Contents

[Install MongoDB 7](#_Toc106622013)

[Ubuntu 7](#_Toc106622014)

[Install MongoDB Community Edition 7](#_Toc106622015)

[Run MongoDB Community Edition 8](#_Toc106622016)

[Uninstall MongoDB Community Edition 10](#_Toc106622017)

[Command Line Shell 11](#_Toc106622018)

[Windows 11](#_Toc106622019)

[Install mongoimport (MongoDB Database Tools) 11](#_Toc106622020)

[Help 11](#_Toc106622021)

[Disadvantages of the mongo Shell 12](#_Toc106622022)

[Alternatives to MongoDB mongo shell 12](#_Toc106622023)

[Concepts Revisited 12](#_Toc106622024)

[Establishing a Connection 13](#_Toc106622025)

[Create the Database & Collections, Add Data 13](#_Toc106622026)

[Selecting Data 15](#_Toc106622027)

[What is “Namespace” in MongoDB 16](#_Toc106622028)

[Things to Remember 16](#_Toc106622029)

[Install MySQL 17](#_Toc106622030)

[Loading data into a table 18](#_Toc106622031)

[Sample Data for Demos 19](#_Toc106622032)

[SQL Tryouts 19](#_Toc106622033)

[SQL vs NoSQL 21](#_Toc106622034)

[What is SQL? 21](#_Toc106622035)

[What is NoSQL? 21](#_Toc106622036)

[KEY DIFFERENCE 21](#_Toc106622037)

[Difference between SQL and NoSQL 22](#_Toc106622038)

[AICD 23](#_Toc106622039)

[BASE 24](#_Toc106622040)

[When to use SQL? 25](#_Toc106622041)

[When to use NoSQL? 25](#_Toc106622042)

[Introduction to MongoDB 26](#_Toc106622043)

[Key Features 26](#_Toc106622044)

[MongoDB Fundamentals: Frequently Asked Questions 27](#_Toc106622045)

[History 27](#_Toc106622046)

[MongoDB: Databases, Schemas, and Tables 28](#_Toc106622047)

[MongoDB and ACID transactions 28](#_Toc106622048)

[MongoDB and caching 28](#_Toc106622049)

[Tools 29](#_Toc106622050)

[Monitoring 29](#_Toc106622051)

[MongoDB CRUD Operations 30](#_Toc106622052)

[Create Operations 30](#_Toc106622053)

[Read the inventory data 30](#_Toc106622054)

[Specify Equality Matches 30](#_Toc106622055)

[Specify Fields to Return (Projection) 31](#_Toc106622056)

[Read Operations 31](#_Toc106622057)

[Update Operations 34](#_Toc106622058)

[Update a Single Document 35](#_Toc106622059)

[Update Multiple Documents 35](#_Toc106622060)

[Replace Single Document 36](#_Toc106622061)

[Replace with Upsert 36](#_Toc106622062)

[Delete Operations 37](#_Toc106622063)

[Delete All Documents 38](#_Toc106622064)

[Delete All Documents that Match a Condition 38](#_Toc106622065)

[Delete Only One Document that Matches a Condition 38](#_Toc106622066)

[Delete Behavior 38](#_Toc106622067)

[Remove All Documents from a Collection 39](#_Toc106622068)

[Remove All Documents that Match a Condition 39](#_Toc106622069)

[Remove a Single Document that Matches a Condition 39](#_Toc106622070)

[CRUD Operation Examples 39](#_Toc106622071)

[Sorting 40](#_Toc106622072)

[Sort Results by a Field 41](#_Toc106622073)

[regex 42](#_Toc106622074)

[$options 42](#_Toc106622075)

[$in Expressions 43](#_Toc106622076)

[Implicit AND Conditions for the Field 43](#_Toc106622077)

[x and s Options 44](#_Toc106622078)

[$regex and $not 44](#_Toc106622079)

[Examples 44](#_Toc106622080)

[Aggregation 46](#_Toc106622081)

[Aggregation Pipeline 46](#_Toc106622082)

[Single Purpose Aggregation Operations 46](#_Toc106622083)

[Map-Reduce**¶** 47](#_Toc106622084)

[Filtered top Subset 47](#_Toc106622085)

[Scenario 47](#_Toc106622086)

[Sample Data Population 47](#_Toc106622087)

[Aggregation Pipeline 49](#_Toc106622088)

[Execution 49](#_Toc106622089)

[Expected Results 49](#_Toc106622090)

[Observations 50](#_Toc106622091)

