Let’s take a look to advanced features in MongoDB.

- **Task 1. Project fields to return from a query**
- **Task 2. Limiting records**
- **Task 3. Sorting records**
- **Task 4. Query for missing fields and null**
- **Task 5. Aggregation operations**
- **Task 6. Working with Cursors**
- **Task 7. Bulk write operations**

Task 1. Project fields to return from a query

Projection means selecting only the necessary data rather than selecting whole of the data of a document. By default, queries in MongoDB return all fields in matching documents. To limit the amount of data that MongoDB sends to applications, you can include a projection document to specify or restrict fields to return.

1. Create a **store** database and collection
2. Add the following items

```
  { item: "postcard", status: "A", quantity: 50, size: { h: 10,
w: 15.25, uom: "cm" }, instock: [ { warehouse: "B", qty: 15 }, {
warehouse: "C", qty: 35 } ] }

  { item: "journal", status: "A", quantity: 5, size: { h: 14, w:
21, uom: "cm" }, instock: [ { warehouse: "A", qty: 5 } ] }

  { item: "paper", status: "D", quantity: 60, size: { h: 8.5, w:
11, uom: "in" }, instock: [ { warehouse: "A", qty: 60 } ] }

  { item: "notebook", status: "A", quantity: 5, size: { h: 8.5,
w: 11, uom: "in" }, instock: [ { warehouse: "C", qty: 5 } ] }

  { item: "planner", status: "D", quantity: 40, size: { h:
22.85, w: 30, uom: "cm" }, instock: [ { warehouse: "A", qty: 40
} ] }
```

The following example returns all fields from all documents in the store collection where the status equals "A":

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

```
store> db.store.find( { status: "A" } )
[
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    status: 'A',
    quantity: 50,
    size: { h: 10, w: 15.25, uom: 'cm' },
    instock: [ { warehouse: 'B', qty: 15 }, { warehouse: 'C', qty: 35 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'journal',
    status: 'A',
    quantity: 5,
    size: { h: 14, w: 21, uom: 'cm' },
    instock: [ { warehouse: 'A', qty: 5 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a9"),
    item: 'notebook',
    status: 'A',
    quantity: 5,
    size: { h: 8.5, w: 11, uom: 'in' },
    instock: [ { warehouse: 'C', qty: 5 } ]
  }
]
```

The following operation returns all documents that match the query with specific fields (the _id is included by default in the results):

```
db.store.find( { status: "A" }, { item: 1, status: 1 } )
```

```
store> db.store.find( { status: "A" }, { item: 1, status: 1 } )
[
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    status: 'A'
  },
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'journal',
    status: 'A'
  },
  {
    _id: ObjectId("63691693aac189b9509a28a9"),
    item: 'notebook',
    status: 'A'
  }
]
```

If you don't want to include the _id field, set it to 0 in the projection:

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

```
store> db.store.find( { status: "A" }, { item: 1, status: 1, _id: 0 } )
[
  { item: 'postcard', status: 'A' },
  { item: 'journal', status: 'A' },
  { item: 'notebook', status: 'A' }
]
```

Instead of listing the fields to return in the matching document, you can use a projection to exclude specific fields. The following example returns all fields except for the status and the instock fields in the matching documents:

```
db.store.find( { status: "A" }, { status: 0, instock: 0 } )
```

```
store> db.store.find( { status: "A" }, { status: 0, instock: 0 } )
[
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    quantity: 50,
    size: { h: 10, w: 15.25, uom: 'cm' }
  },
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'journal',
    quantity: 5,
    size: { h: 14, w: 21, uom: 'cm' }
  },
  {
    _id: ObjectId("63691693aac189b9509a28a9"),
    item: 'notebook',
    quantity: 5,
    size: { h: 8.5, w: 11, uom: 'in' }
  }
]
```

You can return specific fields in an embedded document. Use the dot notation to refer to the embedded field and set to 1 in the projection document.

```
db.store.find(
   { status: "A" },
   { item: 1, status: 1, "size.uom": 1 }
)
```

```
store> db.store.find(
...     { status: "A" },
...     { item: 1, status: 1, "size.uom": 1 }
... );
[
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    status: 'A',
    size: { uom: 'cm' }
  },
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'journal',
    status: 'A',
    size: { uom: 'cm' }
  },
  {
    _id: ObjectId("63691693aac189b9509a28a9"),
    item: 'notebook',
    status: 'A',
    size: { uom: 'in' }
  }
]
```

You can use the same principle to project specific fields inside documents embedded in an array:

```
db.store.find(
   { status: "A" },
   { item: 1, status: 1, "instock.qty": 1 }
)
```

```
store> db.store.find(
...     { status: "A" },
...     { item: 1, status: 1, "instock.qty": 1 }
... )
[
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    status: 'A',
    instock: [ { qty: 15 }, { qty: 35 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'journal',
    status: 'A',
    instock: [ { qty: 5 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a9"),
    item: 'notebook',
    status: 'A',
    instock: [ { qty: 5 } ]
  }
]
```

