    "name": "Pulp Fiction"
  },
  {
    "_id": ObjectId("5a934e000102030405000003"),
    "genre": "Action",
    "gross": 5.34858444e+08,
    "name": "The Dark Knight"
  }
]
```

Sort (1)

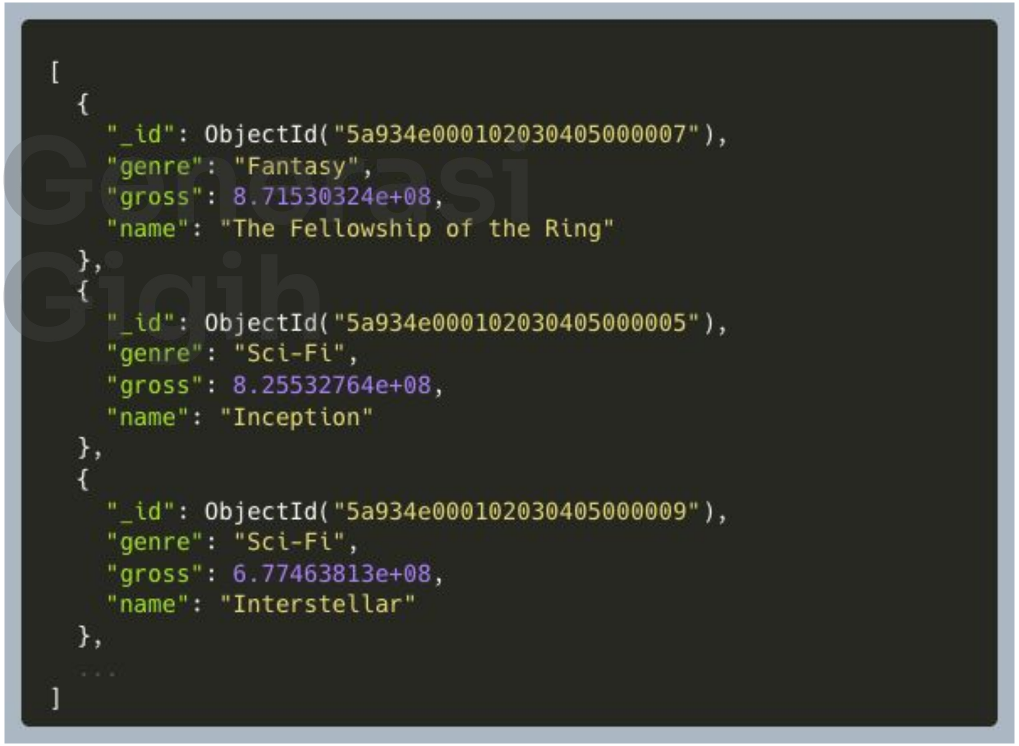
Run the following method:



```
db.movies.aggregate([
    { $sort: { "gross": -1 } }
])
```

Sort (2)

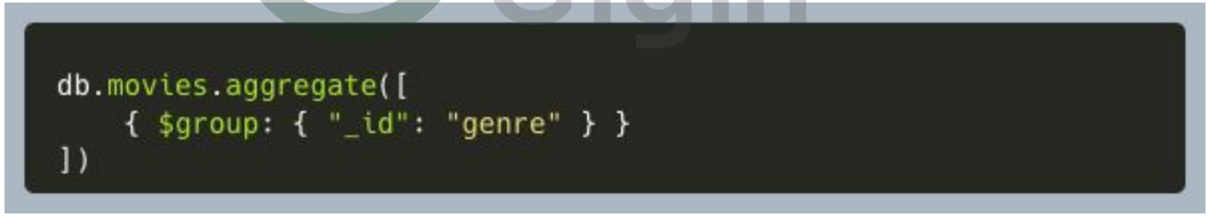
The result should look like:



```
[
  {
    "_id": ObjectId("5a934e000102030405000007"),
    "genre": "Fantasy",
    "gross": 8.71530324e+08,
    "name": "The Fellowship of the Ring"
  },
  {
    "_id": ObjectId("5a934e000102030405000005"),
    "genre": "Sci-Fi",
    "gross": 8.25532764e+08,
    "name": "Inception"
  },
  {
    "_id": ObjectId("5a934e000102030405000009"),
    "genre": "Sci-Fi",
    "gross": 6.77463813e+08,
    "name": "Interstellar"
  },
  ...
]
```

Group (1)

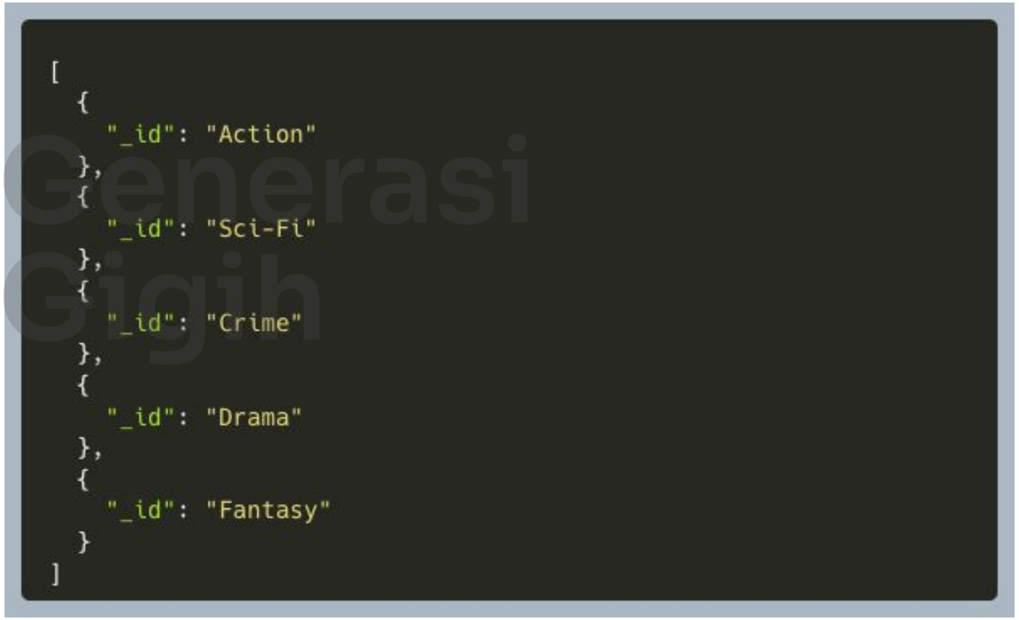
Run the following method:



```
db.movies.aggregate([
    { $group: { "_id": "genre" } }
])
```

Group (2)

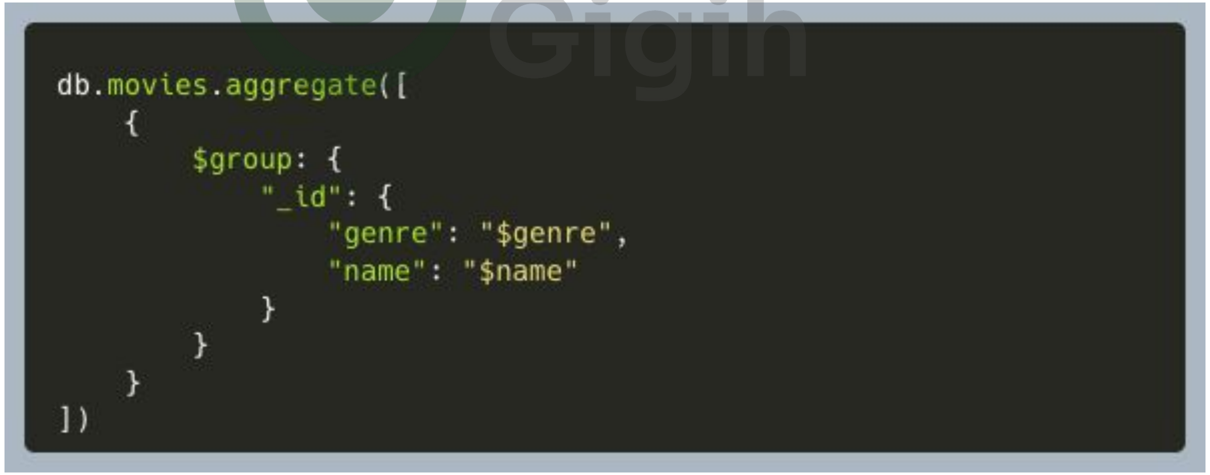
The result should look like:



```
[
  {
    "_id": "Action"
  },
  {
    "_id": "Sci-Fi"
  },
  {
    "_id": "Crime"
  },
  {
    "_id": "Drama"
  },
  {
    "_id": "Fantasy"
  }
]
```

Group (3)

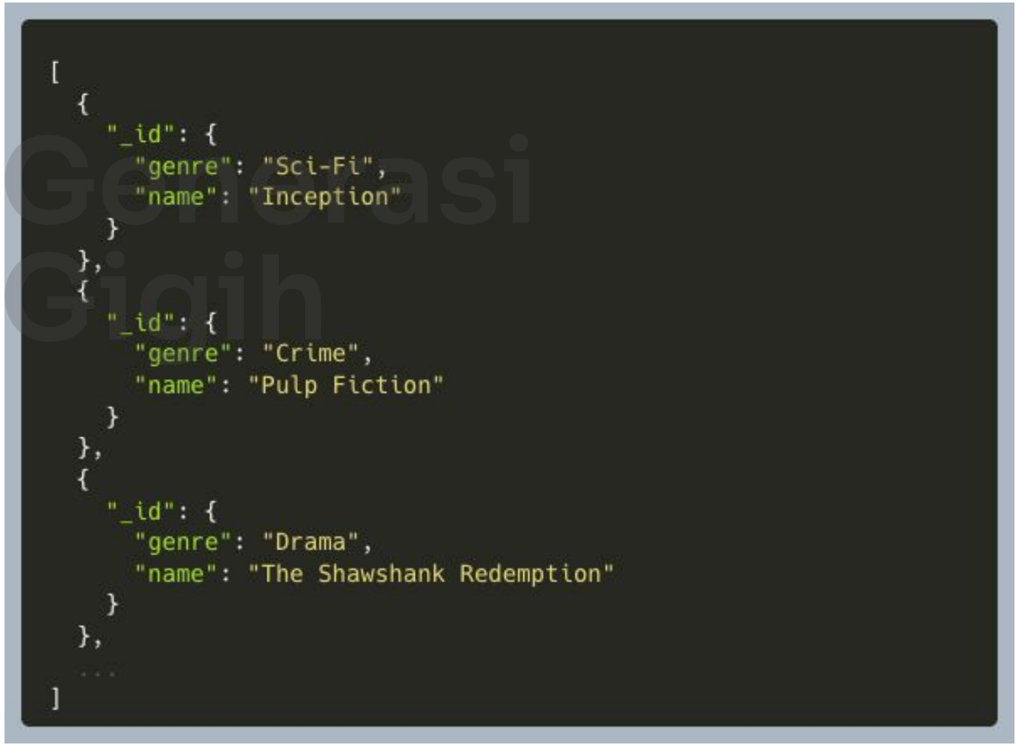
Run the following method:



```
db.movies.aggregate([
    {
        $group: {
            "_id": {
                "genre": "$genre",
                "name": "$name"
            }
        }
    }
])
```

Group (4)

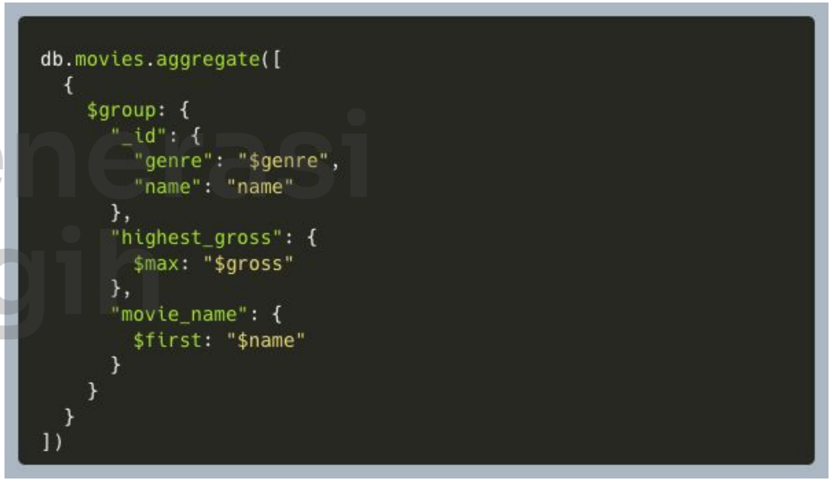
The result should look like:



```
[
  {
    "_id": {
      "genre": "Sci-Fi",
      "name": "Inception"
    }
  },
  {
    "_id": {
      "genre": "Crime",
      "name": "Pulp Fiction"
    }
  },
  {
    "_id": {
      "genre": "Drama",
      "name": "The Shawshank Redemption"
    }
  },
  ...
]
```

Group (5)

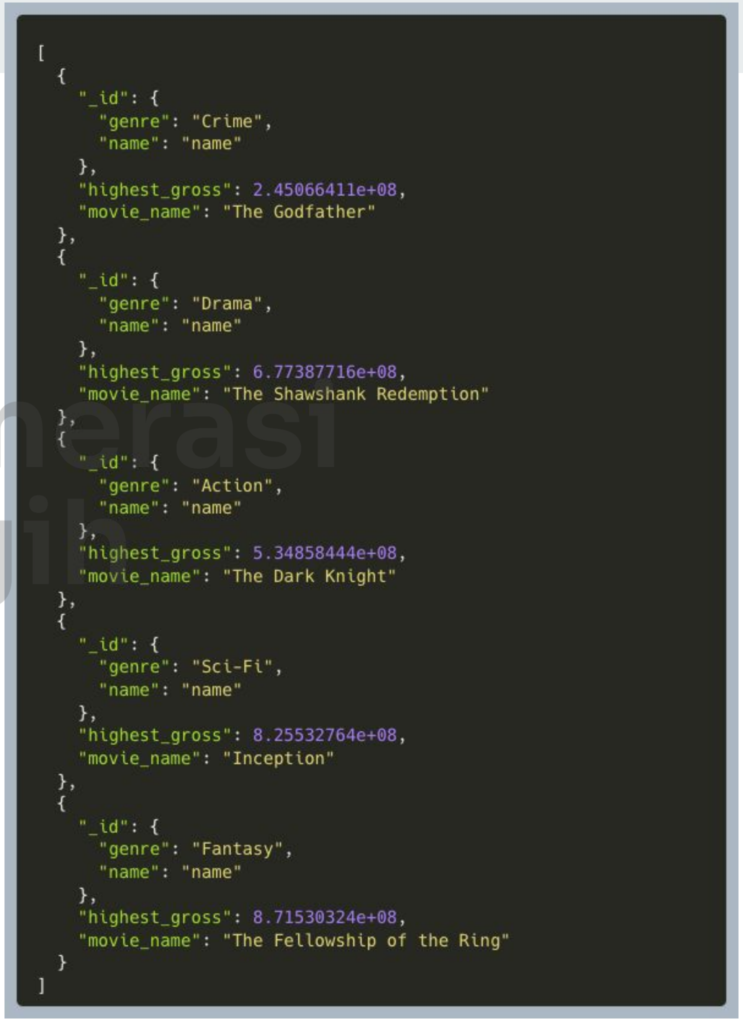
Run the following method:



```
db.movies.aggregate([
  {
    $group: {
      "_id": {
        "genre": "$genre",
        "name": "name"
      },
      "highest_gross": {
        $max: "$gross"
      },
      "movie_name": {
        $first: "$name"
      }
    }
  }
])
```

Group (6)

The result should look like:



```
[
  {
    "_id": {
      "genre": "Crime",
      "name": "name"
    },
    "highest_gross": 2.45066411e+08,
    "movie_name": "The Godfather"
  },
  {
    "_id": {
      "genre": "Drama",
      "name": "name"
    },
    "highest_gross": 6.77387716e+08,
    "movie_name": "The Shawshank Redemption"
  },
  {
    "_id": {
      "genre": "Action",
      "name": "name"
    },
    "highest_gross": 5.34858444e+08,
    "movie_name": "The Dark Knight"
  },
  {
    "_id": {
      "genre": "Sci-Fi",
      "name": "name"
    },
    "highest_gross": 8.25532764e+08,
    "movie_name": "Inception"
  },
  {
    "_id": {
      "genre": "Fantasy",
      "name": "name"
    },
    "highest_gross": 8.71530324e+08,
    "movie_name": "The Fellowship of the Ring"
  }
]
```

Questions?