**What is MongoDB?**

**MongoDB** is a document-oriented NoSQL database used for high volume data storage. Instead of using **tables and rows** as in the traditional relational databases, MongoDB makes use of **collections and documents**.

Documents consist of key-value pairs which are the basic unit of data in MongoDB. Collections contain sets of documents and function which is the equivalent of relational database tables.

MongoDB is a database which came into light around the mid-2000s.

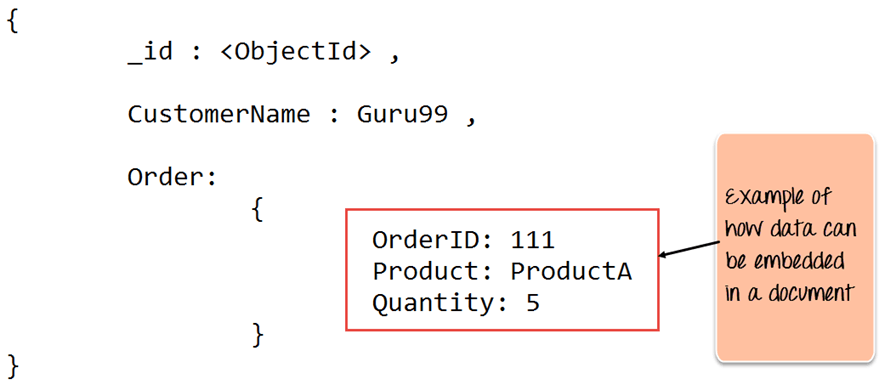
## MongoDB Features

1. Each database contains collections which in turn contains documents. Each document can be different with a varying number of fields. The size and content of each document can be different from each other.
2. The document structure is more in line with how developers construct their classes and objects in their respective programming languages. Developers will often say that their classes are not rows and columns but have a clear structure with key-value pairs.
3. The rows (or documents as called in MongoDB) doesn’t need to have a schema defined beforehand. Instead, the fields can be created on the fly.
4. The data model available within MongoDB allows you to represent hierarchical relationships, to store arrays, and other more complex structures more easily.
5. Scalability – The MongoDB environments are very scalable. Companies across the world have defined clusters with some of them running 100+ nodes with around millions of documents within the database

**MongoDB Example**

The below example shows how a document can be modeled in MongoDB.

1. The \_id field is added by MongoDB to uniquely identify the document in the collection.
2. What you can note is that the Order Data (OrderID, Product, and Quantity ) which in RDBMS will normally be stored in a separate table, while in MongoDB it is actually stored as an embedded document in the collection itself. This is one of the key differences in how data is modeled in MongoDB.



**Key Components of MongoDB Architecture**

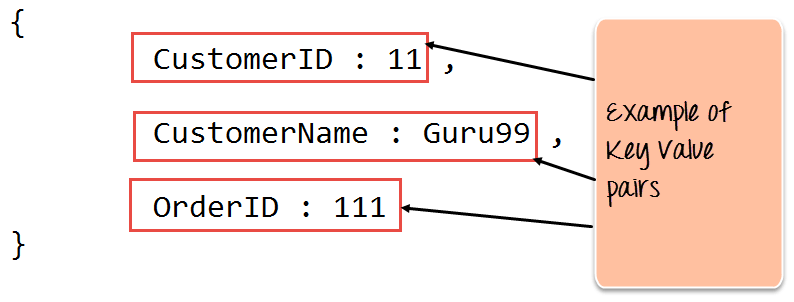
Below are a few of the common terms used in MongoDB

1. **\_id** – This is a field required in every MongoDB document. The \_id field represents a unique value in the MongoDB document. The \_id field is like the document’s primary key. If you create a new document without an \_id field, MongoDB will automatically create the field. So for example, if we see the example of the above customer table, Mongo DB will add a 24 digit unique identifier to each document in the collection.

| **\_Id** | **CustomerID** | **CustomerName** | **OrderID** |
| --- | --- | --- | --- |
| 563479cc8a8a4246bd27d784 | 11 | Guru99 | 111 | |
| 563479cc7a8a4246bd47d784 | 22 | Trevor Smith | 222 | |
| 563479cc9a8a4246bd57d784 | 33 | Nicole | 333 | |

1. **Collection** – This is a grouping of MongoDB documents. A collection is the equivalent of a table which is created in any other RDMS such as Oracle or MS SQL. A collection exists within a single database. As seen from the introduction collections don’t enforce any sort of structure.
2. **Cursor** – This is a pointer to the result set of a query. Clients can iterate through a cursor to retrieve results.
3. **Database** – This is a container for collections like in RDMS wherein it is a container for tables. Each database gets its own set of files on the file system. A MongoDB server can store multiple databases.
4. **Document** – A record in a MongoDB collection is basically called a document. The document, in turn, will consist of field name and values.
5. **Field** – A name-value pair in a document. A document has zero or more fields. Fields are analogous to columns in relational databases.

The following diagram shows an example of Fields with Key value pairs. So in the example below CustomerID and 11 is one of the key value pair’s defined in the document.



1. **JSON** – This is known as[JavaScript](https://www.guru99.com/interactive-javascript-tutorials.html)Object Notation. This is a human-readable, plain text format for expressing structured data. JSON is currently supported in many programming languages.

Just a quick note on the key difference between the \_id field and a normal collection field. The \_id field is used to uniquely identify the documents in a collection and is automatically added by MongoDB when the collection is created.

**Why Use MongoDB?**

Below are the few of the reasons as to why one should start using MongoDB

1. Document-oriented – Since MongoDB is a NoSQL type database, instead of having data in a relational type format, it stores the data in documents. This makes MongoDB very flexible and adaptable to real business world situation and requirements.
2. Ad hoc queries – MongoDB supports searching by field, range queries, and regular expression searches. Queries can be made to return specific fields within documents.
3. Indexing – Indexes can be created to improve the performance of searches within MongoDB. Any field in a MongoDB document can be indexed.
4. Replication – MongoDB can provide high availability with replica sets. A replica set consists of two or more mongo DB instances. Each replica set member may act in the role of the primary or secondary replica at any time. The primary replica is the main server which interacts with the client and performs all the read/write operations. The Secondary replicas maintain a copy of the data of the primary using built-in replication. When a primary replica fails, the replica set automatically switches over to the secondary and then it becomes the primary server.
5. Load balancing – MongoDB uses the concept of sharding to scale horizontally by splitting data across multiple MongoDB instances. MongoDB can run over multiple servers, balancing the load and/or duplicating data to keep the system up and running in case of hardware failure.

**SCHEMA:**

A Mongo schema defines the structure of the document, default values, validators etc.

**MODEL:**

A mongo model is a wrapper on the Mongo schema.

Mongo model provides an interface to the database for creating, querying, updating, deleting records, etc.

Data Modelling in MongoDB

As we have seen from the Introduction section, the data in MongoDB has a flexible schema. Unlike in[SQL](https://www.guru99.com/sql.html)databases, where you must have a table’s schema declared before inserting data, MongoDB’s collections do not enforce document structure. This sort of flexibility is what makes MongoDB so powerful.

## Difference between MongoDB & RDBMS

Below are some of the key term differences between MongoDB and RDBMS

| **RDBMS** | **MongoDB** | **Difference** |
| --- | --- | --- |
| Table | Collection | In RDBMS, the table contains the columns and rows which are used to store the data whereas, in MongoDB, this same structure is known as a collection. The collection contains documents which in turn contains Fields, which in turn are key-value pairs. |
| Row | Document | In RDBMS, the row represents a single, implicitly structured data item in a table. In MongoDB, the data is stored in documents. |
| Column | Field | In RDBMS, the column denotes a set of data values. These in MongoDB are known as Fields. |
| Joins | Embedded documents | In RDBMS, data is sometimes spread across various tables and in order to show a complete view of all data, a join is sometimes formed across tables to get the data. In MongoDB, the data is normally stored in a single collection, but separated by using Embedded documents. So there is no concept of joins in MongoDB. |

**MongoDB Commands:**

**Show Database**

To check your databases list, use the command **show dbs**.

>show dbs

local 0.78125GB

test 0.23012GB

## The use Command

MongoDB **use DATABASE\_NAME** is used to create database. The command will create a new database if it doesn't exist, otherwise it will return the existing database.

**use Databasename**

>use mydb

switched to db mydb

## The createCollection() Method

MongoDB **db.createCollection(name, options)** is used to create collection.

>use test

switched to db test

>db.createCollection("mycollection")

{ "ok" : 1 }

>

You can check the created collection by using the command **show collections**.

>show collections

mycollection

To check your currently selected database, use the command **db**

>db

mydb

## The dropDatabase() Method

MongoDB **db.dropDatabase()** command is used to drop a existing database

db.dropDatabase()

## The insertOne() Method

To insert data into MongoDB collection, you need to use MongoDB's **insertOne()** or **save()** method.

>db.COLLECTION\_NAME.insert(document)\

> db.users.insert({

... title: "students",

... description: "MongoDB course",

... by: "sudhakar reddy",

... })

WriteResult({ "nInserted" : 1 })

>

## The drop() Method

MongoDB's **db.collection.drop()** is used to drop a collection from the database.

db.COLLECTION\_NAME.drop()

## The insertOne() method

If you need to insert only one document into a collection you can use this method.

## The insertMany() method

You can insert multiple documents using the insertMany() method. To this method you need to pass an array of documents.

db.empDetails.insertMany(

[

{

},

{

},

{

}

]

)

## The find() Method

To query data from MongoDB collection, you need to use MongoDB's **find()** method.

**find()** method will display all the documents in a non-structured way.

>db.COLLECTION\_NAME.find()

## The findOne() method

Apart from the find() method, there is **findOne()** method, that returns only one document.