For fields that contain arrays, MongoDB provides the following projection operators for manipulating arrays: **$elemMatch**, and **$slice**. Try the following two examples:

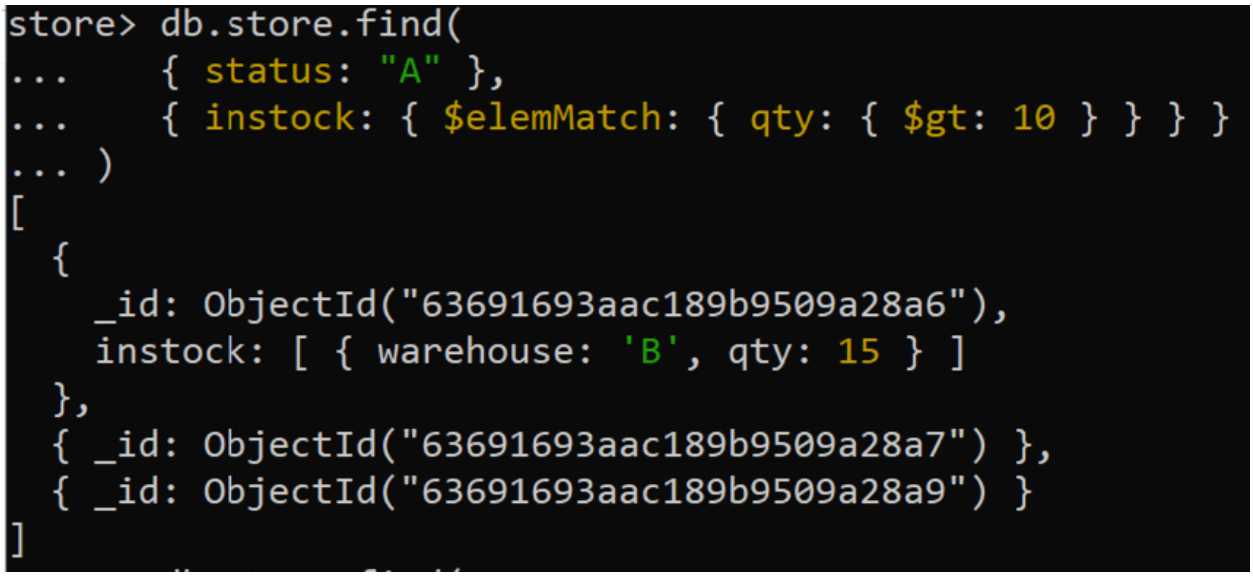
Retrieve the last element in each array

```
db.store.find(
    { status: "A" },
    { item: 1, status: 1, instock: { $slice: -1 } }
)
```

```
store> db.store.find(
...     { status: "A" },
...     { item: 1, status: 1, instock: { $slice: -1 } }
... )
[
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    status: 'A',
    instock: [ { warehouse: 'C', qty: 35 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'journal',
    status: 'A',
    instock: [ { warehouse: 'A', qty: 5 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a9"),
    item: 'notebook',
    status: 'A',
    instock: [ { warehouse: 'C', qty: 5 } ]
  }
]
```

Retrieve only elements with quantity > 10

```
db.store.find(
   { status: "A" },
   { instock: { $elemMatch: { qty: { $gt: 10 } } } }
)
```



Task 2. Limiting records

Use **limit()** method to return only a number of documents.

Try the following query:
`db.store.find( { status: "A" } ).limit(2)`

```
store> db.store.find( { status: "A" } ).limit(2)
[
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    status: 'A',
    quantity: 50,
    size: { h: 10, w: 15.25, uom: 'cm' },
    instock: [ { warehouse: 'B', qty: 15 }, { warehouse: 'C', qty: 35 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'journal',
    status: 'A',
    quantity: 5,
    size: { h: 14, w: 21, uom: 'cm' },
    instock: [ { warehouse: 'A', qty: 5 } ]
  }
]
```

If you don't specify the number argument in limit() method then it will display all documents from the collection.

Apart from limit() method, there is **skip()** to skip a number of documents and retrieve the rest of documents. The following example skips the first element and returns the next two:

```
db.store.find( { status: "A" } ).limit(2).skip(1)
```

```
store> db.store.find( { status: "A" } ).limit(2).skip(1)
[
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'journal',
    status: 'A',
    quantity: 5,
    size: { h: 14, w: 21, uom: 'cm' },
    instock: [ { warehouse: 'A', qty: 5 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a9"),
    item: 'notebook',
    status: 'A',
    quantity: 5,
    size: { h: 8.5, w: 11, uom: 'in' },
    instock: [ { warehouse: 'C', qty: 5 } ]
  }
]
```

If you want to know how many records are available within a query, use the count method:

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

```
store> db.store.find( { status: "A" } ).count()
3
```

Task 3. Sorting records

To return documents ordered by a field, you can use **sort()** method, which accepts a document containing a list of fields along with their sorting order (1 and -1 are used for ascending order and descending order, respectively). If you don't specify the sorting preference, then the documents are displayed in ascending order by default.

