MongoDB

Unit 2

History

- The development of MongoDB began in 2007 by a New York based organization named 10gen which is now known as MongoDB Inc.
- It was initially developed as a PAAS (Platform As A Service).
- Later in 2009, it was introduced in the market as an open source database server that was maintained and supported by MongoDB Inc.
- The first ready production of MongoDB has been considered from version 1.4 which was released in March 2010.
- Latest version is 3.6.

Introduction

- MongoDB is a document-oriented database.
- It is a key feature of MongoDB.
- It offers a document-oriented storage.
- MongoDB stores data in flexible, JSON-like documents, meaning fields can vary from document to document and data structure can be changed over time

```
FirstName = "Ajeet",
Address = "Laxmi Nagar",
Spouse = [{Name: "Chaaru"}].
FirstName ="Ravi",
Address = "Loni"
```

- The document model maps to the objects in your application code, making data easy to work with.
- Ad hoc queries, indexing, and real time aggregation provide powerful ways to access and analyze your data
- MongoDB is a distributed database at its core, so high availability, horizontal scaling, and geographic distribution are built in and easy to use
- MongoDB is free and open-source, published under the GNU Affero General Public License.

MongoDB – Advantages

- Fast, Iterative Development
  - Scope creep and changing business requirements no longer stand between you and successful project delivery.
  - A flexible data model coupled with dynamic schema, and idiomatic drivers, with powerful GUI and command line tools make it fast for developers to build and evolve applications.
  - Automated provisioning and management enable continuous integration and delivery for highly productive operations.
- Flexible Data Model
  - MongoDB stores data in flexible, JSON-like documents, making it easy for you to persist and combine data of any structure.

- The document model maps to the objects in your application code, making data easy to work with, without giving up schema governance controls, data access, complex aggregations, and rich indexing functionality.
- The schema can be dynamically modified without downtime.

- Distributed Data Platform
  - MongoDB can be run within and across geographically distributed data centers and cloud regions, providing new levels of availability and scalability.
  - MongoDB scales elastically with no downtime, and without changing the application.
  - As performance and availability goals evolve, MongoDB lets the developer adapt flexibly, across data centers, with tunable consistency.
- Integrated Feature Set
  - Analytics and data visualization, text and geospatial search, graph processing, event-driven streaming data pipelines, in-memory performance and global replication allow to deliver a wide variety of real-time applications on one technology, reliably and securely.
  - RDBMS systems require additional, complex technologies demanding separate integration overhead and expense to do this well.
- Lower Total Cost of Ownership
  - Application development teams are 2x and more productive when they use MongoDB.
  - The fully managed Atlas cloud service means operations team are as well.
  - MongoDB runs on commodity hardware, dramatically lowering costs.
  - MongoDB offers on-demand, pay-as-you-go pricing and affordable annual subscriptions, including 24x7x365 global support.
  - Applications can be one tenth the cost to deliver compared to using a relational database.

Reasons for moving away from RDBMS

- Expressive query language & secondary Indexes
  - Users should be able to access and manipulate their data in sophisticated ways to support both operational and analytical applications.
  - Indexes play a critical role in providing efficient access to data, supported natively by the database rather than maintained in application code.
- Strong consistency
  - Applications should be able to immediately read what has been written to the database.
  - It is much more complex to build applications around an eventually consistent model, imposing significant work on the developer, even for the most sophisticated engineering teams.
- Enterprise Management and Integrations
  - Databases are just one piece of application infrastructure, and need to fit seamlessly into the enterprise IT stack.

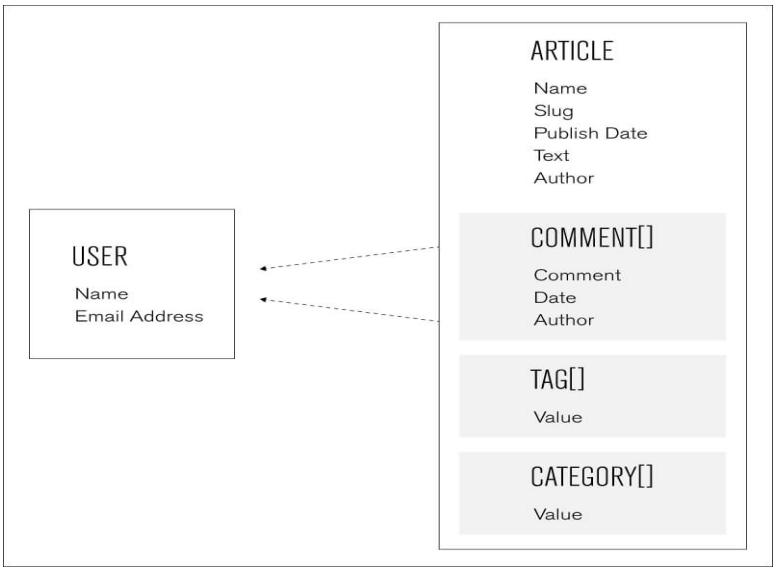
- Organizations need a database that can be secured, monitored, automated, and integrated with their existing technology infrastructure, processes, and staff, including operations teams, DBAs, and data analysts.

Features of architecture

- MongoDB's flexible document data model presents a superset of other database models.
  - It allows data to be represented as simple key-value pairs and flat, table-like structures, through to rich documents and objects with deeply nested arrays and sub-documents.
- With an expressive query language, documents can be queried in many ways.
  - Simple lookups
  - Sophisticated processing pipelines for data analytics and transformations, through to faceted search, JOINs and graph traversals.
- With a flexible storage architecture, application owners can deploy storage engines optimized for different workload and operational requirements.

Data in MongoDB

- MongoDB stores data in a binary representation called BSON (Binary JSON).
- The BSON encoding extends the popular JSON (JavaScript Object Notation) representation to include additional types such as int, long, date, floating point, and decimal128.
- BSON documents contain one or more fields, and each field contains a value of a specific data type, including arrays, binary data and sub-documents.
- MongoDB BSON documents are closely aligned to the structure of objects in the programming language.
  - This makes it simpler and faster for developers to model how data in the application will map to data stored in the database.
- MongoDB documents tend to have all data for a given record in a single document, whereas in a relational database information for a given record is usually spread across many tables.
- Example
  - Consider the data model for a blogging application. In a relational database, the data model would comprise multiple tables such as Categories, Tags, Users, Comments and Articles.
  - In MongoDB the data could be modeled as two collections, one for users, and the other for articles.
    - In each blog document there might be multiple comments, multiple tags, and multiple categories, each expressed as an embedded array.



- Data in MongoDB is more localized, which dramatically reduces the need to JOIN separate tables.
- The result is dramatically higher performance and scalability across commodity hardware as a single read to the database can retrieve the entire document.
- Unlike many NoSQL databases, users don't need to give up JOINs entirely.
  - For additional flexibility, MongoDB provides the ability to perform equi and non-equi JOINs that combine data from multiple collections, typically when executing analytical queries against live, operational data.

Collection

- Collections are simply groups of documents.
  - Since documents exist independently, they can have different fields.
- It is referred to as dynamic schema.

Field Names

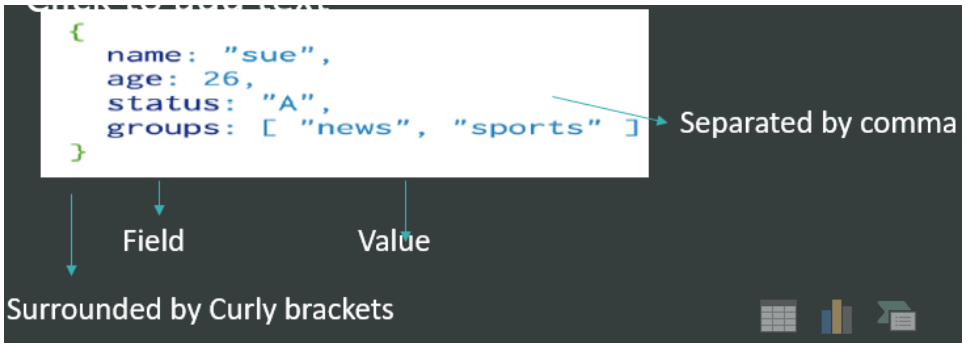
- Field names are strings.
- Documents have the following restrictions on field names:
  - The field name _id is reserved for use as a primary key;
  - Its value must be unique in the collection, is immutable, and may be of any type other than an array.
  - The field names cannot start with the dollar sign ($) character.
  - The field names cannot contain the dot (.) character.
  - The field names cannot contain the null character.

ObjectID

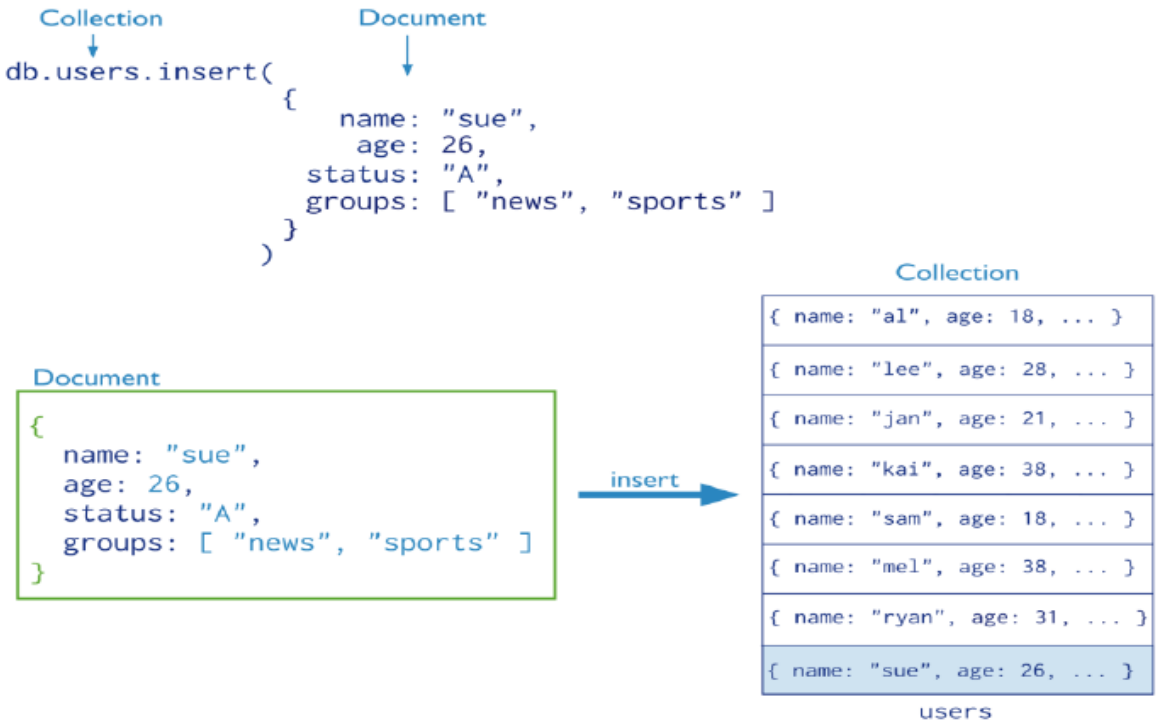
- ObjectId is a 12-byte BSON type, constructed using:
  - a 4-byte value representing the seconds since the Unix epoch,
  - a 3-byte machine identifier,
  - a 2-byte process id, and
  - a 3-byte counter, starting with a random value
- Example