[Group & Total 50](#_Toc106622092)

[Scenario 50](#_Toc106622093)

[Sample Data Population 50](#_Toc106622094)

[Aggregation Pipeline 51](#_Toc106622095)

[Execution 52](#_Toc106622096)

[Expected Results 52](#_Toc106622097)

[Observations 53](#_Toc106622098)

[Unpack Arrays & Group Differently 54](#_Toc106622099)

[Scenario 54](#_Toc106622100)

[Sample Data Population 54](#_Toc106622101)

[Using $unwind (aggregation) 55](#_Toc106622102)

[Unwind Array 55](#_Toc106622103)

[includeArrayIndex and preserveNullAndEmptyArrays 56](#_Toc106622104)

[Group by Unwound Values 57](#_Toc106622105)

[Unwind Embedded Arrays 58](#_Toc106622106)

[Aggregation Pipeline 61](#_Toc106622107)

[Execution 61](#_Toc106622108)

[Expected Results 61](#_Toc106622109)

[Observations 62](#_Toc106622110)

[Distinct List Of Values 62](#_Toc106622111)

[Scenario 62](#_Toc106622112)

[Sample Data Population 63](#_Toc106622113)

[Aggregation Pipeline 64](#_Toc106622114)

[Execution 64](#_Toc106622115)

[Expected Results 64](#_Toc106622116)

[Observations 65](#_Toc106622117)

[Aggregation with the Zip Code Data Set 65](#_Toc106622118)

[Aggregation with User Preference Data (*Exercise*) 68](#_Toc106622119)

[Normalized Documents 73](#_Toc106622120)

[Indexing 74](#_Toc106622121)

[Default \_id Index 74](#_Toc106622122)

[Create an Index 74](#_Toc106622123)

[Index Names 75](#_Toc106622124)

[Index Types 75](#_Toc106622125)

[Single Field 75](#_Toc106622126)

[Compound Index 75](#_Toc106622127)

[Multikey Index 76](#_Toc106622128)

[Geospatial Index 76](#_Toc106622129)

[Text Indexes 77](#_Toc106622130)

[Hashed Indexes 77](#_Toc106622131)

[Examples 77](#_Toc106622132)

[See Indexes on a Collection 77](#_Toc106622133)

[Drop an Index 77](#_Toc106622134)

[Single Field Index 77](#_Toc106622135)

[Create an Ascending Index on a Single Field 78](#_Toc106622136)

[Create an Index on an Embedded Field 78](#_Toc106622137)

[Create an Index on Embedded Document 78](#_Toc106622138)

[Compound Indexes 79](#_Toc106622139)

[Create a Compound Index 79](#_Toc106622140)

[Sort Order 80](#_Toc106622141)

[Multikey Indexes 80](#_Toc106622142)

[Index Basic Arrays 81](#_Toc106622143)

[Index Arrays with Embedded Documents 81](#_Toc106622144)

[Manage Indexes 82](#_Toc106622145)

[List all Indexes on a Collection 82](#_Toc106622146)

[List All Indexes for a Database 82](#_Toc106622147)

[Remove Indexes 83](#_Toc106622148)

[Schemas and Relations 84](#_Toc106622149)

[One-to-One Relationships with Embedded Documents 84](#_Toc106622150)

[Embedded Document Pattern 84](#_Toc106622151)

[Subset Patternhttps://docs.mongodb.com/manual/tutorial/model-embedded-one-to-one-relationships-between-documents/ - subset-pattern 85](#_Toc106622152)

[Trade-Offs of the Subset Pattern 87](#_Toc106622153)

[One-to-Many Relationships with Embedded Documents 87](#_Toc106622154)

[Embedded Document Pattern 87](#_Toc106622155)

[Subset Pattern 88](#_Toc106622156)

[Trade-Offs of the Subset Pattern 90](#_Toc106622157)

[One-to-Many Relationships with Document References 90](#_Toc106622158)

[Pattern 90](#_Toc106622159)

[MongoDB Drivers - PyMongo 93](#_Toc106622160)

[Pre-requisites for PyMongo 93](#_Toc106622161)

[Install PyMongo on Ubuntu 93](#_Toc106622162)

[Install PyMongo on Windows 93](#_Toc106622163)

[Some Useful Commands 94](#_Toc106622164)

[Making a connection to MongoDB 94](#_Toc106622165)

[Getting a database 94](#_Toc106622166)

[Getting a collection 94](#_Toc106622167)

[Sample Database Operations 94](#_Toc106622168)