Try the following two commands:

```
db.store.find().sort( { "item": 1 })
```

```
store> db.store.find().sort( { "item": 1 })
[
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'journal',
    status: 'A',
    quantity: 5,
    size: { h: 14, w: 21, uom: 'cm' },
    instock: [ { warehouse: 'A', qty: 5 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a9"),
    item: 'notebook',
    status: 'A',
    quantity: 5,
    size: { h: 8.5, w: 11, uom: 'in' },
    instock: [ { warehouse: 'C', qty: 5 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a8"),
    item: 'paper',
    status: 'D',
    quantity: 60,
    size: { h: 8.5, w: 11, uom: 'in' },
    instock: [ { warehouse: 'A', qty: 60 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28aa"),
    item: 'planner',
```

```
db.store.find().sort( { "item": -1 })
```

```
store> db.store.find().sort( { "item": -1 })
[
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    status: 'A',
    quantity: 50,
    size: { h: 10, w: 15.25, uom: 'cm' },
    instock: [ { warehouse: 'B', qty: 15 }, { warehouse: 'C', qty: 35 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28aa"),
    item: 'planner',
    status: 'D',
    quantity: 40,
    size: { h: 22.85, w: 30, uom: 'cm' },
    instock: [ { warehouse: 'A', qty: 40 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a8"),
    item: 'paper',
    status: 'D',
    quantity: 60,
    size: { h: 8.5, w: 11, uom: 'in' },
    instock: [ { warehouse: 'A', qty: 60 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a9"),
    item: 'notebook',
```

Of course, you can filter your data first:

```
db.store.find( { status: "A" } ).sort( { "item": -1})
```

```
store> db.store.find( { status: "A" } ).sort( { "item": -1})
[
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    status: 'A',
    quantity: 50,
    size: { h: 10, w: 15.25, uom: 'cm' },
    instock: [ { warehouse: 'B', qty: 15 }, { warehouse: 'C', qty: 35 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a9"),
    item: 'notebook',
    status: 'A',
    quantity: 5,
    size: { h: 8.5, w: 11, uom: 'in' },
    instock: [ { warehouse: 'C', qty: 5 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'journal',
    status: 'A',
    quantity: 5,
    size: { h: 14, w: 21, uom: 'cm' },
    instock: [ { warehouse: 'A', qty: 5 } ]
  }
]
```

What happens if want to sort by status?

```
db.store.find().sort( { "status": 1 })
```

```
store> db.store.find().sort( { "status": 1 })
[
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    status: 'A',
    quantity: 50,
    size: { h: 10, w: 15.25, uom: 'cm' },
    instock: [ { warehouse: 'B', qty: 15 }, { warehouse: 'C', qty: 35 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'journal',
    status: 'A',
    quantity: 5,
    size: { h: 14, w: 21, uom: 'cm' },
    instock: [ { warehouse: 'A', qty: 5 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a9"),
    item: 'notebook',
    status: 'A',
    quantity: 5,
    size: { h: 8.5, w: 11, uom: 'in' },
    instock: [ { warehouse: 'C', qty: 5 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a8"),
    item: 'paper',
```

You might want to sort the documents by two or more fields. Check the following example:

```
db.store.aggregate(
    [
      { $sort : { status : 1, item: 1 } }
    ]
)
```

```
store> db.store.aggregate(
...     [
...       { $sort : { status : 1, item: 1 } }
...     ]
... )
[
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'journal',
    status: 'A',
    quantity: 5,
    size: { h: 14, w: 21, uom: 'cm' },
    instock: [ { warehouse: 'A', qty: 5 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a9"),
    item: 'notebook',
    status: 'A',
    quantity: 5,
    size: { h: 8.5, w: 11, uom: 'in' },
    instock: [ { warehouse: 'C', qty: 5 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    status: 'A',
    quantity: 50,
    size: { h: 10, w: 15.25, uom: 'cm' },
```

Task 4. Query for missing fields and null

Add the following elements to the **store** collection:

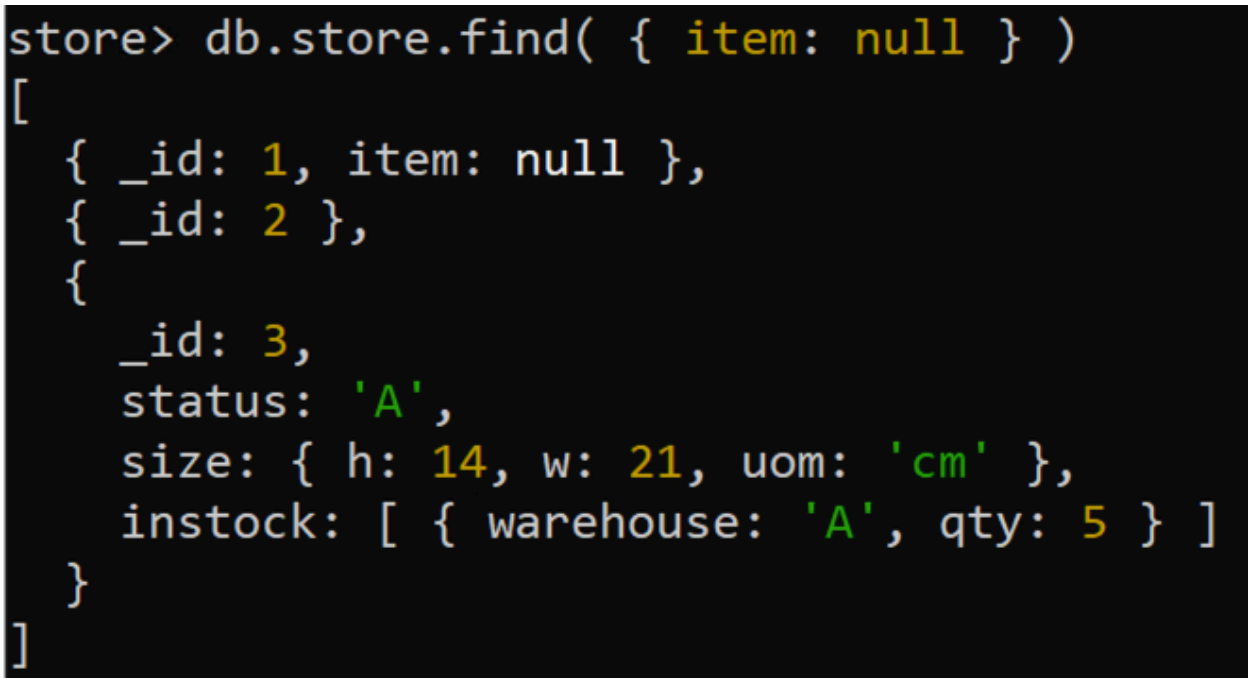
```
  { _id: 1, item: null }

  { _id: 2 }

  { _id: 3, status: "A", size: { h: 14, w: 21, uom: "cm" },
instock: [ { warehouse: "A", qty: 5 } ] }
```

The `{ item : null }` query matches documents that either contain the item field whose value is null or that do not contain the item field.

```
db.store.find( { item: null } )
```



You can use the `{ item : { $type: 10 } }` query to match only documents that contain the item field whose value is null;

```
store> db.store.find( { item: { $type: 10 } } )
[ { _id: 1, item: null } ]
```

BSON Types reference: https://www.mongodb.com/docs/manual/reference/bson-types/

The `{ item : { $exists: false } }` query matches documents that do not contain the item field:

```
db.store.find( { item : { $exists: false } } )
```

```
store> db.store.find( { item: { $exists: false } } )
[
  { _id: 2 },
  {
    _id: 3,
    status: 'A',
    size: { h: 14, w: 21, uom: 'cm' },
    instock: [ { warehouse: 'A', qty: 5 } ]
  }
]
```

Task 5. Aggregation operations

Aggregation operations process multiple documents and return computed results. You can use aggregation operations to:

- Group values from multiple documents together.
- Perform operations on the grouped data to return a single result.
- Analyze data changes over time.

To perform aggregation operations, you can use:

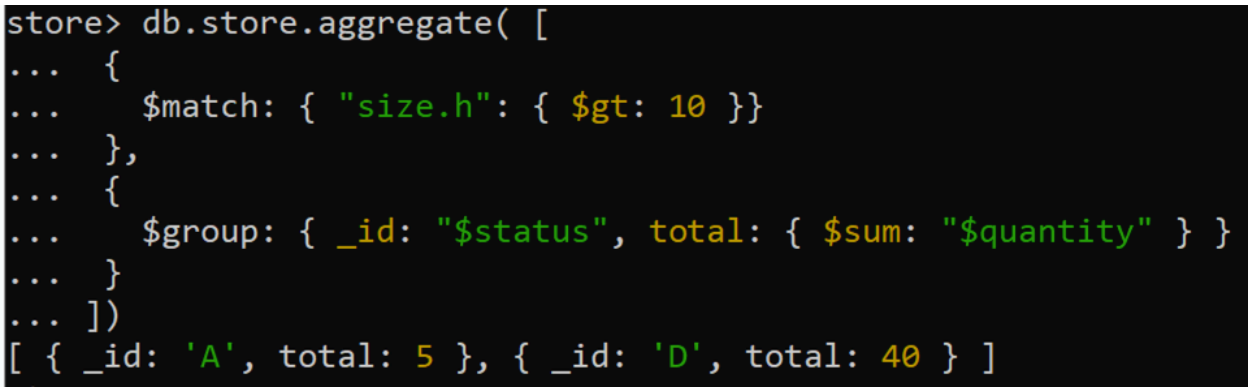
- Aggregation pipelines, which are the preferred method for performing aggregations.
- Single purpose aggregation methods, which are simple but lack the capabilities of an aggregation pipeline.