>db.COLLECTIONNAME.findOne()

DELETE DOCUMENT

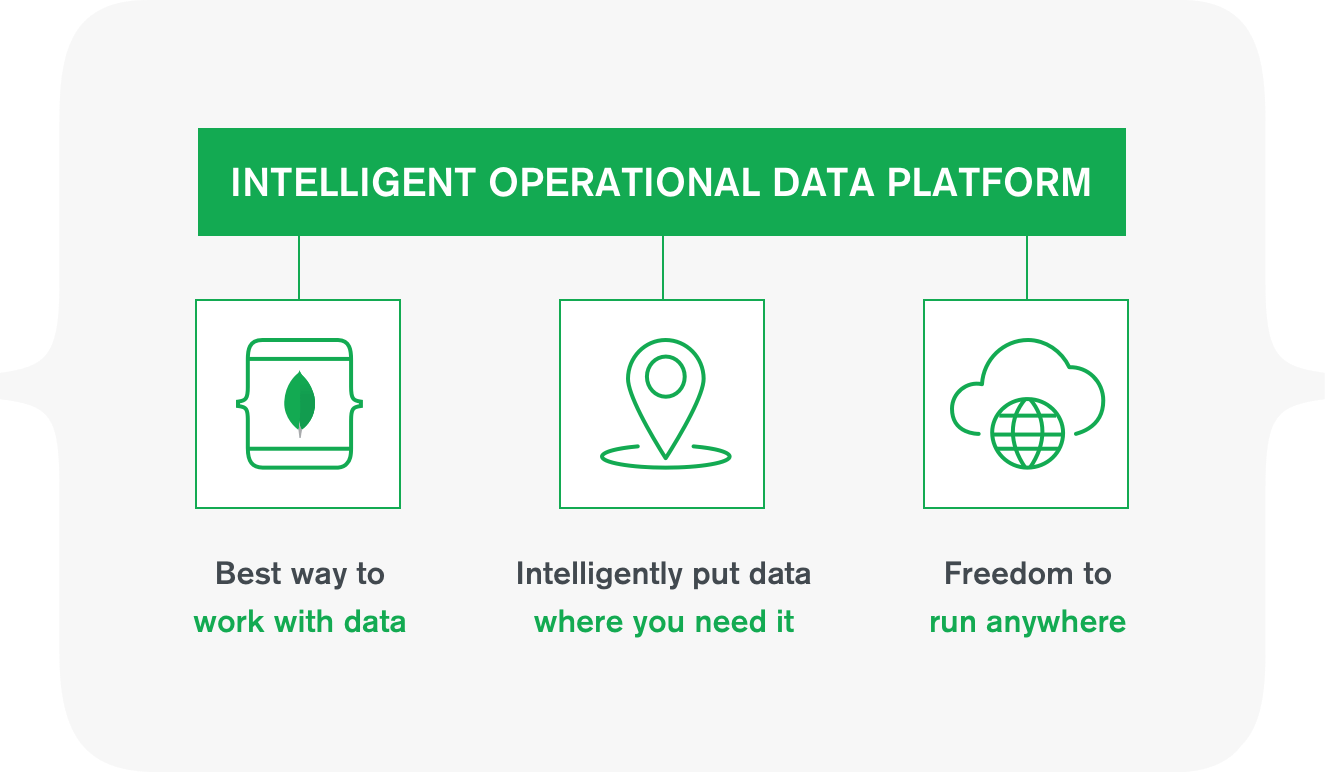
db.nodejs.deleteOne( { "age" : 20 } );

# MongoDB Architecture

MongoDB is designed to meet the demands of modern apps with a technology foundation that enables you through:

* The document data model – presenting you the **best way to work with data**.
* A distributed systems design – allowing you to **intelligently put data where you want it.**
* A unified experience that gives you the **freedom to run anywhere** – allowing you to future-proof your work and eliminate vendor lock-in.

With these capabilities, you can build an Intelligent Operational Data Platform, underpinned by MongoDB.



#### Best way to work with data

* **Easy**: Work with data in a natural, intuitive way
* **Fast**: Get great performance without a lot of work
* **Flexible**: Adapt and make changes quickly
* **Versatile**: Supports a wide variety of data and queries

#### Put data where you need it

* **Availability**: Deliver globally resilient apps through sophisticated replication and self-healing recovery
* **Scalability**: Grow horizontally through native sharding
* **Workload Isolation**: Run operational and analytical workloads in the same cluster
* **Locality**: Place data on specific devices and in specific geographies for governance, class of service, and low-latency access

#### Freedom to run anywhere

* **Portability**: Database that runs the same everywhere
* **Cloud Agnostic**: Leverage the benefits of a multi-cloud strategy with no lock-in
* **Global coverage**: Available as a service in 50+ regions across the major public cloud providers

# Replication

A replica set in MongoDB is a group of [mongod](https://docs.mongodb.com/manual/reference/program/mongod/" \l "mongodb-binary-bin.mongod) processes that maintain the same data set. Replica sets provide redundancy and [high availability](https://docs.mongodb.com/manual/reference/glossary/#std-term-high-availability), and are the basis for all production deployments. This section introduces replication in MongoDB as well as the components and architecture of replica sets. The section also provides tutorials for common tasks related to replica sets.

## Redundancy and Data Availability

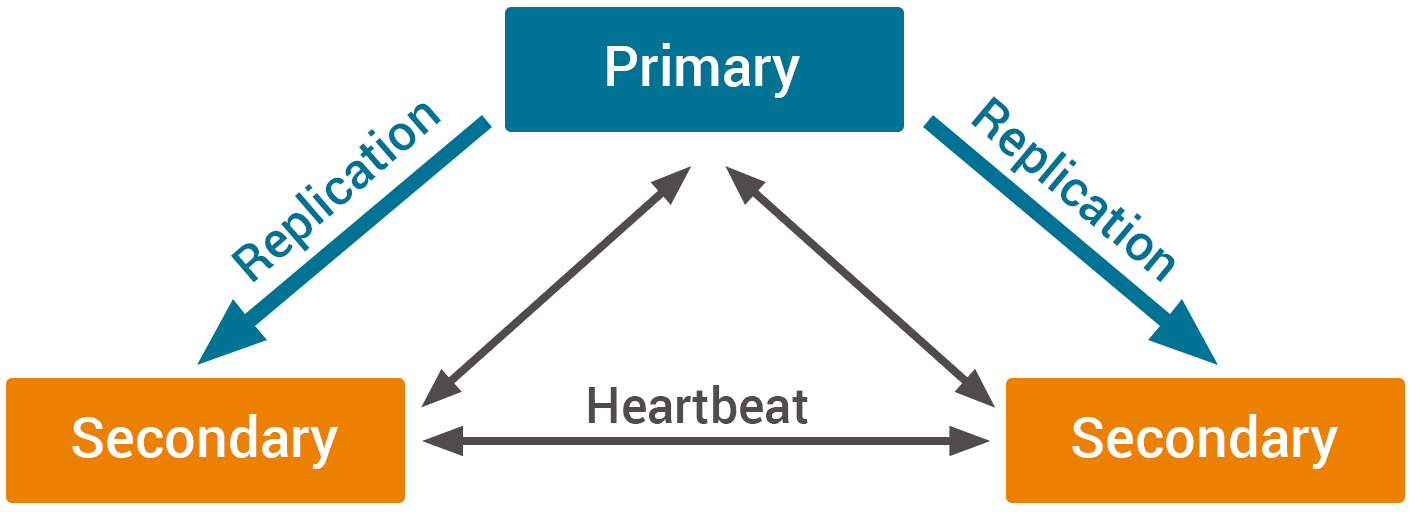
Replication provides redundancy and increases [data availability](https://docs.mongodb.com/manual/reference/glossary/#std-term-high-availability). With multiple copies of data on different database servers, replication provides a level of fault tolerance against the loss of a single database server.

In some cases, replication can provide increased read capacity as clients can send read operations to different servers. Maintaining copies of data in different data centers can increase data locality and availability for distributed applications. You can also maintain additional copies for dedicated purposes, such as disaster recovery, reporting, or backup.

## Replication in MongoDB

A replica set is a group of [mongod](https://docs.mongodb.com/manual/reference/program/mongod/" \l "mongodb-binary-bin.mongod) instances that maintain the same data set. A replica set contains several data bearing nodes and optionally one arbiter node. Of the data bearing nodes, one and only one member is deemed the primary node, while the other nodes are deemed secondary nodes.

The [primary node](https://docs.mongodb.com/manual/core/replica-set-primary/) receives all write operations. A replica set can have only one primary capable of confirming writes with [{ w: "majority" }](https://docs.mongodb.com/manual/reference/write-concern/#mongodb-writeconcern-writeconcern.-majority-) write concern; although in some circumstances, another mongod instance may transiently believe itself to also be primary. [[1]](https://docs.mongodb.com/manual/replication/#footnote-edge-cases-2-primaries) The primary records all changes to its data sets in its operation log, i.e. [oplog](https://docs.mongodb.com/manual/core/replica-set-oplog/).



MongoDB supports many datatypes. Some of them are −

* **String** − This is the most commonly used datatype to store the data. String in MongoDB must be UTF-8 valid.
* **Integer** − This type is used to store a numerical value. Integer can be 32 bit or 64 bit depending upon your server.
* **Boolean** − This type is used to store a boolean (true/ false) value.
* **Double** − This type is used to store floating point values.
* **Min/ Max keys** − This type is used to compare a value against the lowest and highest BSON elements.
* **Arrays** − This type is used to store arrays or list or multiple values into one key.
* **Timestamp** − ctimestamp. This can be handy for recording when a document has been modified or added.
* **Object** − This datatype is used for embedded documents.
* **Null** − This type is used to store a Null value.
* **Symbol** − This datatype is used identically to a string; however, it's generally reserved for languages that use a specific symbol type.
* **Date**− This datatype is used to store the current date or time in UNIX time format. You can specify your own date time by creating object of Date and passing day, month, year into it.
* **Object ID** − This datatype is used to store the document’s ID.
* **Binary data** − This datatype is used to store binary data.
* **Code** − This datatype is used to store JavaScript code into the document.
* **Regular expression** − This datatype is used to store regular expression.