[Inserting a Document 95](#_Toc106622169)

[Getting a Single Document With **find\_one()** 95](#_Toc106622170)

[Querying By ObjectId 96](#_Toc106622171)

[Bulk Inserts 97](#_Toc106622172)

[Querying for More Than One Document 98](#_Toc106622173)

[Counting 99](#_Toc106622174)

[Range Queries 99](#_Toc106622175)

[Indexing 100](#_Toc106622176)

[Delete Data 101](#_Toc106622177)

[Update Data 101](#_Toc106622178)

[Replace Data 102](#_Toc106622179)

[Limit and Projection 103](#_Toc106622180)

[Group By 103](#_Toc106622181)

[Another Example of Group By 104](#_Toc106622182)

[Group By - Having 105](#_Toc106622183)

[MongoDB Drivers – Scala 106](#_Toc106622184)

[The Observable API 106](#_Toc106622185)

[Observable 106](#_Toc106622186)

[Helpers used in the Sample 107](#_Toc106622187)

[Aggregation 112](#_Toc106622188)

[Download and setup the Restaurants Dataset 112](#_Toc106622189)

[Connect to a MongoDB Deployment 113](#_Toc106622190)

[Perform Aggregation 113](#_Toc106622191)

[Use Aggregation Expressions 114](#_Toc106622192)

[Exercises 115](#_Toc106622193)

[Structure of 'restaurants' collection: 115](#_Toc106622194)

[Import data using mongoimport 115](#_Toc106622195)

[Read data from python 115](#_Toc106622196)

[Tasks 116](#_Toc106622197)

[More Exercises 120](#_Toc106622198)

[References 121](#_Toc106622199)

[Sample Datasets 121](#_Toc106622200)

# Install MongoDB

Windows: <https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/>

Linux: <https://docs.mongodb.com/manual/administration/install-on-linux/>

Mac: <https://docs.mongodb.com/manual/tutorial/install-mongodb-on-os-x/>

mongoimport: <https://docs.mongodb.com/database-tools/installation/installation/>

## Ubuntu

### Install MongoDB Community Edition

1. Import the public key used by the package management system.

From a terminal, issue the following command to import the MongoDB public GPG Key from <https://www.mongodb.org/static/pgp/server-4.4.asc>

wget -qO - https://www.mongodb.org/static/pgp/server-4.4.asc | sudo apt-key add -

The operation should respond with an OK.

However, if you receive an error indicating that gnupg is not installed, you can:

1. Install gnupg and its required libraries using the following command:

sudo apt-get install gnupg

1. Once installed, retry importing the key:

wget -qO - https://www.mongodb.org/static/pgp/server-4.4.asc | sudo apt-key add -

1. Create a list file for MongoDB.

Create the list file /etc/apt/sources.list.d/mongodb-org-4.4.list for your version of Ubuntu.

Click on the appropriate tab for your version of Ubuntu. If you are unsure of what Ubuntu version the host is running, open a terminal or shell on the host and execute lsb\_release -dc.

Create the /etc/apt/sources.list.d/mongodb-org-4.4.list file for Ubuntu 20.04 (Focal):

* For Ubuntu 20.04 (Focal)

echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/4.4 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-4.4.list

* For Ubuntu 18.04 (Bionic)

echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu bionic/mongodb-org/4.4 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-4.4.list

* For Ubuntu 16.04 (Xenial)

echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/4.4 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-4.4.list

1. Reload local package database.

Issue the following command to reload the local package database:

sudo apt-get update

1. Install the MongoDB packages.

You can install either the latest stable version of MongoDB or a specific version of MongoDB.

* Latest Version

sudo apt-get install -y mongodb-org

* Specific Version

sudo apt-get install -y mongodb-org=4.4 mongodb-org-server=4.4 mongodb-org-shell=4.4 mongodb-org-mongos=4.4 mongodb-org-tools=4.4

If you only install mongodb-org=4.4 and do not include the component packages, the latest version of each MongoDB package will be installed regardless of what version you specified.

Optional. Although you can specify any available version of MongoDB, apt-get will upgrade the packages when a newer version becomes available. To prevent unintended upgrades, you can pin the package at the currently installed version:

echo "mongodb-org hold" | sudo dpkg --set-selections

echo "mongodb-org-server hold" | sudo dpkg --set-selections

echo "mongodb-org-shell hold" | sudo dpkg --set-selections

echo "mongodb-org-mongos hold" | sudo dpkg --set-selections

echo "mongodb-org-tools hold" | sudo dpkg --set-selections

### Run MongoDB Community Edition

**ulimit Considerations**

Most Unix-like operating systems limit the system resources that a process may use. These limits may negatively impact MongoDB operation, and should be adjusted. See [UNIX ulimit Settings](https://docs.mongodb.com/manual/reference/ulimit/) for the recommended settings for your platform.