An aggregation pipeline consists of one or more stages that process documents:

- Each stage performs an operation on the input documents. For example, a stage can filter documents, group documents, and calculate values.
- The documents that are output from a stage are passed to the next stage.
- An aggregation pipeline can return results for groups of documents. For example, return the total, average, maximum, and minimum values.

The following aggregation pipeline example contains two stages and returns the total quantity of items with height greater than 10 grouped by status:

```
db.store.aggregate( [
 {
   $match: { "size.h": { $gt: 10 }}
 },
 {
   $group: { _id: "$status", total: { $sum: "$quantity" } }
 }
])
```



The $match stage:

- Filters the store documents to elements with a size height greater than 10.
- Passes the remaining documents to the $group stage.

The $group stage:

- Groups the remaining documents by status.
- Uses $sum to calculate the total quantity for each status. The total is stored in the **total** field returned by the aggregation pipeline.

There are other operators you can use. Check the list:
https://www.mongodb.com/docs/manual/reference/operator/aggregation/#accumulators---group---bucket---bucketauto---setwindowfields-

A more segmented list of aggregation expressions is available here:
https://www.tutorialspoint.com/mongodb/mongodb_aggregation.htm

Limit, sort, and projection operations can be included as part of aggregation operations. Check the following 3 examples:

```
db.store.aggregate([ { $limit: 1 } ])
```

```
store> db.store.aggregate([ { $limit: 1 } ])
[
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    status: 'A',
    quantity: 50,
    size: { h: 10, w: 15.25, uom: 'cm' },
    instock: [ { warehouse: 'B', qty: 15 }, { warehouse: 'C', qty: 35 } ]
  }
]
```

```
db.store.aggregate([
  {
    $project: {
      "item": 1,
      "status": 1
    }
  },
  {
    $limit: 2
  }
])
```

```
store> db.store.aggregate([
...   {
...     $project: {
...       "item": 1,
...       "status": 1
...     }
...   },
...   {
...     $limit: 2
...   }
... ])
[
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    status: 'A'
  },
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'journal',
    status: 'A'
  }
]
```

```
db.store.aggregate([
  {
    $sort: { "quantity": -1 }
  },
  {
    $project: {
      "item": 1,
      "status": 1
    }
  },
  {
    $limit: 2
  }
])
```

```
store> db.store.aggregate([
...   {
...     $sort: { "quantity": -1 }
...   },
...   {
...     $project: {
...       "item": 1,
...       "status": 1
...     }
...   },
...   {
...     $limit: 2
...   }
... ])
[
  {
    _id: ObjectId("63691693aac189b9509a28a8"),
    item: 'paper',
    status: 'D'
  },
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    status: 'A'
  }
]
```

The order of your stages matters! Each stage only acts upon the documents that previous stages provide. What happens if you move the project operation to be the first one in the pipeline? What about adding the limit operation at the beginning?

You can add new fields to a pipeline with **$addFields**:

```
db.store.aggregate([
{
     $addFields: {
       avgQty: { $avg: "$instock.qty" }
     }
   },  {
     $project: {
       "item": 1,
       "status": 1,
       "avgQty": 1
     }
   },
   {
     $limit: 5
   }
])
```

Task 6. Working with Cursors

The **find()** method returns a **cursor**. To access the documents, you need to iterate the cursor. If the returned cursor is **NOT** assigned to a variable, then the cursor is automatically iterated up to 20 times to print up to the first 20 documents in the results.

Assign the cursor returned from the find() method to a variable using the **var** keyword (as a result, the cursor does not automatically iterate).

```
var myCursor = db.store.find( { status: "A" } );
```

```
store> var myCursor = db.store.find( { status: "A" } );

store>
```

If you access the cursor, the first 20 documents are retrieved immediately:

```
var myCursor = db.store.find( { status: "A" } );
myCursor
```

```
store> myCursor
[
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'postcard',
    status: 'A',
    quantity: 50,
    size: { h: 10, w: 15.25, uom: 'cm' },
    instock: [ { warehouse: 'B', qty: 15 }, { warehouse: 'C', qty: 35 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'journal',
    status: 'A',
    quantity: 5,
    size: { h: 14, w: 21, uom: 'cm' },
    instock: [ { warehouse: 'A', qty: 5 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a9"),
    item: 'notebook',
    status: 'A',
    quantity: 5,
    size: { h: 8.5, w: 11, uom: 'in' },
    instock: [ { warehouse: 'C', qty: 5 } ]
  },
```

Use the cursor method **next()** to access the documents, as in the following example:

```
var myCursor = db.store.find( { status: "A" } );

while (myCursor.hasNext()) {
   printjson(myCursor.next());
}
```

```
store> var myCursor = db.store.find( { status: "A" } );

store>

store> while (myCursor.hasNext()) {
...     printjson(myCursor.next());
... }
{
  _id: ObjectId("63691693aac189b9509a28a6"),
  item: 'postcard',
  status: 'A',
  quantity: 50,
  size: { h: 10, w: 15.25, uom: 'cm' },
  instock: [ { warehouse: 'B', qty: 15 }, { warehouse: 'C', qty: 35 } ]
}
{
  _id: ObjectId("63691693aac189b9509a28a7"),
  item: 'journal',
  status: 'A',
  quantity: 5,
  size: { h: 14, w: 21, uom: 'cm' },
  instock: [ { warehouse: 'A', qty: 5 } ]
}
{
  _id: ObjectId("63691693aac189b9509a28a9"),
  item: 'notebook',
  status: 'A',
  quantity: 5,
  size: { h: 8.5, w: 11, uom: 'in' },
  instock: [ { warehouse: 'C', qty: 5 } ]
```