## RDBMS Where Clause Equivalents in MongoDB

To query the document on the basis of some condition, you can use following operations.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Syntax** | **Example** | **RDBMS Equivalent** |
| Equality | {<key>:{$eq;<value>}} | db.mycol.find({age:22}) | where age = 22 |
| Less Than | {<key>:{$lt:<value>}} | db.mycol.find({"age":{$lt:25}}) | where age < 25 |
| Less Than Equals | {<key>:{$lte:<value>}} | db.mycol.find({"age":{$lte:25}}) | where likes <= 50 |
| Greater Than | {<key>:{$gt:<value>}} | db.mycol.find({"likes":{$gt:50}}) | where likes > 50 |
| Greater Than Equals | {<key>:{$gte:<value>}} | db.mycol.find({"likes":{$gte:50}}) | where likes >= 50 |
| Not Equals | {<key>:{$ne:<value>}} | db.mycol.find({"likes":{$ne:50}}). | where likes != 50 |
| Values in an array | {<key>:{$in:[<value1>, <value2>,……<valueN>]}} | db.mycol.find({"name":{$in:["Raj", "Ram", "Raghu"]}}).pretty() | Where name matches any of the value in :["Raj", "Ram", "Raghu"] |
| Values not in an array | {<key>:{$nin:<value>}} | db.mycol.find({"name":{$nin:["Ramu", "Raghav"]}}).pretty() | Where name values is not in the array :["Ramu", "Raghav"] or, doesn’t exist at all |

## AND in MongoDB

### Syntax

To query documents based on the AND condition, you need to use $and keyword. Following is the basic syntax of AND −

>db.mycol.find({ $and: [ {<key1>:<value1>}, { <key2>:<value2>} ] })

## OR in MongoDB

### Syntax

To query documents based on the OR condition, you need to use **$or** keyword. Following is the basic syntax of **OR** −

>db.mycol.find({$or:[{key1: value1}, {key2:value2}]}).pretty()

## NOT in MongoDB

### Syntax

To query documents based on the NOT condition, you need to use $not keyword. Following is the basic syntax of **NOT** −

> db.inventory.find( { price: { $not: { $gt: 1.99 } } } )

UPDATE:

The basic syntax of **update()** method is as follows −

>db.COLLECTION\_NAME.updateOne(SELECTION\_CRITERIA, UPDATED\_DATA)

EX:

db.mycol.update({'title':'MongoDB'},{$set:{'title':'MNDB'}})

UPDATEONE:

db.COLLECTION\_NAME.updateOne(<filter>, <update>)

> db.empDetails.updateOne(

{First\_Name: 'shyam'},

{ $set: { Age: '20',e\_mail: shyam@gmail.com'}}

)

## MongoDB updateMany() method

The updateMany() method updates all the documents that matches the given filter.

### Syntax

The basic syntax of updateMany() method is as follows −

>db.COLLECTION\_NAME.update(<filter>, <update>)

# Embedded Documents

These can be considered as de-normalized data models. As the name suggests, embedded documents create relationships between data by storing related data in a single document structure. These data models allow applications to retrieve and manipulate related data in a single database operation.

Embedded documents should be considered when the embedded entity is an integral part of the document and not updated frequently. It should be used when there is a contained relation between entities and they should not grow indefinitely.



MongoDB Document Structure — Embedded Documents

**Capped Collections:**

================

Capped collections are fixed-size collections that support high-throughput operations that insert and retrieve documents based on insertion order.

Capped collections work in a way similar to circular buffers: once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection.

db.createCollection("cap\_col",{capped:true, size:10000, max:2})

then insert documents

this collection will takes only 2 docs, if you insert 3rd document, 1st document will be deleted

first in...first out

# Atomicity

Atomicity in contrast to the database means operations must fail or succeed as a single unit. If a parent transaction has many sub-operations, it will fail even if a single operation fails. Operations in MongoDB happen at the document level. No single write operation can affect more than one collection. Even if it tries to affect multiple collections, these will treat as separate operations. A single write operation can insert or update the data for an entity. Hence, this facilitates atomic write operations.

However, schemas that provide atomicity in write operations may limit the applications to use the data. It may also limit the ways to modify applications. This consideration describes the challenge that comes in a way of data modeling for flexibility.

This was all about MongoDB Data Modeling Tutorial. Hope you like our explanation.

**MONGODB SCHEMA VALIDATION**

A schema in MongoDB is a blueprint that defines the structure of documents in a collection. It specifies the data types, validation rules, and default values for each field.

**What does a schema do?**

* Defines the shape of documents in a collection
* Outlines the organization of data
* Specifies the types of data that can be stored in each field
* Ensures that data is stored properly

Schema validation allows you to define the specific structure of documents in each collection. If anyone tries to insert some documents which don't match with the defined schema, MongoDB can reject this kind of operation or give warnings according to the type of validation action.

MogoDb provides two ways to validate your schema, **Document validation**, and **JSON schema** validation. JSON Schema validation is the extended version of document validation, so let's start with document validation.

Document Validation

Most of the developers who have worked with relational databases know the importance of predictability of the data models or schema. Therefore, MongoDB introduced document validation from version 3.2. Let's see how to add validation rules in MongoDB collections.

Suppose, you have a collection of users which have the following types of documents.

|  |  |
| --- | --- |
| 1  2  3  4  5 | {      "name": "Alex",      "email": "alex@gmail.com",      "mobile": "123-456-7890"  } |

And, following are the validations which we want to check while adding new documents in users collection:

* name, email fields are mandatory
* mobile numbers should follow specific structure: xxx-xxx-xxxx

To add this validation, we can use the “validator” construct while creating a new collection. Run the following query in Mongo shell.

Ex:Validation:

db.createCollection("users", {

    validator: {

          $and: [

              {

                  "name": {$type: "string", $exists: true}

              },

              {

              "mobile": {$type: "string", $regex: /^[0-9]{3}-[0-9]{3}-[0-9]{4}$/}

              },

              {

                  "email": {$type: "string", $exists: true}

              }

          ]

      }

  })

Now, if you try to add any new document without following the validation rules then mongo will throw a validation error. Try to run the following insert queries.

Query:1

|  |  |
| --- | --- |
| 1  2  3 | db.users.insert({      "name": "akash"  }) |

Changing /Editing Validation:

db.runCommand({

collMod:’users’,

validator:{

….

}

})

db.runCommand({

    collMod:'users',

    validator: {

        $and: [

            {

                "name": {$type: "string", $exists: true}

            },

            {

            "mobile": {$type: "number", $regex: /^[0-9]{3}-[0-9]{3}-[0-9]{4}$/}

            },

            {

                "email": {$type: "string", $exists: true}

            }

        ]

    }

  })

Ex: jsonschema validation

db.createCollection('col3',{

    validator:{

        $jsonSchema:{

            required:['name','price'],

            properties:{

                name:{

                    bsonType:'string',

                    description:'must be a string and required'

                },

                price:{

                    bsonType:'number',

                    description:'must be number and required'

                }

            }

        }

    },

    validationAction:'error'

})

|  |  |
| --- | --- |
|  |  |

Output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | WriteResult({      "nInserted" : 0,      "writeError" : {          "code" : 121,          "errmsg" : "Document failed validation"      }  }) |

Query:2

|  |  |
| --- | --- |
| 1  2  3  4  5 | db.users.insert({      "name": "akash",      "email": "akash@gmail.com",      "mobile": "123-456-7890"  }) |

Output:

|  |  |
| --- | --- |
| 1 | WriteResult({ "nInserted" : 1 }) |

However, there are some restrictions with document validation approach such as one can add any number of new key-value pair to the document and insert it into the collection. This can't be prevented by document validation. Consider the following example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | db.users.insert({      "name": "akash",      "email": "akash@gmail.com",      "mobile": "123-456-7890",      "gender": "Male"  }) |

Output:

|  |  |
| --- | --- |
| 1 | WriteResult({ "nInserted" : 1 }) |

Apart from this, document validation only checks for the values. Suppose, if you try to add the document with "nmae"(typo) as a key instead of "name", mongo will consider it as a new field and the document will be inserted in the DB. These things should be avoided when you are working with the production database. To support all this, MongoDB introduced the "jsonSchema" operator with “validator” construct from version 3.6. Let's see how to add the same validation rules as above and avoid adding new/misspelled fields.

## jsonSchema Validation

Run the following command in mongo shell to add the validation rules using "jsonSchema" operator.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | db.createCollection("students", {      validator: {         $jsonSchema: {            bsonType: "object",            title: "Student Object Validation",            required: [ "name", "place", "age"],            properties: {               name: {                  bsonType: "string",                  description: "'name' must be a string and is required"               },               place: {                  bsonType: "string",                  description: "'place' must be a string and is required"               },               age: {                  bsonType: "int",                  description: "'age' must be an integer and is required"               },            }         }      }   }) |

Let's see now, what happens when we try to insert the following document.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | db.users.insert({      "name": "akash",      "email": "akash@gmail.com",      "mobile": "123-456-7890",      "gender": "Male"  }) |

It will throw an error as we haven't defined gender field in the "jsonSchema".

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | WriteResult({      "nInserted" : 0,      "writeError" : {          "code" : 121,          "errmsg" : "Document failed validation"      }  }) |

Same way, if you have typos in any field names, mongo will throw the same error.

The schema defined above is the same as the one which we used in document validation. Additionally, we added the "additionalProperties" field to avoid typos in field names and the addition of new fields in documents. It will allow only fields that are defined under "properties" field. Here is the overview of some properties which we can use under "jsonSchema" operator.

* bsonType: array | object | string | boolean | number | null
* required: an array of all mandatory fields
* enum: an array of only possible values for any field
* minimum: minimum value of the field
* maximum: maximum value of the field
* minLength: minimum length of the field
* mixLength: maximum length of the field
* properties: a collection of valid JSON schemas
* additionalProperties: stops us from adding any other fields than mentioned under properties field
* title: title for any field.
* description: short description for any field.

Apart from schema validation, "jsonSchema" operator can also be used in find and match stage inside the aggregation pipeline.

**Limit**:

db.lpu.find().limit(2)

**distinct**:

db.student.distinct("cust\_id")

**skip:**

db.lpu.find().skip(1)

db.lpu.find().skip(2)

**push(), pop()**

db.courses.updateOne({ \_id: 11 },{ $push: { hours: 89 } })

db.courses.find({})

[

{ \_id: 10, course\_name: 'python', hours: [ 10, 14, 20 ] },

{ \_id: 11, course\_name: 'C++', hours: [ 10, 15] },

db.courses.updateOne({ \_id: 11 },{ $push: { hours: 89 } })

{ \_id: 11, course\_name: 'C++', hours: [ 10, 15, 89] }

{ \_id: 12, course\_name: 'java', hours: [ 10, 11, 12 ] }

]

**REGULAR EXPRESSIONS**

MongoDB provides the functionality to search a pattern in a string during a query by writing a regular expression.