- "_id" : ObjectId("5699348d350ba4219fca39a1")
- "_id" : ObjectId("56993492350ba4219fca39a2")

Document



Documents vs. Collections



Commands

- Connect to mongod
  - mongo
- Clear the screen
  - cls
- Create a new database if it does not exist or returns an existing database
  - use <database_name>
  - output
    - switched to db database_name
- After starting the mongo shell, session will use the test database by default.
- Check the currently selected database_name

- db
- output
  - database_name

- check the databases list
  - show dbs
  - Output
    - table size

- To display a database atleast one document should be inserted into it
- Drop an existing databse
  - db.dropDatabase()
  - Output
    - if not drop the default test database

- Create a collection
  - db.createCollection(name, options)
    - name is name of collection to be created.
    - Options is a document and is used to specify configuration of collection.
    - Options can be
      - Capped
        - It is boolean and optional
        - If true, enables a capped collection.
        - Capped collection is a fixed size collection that automatically overwrites its oldest entries when it reaches its maximum size.
        - If we specify true, we need to specify size parameter also.
          - db.createCollection("MCA", {capped : true, size : 5242880, max : 5000 })
      - autoIndexID
        - It is boolean and optional
        - If true, automatically create index on _id fields.
        - Default value is false.
      - size number
        - It is optional
        - Specifies a maximum size in bytes for a capped collection.
        - If capped is true, then we need to specify this field also.
      - max number
        - It is optional
          - Specifies the maximum number of documents allowed in the capped collection.

CRUD Commands – Create or Insert

- Inserts a document or documents into a collection
  - **db.<collection_name>.insert(<document>)**
  - Output

- WriteResult({"nInserted": 1})
- During insert mongod will create the _id field and assign it a unique ObjectId value

- Insert a document specifying an _id field
  - **db. <collection_name>.insert({_id: num, filed_name: value})**
    - The value of _id must be unique within the collection to avoid duplicate key error.
- Insert multiple documents
  - **db. <collection_name>.insert({<Document1>}, {<Document2>,…})**
- Unordered Insert
  - **db.<collection_name>.insert(**
    [{<Document1>}, {<Document2>},….],
    {ordered: false})
    - With unordered inserts, if an error occurs during an insert of one of the documents, MongoDB continues to insert the remaining documents in the array.
- Insert Single Document
  - **db.<collection_name>.insertOne({<Document1>})**
- Insert a document specifying an _id field
  - **db. <collection_name>.insertOne({_id: num, field_name: value})**
    - The value of _id must be unique within the collection to avoid duplicate key error.
- Insert Multiple Documents with Unordered Insert
  - **db.<collection_name>.insertMany(**
    [{<document 1>}, {<document 2>}, …],
    {ordered: <boolean>})
- Insert a document specifying an _id field
  - **db. <collection_name>.insertMany(**
    {_id: num, field_name: value},
    {_id: num, field_name: value})
    - The value of _id must be unique within the collection to avoid duplicate key error.
- Update an existing document or insert a new document, depending on its document parameter.
  - **db.<collection_name>.save({<document>})**
  - **db.<collection_name>.save({<_id: num, field_name: value, ..>})**
- Modifies an existing document or documents in a collection.
  - The method can modify specific fields of an existing document or documents or replace an existing document entirely, depending on the update parameter.
    - **db.<collection_name>.update(**
      {field_name: existing_value },
      {field_name: existing_value, new_field: new_value, existing_field : new_value },
      {upsert: true})

CRUD Commands – Read

- Selects documents in a collection or view and returns a pointer to the selected documents.

- **db.<collection_name>.find(query, projection)**
  - The query parameter is optional. Specifies selection filter using query operators. To return all documents in a collection, omit this parameter or pass an empty document ({}).
  - The projection parameter determines which fields are returned in the matching documents.

```
db.users.find(                          <-- collection
    { age: { $gt: 18 } },               <-- query criteria
    { name: 1, address: 1 }             <-- projection
).limit(5)                              <-- cursor modifier
```

- The find() method with no parameters returns all documents from a collection and returns all fields for the documents.
  - **db.<collection_name>.find()**

```
> db.emp.emp.find()
{ "_id" : ObjectId("5ac48545bfb80b3ee89f5303"), "id" : "1", "name" : "Lekha", "age" : "23", "country" : "India" }
{ "_id" : ObjectId("5ac485bebfb80b3f443028f6"), "id" : "1", "name" : "Lekha", "age" : "12", "country" : "India" }
```

- Display only id
  - **db.<collection_name>.find({},{_id:1})**

```
> db.cloth.find({}, {_id:1})
{ "_id" : 1 }
{ "_id" : 2 }
{ "_id" : 3 }
{ "_id" : 4 }
```

- Display two fields (id and other)
  - **db.<collection_name>.find({},{_id:1,<field>:1})**

```
> db.cloth.find({}, {_id:1, dept:1})
{ "_id" : 1, "dept" : "A" }
{ "_id" : 2, "dept" : "A" }
{ "_id" : 3, "dept" : "B" }
{ "_id" : 4, "dept" : "A" }
```

- Display all the fields except id
  - **db.<collection_name>.find({},{_id:0})**

```
> db.cloth.find({}, {_id:0})
{ "dept" : "A", "item" : { "sku" : "111", "color" : "red" }, "sizes" : [ "S", "M" ] }
{ "dept" : "A", "item" : { "sku" : "111", "color" : "blue" }, "sizes" : [ "M", "L" ] }
{ "dept" : "B", "item" : { "sku" : "222", "color" : "blue" }, "sizes" : "S" }
{ "dept" : "A", "item" : { "sku" : "333", "color" : "black" }, "sizes" : [ "S" ] }
```

- Display fields except id and dept

```
> db.cloth.find({}, {_id:0, dept:0})
{ "item" : { "sku" : "111", "color" : "red" }, "sizes" : [ "S", "M" ] }
{ "item" : { "sku" : "111", "color" : "blue" }, "sizes" : [ "M", "L" ] }
{ "item" : { "sku" : "222", "color" : "blue" }, "sizes" : "S" }
{ "item" : { "sku" : "333", "color" : "black" }, "sizes" : [ "S" ] }
```

- Display only one field (other than id)
  - **db.<collection_name>.find({},{_id:0, <field>: 1})**

```
> db.cloth.find({}, {_id:0, dept:1})
{ "dept" : "A" }
{ "dept" : "A" }
{ "dept" : "B" }
{ "dept" : "A" }
```

- Find Documents that Match Query Criteria
  - **db.<collection_name>.find(<CRITERIA>)**
  - Example
    - Return all the documents from the collection products where qty is greater than 25
      - **db.products.find({ qty: { $gt: 25 } })**

Operator	Description

$eq	⇒ Matches values that are equal to a specified value.
	⇒ **{field: {$eq: value}}**
	`> db.inven.find()`
	`{ "_id" : 1, "item" : "abc1", "description" : "product 1", "qty" : 300 }`
	`{ "_id" : 2, "item" : "abc2", "description" : "product 2", "qty" : 200 }`
	`{ "_id" : 3, "item" : "xyz1", "description" : "product 3", "qty" : 250 }`
	`{ "_id" : 4, "item" : "VWZ1", "description" : "product 4", "qty" : 300 }`
	`{ "_id" : 5, "item" : "VWZ2", "description" : "product 5", "qty" : 180 }`
	`> db.inven.aggregate([{$project:{item: 1, qty: 1, qty250: { $eq: [ "$qty", 250 ] },_id: 0}}])`
	`{ "item" : "abc1", "qty" : 300, "qty250" : false }`
	`{ "item" : "abc2", "qty" : 200, "qty250" : false }`
	`{ "item" : "xyz1", "qty" : 250, "qty250" : true }`
	`{ "item" : "VWZ1", "qty" : 300, "qty250" : false }`
	`{ "item" : "VWZ2", "qty" : 180, "qty250" : false }`

$gt	⇒ Matches values that are greater than a specified value.
	⇒ **{field: {$gt: value}}**
	`> db.inven.find()`
	`{ "_id" : 1, "item" : "abc1", "description" : "product 1", "qty" : 300 }`
	`{ "_id" : 2, "item" : "abc2", "description" : "product 2", "qty" : 200 }`
	`{ "_id" : 3, "item" : "xyz1", "description" : "product 3", "qty" : 250 }`
	`{ "_id" : 4, "item" : "VWZ1", "description" : "product 4", "qty" : 300 }`
	`{ "_id" : 5, "item" : "VWZ2", "description" : "product 5", "qty" : 180 }`
	`> db.inven.aggregate([{$project:{item: 1, qty: 1, qtygt250: { $gt: [ "$qty", 250 ] },_id: 0}}])`
	`{ "item" : "abc1", "qty" : 300, "qtygt250" : true }`
	`{ "item" : "abc2", "qty" : 200, "qtygt250" : false }`
	`{ "item" : "xyz1", "qty" : 250, "qtygt250" : false }`
	`{ "item" : "VWZ1", "qty" : 300, "qtygt250" : true }`
	`{ "item" : "VWZ2", "qty" : 180, "qtygt250" : false }`

$gte

⇒ Matches values that are greater than or equal to a specified value.

⇒ **{field: {$gte: value}}**

```
> db.inven.find()
{ "_id" : 1, "item" : "abc1", "description" : "product 1", "qty" : 300 }
{ "_id" : 2, "item" : "abc2", "description" : "product 2", "qty" : 200 }
{ "_id" : 3, "item" : "xyz1", "description" : "product 3", "qty" : 250 }
{ "_id" : 4, "item" : "VWZ1", "description" : "product 4", "qty" : 300 }
{ "_id" : 5, "item" : "VWZ2", "description" : "product 5", "qty" : 180 }
>  db.inven.aggregate([{$project:{item: 1, qty: 1, qtygte250: { $gte: [ "$qty", 250 ] },_id: 0}}])
{ "item" : "abc1", "qty" : 300, "qtygte250" : true }
{ "item" : "abc2", "qty" : 200, "qtygte250" : false }
{ "item" : "xyz1", "qty" : 250, "qtygte250" : true }
{ "item" : "VWZ1", "qty" : 300, "qtygte250" : true }
{ "item" : "VWZ2", "qty" : 180, "qtygte250" : false }
```

$in

⇒ Matches any of the values specified in an array.