The next() method exhausts the cursor, which means that trying to access the cursor again will result in an empty object

Another option:

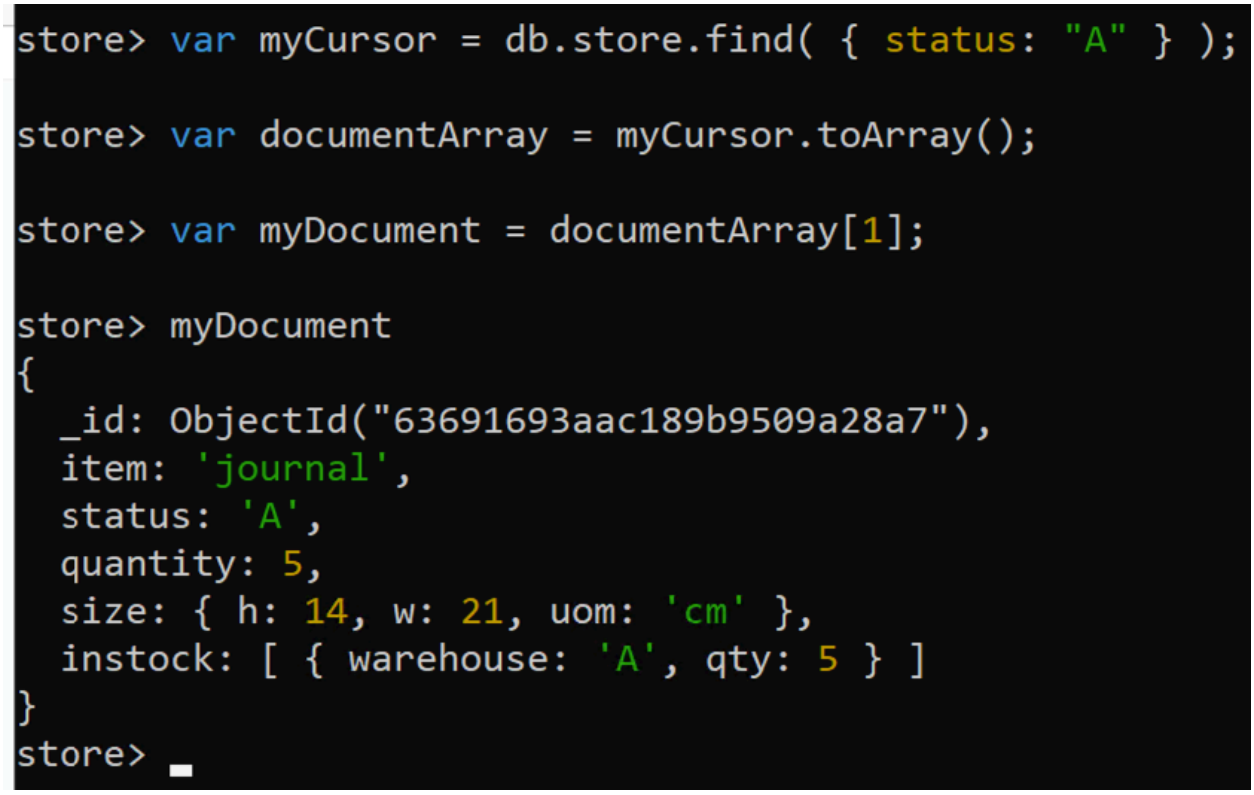
```
var myCursor = db.store.find( { status: "A" } );
myCursor.forEach(printjson);
```

```
store> var myCursor = db.store.find( { status: "A" } );

store> myCursor.forEach(printjson);
{
  _id: ObjectId("63691693aac189b9509a28a6"),
  item: 'postcard',
  status: 'A',
  quantity: 50,
  size: { h: 10, w: 15.25, uom: 'cm' },
  instock: [ { warehouse: 'B', qty: 15 }, { warehouse: 'C', qty: 35 } ]
}
{
  _id: ObjectId("63691693aac189b9509a28a7"),
  item: 'journal',
  status: 'A',
  quantity: 5,
  size: { h: 14, w: 21, uom: 'cm' },
  instock: [ { warehouse: 'A', qty: 5 } ]
}
{
  _id: ObjectId("63691693aac189b9509a28a9"),
  item: 'notebook',
  status: 'A',
  quantity: 5,
  size: { h: 8.5, w: 11, uom: 'in' },
  instock: [ { warehouse: 'C', qty: 5 } ]
}
{
  _id: 3,
  status: 'A',
```