A regular expression is a generalized way to match patterns with sequences of characters.

db.two.find({product:{$regex:/pi/}})

db.col.find({name:{$regex:/ar/}})

db.one.find({fname:{$regex:/m/}})

db.one.find({fname:{$regex:/M/}})

db.one.find({fname:{$regex:/M/i}}) ==>i stands for ignore case sensitivity

$regex:"e$" ->ends with e

db.one.find({fname:{$regex:"e$"}})

$regex:"^B" ->starts with B

db.one.find({fname:{$regex:"^M"}})

Regular Expressions are frequently used in all languages to search for a pattern or word in any string. MongoDB also provides functionality of regular expression for string pattern matching using the **$regex** operator.

Unlike text search, we do not need to do any configuration or command to use regular expressions.

Assume we have inserted a document in a database named **posts** as shown below −

> db.posts.insert(

{

"post\_text": "learn mongodb articles by sudhakarreddy",

"tags": [

"mongodb",

"sudhakarreddy"

]

}

WriteResult({ "nInserted" : 1 })

## Using regex Expression

The following regex query searches for all the posts containing string **mongodb** in it −

> db.posts.find({post\_text:{$regex:"mongodb"}}).pretty()

{

"\_id" : ObjectId("5dd7ce28f1dd4583e7103fe0"),

"post\_text" : " learn mongodb articles by sudhakarreddy ",

"tags" : [

"mongodb",

"sudhakarreddy"

]

}

{

"\_id" : ObjectId("5dd7d111f1dd4583e7103fe2"),

"post\_text" : "learn mongodb articles by sudhakarreddy ",

"tags" : [

"mongodb",

"sudhakarreddy"

]

}

>

The same query can also be written as −

>db.posts.find({post\_text:/mongodb/})

## Using regex Expression with Case Insensitive

To make the search case insensitive, we use the **$options** parameter with value **$i**. The following command will look for strings having the word **mongodb**, irrespective of smaller or capital case −

>db.posts.find({post\_text:{$regex:"mongodb",$options:"$i"}})

{

"\_id" : ObjectId("53493d37d852429c10000004"),

"post\_text" : "learn mongodb articles by sudhakarreddy ",

"tags" : [ "mongodb" ]

}

**Indexing**

db.COLLECTION\_NAME.createIndex({KEY:1})

db.mycol.createIndex({“age”:1})

db.NAME\_OF\_COLLECTION.dropIndex({KEY:1})

db.NAME\_OF\_COLLECTION.getIndexes()

Indexes are special data structures that store a small portion of the collection's data set in an easy-to-traverse form. MongoDB indexes use a B-tree data structure. The index stores the value of a specific field or set of fields, ordered by the value of the field.

**AGGREGATION:**

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.

**Aggregation Pipelines**

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

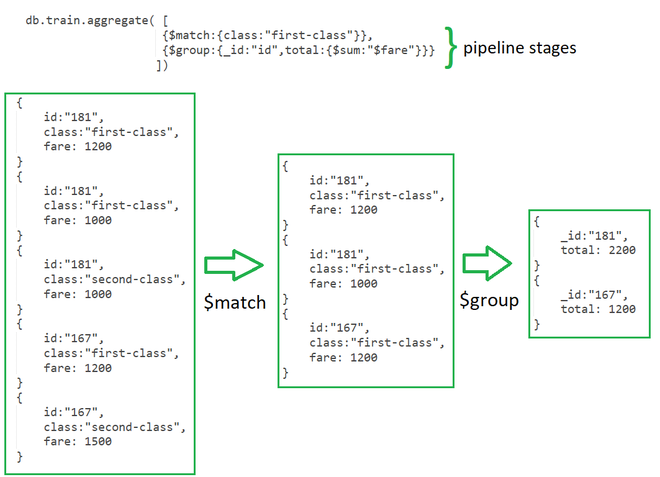
Each stage performs an operation on the input documents.

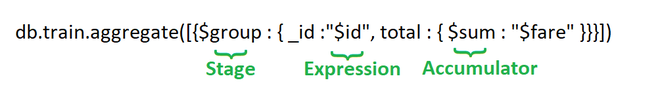
For example, a stage can filter documents, group documents, and calculate values.

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.

**Example1:**





**Example 1:**

db.orders.insertMany([

    {cust\_id:1, qty:"2", amount:300},

    {cust\_id:2, qty:"2", amount:400},

    {cust\_id:1, qty:"1", amount:500},

    {cust\_id:2, qty:"1", amount:100},

    {cust\_id:1, qty:"2", amount:500}

])

db.orders.aggregate([

    {$match:{cust\_id:1}},

    {$group:{\_id:'$cust\_id', total:{$sum:'$amount'}}}

])

**Example 2:**

db.col1.insertMany([

    {\_id:10, Fname:'Mark', Lname:'Masen',Gender:'Male', Age:14},

    {\_id:11, Fname:'Mel', Lname:'Gibson',Gender:'Female', Age:17},

    {\_id:12, Fname:'Deno', Lname:'Socratus',Gender:'Male', Age:24},

    {\_id:13, Fname:'Charls', Lname:'Trippy',Gender:'Female', Age:40},

    {\_id:14, Fname:'Shay', Lname:'Whatson',Gender:'Male', Age:19},

    {\_id:15, Fname:'Katilet', Lname:'Ben',Gender:'Female', Age:34},

])

db.col1.aggregate([{$group:{\_id:"$Gender", MyResult : {$sum:1}}}])

db.col1.aggregate([{$group:{\_id:"$Gender", MaxAge : {$max:"$Age"}}}])

db.col1.aggregate([{$group:{\_id:"$Gender", MaxAge : {$min:"$Age"}}}])

## ­Example 3:

|  |
| --- |
| db.orders.insertMany( [ |
| { \_id: 0, name: **"Pepperoni"**, size: **"small"**, price: 19, |
| quantity: 10, }, |
| { \_id: 1, name: **"Pepperoni"**, size: **"medium"**, price: 20, |
| quantity: 20}, |
| { \_id: 2, name: **"Pepperoni"**, size: **"large"**, price: 21, |
| quantity: 30}, |
| { \_id: 3, name: **"Cheese"**, size: **"small"**, price: 12, |
| quantity: 15}, |
| { \_id: 4, name: **"Cheese"**, size: **"medium"**, price: 13, |
| quantity:50}, |
| { \_id: 5, name: **"Cheese"**, size: **"large"**, price: 14, |
| quantity: 10}, |
| { \_id: 6, name: **"Vegan"**, size: **"small"**, price: 17, |
| quantity: 10}, |
| { \_id: 7, name: **"Vegan"**, size: **"medium"**, price: 18, |
| quantity: 10,} |
| ] )  db.orders.aggregate( [  // Stage 1: Filter pizza order documents by pizza size  {  $match: { size: "medium" }  },  // Stage 2: Group remaining documents by pizza name and calculate total quantity  {  $group: { \_id: "$name", totalQuantity: { $sum: "$quantity" } }  }  ] ) |

## Example:$floor and $ceil

|  |
| --- |
| db.samples.insertMany( |
| [ |
| { \_id: 1, value: 9.25 }, |
| { \_id: 2, value: 8.73 }, |
| { \_id: 3, value: 4.32 }, |
| { \_id: 4, value: -5.34 } |
| ] |
| ) |
| db.samples.aggregate([ |
| { $project: { value: 1, floorValue: { $floor: "$value" } } } |
| ]) |

## Output:

|  |
| --- |
| { **"\_id"** : 1, **"value"** : 9.25, **"floorValue"** : 9 } |
| { **"\_id"** : 2, **"value"** : 8.73, **"floorValue"** : 8 } |
| { **"\_id"** : 3, **"value"** : 4.32, **"floorValue"** : 4 } |
| { **"\_id"** : 4, **"value"** : -5.34, **"floorValue"** : -6 } |

## Ceil:

|  |
| --- |
| db.samples.aggregate([ |
| { $project: { value: 1, ceilingValue: { $ceil: **"$value"** } } } |
| ]) |

The operation returns the following results:

|  |
| --- |
| { **"\_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 } |

## [Understanding Aggregation Pipelines](https://www.digitalocean.com/community/tutorials/how-to-use-aggregations-in-mongodb#understanding-aggregation-pipelines):

When working with a database management system, any time you want to retrieve data from the database you must execute an operation known as a query. However, queries only return the data that already exists in the database. In order to analyze your data to find patterns or other information about the data — rather than the data itself — you’ll often need to perform another kind of operation known as an aggregation.

Aggregations group data from multiple sources and then process that data in some way to return a single result. In a relational database, the database management system will typically pull data from multiple rows in the same table to execute an aggregate function. In a document-oriented database like MongoDB, though, the database will pull data from multiple documents in the same collection.

MongoDB enables you to perform aggregation operations through the mechanism called aggregation pipelines. These are built as a sequential series of declarative data processing operations known as stages. Each stage inspects and transforms the documents as they pass through the pipeline, feeding the transformed results into the subsequent stages for further processing. Documents from a chosen collection enter the pipeline and go through each stage, where the output coming from one stage forms the input for the next one and the final result comes at the end of the pipeline.

It may help to think of this process like vegetables going through an assembly line in a restaurant kitchen. In this analogy, vegetables go through a set of stations, each responsible for a single action: washing, peeling, chopping, cooking, and plating. Likewise, data entering an aggregation pipeline must go through a number of stages, each of which are responsible for a specific operation.

Stages can perform operations on data such as:

* **filtering**: This resembles queries, where the list of documents is narrowed down through a set of criteria
* **sorting**: You can reorder documents based on a chosen field
* **transforming**: The ability to change the structure of documents means you can remove or rename certain fields, or perhaps rename or group fields within an embedded document for legibility
* **grouping**: You can also process multiple documents together to form a summarized result

Pipeline stages do not need to produce the same number of documents they receive. Continuing with our kitchen analogy, imagine a chopping station that takes whole vegetables and passes them along as multiple slices, or a quality control station that rejects vegetables with faults and passes over to the next station only a handful of the healthy ones. Similarly, stages can generate new documents or filter out existing ones from the collection that was entered into the start of the pipeline. Additionally, the same stage can appear more than once in an aggregation pipeline, applying multiple operations one after another.

In the following steps, you’ll prepare a test database to serve as an example data set. You’ll then learn how to use a few of the most common aggregation pipeline stages individually. Finally, you’ll combine these stages together to form a complete example pipeline.