⇒ **{ field: { $in: [<value1>, <value2>, … <valueN>] } }**

⇒ Project true if the documents have "bananas" in in_stock

```
> db.fruits.find()
{ "_id" : 1, "location" : "24th Street", "in_stock" : [ "apples", "oranges", "bananas" ] }
{ "_id" : 2, "location" : "36th Street", "in_stock" : [ "bananas", "pears", "grapes" ] }
{ "_id" : 3, "location" : "82nd Street", "in_stock" : [ "cantaloupes", "watermelons", "apples" ] }
{ "_id" : 4, "location" : "45th Street", "in_stock" : [ "strawberry", "blueberry", "bananas" ] }
{ "_id" : 5, "location" : "1st Street", "in_stock" : [ "bananas", "kiwi", "passion fruit" ] }
{ "_id" : 6, "location" : "4th Street", "in_stock" : [ "mango", "squash", "wood apple" ] }
> db.fruits.aggregate([{ $project: {"store location" : "$location","has bananas" : {$in: [ "bananas", "$in_stock" ]}}}
])
{ "_id" : 1, "store location" : "24th Street", "has bananas" : true }
{ "_id" : 2, "store location" : "36th Street", "has bananas" : true }
{ "_id" : 3, "store location" : "82nd Street", "has bananas" : false }
{ "_id" : 4, "store location" : "45th Street", "has bananas" : true }
{ "_id" : 5, "store location" : "1st Street", "has bananas" : true }
{ "_id" : 6, "store location" : "4th Street", "has bananas" : false }
```

⇒ Display all documents that have "bananas" in the in_stock

```
> db.fruits.find()
{ "_id" : 1, "location" : "24th Street", "in_stock" : [ "apples", "oranges", "bananas" ] }
{ "_id" : 2, "location" : "36th Street", "in_stock" : [ "bananas", "pears", "grapes" ] }
{ "_id" : 3, "location" : "82nd Street", "in_stock" : [ "cantaloupes", "watermelons", "apples" ] }
{ "_id" : 4, "location" : "45th Street", "in_stock" : [ "strawberry", "blueberry", "bananas" ] }
{ "_id" : 5, "location" : "1st Street", "in_stock" : [ "bananas", "kiwi", "passion fruit" ] }
{ "_id" : 6, "location" : "4th Street", "in_stock" : [ "mango", "squash", "wood apple" ] }
> db.fruits.find({in_stock: {$in : ["bananas"]}})
{ "_id" : 1, "location" : "24th Street", "in_stock" : [ "apples", "oranges", "bananas" ] }
{ "_id" : 2, "location" : "36th Street", "in_stock" : [ "bananas", "pears", "grapes" ] }
{ "_id" : 4, "location" : "45th Street", "in_stock" : [ "strawberry", "blueberry", "bananas" ] }
{ "_id" : 5, "location" : "1st Street", "in_stock" : [ "bananas", "kiwi", "passion fruit" ] }
```

⇒

$lt

⇒ Matches values that are less than a specified value.

⇒ **{field: {$lt: value}}**

```
> db.inven.find()
{ "_id" : 1, "item" : "abc1", "description" : "product 1", "qty" : 300 }
{ "_id" : 2, "item" : "abc2", "description" : "product 2", "qty" : 200 }
{ "_id" : 3, "item" : "xyz1", "description" : "product 3", "qty" : 250 }
{ "_id" : 4, "item" : "VWZ1", "description" : "product 4", "qty" : 300 }
{ "_id" : 5, "item" : "VWZ2", "description" : "product 5", "qty" : 180 }
> db.inven.aggregate([{$project:{item: 1,qty: 1, qtyLt250: { $lt: [ "$qty", 250 ] },_id: 0}}])
{ "item" : "abc1", "qty" : 300, "qtyLt250" : false }
{ "item" : "abc2", "qty" : 200, "qtyLt250" : true }
{ "item" : "xyz1", "qty" : 250, "qtyLt250" : false }
{ "item" : "VWZ1", "qty" : 300, "qtyLt250" : false }
{ "item" : "VWZ2", "qty" : 180, "qtyLt250" : true }
```

$lte

⇒ Matches values that are less than or equal to a specified value.

⇒ **{field: {$lte: value}}**

```
> db.inven.find()
{ "_id" : 1, "item" : "abc1", "description" : "product 1", "qty" : 300 }
{ "_id" : 2, "item" : "abc2", "description" : "product 2", "qty" : 200 }
{ "_id" : 3, "item" : "xyz1", "description" : "product 3", "qty" : 250 }
{ "_id" : 4, "item" : "VWZ1", "description" : "product 4", "qty" : 300 }
{ "_id" : 5, "item" : "VWZ2", "description" : "product 5", "qty" : 180 }
> db.inven.aggregate([{$project:{item: 1,qty: 1, qtyLe250: { $lte: [ "$qty", 250 ] },_id: 0}}])
{ "item" : "abc1", "qty" : 300, "qtyLe250" : false }
{ "item" : "abc2", "qty" : 200, "qtyLe250" : true }
{ "item" : "xyz1", "qty" : 250, "qtyLe250" : true }
{ "item" : "VWZ1", "qty" : 300, "qtyLe250" : false }
{ "item" : "VWZ2", "qty" : 180, "qtyLe250" : true }
```

$ne

⇒ Matches all values that are not equal to a specified value.

⇒ **{field: {$ne: value}}**

```
> db.inven.aggregate([{$project:{item: 1, qty: 1, qtyNe250: { $ne: [ "$qty", 250 ] },_id: 0}}])
{ "item" : "abc1", "qty" : 300, "qtyNe250" : true }
{ "item" : "abc2", "qty" : 200, "qtyNe250" : true }
{ "item" : "xyz1", "qty" : 250, "qtyNe250" : false }
{ "item" : "VWZ1", "qty" : 300, "qtyNe250" : true }
{ "item" : "VWZ2", "qty" : 180, "qtyNe250" : true }
```

$nin

⇒ Matches none of the values specified in an array.

⇒ **{field: {$nin: [<value1>, <value2> … <valueN>]}}**

⇒ Display all documents that do not have "bananas" in in_stock

```
> db.fruits.find()
{ "_id" : 1, "location" : "24th Street", "in_stock" : [ "apples", "oranges", "bananas" ] }
{ "_id" : 2, "location" : "36th Street", "in_stock" : [ "bananas", "pears", "grapes" ] }
{ "_id" : 3, "location" : "82nd Street", "in_stock" : [ "cantaloupes", "watermelons", "apples" ] }
{ "_id" : 4, "location" : "45th Street", "in_stock" : [ "strawberry", "blueberry", "bananas" ] }
{ "_id" : 5, "location" : "1st Street", "in_stock" : [ "bananas", "kiwi", "passion fruit" ] }
{ "_id" : 6, "location" : "4th Street", "in_stock" : [ "mango", "squash", "wood apple" ] }
> db.fruits.find({in_stock: {$nin: [ 'bananas']}})
{ "_id" : 3, "location" : "82nd Street", "in_stock" : [ "cantaloupes", "watermelons", "apples" ] }
{ "_id" : 6, "location" : "4th Street", "in_stock" : [ "mango", "squash", "wood apple" ] }
```

- Returns one document that satisfies the specified query criteria on the collection.
  - **db.<collection_name>.findOne(query, projection)**
    - If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk.

- In capped collections, natural order is the same as insertion order. If no document satisfies the query, the method returns null.

- Returns a single document without any query.
  - **db.<collection_name>.findOne()**

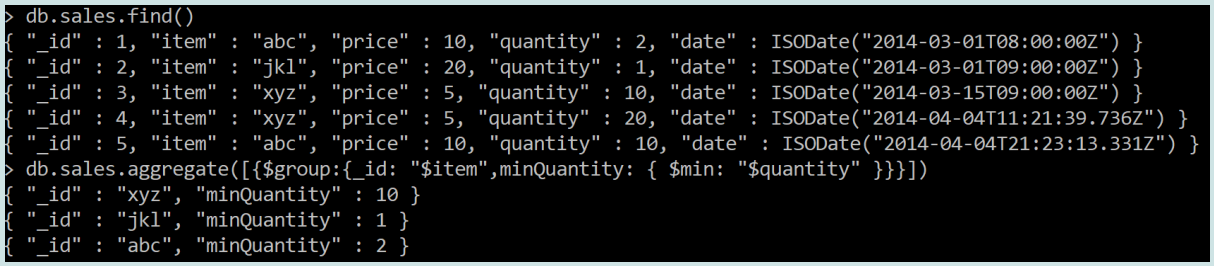
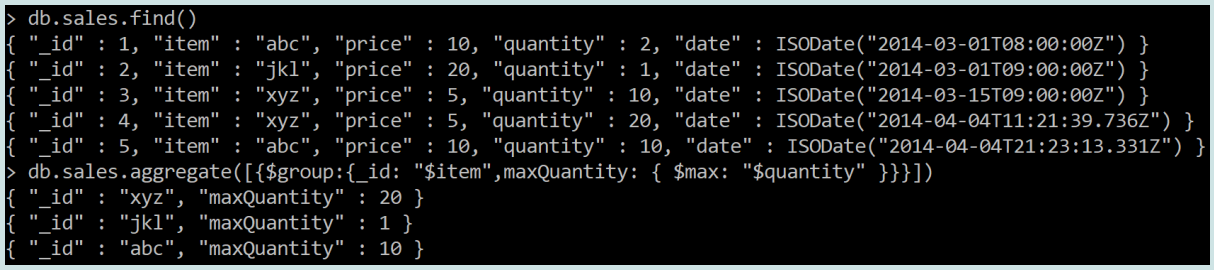
```
> db.bios.findOne(
...     { },
...     { name: 1, contribs: 1 }
... )
{
        "_id" : 1,
        "name" : {
                "first" : "John",
                "last" : "Backus"
        },
        "contribs" : [
                "Fortran",
                "ALGOL",
                "Backus-Naur Form",
                "FP"
        ]
}
```

CRUD Commands – Update

- Modifies an existing document or documents in a collection.
  - The method can modify specific fields of an existing document or documents or replace an existing document entirely, depending on the update parameter.
  - **db.<collection_name>.update({ field_name: existing_value },**
    {field_name: existing_value, new_field: new_value, existing_field : new_value },
    {upsert: true})
- Replace a single document within the collection based on the filter.
  - **db.<collection_name>.replaceOne(<filter>, <replacement>)**
    - In a capped collection if a replacement operation changes the document size, the operation will fail.