NOTE

Starting in MongoDB 4.4, a startup error is generated if the ulimit value for number of open files is under 64000.

**Directories**

If you installed via the package manager, the data directory /var/lib/mongodb and the log directory /var/log/mongodb are created during the installation.

By default, MongoDB runs using the mongodb user account. If you change the user that runs the MongoDB process, you must also modify the permission to the data and log directories to give this user access to these directories.

**Configuration File**

The official MongoDB package includes a [configuration file](https://docs.mongodb.com/manual/reference/configuration-options/#std-label-conf-file) (/etc/mongod.conf). These settings (such as the data directory and log directory specifications) take effect upon startup. That is, if you change the configuration file while the MongoDB instance is running, you must restart the instance for the changes to take effect.

**Procedure**

Follow these steps to run MongoDB Community Edition on your system. These instructions assume that you are using the official mongodb-org package -- not the unofficial mongodb package provided by Ubuntu -- and are using the default settings.

**Init System**

To run and manage your [mongod](https://docs.mongodb.com/manual/reference/program/mongod/" \l "mongodb-binary-bin.mongod) process, you will be using your operating system's built-in [init system](https://docs.mongodb.com/manual/reference/glossary/" \l "std-term-init-system). Recent versions of Linux tend to use **systemd** (which uses the systemctl command), while older versions of Linux tend to use **System V init** (which uses the service command).

If you are unsure which init system your platform uses, run the following command:

ps --no-headers -o comm 1

1. **Start MongoDB**

You can start the [mongod](https://docs.mongodb.com/manual/reference/program/mongod/" \l "mongodb-binary-bin.mongod) process by issuing the following command:

|  |
| --- |
| sudo systemctl start mongod |

If you receive an error similar to the following when starting [mongod](https://docs.mongodb.com/manual/reference/program/mongod/" \l "mongodb-binary-bin.mongod):

Failed to start mongod.service: Unit mongod.service not found.

Run the following command first:

|  |
| --- |
| sudo systemctl daemon-reload |

Then run the start command above again.

1. **Verify that MongoDB has started successfully.**

|  |
| --- |
| sudo systemctl status mongod  # Press q to quit |

You can optionally ensure that MongoDB will start following a system reboot by issuing the following command:

|  |
| --- |
| sudo systemctl enable mongod |

1. **Stop MongoDB.**

As needed, you can stop the [mongod](https://docs.mongodb.com/manual/reference/program/mongod/" \l "mongodb-binary-bin.mongod) process by issuing the following command:

|  |
| --- |
| sudo systemctl stop mongod |

**4**

1. **Restart MongoDB.**

You can restart the [mongod](https://docs.mongodb.com/manual/reference/program/mongod/" \l "mongodb-binary-bin.mongod) process by issuing the following command:

|  |
| --- |
| sudo systemctl restart mongod |

You can follow the state of the process for errors or important messages by watching the output in the /var/log/mongodb/mongod.log file.

1. **Begin using MongoDB.**

Start a [mongo](https://docs.mongodb.com/manual/reference/program/mongo/#mongodb-binary-bin.mongo) shell on the same host machine as the [mongod](https://docs.mongodb.com/manual/reference/program/mongod/" \l "mongodb-binary-bin.mongod). You can run the [mongo](https://docs.mongodb.com/manual/reference/program/mongo/#mongodb-binary-bin.mongo) shell without any command-line options to connect to a [mongod](https://docs.mongodb.com/manual/reference/program/mongod/" \l "mongodb-binary-bin.mongod) that is running on your localhost with default port 27017:

|  |
| --- |
| mongo |

### Uninstall MongoDB Community Edition

To completely remove MongoDB from a system, you must remove the MongoDB applications themselves, the configuration files, and any directories containing data and logs.