To understand how the aggregation pipelines work, you’ll need a collection of documents with multiple fields of different types you can filter, sort, group, and summarize in different ways. This guide will use a sample collection describing the twenty most populated cities in the world. These documents will have the same format as the following sample document, which describes the city of Tokyo:

The Tokyo document

{

"name": "Tokyo",

"country": "Japan",

"continent": "Asia",

"population": 37.400

}

This document contains the following information:

* name: the city’s name.
* country: the country where the city is located.
* continent: the continent where the city is located.
* population: the city’s population, in millions.

Run the following insertMany() method in the MongoDB shell to simultaneously create a collection named cities and insert twenty sample documents into it. These documents describe the twenty most populated cities in the world:

1. db.cities.insertMany([
2. {"name": "Seoul", "country": "South Korea", "continent": "Asia", "population": 25.674 },
3. {"name": "Mumbai", "country": "India", "continent": "Asia", "population": 19.980 },
4. {"name": "Lagos", "country": "Nigeria", "continent": "Africa", "population": 13.463 },
5. {"name": "Beijing", "country": "China", "continent": "Asia", "population": 19.618 },
6. {"name": "Shanghai", "country": "China", "continent": "Asia", "population": 25.582 },
7. {"name": "Osaka", "country": "Japan", "continent": "Asia", "population": 19.281 },
8. {"name": "Cairo", "country": "Egypt", "continent": "Africa", "population": 20.076 },
9. {"name": "Tokyo", "country": "Japan", "continent": "Asia", "population": 37.400 },
10. {"name": "Karachi", "country": "Pakistan", "continent": "Asia", "population": 15.400 },
11. {"name": "Dhaka", "country": "Bangladesh", "continent": "Asia", "population": 19.578 },
12. {"name": "Rio de Janeiro", "country": "Brazil", "continent": "South America", "population": 13.293 },
13. {"name": "São Paulo", "country": "Brazil", "continent": "South America", "population": 21.650 },
14. {"name": "Mexico City", "country": "Mexico", "continent": "North America", "population": 21.581 },
15. {"name": "Delhi", "country": "India", "continent": "Asia", "population": 28.514 },
16. {"name": "Buenos Aires", "country": "Argentina", "continent": "South America", "population": 14.967 },
17. {"name": "Kolkata", "country": "India", "continent": "Asia", "population": 14.681 },
18. {"name": "New York", "country": "United States", "continent": "North America", "population": 18.819 },
19. {"name": "Manila", "country": "Philippines", "continent": "Asia", "population": 13.482 },
20. {"name": "Chongqing", "country": "China", "continent": "Asia", "population": 14.838 },
21. {"name": "Istanbul", "country": "Turkey", "continent": "Europe", "population": 14.751 }
22. ])

The output will contain a list of object identifiers assigned to the newly inserted objects.

Output

{

"acknowledged" : true,

"insertedIds" : [

ObjectId("612d1e835ebee16872a109a4"),

ObjectId("612d1e835ebee16872a109a5"),

ObjectId("612d1e835ebee16872a109a6"),

ObjectId("612d1e835ebee16872a109a7"),

ObjectId("612d1e835ebee16872a109a8"),

ObjectId("612d1e835ebee16872a109a9"),

ObjectId("612d1e835ebee16872a109aa"),

ObjectId("612d1e835ebee16872a109ab"),

ObjectId("612d1e835ebee16872a109ac"),

ObjectId("612d1e835ebee16872a109ad"),

ObjectId("612d1e835ebee16872a109ae"),

ObjectId("612d1e835ebee16872a109af"),

ObjectId("612d1e835ebee16872a109b0"),

ObjectId("612d1e835ebee16872a109b1"),

ObjectId("612d1e835ebee16872a109b2"),

ObjectId("612d1e835ebee16872a109b3"),

ObjectId("612d1e835ebee16872a109b4"),

ObjectId("612d1e835ebee16872a109b5"),

ObjectId("612d1e835ebee16872a109b6"),

ObjectId("612d1e835ebee16872a109b7")

]

}

You can verify that the documents were properly inserted by running the find() method on the cities collection with no arguments. This will retrieve all the documents in the collection:

1. db.cities.find()

Copy

Output

{ "\_id" : ObjectId("612d1e835ebee16872a109a4"), "name" : "Seoul", "country" : "South Korea", "continent" : "Asia", "population" : 25.674 }

. . .

With the sample data in place, you can continue on to the next step to learn how to build an aggregation pipeline using the $match stage.

## [Step 2 — Using the $match Aggregation Stage](https://www.digitalocean.com/community/tutorials/how-to-use-aggregations-in-mongodb#step-2-using-the-match-aggregation-stage)

To create an aggregation pipeline, you can use can use MongoDB’s aggregate() method. This method uses a syntax that is fairly similar to the find() method used to query data in a collection, but aggregate() accepts one or more stage names as arguments. This step focuses on how to use the $match aggregation stage.

Whether you want to do light document structure processing, summarizing, or complex transformations, you’ll usually want to focus your analysis on just a selection of documents matching specific criteria. $match can be used to narrow down the list of documents at any given step of a pipeline, and can be used to ensure that all subsequent operations will be executed on a limited list of entries.

As an example, run the following operation. This will construct an aggregation pipeline using a single $match stage without any particular filtering query:

1. db.cities.aggregate([
2. { $match: { } }
3. ])

The aggregate() method executed on the cities collection instructs MongoDB to run an aggregation pipeline passed as the method argument. Because aggregation pipelines are multi-step processes, the argument is a list of stages, hence the use of square brackets [] denoting an array of multiple elements.

Each element inside this array is an object describing a processing stage. The stage here is written as { $match: { } }. In this document describing the processing stage, the key $match refers to the stage type, and the value { } describes its parameters. In our example, the $match stage uses the empty query document as its parameter and is the only stage in the whole processing pipeline.

Remember that $match narrows down the list of documents from the collection. With no filtering parameters applied, MongoDB will return the list of all cities from the collection:

Output

{ "\_id" : ObjectId("612d1e835ebee16872a109a4"), "name" : "Seoul", "country" : "South Korea", "continent" : "Asia", "population" : 25.674 }

{ "\_id" : ObjectId("612d1e835ebee16872a109a5"), "name" : "Mumbai", "country" : "India", "continent" : "Asia", "population" : 19.98 }

{ "\_id" : ObjectId("612d1e835ebee16872a109a6"), "name" : "Lagos", "country" : "Nigeria", "continent" : "Africa", "population" : 13.463 }

{ "\_id" : ObjectId("612d1e835ebee16872a109a7"), "name" : "Beijing", "country" : "China", "continent" : "Asia", "population" : 19.618 }

{ "\_id" : ObjectId("612d1e835ebee16872a109a8"), "name" : "Shanghai", "country" : "China", "continent" : "Asia", "population" : 25.582 }

. . .

Next, run the aggregate() method again, but this time include a query document as a parameter to the $match stage. Any valid query document can be used here.

You can think of using the $match stage as equivalent to querying the collection with find() as described in the [How To Create Queries in MongoDB](https://www.digitalocean.com/community/tutorials/how-to-create-queries-in-mongodb) tutorial listed in the Prerequisites section. The biggest difference is that $match can be used multiple times in the aggregation pipeline, allowing you to query documents that have already been processed and transformed earlier in the pipeline. You’ll learn more about using the same stage multiple times in the same aggregation pipeline later on in this guide.

Run the following aggregate() method. This example includes a $match stage to select only cities from North America:

1. db.cities.aggregate([
2. { $match: { "continent": "North America" } }
3. ])

Copy

This time the { "continent": "North America" } query document appears as the parameter to the $match stage. Consequently, MongoDB returns two cities from North America:

Output

{ "\_id" : ObjectId("612d1e835ebee16872a109b0"), "name" : "Mexico City", "country" : "Mexico", "continent" : "North America", "population" : 21.581 }

{ "\_id" : ObjectId("612d1e835ebee16872a109b4"), "name" : "New York", "country" : "United States", "continent" : "North America", "population" : 18.819 }

This command returns the same output as the following one which instead uses the find() method to query the database:

1. db.cities.find({ "continent": "North America" })

The previous aggregate() method only returns two cities, so there isn’t much to experiment with. To return more results, alter this command so it returns cities from North America and Asia:

1. db.cities.aggregate([
2. { $match: { "continent": { $in: ["North America", "Asia"] } } }
3. ])

Notice that the query document syntax is once again identical to how you’d retrive the same data using the find() method. This time MongoDB returns 14 different cities:

Output

{ "\_id" : ObjectId("612d1e835ebee16872a109a4"), "name" : "Seoul", "country" : "South Korea", "continent" : "Asia", "population" : 25.674 }

{ "\_id" : ObjectId("612d1e835ebee16872a109a5"), "name" : "Mumbai", "country" : "India", "continent" : "Asia", "population" : 19.98 }

. . .

With that, you’ve learned how to execute an aggregation pipeline and using the $match stage to narrow down the collection’s documents. Continue reading to learn how to build more complex pipelines by using $sort stage to order the results and by combining multiple stages together.

## [Step 3 — Using the $sort Aggregation Stage](https://www.digitalocean.com/community/tutorials/how-to-use-aggregations-in-mongodb#step-3-using-the-sort-aggregation-stage)

The $match stage is useful for narrowing down the list of documents that are moved on to the next aggregation stage. However, $match does nothing to change or transform the data as it passes through the pipeline.

When querying the database, it’s common to expect a certain order when retrieving the results. Using the standard query mechanism, you can specify the document order by appending a sort() method to the end of a find() query. For example, to retrieve every city in the collection and sort them in descending order by population, you could run an operation like this:

1. db.cities.find().sort({ "population": -1 })

Copy

MongoDB will return each city starting with Tokyo, followed by Delhi, Seoul, and so on:

Output

{ "\_id" : ObjectId("612d1e835ebee16872a109ab"), "name" : "Tokyo", "country" : "Japan", "continent" : "Asia", "population" : 37.4 }

{ "\_id" : ObjectId("612d1e835ebee16872a109b1"), "name" : "Delhi", "country" : "India", "continent" : "Asia", "population" : 28.514 }

{ "\_id" : ObjectId("612d1e835ebee16872a109a4"), "name" : "Seoul", "country" : "South Korea", "continent" : "Asia", "population" : 25.674 }

. . .