CRUD Commands – Update Operators

Name	Description
$currentDate	⇒ Sets the value of a field to current date, either as a Date or a Timestamp. ⇒ **{$currentDate: {<field1>: <typeSpecification1>, ...}}** ⇒ <typeSpecification> can be either: χ a boolean true to set the field value to the current date as a Date, or χ a document { $type: "timestamp" } or { $type: "date" } which explicitly specifies the type.

<table>
<tr><td></td><td>⇒ The operator is case-sensitive and accepts only the lowercase "timestamp" or the lowercase "date".
⇒ Update the lastModified field to the current date, the "cancellation.date" field to the current timestamp as well as update the status field to "D" and the "cancellation.reason" to "user request".
 
</td></tr>
<tr><td>$min</td><td>⇒ Only updates the field if the specified value is less than the existing field value.
⇒ {$min: {<field1>: <value1>, ...}}
⇒ If the field does not exist, the $min operator sets the field to the specified value.
⇒ For comparisons between values of different types, such as a number and a null, $min uses the BSON comparison order.
⇒ Compute the minimum amount and minimum quantity for each grouping.
 
</td></tr>
<tr><td>$max</td><td>⇒ Only updates the field if the specified value is greater than the existing field value.
⇒ {$max: {<field1>: <value1>, ...}}
⇒ The $max operator updates the value of the field to a specified value if the specified value is greater than the current value of the field.
⇒ The $max operator can compare values of different types, using the BSON comparison order.
⇒ Compute the maximum amount and maximum quantity for each grouping.
 
</td></tr>
<tr><td>$mul</td><td>⇒ Multiplies the value of the field by the specified amount.
⇒ {$mul: {field: <number>}}</td></tr>
</table>

⇒ If the field does not exist in a document, $mul creates the field and sets the value to zero of the same numeric type as the multiplier.

⇒

	32-bit Integer	64-bit Integer	Float
32-bit Integer	32-bit or 64-bit Integer	64-bit Integer	Float
64-bit Integer	64-bit Integer	64-bit Integer	Float
Float	Float	Float	Float

$set

⇒ Sets the value of a field in a document.

⇒ **{$set: {<field1>: <value1>, …}}**

⇒ If the field does not exist, $set will add a new field with the specified value, provided that the new field does not violate a type constraint.

⇒ For the document matching the criteria _id equal to 100, update the value of the quantity field, details field, and the tags field.

```
> db.prod.find()
{ "_id" : 100, "sku" : "abc123", "quantity" : 250, "instock" : true, "reorder" : false, "details" : { "model" : "14Q2"
, "make" : "xyz" }, "tags" : [ "apparel", "clothing" ], "ratings" : [ { "by" : "ijk", "rating" : 4 } ] }
{ "_id" : 110, "sku" : "pqr567", "quantity" : 150, "instock" : true, "reorder" : false, "details" : { "model" : "18Q2"
, "make" : "xyz" }, "tags" : [ "apparel", "clothing" ], "ratings" : [ { "by" : "ijk", "rating" : 5 } ] }
> db.prod.update({ _id: 100 },{ $set:{quantity: 500,details: { model: "14Q3", make: "xyz" },tags: [ "coats", "outerwea
r", "clothing" ]}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.prod.find()
{ "_id" : 100, "sku" : "abc123", "quantity" : 500, "instock" : true, "reorder" : false, "details" : { "model" : "14Q3"
, "make" : "xyz" }, "tags" : [ "coats", "outerwear", "clothing" ], "ratings" : [ { "by" : "ijk", "rating" : 4 } ] }
{ "_id" : 110, "sku" : "pqr567", "quantity" : 150, "instock" : true, "reorder" : false, "details" : { "model" : "18Q2"
, "make" : "xyz" }, "tags" : [ "apparel", "clothing" ], "ratings" : [ { "by" : "ijk", "rating" : 5 } ] }
```

$setOnInsert

⇒ Sets the value of a field if an update results in an insert of a document. Has no effect on update operations that modify existing documents.

⇒ **{$setOnInsert: {<field1>: <value1>, …}}**

```
> db.prod.find()
{ "_id" : 100, "sku" : "abc123", "quantity" : 500, "instock" : true, "reorder" : false, "details" : { "model" : "14Q3"
, "make" : "xyz" }, "tags" : [ "coats", "outerwear", "clothing" ], "ratings" : [ { "by" : "ijk", "rating" : 4 } ] }
{ "_id" : 110, "sku" : "pqr567", "quantity" : 150, "instock" : true, "reorder" : false, "details" : { "model" : "18Q2"
, "make" : "xyz" }, "tags" : [ "apparel", "clothing" ], "ratings" : [ { "by" : "ijk", "rating" : 5 } ] }
> db.prod.update(
...   { _id: 1 },
...   {
...     $set: { sku: "xyz123" },
...     $setOnInsert: { quantity: 100 }
...   },
...   { upsert: true }
... )
WriteResult({ "nMatched" : 0, "nUpserted" : 1, "nModified" : 0, "_id" : 1 })
> db.prod.find()
{ "_id" : 100, "sku" : "abc123", "quantity" : 500, "instock" : true, "reorder" : false, "details" : { "model" : "14Q3"
, "make" : "xyz" }, "tags" : [ "coats", "outerwear", "clothing" ], "ratings" : [ { "by" : "ijk", "rating" : 4 } ] }
{ "_id" : 110, "sku" : "pqr567", "quantity" : 150, "instock" : true, "reorder" : false, "details" : { "model" : "18Q2"
, "make" : "xyz" }, "tags" : [ "apparel", "clothing" ], "ratings" : [ { "by" : "ijk", "rating" : 5 } ] }
{ "_id" : 1, "quantity" : 100, "sku" : "xyz123" }
```

$unset

⇒ Removes the specified field from a document.

⇒ **{$unset: {<field1>: "", …}}**

	⇒ If the field does not exist, then $unset does nothing (i.e. no operation). ⇒ Delete the purqty field from the document for _id : 1

```
> db.mycol.find()
{ "_id" : 1, "description" : "item1", "op_stock" : 100, "purqty" : 100 }
> db.mycol.update( { _id: 1 },{ $unset: {"purqty": ""}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.mycol.find()
{ "_id" : 1, "description" : "item1", "op_stock" : 100 }
```

CRUD Commands – Update Operators on Arrays

Name	Description
$	⇒ The positional $ operator identifies an element in an array to update without explicitly specifying the position of the element in the array. ⇒ **{"<array>.$" : value }** ⇒ Update the value of the std field in the first embedded document that has grade field with a value less than or equal to 90 and a mean field with a value greater than 80

```
> db.students.find()
{ "_id" : 1, "grades" : [ 85, 80, 80 ] }
{ "_id" : 2, "grades" : [ 88, 90, 92 ] }
{ "_id" : 3, "grades" : [ 85, 100, 90 ] }
> db.students.updateOne(
...     { _id: 1, grades: 80 },
...     { $set: { "grades.$" : 82 } }
... )
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.students.find()
{ "_id" : 1, "grades" : [ 85, 82, 80 ] }
{ "_id" : 2, "grades" : [ 88, 90, 92 ] }
{ "_id" : 3, "grades" : [ 85, 100, 90 ] }
```

$[]	⇒ The all positional operator $[] indicates that the update operator should modify all elements in the specified array field. ⇒ **{<update operator>: {"<array>.$[]" : value } }** ⇒ To increment all elements in the grades array by 10 for all documents in the collection

```
> db.students.find()
{ "_id" : 1, "grades" : [ 85, 82, 80 ] }
{ "_id" : 2, "grades" : [ 88, 90, 92 ] }
{ "_id" : 3, "grades" : [ 85, 100, 90 ] }
> db.students.update(
...     { },
...     { $inc: { "grades.$[]": 10 } },
...     { multi: true }
... )
WriteResult({ "nMatched" : 3, "nUpserted" : 0, "nModified" : 3 })
> db.students.find()
{ "_id" : 1, "grades" : [ 95, 92, 90 ] }
{ "_id" : 2, "grades" : [ 98, 100, 102 ] }
{ "_id" : 3, "grades" : [ 95, 110, 100 ] }
```

$[<identifier>]

⇒ The filtered positional operator $[<identifier>] identifies the array elements that match the arrayFilters conditions for an update operation.

⇒ **{<update operator>:**
{"<array>.$[<identifier>]" : value } },
{arrayFilters: [{<identifier>: <condition>}}]}

⇒ To increment all elements in the grades array by 10 for all documents in the collection

```
> db.students.find()
{ "_id" : 1, "grades" : [ 95, 92, 90 ] }
{ "_id" : 2, "grades" : [ 98, 100, 102 ] }
{ "_id" : 3, "grades" : [ 95, 110, 100 ] }
> db.students.update(
...     { },
...     { $set: { "grades.$[element]" : 100 } },
...     { multi: true,
...       arrayFilters: [ { "element": { $gte: 100 } } ]
...     }
... )
WriteResult({ "nMatched" : 3, "nUpserted" : 0, "nModified" : 2 })
> db.students.find()
{ "_id" : 1, "grades" : [ 95, 92, 90 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100 ] }
```

$addToSet

⇒ The $addToSet operator adds a value to an array unless the value is already present, in which case $addToSet does nothing to that array.

⇒ **{$addToSet: {<field1>: <value1>, ...}}**

⇒ Add the element "accessories" to the tags array since "accessories" does not exist in the array

```
> db.createCollection("inventory")
{ "ok" : 1 }
> db.inventory.insert({ _id: 1, item: "polarizing_filter", tags: [ "electronics", "camera" ]})
WriteResult({ "nInserted" : 1 })
> db.inventory.find()
{ "_id" : 1, "item" : "polarizing_filter", "tags" : [ "electronics", "camera" ] }
> db.inventory.update(
...    { _id: 1 },
...    { $addToSet: { tags: "accessories" } }
... )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.inventory.find()
{ "_id" : 1, "item" : "polarizing_filter", "tags" : [ "electronics", "camera", "accessories" ] }
```

```
> db.inventory.find()
{ "_id" : 1, "item" : "polarizing_filter", "tags" : [ "electronics", "camera", "accessories" ] }
{ "_id" : 2, "item" : "cable", "tags" : [ "electronics", "supplies" ] }
> db.inventory.update(
...    { _id: 2 },
...    { $addToSet: { tags: { $each: [ "camera", "electronics", "accessories" ] } } }
...  )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.inventory.find()
{ "_id" : 1, "item" : "polarizing_filter", "tags" : [ "electronics", "camera", "accessories" ] }
{ "_id" : 2, "item" : "cable", "tags" : [ "electronics", "supplies", "camera", "accessories" ] }
```

$pop

⇒ The $pop operator removes the first or last element of an array.