You can use the **toArray()** method to iterate the cursor and return the documents in an array, as in the following:

```
var myCursor = db.store.find( { status: "A" } );
var documentArray = myCursor.toArray();
var myDocument = documentArray[1];
myDocument
```



Finally, you can iterate and perform actions (such as updating or transforming) over your documents. Here is an example:

```
var myCursor = db.store.find( { status: "A" } );

myCursor.forEach(function(doc) {
  var updated_item = "New " + doc.item;
  db.store.updateOne(
    { _id: doc._id },
    { $set: { item: updated_item } }
  )
});

db.store.find( { status: "A" } );
```

```
store> var myCursor = db.store.find( { status: "A" } );

store>

store> myCursor.forEach(function(doc) {
...   var updated_item = "New " + doc.item;
...   db.store.updateOne(
...     { _id: doc._id },
...     { $set: { item: updated_item } }
...   )
... });

store> db.store.find( { status: "A" } );
[
  {
    _id: ObjectId("63691693aac189b9509a28a6"),
    item: 'New postcard',
    status: 'A',
    quantity: 50,
    size: { h: 10, w: 15.25, uom: 'cm' },
    instock: [ { warehouse: 'B', qty: 15 }, { warehouse: 'C', qty: 35 } ]
  },
  {
    _id: ObjectId("63691693aac189b9509a28a7"),
    item: 'New journal',
    status: 'A',
    quantity: 5,
    size: { h: 14, w: 21, uom: 'cm' },
    instock: [ { warehouse: 'A', qty: 5 } ]
```

Another example, with data transformation:

```
var myCursor = db.store.find( { status: "A" } );

myCursor.map(function(doc) {
  return doc.item.split(' ').reverse();
});
```

```
store> var myCursor = db.store.find( { status: "A" } );

store>

store> myCursor.map(function(doc) {
...   return doc.item.split(' ').reverse();
... });
[
  [ 'postcard', 'New' ],
  [ 'journal', 'New' ],
  [ 'notebook', 'New' ],
  [ 'undefined', 'New' ]
]
store>
```

Check available cursor methods here:
https://www.mongodb.com/docs/manual/reference/method/#std-label-js-query-cursor-methods

Task 7. Bulk write operations

The **bulkWrite()** method performs multiple write operations with controls for order of execution.

Syntax:

```
db.collection.bulkWrite(
   [ <operation 1>, <operation 2>, ... ],
   {
      writeConcern : <document>,
      ordered : <boolean>
   }
)
```

Example:
First of all, create a pizzas collection, then insert the following documents:

```
{ _id: 0, type: "pepperoni", size: "small", price: 4 }

{ _id: 1, type: "cheese", size: "medium", price: 7 }

{ _id: 2, type: "vegan", size: "large", price: 8 }
```

The code in the next page runs these operations on the pizzas collection:

- Adds two documents using insertOne.
- Updates a document using updateOne.
- Deletes a document using deleteOne.
- Replaces a document using replaceOne.

```
try {
   db.pizzas.bulkWrite( [
      { insertOne: { document: { _id: 3, type: "beef", size:
"medium", price: 6 } } },
      { insertOne: { document: { _id: 4, type: "sausage", size:
"large", price: 10 } } },
      { updateOne: {
         filter: { type: "cheese" },
         update: { $set: { price: 8 } }
      } },
      { deleteOne: { filter: { type: "pepperoni"} } },
      { replaceOne: {
         filter: { type: "vegan" },
         replacement: { type: "tofu", size: "small", price: 4 }
      } }
   ] )
} catch( error ) {
   print( error )
}
```

```
store> try {
...    db.pizzas.bulkWrite( [
...       { insertOne: { document: { _id: 3, type: "beef", size: "medium", price: 6 } } },
...       { insertOne: { document: { _id: 4, type: "sausage", size: "large", price: 10 } } },
...       { updateOne: {
...          filter: { type: "cheese" },
...          update: { $set: { price: 8 } }
...       } },
...       { deleteOne: { filter: { type: "pepperoni"} } },
...       { replaceOne: {
...          filter: { type: "vegan" },
...          replacement: { type: "tofu", size: "small", price: 4 }
...       } }
...    ] )
... } catch( error ) {
...    print( error )
... }
{
  acknowledged: true,
  insertedCount: 2,
  insertedIds: { '0': 3, '1': 4 },
  matchedCount: 2,
  modifiedCount: 2,
  deletedCount: 1,
  upsertedCount: 0,
  upsertedIds: {}
}
```

This is the content of pizzas collection:

```
store> db.pizzas.find()
[
  { _id: 1, type: 'cheese', size: 'medium', price: 8 },
  { _id: 2, type: 'tofu', size: 'small', price: 4 },
  { _id: 3, type: 'beef', size: 'medium', price: 6 },
  { _id: 4, type: 'sausage', size: 'large', price: 10 }
]
```