You can alternatively sort the documents in an aggregation pipeline by including a $sort stage. To illustrate this, run the following aggregate() method. This follows a similar syntax to the previous examples that used a $match stage:

1. db.cities.aggregate([
2. { $sort: { "population": -1 } }
3. ])

Again, the list of stages used in an aggregation pipeline is passed as an array of stage definitions between a pair of square brackets ([]). This example’s stage definition only includes a single $sort stage as the key, with its value being a document holding the sorting parameters. Any valid sort document can be used here.

MongoDB will return the same result set as the previous find() operation since using an aggregation pipeline with just a sorting stage is equivalent to a standard query with a sort order applied:

Output

{ "\_id" : ObjectId("612d1e835ebee16872a109ab"), "name" : "Tokyo", "country" : "Japan", "continent" : "Asia", "population" : 37.4 }

{ "\_id" : ObjectId("612d1e835ebee16872a109b1"), "name" : "Delhi", "country" : "India", "continent" : "Asia", "population" : 28.514 }

{ "\_id" : ObjectId("612d1e835ebee16872a109a4"), "name" : "Seoul", "country" : "South Korea", "continent" : "Asia", "population" : 25.674 }

. . .

Suppose that you want to retrieve cities just from North America sorted by population in ascending order. To do so, you could apply two processing stages one after the other: the first to narrow down the result set with a filtering $match stage and then a second to apply the required ordering using a $sort stage:

1. db.cities.aggregate([
2. { $match: { "continent": "North America" } },
3. { $sort: { "population": 1 } }
4. ])

Notice the two separate stages in the command syntax, separated by a comma within the stages array.

This time, MongoDB will return documents representing New York and Mexico City, the only two cities from North America, starting with New York as it has a lower population:

Output

{ "\_id" : ObjectId("612d1e835ebee16872a109b4"), "name" : "New York", "country" : "United States", "continent" : "North America", "population" : 18.819 }

{ "\_id" : ObjectId("612d1e835ebee16872a109b0"), "name" : "Mexico City", "country" : "Mexico", "continent" : "North America", "population" : 21.581 }

To obtain these results, MongoDB first passed the document collection through the $match stage, filtered the documents against the query criteria, and then forwarded the results to the next stage in line responsible for sorting the results. Just like the $match stage, $sort can appear multiple times in the aggregation pipeline and can sort documents by any field you might need, including fields that will only appear in the document structure during the aggregation.

**Note:** When running filtering and sorting stages at the beginning of the aggregation pipeline, before any projection, grouping, or other transformation stages, MongoDB will use indexes to maximize the performance just like it would with standard query. You can learn more about indexes in our guide on [How to Use Indexes in MongoDB](https://www.digitalocean.com/community/tutorials/how-to-use-indexes-in-mongodb).

## [Step 4 — Using the $group Aggregation Stage](https://www.digitalocean.com/community/tutorials/how-to-use-aggregations-in-mongodb#step-4-using-the-group-aggregation-stage)

The $group aggregation stage is responsible for grouping and summarizing documents. It takes in multiple documents and arranges them into several separate batches based on grouping expression values and outputs a single document for each distinct batch. The output documents hold information about the group and can contain additional computed fields like sums or averages across the list of documents from the group.

To illustrate, run the following aggregate() method. This includes a $group stage that will group the resulting documents by the continent in which each city is located:

1. db.cities.aggregate([
2. { $group: { "\_id": "$continent" } }
3. ])

Copy

In MongoDB, every document must have an \_id field to be used as a primary key. Recall from Step 1 that the insertMany() method used to create the sample collection didn’t include this field in any sample documents. This is because MongoDB automatically creates this field and generates unique identification numbers in the form of ObjectId fields. For $group stages within an aggregation pipeline, though, it is required that you specify an \_id field with a valid expression.

This aggregate() method, though, does specify an \_id value; namely, each value found in the continent fields of each document in the cities collection. Any time you want to refer the values of a field in an aggregation pipeline like this, you must precede the name of the field with a dollar sign ($). In MongoDB, this is referred to as a field path, as it directs the operation to the appropriate field where it can find the values to be used in the pipeline stage.

Here in this example, "$continent" tells MongoDB to take the continent field from the original document and use its value to construct the expression value in the aggregation pipeline. MongoDB will output a single document for each unique value of that grouping expression:

Output

{ "\_id" : "Africa" }

{ "\_id" : "Asia" }

{ "\_id" : "South America" }

{ "\_id" : "Europe" }

{ "\_id" : "North America" }

This example outputs a single document for each of the five continents represented in the collection. By default, the grouping stage doesn’t include any additional fields from the original document, since it wouldn’t know how or from which document to source the other values.

You can, however, specify multiple single-field values in a grouping expression. The following example method will group documents based on the values in the continent and country documents:

1. db.cities.aggregate([
2. {
3. $group: {
4. "\_id": {
5. "continent": "$continent",
6. "country": "$country"
7. }
8. }
9. }
10. ])

Copy

Notice that the \_id field for this example’s grouping expression uses an embedded document which, in turn, has two fields inside: one for the continent name and another for the country name. Both fields refer to fields from the original documents using the field path dollar sign notation.

This time MongoDB returns 14 results as there are 14 distinct country-continents pairs in the collection:

Output

{ "\_id" : { "continent" : "Europe", "country" : "Turkey" } }

{ "\_id" : { "continent" : "South America", "country" : "Argentina" } }

{ "\_id" : { "continent" : "Asia", "country" : "Bangladesh" } }

{ "\_id" : { "continent" : "Asia", "country" : "Philippines" } }

{ "\_id" : { "continent" : "Asia", "country" : "South Korea" } }

{ "\_id" : { "continent" : "Asia", "country" : "Japan" } }

{ "\_id" : { "continent" : "Asia", "country" : "China" } }

{ "\_id" : { "continent" : "North America", "country" : "United States" } }

{ "\_id" : { "continent" : "North America", "country" : "Mexico" } }

{ "\_id" : { "continent" : "Africa", "country" : "Nigeria" } }

{ "\_id" : { "continent" : "Asia", "country" : "India" } }

{ "\_id" : { "continent" : "Asia", "country" : "Pakistan" } }

{ "\_id" : { "continent" : "Africa", "country" : "Egypt" } }

{ "\_id" : { "continent" : "South America", "country" : "Brazil" } }

These results aren’t ordered in any meaningful way. As you work with data more and more, you’ll likely encounter situations where you want to perform more complex data analysis. To this end, MongoDB provides a number of accumulator operators which allow you to find more granular details about your data. An accumulator operator, sometimes just referred to simply as an accumulator, is a special type of operation that maintains its value or state as it passes through an aggregation pipeline, such as a sum or average of more than one value.

To illustrate, run the following aggregate() method. This method’s $group stage creates the required \_id grouping expression as well as three additional computed fields. These computed fields all include an accumulator operator and its value. Here’s a breakdown of these computed fields:

* highest\_population: this field contains the maximum population value in the group. The $max accumulator operator computes the maximum value for "$population" across all documents in a group.
* first\_city: contains the name of the first city in the group. The $first accumulator operator takes the value of "$name" from the first document appearing in the group. Notice that since the list of documents is now unordered, this doesn’t automatically make it the city with the highest population, but rather the first city MongoDB finds within each group.
* cities\_in\_top\_20: holds the number of cities in the collection for each continent-country pair. To accomplish this, the $sum accumulator operator is used to compute the sum of all the pairs in the list. In this example, the sum takes one for each document and doesn’t refer to a particular field in the source document.

You can add as many additional computed fields as needed for your use case, but for now run this example query:

1. db.cities.aggregate([
2. {
3. $group: {
4. "\_id": {
5. "continent": "$continent",
6. "country": "$country"
7. },
8. "highest\_population": { $max: "$population" },
9. "first\_city": { $first: "$name" },
10. "cities\_in\_top\_20": { $sum: 1 }
11. }
12. }
13. ])

Copy

MongoDB returns the following 14 documents, one for each unique group defined by the grouping expression:

Output

{ "\_id" : { "continent" : "North America", "country" : "United States" }, "highest\_population" : 18.819, "first\_city" : "New York", "cities\_in\_top\_20" : 1 }

{ "\_id" : { "continent" : "Asia", "country" : "Philippines" }, "highest\_population" : 13.482, "first\_city" : "Manila", "cities\_in\_top\_20" : 1 }

{ "\_id" : { "continent" : "North America", "country" : "Mexico" }, "highest\_population" : 21.581, "first\_city" : "Mexico City", "cities\_in\_top\_20" : 1 }

{ "\_id" : { "continent" : "Africa", "country" : "Nigeria" }, "highest\_population" : 13.463, "first\_city" : "Lagos", "cities\_in\_top\_20" : 1 }

{ "\_id" : { "continent" : "Asia", "country" : "India" }, "highest\_population" : 28.514, "first\_city" : "Mumbai", "cities\_in\_top\_20" : 3 }

{ "\_id" : { "continent" : "Asia", "country" : "Pakistan" }, "highest\_population" : 15.4, "first\_city" : "Karachi", "cities\_in\_top\_20" : 1 }

{ "\_id" : { "continent" : "Africa", "country" : "Egypt" }, "highest\_population" : 20.076, "first\_city" : "Cairo", "cities\_in\_top\_20" : 1 }

{ "\_id" : { "continent" : "South America", "country" : "Brazil" }, "highest\_population" : 21.65, "first\_city" : "Rio de Janeiro", "cities\_in\_top\_20" : 2 }

{ "\_id" : { "continent" : "Europe", "country" : "Turkey" }, "highest\_population" : 14.751, "first\_city" : "Istanbul", "cities\_in\_top\_20" : 1 }

{ "\_id" : { "continent" : "Asia", "country" : "Bangladesh" }, "highest\_population" : 19.578, "first\_city" : "Dhaka", "cities\_in\_top\_20" : 1 }

{ "\_id" : { "continent" : "South America", "country" : "Argentina" }, "highest\_population" : 14.967, "first\_city" : "Buenos Aires", "cities\_in\_top\_20" : 1 }

{ "\_id" : { "continent" : "Asia", "country" : "South Korea" }, "highest\_population" : 25.674, "first\_city" : "Seoul", "cities\_in\_top\_20" : 1 }

{ "\_id" : { "continent" : "Asia", "country" : "Japan" }, "highest\_population" : 37.4, "first\_city" : "Osaka", "cities\_in\_top\_20" : 2 }

{ "\_id" : { "continent" : "Asia", "country" : "China" }, "highest\_population" : 25.582, "first\_city" : "Beijing", "cities\_in\_top\_20" : 3 }

The field names in the returned documents correspond to the computed field names in the grouping stage document. To examine the results more closely, let’s narrow our focus to a single document:

A summarized document representing Japan

{ "\_id" : { "continent" : "Asia", "country" : "Japan" }, "highest\_population" : 37.4, "first\_city" : "Osaka", "cities\_in\_top\_20" : 2 }

The \_id field holds the grouping expression values for Japan and Asia. The cities\_in\_top\_20 field shows that two Japanese cities are on the list of the 20 most populated cities. Recall from Step 1 that you only added two documents representing cities in Japan (Tokyo and Osaka), so this value is correct. The highest\_population corresponds to the population of Tokyo, which is indeed the higher population of the two.