⇒ Pass $pop a value of -1 to remove the first element of an array and 1 to remove the last element in an array.

⇒ **{$pop: {<field>: <-1 | 1>, ...}} {$addToSet: {<field1>: <value1>, ...}}**

⇒ Remove the first element in the grades array

```
> db.students.find()
{ "_id" : 1, "grades" : [ 95, 92, 90 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100 ] }
> db.students.update( { _id: 1 }, { $pop: { grades: -1 } } )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.students.find()
{ "_id" : 1, "grades" : [ 92, 90 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100 ] }
```

$push

⇒ The $push operator appends a specified value to an array.

⇒ **{$push: {<field1>: <value1>, ...}}**

⇒ Appends 89 to the grades array

```
> db.students.find()
{ "_id" : 1, "grades" : [ 92, 90 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100 ] }
> db.students.update(
...     { _id: 1 },
...     { $push: { grades: 89 } }
... )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.students.find()
{ "_id" : 1, "grades" : [ 92, 90, 89 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100 ] }
```

```
> db.students.update(
...     { _id: 3 },
...     { $push: { grades: { $each: [ 90, 92, 85 ] } } }
... )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.students.find()
{ "_id" : 1, "grades" : [ 92, 90, 89 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100, 90, 92, 85 ] }
```

$pull

⇒ The $pull operator removes from an existing array all instances of a value or values that match a specified condition.

⇒ **{$pull: {<field1>: <value|condition>, <field2>: <value|condition>, ...}}**

⇒ Remove the arrays that have elements lesser than 95

```
> db.students.find()
{ "_id" : 1, "grades" : [ 92, 90, 89 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100, 90, 92, 85 ] }
> db.students.update( { _id: 1 }, { $pull: { grades: { $lte: 95 } } } )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.students.find()
{ "_id" : 1, "grades" : [ ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100, 90, 92, 85 ] }
```

$pullAll

⇒ The $pullAll operator removes all instances of the specified values from an existing array.

⇒ Unlike the $pull operator that removes elements by specifying a query, $pullAll removes elements that match the listed values.

⇒ **{$pullAll: {<field1>: [<value1>, <value2> …], … }}**

⇒ Remove the instances of all value 75, 80 from the grades array

```
> db.students.find()
{ "_id" : 1, "grades" : [ ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100, 90, 92, 85 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 5, "grades" : [ 98, 90, 92 ] }
{ "_id" : 6, "grades" : [ 85, 100, 80 ] }
{ "_id" : 7, "grades" : [ 75, 80, 79 ] }
```

```
> db.students.update({_id:7}, { $pullAll: { grades: [ 75, 80 ] } } )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.students.find()
{ "_id" : 1, "grades" : [ ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100, 90, 92, 85 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 5, "grades" : [ 98, 90, 92 ] }
{ "_id" : 6, "grades" : [ 85, 100, 80 ] }
{ "_id" : 7, "grades" : [ 79 ] }
```

CRUD Commands – Update Modifiers

Name	Description
$each	⇒ The $each modifier is available for use with the $addToSet operator and the $push operator. ⇒ Use with the $addToSet operator to add multiple values to an array <field> if the values do not exist in the <field>. ⇒ **{$addToSet: {<field>: {$each: [<value1>, <value2> …]}}}** ⇒ Use with the $push operator to append multiple values to an array <field>. ⇒ **{$push: {<field>: {$each: [<value1>, <value2> …]}}}** ⇒ Append each element of [90, 92, 85] to the scores array for the document where the _id field is 1

```
> db.students.find()
{ "_id" : 1, "grades" : [ ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100, 90, 92, 85 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 5, "grades" : [ 98, 90, 92 ] }
{ "_id" : 6, "grades" : [ 85, 100, 80 ] }
{ "_id" : 7, "grades" : [ 79 ] }
> db.students.update(
...     { _id: 1 },
...     { $push: { grades: { $each: [ 90, 92, 85 ] } } }
... )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.students.find()
{ "_id" : 1, "grades" : [ 90, 92, 85 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100, 90, 92, 85 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 5, "grades" : [ 98, 90, 92 ] }
{ "_id" : 6, "grades" : [ 85, 100, 80 ] }
{ "_id" : 7, "grades" : [ 79 ] }
```

⇒ Add multiple elements to the tags array

```
> db.inventory.update(
...    { _id: 2 },
...    { $addToSet: { tags: { $each: [ "tripod", "electronics", "accessories" ] } } }
... )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.inventory.find()
{ "_id" : 1, "item" : "polarizing_filter", "tags" : [ "electronics", "camera", "accessories" ] }
{ "_id" : 2, "item" : "cable", "tags" : [ "electronics", "supplies", "camera", "accessories", "tripod" ] }
```

$position

⇒ The $position modifier specifies the location in the array at which the $push operator inserts elements.

⇒ Without the $position modifier, the $push operator inserts elements to the end of the array.

⇒ To use the $position modifier, it must appear with the $each modifier.

```
{
  $push: {
    <field>: {
      $each: [ <value1>, <value2>, ... ],
      $position: <num>
    }
  }
}
```

```
> db.students.find()
{ "_id" : 1, "grades" : [ 90, 92, 85 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100, 90, 92, 85 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 5, "grades" : [ 98, 90, 92 ] }
{ "_id" : 6, "grades" : [ 85, 100, 80 ] }
```

⇒ Update the grades field to add the elements 50, 60 and 70 to the beginning of the array

```
> db.students.update({ _id: 6 }, {$push: {grades: {$each: [ 50, 60, 70 ], $position: 0}}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.students.find()
{ "_id" : 1, "grades" : [ 90, 92, 85 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100, 90, 92, 85 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 5, "grades" : [ 98, 90, 92 ] }
{ "_id" : 6, "grades" : [ 50, 60, 70, 85, 100, 80 ] }
```

⇒ Update the grades field to add the elements 90 and 80 at the array index of 2

```
> db.students.update({ _id: 1 },{$push: {grades: {$each: [ 90, 80 ],$position: -2}}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.students.find()
{ "_id" : 1, "grades" : [ 90, 90, 80, 92, 85 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100, 90, 92, 85 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 5, "grades" : [ 98, 90, 92 ] }
{ "_id" : 6, "grades" : [ 50, 60, 70, 85, 100, 80 ] }
```

$slice

⇒ The $slice modifier limits the number of array elements during a $push operation.

⇒ To use the $slice modifier, it must appear with the $each modifier.

⇒ Can pass an empty array [] to the $each modifier such that only the $slice modifier has an effect.

```
{
  $push: {
     <field>: {
       $each: [ <value1>, <value2>, ... ],
       $slice: <num>
     }
  }
}
```

⇒ Add new elements to the grades array and use the $slice modifier to trim the array to the last five elements.

```
> db.students.find()
{ "_id" : 1, "grades" : [ 90, 90, 80, 92, 85 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100, 90, 92, 85 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 5, "grades" : [ 98, 90, 92 ] }
{ "_id" : 6, "grades" : [ 50, 60, 70, 85, 100, 80 ] }
> db.students.update({ _id: 1 },{$push: {grades: {$each: [ 80, 78, 86 ],$slice: -5}}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.students.find()
{ "_id" : 1, "grades" : [ 92, 85, 80, 78, 86 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100, 90, 92, 85 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 5, "grades" : [ 98, 90, 92 ] }
{ "_id" : 6, "grades" : [ 50, 60, 70, 85, 100, 80 ] }
```

⇒ Add new elements to the grades array and use the $slice modifier to trim the array from first three elements.

```
> db.students.update({ _id: 6 },{$push: {grades: {$each: [ 100, 20 ],$slice: 3}}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.students.find()
{ "_id" : 1, "grades" : [ 92, 85, 80, 78, 86 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100, 90, 92, 85 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 5, "grades" : [ 98, 90, 92 ] }
{ "_id" : 6, "grades" : [ 50, 60, 70 ] }
```

⇒

$sort

⇒ The $sort modifier orders the elements of an array during a $push operation.

⇒ To use the $sort modifier, it must appear with the $each modifier.

```
{
  $push: {
     <field>: {
       $each: [ <value1>, <value2>, ... ],
       $sort: <sort specification>
     }
  }
}
```

```
> db.students.update(
...     { _id: 2 },
...     { $push: { grades: { $each: [ 40, 60 ], $sort: 1 } } }
... )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.students.find()
{ "_id" : 1, "grades" : [ 92, 85, 80, 78, 86 ] }
{ "_id" : 2, "grades" : [ 40, 60, 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100, 90, 92, 85 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 5, "grades" : [ 98, 90, 92 ] }
{ "_id" : 6, "grades" : [ 50, 60, 70 ] }
```

⇒ Push the grades in sorted order and sort the grades in descending order.

```
> db.students.update({ _id: 3 },{ $push: { grades: { $each: [ ], $sort: -1}}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.students.find()
{ "_id" : 1, "grades" : [ 92, 85, 80, 78, 86 ] }
{ "_id" : 2, "grades" : [ 40, 60, 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 100, 100, 95, 92, 90, 85 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 5, "grades" : [ 98, 90, 92 ] }
{ "_id" : 6, "grades" : [ 50, 60, 70 ] }
```

CRUD Commands – Delete

- Remove all documents from a collection.
  - **db.<collection_name>.remove({})**
- Remove all documents that match a condition
  - **db.<collection_name>.remove({ field_name: { query } })**
- Remove a single document that matches a condition
  - Call the remove method with the query criteria and the justOne parameter set to true or 1.
    - Delete all grades arrays that have at least one value that is greater than 95

```
> db.students.find()
{ "_id" : 1, "grades" : [ 92, 85, 80, 78, 86 ] }
{ "_id" : 2, "grades" : [ 40, 60, 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 100, 100, 95, 92, 90, 85 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 5, "grades" : [ 98, 90, 92 ] }
{ "_id" : 6, "grades" : [ 50, 60, 70 ] }
> db.students.remove( { grades: { $gt: 95 } } )
WriteResult({ "nRemoved" : 3 })
> db.students.find()
{ "_id" : 1, "grades" : [ 92, 85, 80, 78, 86 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 6, "grades" : [ 50, 60, 70 ] }
```

- Removes a single document from a collection
  - **db.<collection_name>.deleteOne(<filter>)**
    - Use a field that is part of a unique index such as _id for precise deletions.
- Removes all documents that match the filter from a collection.
  - **db.<collection_name>.deleteMany(<filter>)**
    - Delete the row that contains id=1

```
> db.students.find()
{ "_id" : 1, "grades" : [ 92, 85, 80, 78, 86 ] }
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 6, "grades" : [ 50, 60, 70 ] }
> db.students.deleteOne({_id: 1})
{ "acknowledged" : true, "deletedCount" : 1 }
> db.students.find()
{ "_id" : 4, "grades" : [ 85, 80, 80 ] }
{ "_id" : 6, "grades" : [ 50, 60, 70 ] }
```