If there is an error (for example, the collection already contained a document with an _id of 3 before running the previous bulkWrite() example) an exception is returned **and the process stops** (this is known as an ordered bulkWrite operation).

```
store> try { db.pizzas.bulkWrite([ { insertOne: { document: { _id: 3, type: "beef", size: "medium", price: 6 }
 } }, { insertOne: { document: { _id: 4, type: "sausage", size: "large", price: 10 } } }, { updateOne: { filte
r: { type: "cheese" }, update: { $set: { price: 8 } } } }, { deleteOne: { filter: { type: "pepperoni" } } }, {
 replaceOne: { filter: { type: "vegan" }, replacement: { type: "tofu", size: "small", price: 4 } } }]); } catc
h (error) { print(error); }
MongoBulkWriteError: E11000 duplicate key error collection: store.pizzas index: _id_ dup key: { _id: 3 }
    at OrderedBulkOperation.handleWriteError (C:\MongoShell\mongosh-1.6.0-win32-x64\bin\mongosh.exe:56113:16)
    at resultHandler (C:\MongoShell\mongosh-1.6.0-win32-x64\bin\mongosh.exe:55498:23)
    at C:\MongoShell\mongosh-1.6.0-win32-x64\bin\mongosh.exe:52305:5
    at C:\MongoShell\mongosh-1.6.0-win32-x64\bin\mongosh.exe:53841:9
    at C:\MongoShell\mongosh-1.6.0-win32-x64\bin\mongosh.exe:75175:13
    at C:\MongoShell\mongosh-1.6.0-win32-x64\bin\mongosh.exe:76300:9
    at handleOperationResult (C:\MongoShell\mongosh-1.6.0-win32-x64\bin\mongosh.exe:76393:14)
    at Connection.onMessage (C:\MongoShell\mongosh-1.6.0-win32-x64\bin\mongosh.exe:72974:5)
    at MessageStream.<anonymous> (C:\MongoShell\mongosh-1.6.0-win32-x64\bin\mongosh.exe:72777:56)
    at MessageStream.emit (node:events:513:28) {
  code: 11000,
  writeErrors: [
    WriteError {
      err: {
        index: 0,
        code: 11000,
        errmsg: 'E11000 duplicate key error collection: store.pizzas index: _id_ dup key: { _id: 3 }',
        errInfo: undefined,
        op: { _id: 3, type: 'beef', size: 'medium', price: 6 }
      }
    }
  ],
```

```
result: BulkWriteResult {
  result: {
    ok: 1,
    writeErrors: [
      WriteError {
        err: {
          index: 0,
          code: 11000,
          errmsg: 'E11000 duplicate key error collection: store.pizzas index: _id_ dup key: { _id: 3 }',
          errInfo: undefined,
          op: { _id: 3, type: 'beef', size: 'medium', price: 6 }
        }
      }
    ],
    writeConcernErrors: [],
    insertedIds: [ { index: 0, _id: 3 }, { index: 1, _id: 4 } ],
    nInserted: 0,
    nUpserted: 0,
    nMatched: 0,
    nModified: 0,
    nRemoved: 0,
    upserted: []
  }
},
[Symbol(errorLabels)]: Set(0) {}
```

To complete all operations that do not have errors, run bulkWrite() with **ordered** set to **false**.

- Drop the pizzas collection.
- Insert the elements again:

```
{ _id: 0, type: "pepperoni", size: "small", price: 4 }

{ _id: 1, type: "cheese", size: "medium", price: 7 }

{ _id: 2, type: "vegan", size: "large", price: 8 }
```

- Use the following code:

```
try {
   db.pizzas.bulkWrite( [
      { insertOne: { document: { _id: 3, type: "beef", size:
"medium", price: 6 } } },
      { insertOne: { document: { _id: 3, type: "sausage", size:
"large", price: 10 } } },
      { updateOne: {
         filter: { type: "cheese" },
         update: { $set: { price: 8 } }
      } },
      { deleteOne: { filter: { type: "pepperoni"} } },
      { replaceOne: {
         filter: { type: "vegan" },
         replacement: { type: "tofu", size: "small", price: 4 }
      } }
   ],
   { ordered: false } )
} catch( error ) {
   print( error )
}
```

Results on the next page:

Despite the error (duplicate key), the rest of the commands were executed.

```
result: BulkWriteResult {
  result: {
    ok: 1,
    writeErrors: [
      WriteError {
        err: {
          index: 1,
          code: 11000,
          errmsg: 'E11000 duplicate key error collection: store.pizzas index: _id_ dup key: { _id: 3 }',
          errInfo: undefined,
          op: { _id: 3, type: 'sausage', size: 'large', price: 10 }
        }
      }
    ],
    writeConcernErrors: [],
    insertedIds: [ { index: 0, _id: 3 }, { index: 1, _id: 3 } ],
    nInserted: 1,
    nUpserted: 0,
    nMatched: 2,
    nModified: 2,
    nRemoved: 1,
    upserted: []
  }
},
[Symbol(errorLabels)]: Set(0) {}
```

The **writeConcern** argument is used for replica set members acknowledgement. It will be explored later.