However, the first\_city shows Osaka and not Tokyo, as one might expect. That’s because the grouping stage used a list of source documents that were not ordered by population, so it couldn’t guarantee the logical meaning of “first” in that scenario. Osaka was processed by the pipeline first and hence appears in the first\_city field value.

You’ll learn how to change that by strategically combining sorting and grouping stages in Step 6. For now, though, you can continue on to Step 5 which outlines how to transform the structure of documents in a pipeline using projections.

**Note:** In addition to the three described in this step, there are several more accumulator operators available in MongoDB that can be used for a variety of aggregations. To learn more about grouping and accumulator operators, we encourage you to follow the [official MongoDB documentation](https://docs.mongodb.com/v4.4/mongo/).

## [Step 5 — Using the $project Aggregation Stage](https://www.digitalocean.com/community/tutorials/how-to-use-aggregations-in-mongodb#step-5-using-the-project-aggregation-stage)

When working with aggregation pipelines, you’ll sometimes want to return only a few of a document collection’s multiple fields or change the structure slightly to move some fields into embedded documents. You could use this strategy to redact fields that should not be included in a report, or to prepare results in a format ready for certain application requirements.

Say, for example, that you’d like to retrieve the population for each of the cities in the sample collection, but you’d like the results to take the following format:

Required document structure

{

"location" : {

"country" : "South Korea",

"continent" : "Asia"

},

"name" : "Seoul",

"population" : 25.674

}

The location field contains the country and continent pair, the city’s name and population are shown in name and population fields, respectively, and the document identifier \_id doesn’t appear in the outputted document.

You can use the $project stage to construct new document structures in an aggregation pipeline, thereby altering the way resulting documents appear in the result set.

To illustrate, run the following aggregate() method which includes a $project stage:

1. db.cities.aggregate([
2. {
3. $project: {
4. "\_id": 0,
5. "location": {
6. "country": "$country",
7. "continent": "$continent",
8. },
9. "name": "$name",
10. "population": "$population"
11. }
12. }
13. ])

Copy

The value for this $project stage is a projection document describing the output structure. These projection documents follow the same format as [those used in queries](https://www.digitalocean.com/community/tutorials/how-to-create-queries-in-mongodb#step-6-%E2%80%94-returning-a-subset-of-fields), constructed as inclusion projections or exclusion projections. The projection document keys correspond to the keys from input documents entering the $project stage.

When the projection document contains keys with 1 as their values, it describes the list of fields that will be included in the result. If, on the other hand, projection keys are set to 0, the projection document describes the list of fields that will be excluded from the result.

In an aggregation pipeline, projections can also include additional computed fields. In such cases, the projection automatically becomes an inclusion projection, and only the \_id field can be suppressed by appending "\_id": 0 to the projection document. Computed fields use the dollar sign field path notation for their values and can refer to the values from input documents.

In this example, the document identifier is suppressed with "\_id": 0, the name and population are computed fields referring to the name and population fields from the input documents, respectively. The location field becomes an embedded document with two additional keys: country and continent, referring to fields from the input documents.

Using this projection stage, MongoDB will return the following documents:

Output

{ "location" : { "country" : "South Korea", "continent" : "Asia" }, "name" : "Seoul", "population" : 25.674 }

{ "location" : { "country" : "India", "continent" : "Asia" }, "name" : "Mumbai", "population" : 19.98 }

{ "location" : { "country" : "Nigeria", "continent" : "Africa" }, "name" : "Lagos", "population" : 13.463 }

{ "location" : { "country" : "China", "continent" : "Asia" }, "name" : "Beijing", "population" : 19.618 }

{ "location" : { "country" : "China", "continent" : "Asia" }, "name" : "Shanghai", "population" : 25.582 }

{ "location" : { "country" : "Japan", "continent" : "Asia" }, "name" : "Osaka", "population" : 19.281 }

{ "location" : { "country" : "Egypt", "continent" : "Africa" }, "name" : "Cairo", "population" : 20.076 }

{ "location" : { "country" : "Japan", "continent" : "Asia" }, "name" : "Tokyo", "population" : 37.4 }

{ "location" : { "country" : "Pakistan", "continent" : "Asia" }, "name" : "Karachi", "population" : 15.4 }

{ "location" : { "country" : "Bangladesh", "continent" : "Asia" }, "name" : "Dhaka", "population" : 19.578 }

{ "location" : { "country" : "Brazil", "continent" : "South America" }, "name" : "Rio de Janeiro", "population" : 13.293 }

{ "location" : { "country" : "Brazil", "continent" : "South America" }, "name" : "São Paulo", "population" : 21.65 }

{ "location" : { "country" : "Mexico", "continent" : "North America" }, "name" : "Mexico City", "population" : 21.581 }

{ "location" : { "country" : "India", "continent" : "Asia" }, "name" : "Delhi", "population" : 28.514 }

{ "location" : { "country" : "Argentina", "continent" : "South America" }, "name" : "Buenos Aires", "population" : 14.967 }

{ "location" : { "country" : "India", "continent" : "Asia" }, "name" : "Kolkata", "population" : 14.681 }

{ "location" : { "country" : "United States", "continent" : "North America" }, "name" : "New York", "population" : 18.819 }

{ "location" : { "country" : "Philippines", "continent" : "Asia" }, "name" : "Manila", "population" : 13.482 }

{ "location" : { "country" : "China", "continent" : "Asia" }, "name" : "Chongqing", "population" : 14.838 }

{ "location" : { "country" : "Turkey", "continent" : "Europe" }, "name" : "Istanbul", "population" : 14.751 }

Each document now follows the new format transformed through the projection stage.

Now that you’ve learned how to use $project stage to construct a new document structure for the documents going through an aggregation pipeline, you’re ready to combine all the pipeline stages covered throughout this guide in a single aggregation pipeline.

## [Step 6 — Putting All the Stages Together](https://www.digitalocean.com/community/tutorials/how-to-use-aggregations-in-mongodb#step-6-putting-all-the-stages-together)

You’re now ready to join together all the stages that you’ve practiced using in the previous steps to form a fully functional aggregation pipeline that both filters and transforms documents.

Suppose the task at hand is to find the most populated city for each country in country in Asia and North America and return both its name and population. The results should be sorted by the highest population, returning countries with the largest cities first, and you are interested only in countries where the most populated city crosses the threshold of 20 million people. Lastly, the document structure you aim for should replicate the following:

Example document

{

"location" : {

"country" : "Japan",

"continent" : "Asia"

},

"most\_populated\_city" : {

"name" : "Tokyo",

"population" : 37.4

}

}

To illustrate how to retrieve a data set that would satisfy these requirements, this step outlines how to build the appropriate aggregation pipeline.

Begin by running the following query, which filters the initial documents coming from the cities collection so the result set will only contain countries in Asia and North America. Even though it would be possible to narrow down the document selection later, doing so upfront will optimize the pipeline’s efficiency. After all, limiting the number of documents to be processed early will minimize the amount of processing needed in later stages:

1. db.cities.aggregate([
2. {
3. $match: {
4. "continent": { $in: ["North America", "Asia"] }
5. }
6. }
7. ])

This pipeline’s $match stage will only find cities in North America and Asia, and the documents representing these cities will be returned in their full original structure and with default ordering:

Output

{ "\_id" : ObjectId("612d1e835ebee16872a109a4"), "name" : "Seoul", "country" : "South Korea", "continent" : "Asia", "population" : 25.674 }

{ "\_id" : ObjectId("612d1e835ebee16872a109a5"), "name" : "Mumbai", "country" : "India", "continent" : "Asia", "population" : 19.98 }

{ "\_id" : ObjectId("612d1e835ebee16872a109a7"), "name" : "Beijing", "country" : "China", "continent" : "Asia", "population" : 19.618 }

{ "\_id" : ObjectId("612d1e835ebee16872a109a8"), "name" : "Shanghai", "country" : "China", "continent" : "Asia", "population" : 25.582 }

. . .

In Step 4, you learned that passing an unordered list of documents to the grouping stage can have unexpected results if you need to access fields from the first document in the group. You must do this to find the name of the most populated city later, so to head off this problem you can order the cities from the highest to the lowest population by following the $match stage with a $sort stage:

1. db.cities.aggregate([
2. {
3. $match: {
4. "continent": { $in: ["North America", "Asia"] }
5. }
6. },
7. {
8. $sort: { "population": -1 }
9. }
10. ])

Copy

The second pipeline stage in this aggregate() method tells MongoDB to order the documents by population in descending order, as indicated by the { "population": -1 } sorting document.

Once again the returned documents have the same structure, but this time Tokyo comes first since it has the highest population:

Output

{ "\_id" : ObjectId("612d1e835ebee16872a109ab"), "name" : "Tokyo", "country" : "Japan", "continent" : "Asia", "population" : 37.4 }

. . .

You now have the list of cities sorted by the population coming from the expected continents, so the next necessary action for this scenario is to group cities by their countries, choosing only the most populated city from each group. To do so, add a $group stage to the pipeline:

1. db.cities.aggregate([
2. {
3. $match: {
4. "continent": { $in: ["North America", "Asia"] }
5. }
6. },
7. {
8. $sort: { "population": -1 }
9. },
10. {
11. $group: {
12. "\_id": {
13. "continent": "$continent",
14. "country": "$country"
15. },
16. "first\_city": { $first: "$name" },
17. "highest\_population": { $max: "$population" }
18. }
19. }
20. ])

Copy

This new stage’s grouping expression tells MongoDB to group cities by unique continent and country pairs. For each group, two computed values summarize the groups. The highest\_population value uses the $max accumulator operator to find the highest population in the group. The first\_city gets the name of the first city in the group of documents. Thanks to the previously applied sorting stage, you can be sure this first city will also the most populated city in the group and that it will match the numerical population value.