    - Delete all rows that contain camera in its tags.

```
> db.inventory.find()
{ "_id" : 1, "item" : "polarizing_filter", "tags" : [ "electronics", "camera", "accessories" ] }
{ "_id" : 2, "item" : "cable", "tags" : [ "electronics", "supplies", "camera", "accessories", "tripod" ] }
{ "_id" : 3, "item" : "switch", "tags" : [ "electronics", "accessories" ] }
{ "_id" : 4, "item" : "modem", "tags" : [ "electronics", "accessories", "internet" ] }
> db.inventory.deleteMany({tags:"camera"})
{ "acknowledged" : true, "deletedCount" : 2 }
> db.inventory.find()
{ "_id" : 3, "item" : "switch", "tags" : [ "electronics", "accessories" ] }
{ "_id" : 4, "item" : "modem", "tags" : [ "electronics", "accessories", "internet" ] }
```

- Delete a single document based on the filter and sort criteria, returning the deleted document.
  - **db.<collection_name>.findOneAndDelete(filter, options)**
    - Find the first document where name is 'M. MMM' and deletes it

```
> db.details.find()
{ "_id" : 1, "name" : "A. AAA", "assignment" : 5, "points" : 24 }
{ "_id" : 2, "name" : "B. BBB", "assignment" : 3, "points" : 22 }
{ "_id" : 3, "name" : "M. MMM", "assignment" : 5, "points" : 30 }
{ "_id" : 4, "name" : "R. RRR", "assignment" : 2, "points" : 12 }
{ "_id" : 5, "name" : "A. DDD", "assignment" : 2, "points" : 14 }
{ "_id" : 6, "name" : "R. SSS", "assignment" : 1, "points" : 10 }
> db.details.findOneAndDelete({ "name" : "M. MMM" })
{ "_id" : 3, "name" : "M. MMM", "assignment" : 5, "points" : 30 }
> db.details.find()
{ "_id" : 1, "name" : "A. AAA", "assignment" : 5, "points" : 24 }
{ "_id" : 2, "name" : "B. BBB", "assignment" : 3, "points" : 22 }
{ "_id" : 4, "name" : "R. RRR", "assignment" : 2, "points" : 12 }
{ "_id" : 5, "name" : "A. DDD", "assignment" : 2, "points" : 14 }
{ "_id" : 6, "name" : "R. SSS", "assignment" : 1, "points" : 10 }
```

CRUD Commands – Modify

- Modify and return a single document.
  - **db.<collection_name>.findAndModify(document)**
    - Increment the field 'points' based on a query.

```
> db.details.findAndModify({query:{_id:1}, update: {$inc: {points: 1}}, upsert: true})
{ "_id" : 1, "name" : "A. AAA", "assignment" : 5, "points" : 24 }
> db.details.find()
{ "_id" : 1, "name" : "A. AAA", "assignment" : 5, "points" : 25 }
{ "_id" : 2, "name" : "B. BBB", "assignment" : 3, "points" : 22 }
{ "_id" : 4, "name" : "R. RRR", "assignment" : 2, "points" : 12 }
{ "_id" : 5, "name" : "A. DDD", "assignment" : 2, "points" : 14 }
{ "_id" : 6, "name" : "R. SSS", "assignment" : 1, "points" : 10 }
```

MongoDB Commands

- Perform multiple write operations with controls for order of execution.
  - **db.collection.bulkWrite([<operation 1>, <operation 2>, ...])**
    - By default, operations are executed in order.

Variations in Insert

- Create documents using JavaScript operations
  - **d1={_id: 1, name: "Lekha", city: "Bangalore", age: 18, marks: [90, 90, 90]}**
  - **d2={_id: 2, name: "Manish", city: "Mangalore", age: 18, marks: [99, 90, 99]}**
  - **db.Semfour.insert(d1)**
  - **db.Semfour.insert(d2)**
  - **db.Semfour.find()**
    - { "_id" : 1, "name" : "Lekha", "city" : "Bangalore", "age" : 18, "marks" : [90, 90, 90] }
    - { "_id" : 2, "name" : "Manish", "city" : "Mangalore", "age" : 18, "marks" : [99, 90, 99] }
- Insert an array of documents

- **var MyDocument=[**
  {_id:1, name:"Lekha", age:18, city: "Bangalore", marks:[90, 90, 90]},
  {_id:2, name:"Manish", age:18, city: "Mangalore", marks:[99, 90, 90]},
  {_id:3, name:"Isha", age: 17, city: "Bangalore", marks: [99,98,97]}]
- **db.Semfour.insert(MyDocument)**
- Output
  - BulkWriteResult({"writeErrors" : [], "writeConcernErrors" : [], "nInserted" : 3, "nUpserted" : 0, "nMatched" : 0, "nModified" : 0, "nRemoved" : 0, "upserted" : [] })

DB Commands – aggregate

- Calculates aggregate values for the data in a collection or a view.
  - db.<collection_name>.aggregate(pipeline, options)
    - db.Semfour.aggregate([{$match: {city : "Bangalore"}}, {$group: {_id: "$_id", total: {$sum: "$age"}}}, {$sort: {total: -1}}])
- Select documents with status equal to "A", group the matching documents by the cust_id field and calculate the total for each cust_id field from the sum of the amount field, and sort the results by the total field in descending order.

```
> db.orders.find()
{ "_id" : 1, "cust_id" : "abc1", "ord_date" : ISODate("2012-11-02T17:04:11.102Z"), "status" : "A", "amount" : 50 }
{ "_id" : 2, "cust_id" : "xyz1", "ord_date" : ISODate("2013-10-01T17:04:11.102Z"), "status" : "A", "amount" : 100 }
{ "_id" : 3, "cust_id" : "xyz1", "ord_date" : ISODate("2013-10-12T17:04:11.102Z"), "status" : "D", "amount" : 25 }
{ "_id" : 4, "cust_id" : "xyz1", "ord_date" : ISODate("2013-10-11T17:04:11.102Z"), "status" : "D", "amount" : 125 }
{ "_id" : 5, "cust_id" : "abc1", "ord_date" : ISODate("2013-11-12T17:04:11.102Z"), "status" : "A", "amount" : 25 }
> db.orders.aggregate([
...                     { $match: { status: "A" } },
...                     { $group: { _id: "$cust_id", total: { $sum: "$amount" } } },
...                     { $sort: { total: -1 } }
...                   ])
{ "_id" : "xyz1", "total" : 100 }
{ "_id" : "abc1", "total" : 75 }
```

DB Commands - $match

- Filters the documents to pass only the documents that match the specified condition(s) to the next pipeline stage.
- The $match stage has the following prototype form:
  - { $match: { <query> } }
- $match takes a document that specifies the query conditions.
  - Perform a simple equality match.

```
> db.articles.find()
{ "_id" : ObjectId("512bc95fe835e68f199c8686"), "author" : "dave", "score" : 80, "views" : 100 }
{ "_id" : ObjectId("512bc962e835e68f199c8687"), "author" : "dave", "score" : 85, "views" : 521 }
{ "_id" : ObjectId("55f5a192d4bede9ac365b257"), "author" : "ahn", "score" : 60, "views" : 1000 }
{ "_id" : ObjectId("55f5a192d4bede9ac365b258"), "author" : "li", "score" : 55, "views" : 5000 }
{ "_id" : ObjectId("55f5a1d3d4bede9ac365b259"), "author" : "annT", "score" : 60, "views" : 50 }
{ "_id" : ObjectId("55f5a1d3d4bede9ac365b25a"), "author" : "li", "score" : 94, "views" : 999 }
{ "_id" : ObjectId("55f5a1d3d4bede9ac365b25b"), "author" : "ty", "score" : 95, "views" : 1000 }
> db.articles.aggregate(
...     [ { $match : { author : "dave" } } ]
... );
{ "_id" : ObjectId("512bc95fe835e68f199c8686"), "author" : "dave", "score" : 80, "views" : 100 }
{ "_id" : ObjectId("512bc962e835e68f199c8687"), "author" : "dave", "score" : 85, "views" : 521 }
```

- Select the documents where either the score is greater than 70 and less than 90 or the views is greater than or equal to 1000 and perform a count.

```
> db.articles.aggregate( [
...   { $match: { $or: [ { score: { $gt: 70, $lt: 90 } }, { views: { $gte: 1000 } } ] } },
...   { $group: { _id: null, count: { $sum: 1 } } }
... ] );
{ "_id" : null, "count" : 5 }
```

DB Commands - $group

- Groups documents by some specified expression and outputs to the next stage a document for each distinct grouping.
- The output documents contain an _id field which contains the distinct group by key.
- The output documents can also contain computed fields that hold the values of some accumulator expression grouped by the $group's _id field.
- $group does not order its output documents.
    - { $group: { _id: <expression>, <field1>: { <accumulator1> : <expression1> }, ... } }
- The _id field is mandatory.
- Group the documents by the item to retrieve the distinct item values.

```
> db.sales.find()
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-03-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-03-01T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-03-15T09:00:00Z") }
{ "_id" : 4, "item" : "xyz", "price" : 5, "quantity" : 20, "date" : ISODate("2014-04-04T11:21:39.736Z") }
{ "_id" : 5, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-04-04T21:23:13.331Z") }
> db.sales.aggregate( [ { $group : { _id : "$item" } } ] )
{ "_id" : "xyz" }
{ "_id" : "jkl" }
{ "_id" : "abc" }
```

- Group the documents by the month, day, and year and calculates the total price and the average quantity as well as counts the documents per each group

```
> db.sales.find()
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-03-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-03-01T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-03-15T09:00:00Z") }
{ "_id" : 4, "item" : "xyz", "price" : 5, "quantity" : 20, "date" : ISODate("2014-04-04T11:21:39.736Z") }
{ "_id" : 5, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-04-04T21:23:13.331Z") }
> db.sales.aggregate([{
...     $group : { _id : { month: { $month: "$date" },
...                day: { $dayOfMonth: "$date" }, year: { $year: "$date" } },
...                totalPrice: { $sum: { $multiply: [ "$price", "$quantity" ] } },
...                averageQuantity: { $avg: "$quantity" }, count: { $sum: 1 }}}])
{ "_id" : { "month" : 4, "day" : 4, "year" : 2014 }, "totalPrice" : 200, "averageQuantity" : 15, "count" : 2 }
{ "_id" : { "month" : 3, "day" : 15, "year" : 2014 }, "totalPrice" : 50, "averageQuantity" : 10, "count" : 1 }
{ "_id" : { "month" : 3, "day" : 1, "year" : 2014 }, "totalPrice" : 40, "averageQuantity" : 1.5, "count" : 2 }
```