Adding this $group stage changes the number of documents returned by this method as well as their structure. This time, the method only returns nine documents, as there are only nine unique country and continent pairs in the previously filtered cities list. Each document corresponds to one of these pairs, and consists of the grouping expression value in the \_id field and two computed fields:

Output

{ "\_id" : { "continent" : "North America", "country" : "United States" }, "first\_city" : "New York", "highest\_population" : 18.819 }

{ "\_id" : { "continent" : "Asia", "country" : "China" }, "first\_city" : "Shanghai", "highest\_population" : 25.582 }

{ "\_id" : { "continent" : "Asia", "country" : "Japan" }, "first\_city" : "Tokyo", "highest\_population" : 37.4 }

{ "\_id" : { "continent" : "Asia", "country" : "South Korea" }, "first\_city" : "Seoul", "highest\_population" : 25.674 }

{ "\_id" : { "continent" : "Asia", "country" : "Bangladesh" }, "first\_city" : "Dhaka", "highest\_population" : 19.578 }

{ "\_id" : { "continent" : "Asia", "country" : "Philippines" }, "first\_city" : "Manila", "highest\_population" : 13.482 }

{ "\_id" : { "continent" : "Asia", "country" : "India" }, "first\_city" : "Delhi", "highest\_population" : 28.514 }

{ "\_id" : { "continent" : "Asia", "country" : "Pakistan" }, "first\_city" : "Karachi", "highest\_population" : 15.4 }

{ "\_id" : { "continent" : "North America", "country" : "Mexico" }, "first\_city" : "Mexico City", "highest\_population" : 21.581 }

Notice that the resulting documents for each group are not ordered by the population value. New York comes first, but the second city — Shanghai — has a population of almost 7 million people more. Also, several countries have the most populated cities below the expected threshold of 20 million.

Remember that filtering and sorting stages can appear multiple times in the pipeline. Also, for each aggregation stage, the last stage’s output is the next stage’s input. Use another $match stage to filter the groups to contain only countries with the cities satisfying population minimum of 20 million:

1. db.cities.aggregate([
2. {
3. $match: {
4. "continent": { $in: ["North America", "Asia"] }
5. }
6. },
7. {
8. $sort: { "population": -1 }
9. },
10. {
11. $group: {
12. "\_id": {
13. "continent": "$continent",
14. "country": "$country"
15. },
16. "first\_city": { $first: "$name" },
17. "highest\_population": { $max: "$population" }
18. }
19. },
20. {
21. $match: {
22. "highest\_population": { $gt: 20.0 }
23. }
24. }
25. ])

Copy

This filtering $match stage refers to the highest\_population field available in the documents coming from the grouping stage, even though such a field is not part of the structure of the original documents.

This time, five countries appear in the output:

Output

{ "\_id" : { "continent" : "Asia", "country" : "China" }, "first\_city" : "Shanghai", "highest\_population" : 25.582 }

{ "\_id" : { "continent" : "Asia", "country" : "Japan" }, "first\_city" : "Tokyo", "highest\_population" : 37.4 }

{ "\_id" : { "continent" : "Asia", "country" : "South Korea" }, "first\_city" : "Seoul", "highest\_population" : 25.674 }

{ "\_id" : { "continent" : "Asia", "country" : "India" }, "first\_city" : "Delhi", "highest\_population" : 28.514 }

{ "\_id" : { "continent" : "North America", "country" : "Mexico" }, "first\_city" : "Mexico City", "highest\_population" : 21.581 }

Next, sort the results according by their highest\_population value. To do so, add another $sort stage:

1. db.cities.aggregate([
2. {
3. $match: {
4. "continent": { $in: ["North America", "Asia"] }
5. }
6. },
7. {
8. $sort: { "population": -1 }
9. },
10. {
11. $group: {
12. "\_id": {
13. "continent": "$continent",
14. "country": "$country"
15. },
16. "first\_city": { $first: "$name" },
17. "highest\_population": { $max: "$population" }
18. }
19. },
20. {
21. $match: {
22. "highest\_population": { $gt: 20.0 }
23. }
24. },
25. {
26. $sort: { "highest\_population": -1 }
27. }
28. ])

Copy

The document structure doesn’t change, and MongoDB still returns five documents corresponding to the country groups. This time, however, Japan appears first since Tokyo is the most populated city of all in the data set:

Output

{ "\_id" : { "continent" : "Asia", "country" : "Japan" }, "first\_city" : "Tokyo", "highest\_population" : 37.4 }

{ "\_id" : { "continent" : "Asia", "country" : "India" }, "first\_city" : "Delhi", "highest\_population" : 28.514 }

{ "\_id" : { "continent" : "Asia", "country" : "South Korea" }, "first\_city" : "Seoul", "highest\_population" : 25.674 }

{ "\_id" : { "continent" : "Asia", "country" : "China" }, "first\_city" : "Shanghai", "highest\_population" : 25.582 }

{ "\_id" : { "continent" : "North America", "country" : "Mexico" }, "first\_city" : "Mexico City", "highest\_population" : 21.581 }

The last requirement is to transform the document structure to match the sample shown previously. For your review, here’s that sample once more:

Example document

{

"location" : {

"country" : "Japan",

"continent" : "Asia"

},

"most\_populated\_city" : {

"name" : "Tokyo",

"population" : 37.4

}

}

This sample’s location embedded document resembles the \_id grouping expression value, as both include country and continent fields. The most populated city name and population are nested as an embedded document under the most\_populated\_city field. This is different from the grouping results, where all computed fields are top-level fields.

To transform the results to align with this structure, add a $project stage to the pipeline:

1. db.cities.aggregate([
2. {
3. $match: {
4. "continent": { $in: ["North America", "Asia"] }
5. }
6. },
7. {
8. $sort: { "population": -1 }
9. },
10. {
11. $group: {
12. "\_id": {
13. "continent": "$continent",
14. "country": "$country"
15. },
16. "first\_city": { $first: "$name" },
17. "highest\_population": { $max: "$population" }
18. }
19. },
20. {
21. $match: {
22. "highest\_population": { $gt: 20.0 }
23. }
24. },
25. {
26. $sort: { "highest\_population": -1 }
27. },
28. {
29. $project: {
30. "\_id": 0,
31. "location": {
32. "country": "$\_id.country",
33. "continent": "$\_id.continent",
34. },
35. "most\_populated\_city": {
36. "name": "$first\_city",
37. "population": "$highest\_population"
38. }
39. }
40. }
41. ])

This $project stage first suppresses the \_id field from appearing in the output. Then it creates a location field as a nested document containing two fields: country and continent. Using the dollar-sign notation, each of these fields refers to values from the input documents. "$\_id.country" pulls values from the country field from inside the \_id embedded document of the input, and $\_id.continent pulls values from its continent field. most\_populated\_city follows a similar structure, nesting the name and population fields inside. These refer to the top-level fields first\_city and highest\_population, respectively.

This projection stage effectively constructs an entirely new structure for the output, which is as follows:

Output

{ "location" : { "country" : "Japan", "continent" : "Asia" }, "most\_populated\_city" : { "name" : "Tokyo", "population" : 37.4 } }

{ "location" : { "country" : "India", "continent" : "Asia" }, "most\_populated\_city" : { "name" : "Delhi", "population" : 28.514 } }

{ "location" : { "country" : "South Korea", "continent" : "Asia" }, "most\_populated\_city" : { "name" : "Seoul", "population" : 25.674 } }

{ "location" : { "country" : "China", "continent" : "Asia" }, "most\_populated\_city" : { "name" : "Shanghai", "population" : 25.582 } }

{ "location" : { "country" : "Mexico", "continent" : "North America" }, "most\_populated\_city" : { "name" : "Mexico City", "population" : 21.581 } }

This output meets all the requirements defined at the beginning of this step:

* It only includes cities from Asia and North America in the lists.
* For each country and continent pair, a single city is selected, and it’s the city with the highest population.
* The selected city’s name and population are listed.
* Cities are sorted from the most populated to least populated.
* The output format is altered to align with the example document.

**ATOMIC OPERATIONS**

The recommended approach to maintain atomicity would be to keep all the related information, which is frequently updated together in a single document using **embedded documents**. This would make sure that all the updates for a single document are atomic.

db.productDetails.insert(

{

"\_id":1,

"product\_name": "Samsung S3",

"category": "mobiles",

"product\_total": 5,

"product\_available": 3,

"product\_bought\_by": [

{

"customer": "john",

"date": "7-Jan-2014"

},

{

"customer": "mark",

"date": "8-Jan-2014"

}

]

}

)

In this document, we have embedded the information of the customer who buys the product in the **product\_bought\_by** field. Now, whenever a new customer buys the product, we will first check if the product is still available using **product\_available** field. If available, we will reduce the value of product\_available field as well as insert the new customer's embedded document in the product\_bought\_by field. We will use **findAndModify** command for this functionality because it searches and updates the document in the same go.

>db.products.findAndModify({

query:{\_id:2,product\_available:{$gt:0}},

update:{

$inc:{product\_available:-1},

$push:{product\_bought\_by:{customer:"rob",date:"9-Jan-2014"}}

}

})

Our approach of embedded document and using findAndModify query makes sure that the product purchase information is updated only if it the product is available. And the whole of this transaction being in the same query, is atomic.

**User Creation:**

Db.createUser({user:”test”, pwd:”123456”, roles:[{role:”read”, db:”schooldb”}]})

Show users

Show roles

**Authentication**

**Enabling Authorization:**

=>open mongodb config (C:\Program Files\MongoDB\Server\5.0\bin\mongod.cnf)

=>enable security (before enabling, come out of mongosh(exit))

Security:

Authorization:enabled

=>after enabling security, need to restart mongo server

Windows=>run=>services.msc

Goto mongodb=>right click=>restart(important)

After restarting server, test by using find method, it shows error=>command find requires authentication

To authenticate user for a db, exit from mongosh goto cmd and type below code

Mongosh –authenticationDatabase “schooldb” –u “test” –p “123456”

Now you find documents, it will show