DB Commands – Count

- Returns the count of documents that would match a find() query for the collection or view.
    - db.<collection_name>.count(query, options)
        - The db.<collection_name>.count() method does not perform the find() operation but instead counts and returns the number of results that match a query.
        - The options can be

- limit – the maximum number of documents to count
- skip – the number of documents to skip before counting
- hint – an index name hint
- maxTimeMS – the maximum amount of time to allow the query to run

- Count the number of all documents

```
> db.sales.find()
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-03-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-03-01T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-03-15T09:00:00Z") }
{ "_id" : 4, "item" : "xyz", "price" : 5, "quantity" : 20, "date" : ISODate("2014-04-04T11:21:39.736Z") }
{ "_id" : 5, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-04-04T21:23:13.331Z") }
> db.sales.count()
5
```

- Count the number of the documents with the field ord_dt greater than new Date('01/01/2013')

```
> db.orders.find()
{ "_id" : 1, "cust_id" : "abc1", "ord_date" : ISODate("2012-11-02T17:04:11.102Z"), "status" : "A", "amount" : 50 }
{ "_id" : 2, "cust_id" : "xyz1", "ord_date" : ISODate("2013-10-01T17:04:11.102Z"), "status" : "A", "amount" : 100 }
{ "_id" : 3, "cust_id" : "xyz1", "ord_date" : ISODate("2013-10-12T17:04:11.102Z"), "status" : "D", "amount" : 25 }
{ "_id" : 4, "cust_id" : "xyz1", "ord_date" : ISODate("2013-10-11T17:04:11.102Z"), "status" : "D", "amount" : 125 }
{ "_id" : 5, "cust_id" : "abc1", "ord_date" : ISODate("2013-11-12T17:04:11.102Z"), "status" : "A", "amount" : 25 }
> db.orders.count( { ord_date: { $gt: new Date('01/11/2013') } } )
4
```

DB Command - $count

- Passes a document to the next stage that contains a count of the number of documents input to the stage.
  - { $count: <string> }
  - Exclude documents that have a score value of less than or equal to 80 to pass along the documents with score greater than 80 to the next stage and returns count of the remaining documents in the aggregation pipeline and assigns the value to a field called passing_scores.

```
> db.scores.find()
{ "_id" : 1, "subject" : "History", "score" : 88 }
{ "_id" : 2, "subject" : "History", "score" : 92 }
{ "_id" : 3, "subject" : "History", "score" : 97 }
{ "_id" : 4, "subject" : "History", "score" : 71 }
{ "_id" : 5, "subject" : "History", "score" : 79 }
{ "_id" : 6, "subject" : "History", "score" : 83 }
> db.scores.aggregate([{$match: {score: {$gt: 80}}},{$count: "passing_scores"}])
{ "passing_scores" : 4 }
```

DB Commands – distinct

- Finds the distinct values for a specified field across a single collection or view and return the results in an array.
  - db.<collection_name>.distinct(field, query, options)
  - Returns the distinct values for the field dept from all documents

```
> db.cloth.find()
{ "_id" : 1, "dept" : "A", "item" : { "sku" : "111", "color" : "red" }, "sizes" : [ "S", "M" ] }
{ "_id" : 2, "dept" : "A", "item" : { "sku" : "111", "color" : "blue" }, "sizes" : [ "M", "L" ] }
{ "_id" : 3, "dept" : "B", "item" : { "sku" : "222", "color" : "blue" }, "sizes" : "S" }
{ "_id" : 4, "dept" : "A", "item" : { "sku" : "333", "color" : "black" }, "sizes" : [ "S" ] }
> db.cloth.distinct( "dept" )
[ "A", "B" ]
```

- Embedded documents
  - Return the distinct values for the field sku, embedded in the item field

```
> db.cloth.distinct( "item.sku" )
[ "111", "222", "333" ]
```

- Return the distinct values for the field sizes from all documents in the cloth collection.

```
> db.cloth.find()
{ "_id" : 1, "dept" : "A", "item" : { "sku" : "111", "color" : "red" }, "sizes" : [ "S", "M" ] }
{ "_id" : 2, "dept" : "A", "item" : { "sku" : "111", "color" : "blue" }, "sizes" : [ "M", "L" ] }
{ "_id" : 3, "dept" : "B", "item" : { "sku" : "222", "color" : "blue" }, "sizes" : "S" }
{ "_id" : 4, "dept" : "A", "item" : { "sku" : "333", "color" : "black" }, "sizes" : [ "S" ] }
> db.cloth.distinct( "sizes" )
[ "M", "S", "L" ]
```

- Return the distinct values for the field sku, embedded in the item field, from the documents whose dept is equal to "A"

```
> db.cloth.find()
{ "_id" : 1, "dept" : "A", "item" : { "sku" : "111", "color" : "red" }, "sizes" : [ "S", "M" ] }
{ "_id" : 2, "dept" : "A", "item" : { "sku" : "111", "color" : "blue" }, "sizes" : [ "M", "L" ] }
{ "_id" : 3, "dept" : "B", "item" : { "sku" : "222", "color" : "blue" }, "sizes" : "S" }
{ "_id" : 4, "dept" : "A", "item" : { "sku" : "333", "color" : "black" }, "sizes" : [ "S" ] }
> db.cloth.distinct( "item.sku", { dept: "A" } )
[ "111", "333" ]
```

DB Commands – $substrBytes

- Return the substring of a string.
  - { $substrBytes: [<string expression>, <byte index>, <byte count>] }
    - Create a three byte menuCode from the name value

```
> db.food.find()
{ "_id" : 1, "name" : "apple" }
{ "_id" : 2, "name" : "banana" }
{ "_id" : 3, "name" : "éclair" }
{ "_id" : 4, "name" : "hamburger" }
{ "_id" : 5, "name" : "jalapeño" }
{ "_id" : 6, "name" : "pizza" }
{ "_id" : 7, "name" : "tacos" }
{ "_id" : 8, "name" : "□ □ sushi" }
> db.food.aggregate([{$project: {"name": 1,"menuCode": { $substrBytes: [ "$name", 0, 3]}}}])
{ "_id" : 1, "name" : "apple", "menuCode" : "app" }
{ "_id" : 2, "name" : "banana", "menuCode" : "ban" }
{ "_id" : 3, "name" : "éclair", "menuCode" : "éc" }
{ "_id" : 4, "name" : "hamburger", "menuCode" : "ham" }
{ "_id" : 5, "name" : "jalapeño", "menuCode" : "jal" }
{ "_id" : 6, "name" : "pizza", "menuCode" : "piz" }
{ "_id" : 7, "name" : "tacos", "menuCode" : "tac" }
{ "_id" : 8, "name" : "□ □ sushi", "menuCode" : "□ " }
```

- Separate the quarter value (containing only single byte US-ASCII characters) into a yearSubstring and a quarterSubstring.

```
> db.items.find()
{ "_id" : 1, "item" : "ABC1", "quarter" : "13Q1", "description" : "product 1" }
{ "_id" : 2, "item" : "ABC2", "quarter" : "13Q4", "description" : "product 2" }
{ "_id" : 3, "item" : "XYZ1", "quarter" : "14Q2", "description" : null }
> db.items.aggregate([{
...       $project: {item: 1, yearSubstring: { $substrBytes: [ "$quarter", 0, 2 ] },
...         quarterSubtring: {$substrBytes: [
...             "$quarter", 2, { $subtract: [ { $strLenBytes: "$quarter" }, 2 ] }
...           ]}}}])
{ "_id" : 1, "item" : "ABC1", "yearSubstring" : "13", "quarterSubtring" : "Q1" }
{ "_id" : 2, "item" : "ABC2", "yearSubstring" : "13", "quarterSubtring" : "Q4" }
{ "_id" : 3, "item" : "XYZ1", "yearSubstring" : "14", "quarterSubtring" : "Q2" }
```

DB Commands – $ceil

- Return the smallest integer greater than or equal to the specified number.
  - { $ceil: <number> }
    - Return both the original value and the ceiling value

```
> db.samp.find()
{ "_id" : 1, "value" : 9.25 }
{ "_id" : 2, "value" : 8.73 }
{ "_id" : 3, "value" : 4.32 }
{ "_id" : 4, "value" : -5.34 }
> db.samp.aggregate([{ $project: { value: 1, ceilingValue: { $ceil: "$value" } } }])
{ "_id" : 1, "value" : 9.25, "ceilingValue" : 10 }
{ "_id" : 2, "value" : 8.73, "ceilingValue" : 9 }
{ "_id" : 3, "value" : 4.32, "ceilingValue" : 5 }
{ "_id" : 4, "value" : -5.34, "ceilingValue" : -5 }
```

DB Commands – $cmp

- Compares two values and returns:
  - -1 if the first value is less than the second.
  - 1 if the first value is greater than the second.
  - 0 if the two values are equivalent.
    - { $cmp: [<expression1>, <expression2>] }
- Compare the qty value with 250

```
> db.ord.find()
{ "_id" : 1, "item" : "abc1", "description" : "product 1", "qty" : 300 }
{ "_id" : 2, "item" : "abc2", "description" : "product 2", "qty" : 200 }
{ "_id" : 3, "item" : "xyz1", "description" : "product 3", "qty" : 250 }
{ "_id" : 4, "item" : "VWZ1", "description" : "product 4", "qty" : 300 }
{ "_id" : 5, "item" : "VWZ2", "description" : "product 5", "qty" : 180 }
> db.ord.aggregate([{
...        $project:{item: 1, qty: 1, cmpTo250: { $cmp: [ "$qty", 250 ] },_id: 0 }}])
{ "item" : "abc1", "qty" : 300, "cmpTo250" : 1 }
{ "item" : "abc2", "qty" : 200, "cmpTo250" : -1 }
{ "item" : "xyz1", "qty" : 250, "cmpTo250" : 0 }
{ "item" : "VWZ1", "qty" : 300, "cmpTo250" : 1 }
{ "item" : "VWZ2", "qty" : 180, "cmpTo250" : -1 }
```

DB Commands – $divide

- Divide one number by another and return the result.
  - Pass the arguments to $divide in an array.
  - { $divide: [<expression1>, <expression2>] }
- Divide the hours field by a literal 8 to compute the number of work days.

```
> db.plan.find()
{ "_id" : 1, "name" : "A", "hours" : 80, "resources" : 7 }
{ "_id" : 2, "name" : "B", "hours" : 40, "resources" : 4 }
{ "_id" : 3, "name" : "C", "hours" : 180, "resources" : 3 }
{ "_id" : 4, "name" : "D", "hours" : 45, "resources" : 5 }
{ "_id" : 5, "name" : "E", "hours" : 90, "resources" : 5 }
{ "_id" : 6, "name" : "F", "hours" : 30, "resources" : 4 }
{ "_id" : 7, "name" : "G", "hours" : 280, "resources" : 9 }
{ "_id" : 8, "name" : "H", "hours" : 55, "resources" : 6 }
> db.plan.aggregate([{ $project: { name: 1, workdays: { $divide: [ "$hours", 8 ] } } }])
{ "_id" : 1, "name" : "A", "workdays" : 10 }
{ "_id" : 2, "name" : "B", "workdays" : 5 }
{ "_id" : 3, "name" : "C", "workdays" : 22.5 }
{ "_id" : 4, "name" : "D", "workdays" : 5.625 }
{ "_id" : 5, "name" : "E", "workdays" : 11.25 }
{ "_id" : 6, "name" : "F", "workdays" : 3.75 }
{ "_id" : 7, "name" : "G", "workdays" : 35 }
{ "_id" : 8, "name" : "H", "workdays" : 6.875 }
```

DB Commands – $project

- Pass along the documents with only the specified fields to the next stage in the pipeline.
  - This may be the existing fields from the input documents or newly computed fields.
  - { $project: { <specifications> } }
    - The specification for $project command contain the inclusion of fields, the suppression of the _id field, the addition of new fields, and the resetting the values of existing fields.
- Include only the title field in the embedded document in the stop field.

```
> db.Bmarks.find()
{ "_id" : 1, "user" : "1234", "stop" : { "title" : "book1", "author" : "xyz", "page" : 32 } }
{ "_id" : 2, "user" : "7890", "stop" : [ { "title" : "book2", "author" : "abc", "page" : 5 }, { "title" : "book3", "au
thor" : "ijk", "page" : 100 } ] }
> db.Bmarks.aggregate( [ { $project: { "stop.title": 1 } } ] )
{ "_id" : 1, "stop" : { "title" : "book1" } }
{ "_id" : 2, "stop" : [ { "title" : "book2" }, { "title" : "book3" } ] }
```

```
> db.Bmarks.find()
{ "_id" : 1, "user" : "1234", "stop" : { "title" : "book1", "author" : "xyz", "page" : 32 } }
{ "_id" : 2, "user" : "7890", "stop" : [ { "title" : "book2", "author" : "abc", "page" : 5 }, { "title" : "book3", "au
thor" : "ijk", "page" : 100 } ] }
> db.Bmarks.aggregate( [ { $project: { stop: { title: 1 } } } ] )
{ "_id" : 1, "stop" : { "title" : "book1" } }
{ "_id" : 2, "stop" : [ { "title" : "book2" }, { "title" : "book3" } ] }
```

- Add new fields isbn, lastName, and copiesSold

```
> db.Books.find()
{ "_id" : 1, "title" : "abc123", "isbn" : "0001122223334", "author" : { "last" : "zzz", "first" : "aaa" }, "copies" :
5, "lastModified" : "2016-07-28" }
{ "_id" : 2, "title" : "Baked Goods", "isbn" : "9999999999999", "author" : { "last" : "xyz", "first" : "abc", "middle"
 : "" }, "copies" : 2, "lastModified" : "2017-07-21" }
{ "_id" : 3, "title" : "Ice Cream Cakes", "isbn" : "8888888888888", "author" : { "last" : "xyz", "first" : "abc", "mid
dle" : "mmm" }, "copies" : 5, "lastModified" : "2017-07-22" }
> db.Books.aggregate([{$project: {title: 1, isbn: {prefix: { $substr: [ "$isbn", 0, 3 ] },group: { $substr: [ "$isbn",
 3, 2 ] },publisher: { $substr: [ "$isbn", 5, 4 ] },title: { $substr: [ "$isbn", 9, 3 ] },checkDigit: { $substr: [ "$i
sbn", 12, 1] }},lastName: "$author.last",copiesSold: "$copies"}}])
{ "_id" : 1, "title" : "abc123", "isbn" : { "prefix" : "000", "group" : "11", "publisher" : "2222", "title" : "333", "
checkDigit" : "4" }, "lastName" : "zzz", "copiesSold" : 5 }
{ "_id" : 2, "title" : "Baked Goods", "isbn" : { "prefix" : "999", "group" : "99", "publisher" : "9999", "title" : "99
9", "checkDigit" : "9" }, "lastName" : "xyz", "copiesSold" : 2 }
{ "_id" : 3, "title" : "Ice Cream Cakes", "isbn" : { "prefix" : "888", "group" : "88", "publisher" : "8888", "title" :
 "888", "checkDigit" : "8" }, "lastName" : "xyz", "copiesSold" : 5 }
```

- Use the REMOVE variable to excludes the author.middle field only if it equals ""

```
> db.Books.aggregate( [
...    {
...       $project: {
...          title: 1,
...          "author.first": 1,
...          "author.last" : 1,
...          "author.middle": {
...             $cond: {
...                if: { $eq: [ "", "$author.middle" ] },
...                then: "$$REMOVE",
...                else: "$author.middle"
...             }
...          }
...       }
...    }
... ] )
{ "_id" : 1, "title" : "abc123", "author" : { "last" : "zzz", "first" : "aaa" } }
{ "_id" : 2, "title" : "Baked Goods", "author" : { "last" : "xyz", "first" : "abc" } }
{ "_id" : 3, "title" : "Ice Cream Cakes", "author" : { "last" : "xyz", "first" : "abc", "middle" : "mmm" } }
```

- Project the fields x and y as elements in a new field myArray.

```
> db.coll.find()
{ "_id" : ObjectId("55ad167f320c6be244eb3b95"), "x" : 1, "y" : 1 }
{ "_id" : ObjectId("56ad167f320c6be244eb3b95"), "x" : 2, "y" : 2 }
> db.coll.aggregate( [ { $project: { myArray: [ "$x", "$y" ] } } ] )
{ "_id" : ObjectId("55ad167f320c6be244eb3b95"), "myArray" : [ 1, 1 ] }
{ "_id" : ObjectId("56ad167f320c6be244eb3b95"), "myArray" : [ 2, 2 ] }
```

DB Commands – $out

- Create a new collection in the current database if the specified one does not already exist based on an aggregation.
    - { $out: "<output-collection>" }
        - It should be the last stage in the pipeline operator.
        - Pivot the data in the books collection to have titles grouped by authors and then write the results to the authors collection.

```
> db.books.aggregate([{ $group : { _id : "$author", books: { $push: "$title" } } },{ $out : "authors" }] )
> db.books.find()
{ "_id" : 8751, "title" : "The Banquet", "author" : "Dante", "copies" : 2 }
{ "_id" : 8752, "title" : "Divine Comedy", "author" : "Dante", "copies" : 1 }
{ "_id" : 8645, "title" : "Eclogues", "author" : "Dante", "copies" : 2 }
{ "_id" : 7000, "title" : "The Odyssey", "author" : "Homer", "copies" : 10 }
{ "_id" : 7020, "title" : "Iliad", "author" : "Homer", "copies" : 10 }
> db.authors.find()
{ "_id" : "Homer", "books" : [ "The Odyssey", "Iliad" ] }
{ "_id" : "Dante", "books" : [ "The Banquet", "Divine Comedy", "Eclogues" ] }
```

DB Commands – $filter

- Select a subset of an array to return based on the specified condition.
  - { $filter: { input: <array>, as: <string>, cond: <expression> } }
    - input - An expression that resolves to an array.
    - as - A name for the variable that represents each individual element of the input array.

    If no name is specified, the variable name defaults to this.
    - cond - Expression that resolves to a boolean value.

  Used to determine if an element should be included in the output array.

  Expression references each element of the input array individually with the variable name specified in as.
    - Returns an array with only those elements that match the condition.
    - The returned elements are in the original order.
- Filter the items array to only include documents that have a price greater than or equal to 100

```
> db.Saldat.find()
{ "_id" : 0, "items" : [ { "item_id" : 43, "quantity" : 2, "price" : 10 }, { "item_id" : 2, "quantity" : 1, "price" :
240 } ] }
{ "_id" : 1, "items" : [ { "item_id" : 23, "quantity" : 3, "price" : 110 }, { "item_id" : 103, "quantity" : 4, "price"
 : 5 }, { "item_id" : 38, "quantity" : 1, "price" : 300 } ] }
{ "_id" : 2, "items" : [ { "item_id" : 4, "quantity" : 1, "price" : 23 } ] }
> db.Saldat.aggregate([{$project: {items: {$filter: {input: "$items", as: "item", cond: { $gte: [ "$$item.price", 100
] }}}}}])
{ "_id" : 0, "items" : [ { "item_id" : 2, "quantity" : 1, "price" : 240 } ] }
{ "_id" : 1, "items" : [ { "item_id" : 23, "quantity" : 3, "price" : 110 }, { "item_id" : 38, "quantity" : 1, "price"
: 300 } ] }
{ "_id" : 2, "items" : [ ] }
```

DB Commands – Contd..

- Return the specific option for a date
  - { $dayOfMonth: <dateExpression> }
  - year: { $year: "$<date expression>" },
  - month: { $month: "$<date expression>" },
  - day: { $dayOfMonth: "$<date expression>" },
  - hour: { $hour: "$<date expression>" },
  - minutes: { $minute: "$<date expression>" },
  - seconds: { $second: "$<date expression>" },
  - milliseconds: { $millisecond: "$<date expression>" },
  - dayOfYear: { $dayOfYear: "$<date expression>" },
  - dayOfWeek: { $dayOfWeek: "$<date expression>" },

- week: { $week: "$<date expression>" }

Administrative Commands

- Rename an existing collection
  - db.adminCommand({ renameCollection: "<db_name>.<collection_name>", to: "<db>.<newcollection_name>" })
  - db.collection.renameCollection("<new_name>")
- Drop a database
  - use <database_name>
  - db.runCommand({dropDatabase: 1})
- Create a collection
  - db.runCommand({ create: "<collection_name", capped : <Boolean>, size: <bytes>})
- Drop a collection given the collection_name
  - db.<collection_name>. drop()

Embedded Data Model

- Also called as denormalized data model.
- Embedded data models allow applications to store related pieces of information in the same database record.
  - As a result, applications may need to issue fewer queries and updates to complete common operations.

Normalized Data Models

- Normalized data models describe relationships using references between documents.
- References provides more flexibility than embedding.
  - However, client-side applications must issue follow-up queries to resolve the references.
  - In other words, normalized data models can require more round trips to the server.