1. **Stop MongoDB.**

Stop the [mongod](https://docs.mongodb.com/manual/reference/program/mongod/" \l "mongodb-binary-bin.mongod) process by issuing the following command:

|  |
| --- |
| sudo service mongod stop |

1. **Remove Packages.**

Remove any MongoDB packages that you had previously installed.

|  |
| --- |
| sudo apt-get purge mongodb-org\* |

1. **Remove Data Directories.**

Remove MongoDB databases and log files.

|  |
| --- |
| sudo rm -r /var/log/mongodb |
| sudo rm -r /var/lib/mongodb |

### Command Line Shell

Start mongo db, verify and open the shell

|  |
| --- |
| sudo systemctl start mongod |
| sudo systemctl status mongod |
| mongo |

## Windows

<https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/>

Download the installer from <https://www.mongodb.com/try/download/community?tck=docs_server>

* Start the installation
  + Do not start MongoDB as a service
  + Do not install MongoDB Compass
* Create the database directory. Open a command prompt and run:

mkdir c:\data\db

* Start MongoDB from the command line with:

"C:\Program Files\MongoDB\Server\4.4\bin\mongod.exe" --dbpath="c:\data\db"

* Open another command prompt and connect to MongoDB with:

"C:\Program Files\MongoDB\Server\4.4\bin\mongo.exe"

## Install mongoimport (MongoDB Database Tools)

<https://docs.mongodb.com/database-tools/installation/installation/>

**Syntax**:

“C:\Program Files\MongoDB\Server\mongodb-database-tools\bin\mongoimport” –file <path to json file> --db <db nme> --collection <collection name>

**Import sample inventory data**:

"C:\Program Files\MongoDB\Server\mongodb-database-tools\bin\mongoimport" --file .\inventory.crud.json --db test --collection inventory

## Help

db // Show current DB in use.

use company // Switch to a different db.

db.employee.insertOne({name: "mark"}); // Insert one document.

**db.employee.find().forEach(printjson) // Fetch data, print in JSON format.**

show dbs // Show all dbs.

help

db.help()

db.dropDatabase() // Drop the current database and associated data files.

db.employee.drop() // Drop a collection.

## Disadvantages of the mongo Shell

* The Mongo shell is strictly a console centric method of data manipulation. While some find it easy and quick, others might not find those characteristics appealing.
* If you are working on multiple sessions, you need multiple terminals.
* If the results are too long, they scroll away.
* Repetitive commands or debugging a function need the programmer to traverse the long command line history manually.

## Alternatives to MongoDB mongo shell

* [MongoDB Compass](https://www.mongodb.com/products/compass) ([MongoDB Compass](https://www.mongodb.com/products/compass))
* [NoSQLBooster](https://nosqlbooster.com/) ([NoSQLBooster](https://nosqlbooster.com/" \t "_blank))
* [Mongo Management Studio](https://mms.litixsoft.de/) ([Mongo Management Studio](https://mms.litixsoft.de/))
* [Robo 3T](https://robomongo.org/) ([Robo 3T](https://robomongo.org/))

## Concepts Revisited

REF: <https://www.bmc.com/blogs/mongodb-overview-getting-started-with-mongodb/>

MongoDB records are called **documents**. Each MongoDB database (You can have many.) includes **collections**, which are a set of JSON documents. Each collection and document has an ObjectID created by MongoDB or supplied by the programmer.

In an RDBMS database:

|  |
| --- |
| **PRODUCT TABLE** |
| Fields: product number, product category |
| **PRODUCT CATEGORY TABLE** |
| Fields: product category, weight, color |

But then you have to do a **join** operation if you want to know the color or weight of a product. But a join is a computationally expensive operation.

MongoDB would store the data like this:

|  |
| --- |
| **PRODUCT DOCUMENTS** |
| Fields: product number, product category, weight, color |

## Establishing a Connection

When you run the mongo command without arguments, it launches the shell and connects to the default local server provided by the mongod process at mongod://127.0.0.1:27017. This means you’re connected to the local host through port 27017.

By default, the mongo shell starts the session by establishing a connection to the test database. You can access the current database through the db object:

> db

test

>

## Create the Database & Collections, Add Data

**Create database “**products**”:**

> use products

> switched to db products

MongoDB doesn’t create the physical database file on the file system until you insert real data into the database. So in this case, products won’t show up in your current database list:

> show dbs

admin 0.000GB

config 0.000GB

local 0.000GB

>

**List collections:**

> db.getCollectionNames()

**Create two collections:**

> db.createCollection("boyDiapers")

{ "ok" : 1 }

> db.createCollection("girlDiapers")

{ "ok" : 1 }

>

**Drop database and collection**:

> use test

> db.somecollection.drop()

> db.dropDatabase()

Add some data:

db.boyDiapers.insert([

{

size: 1,

color: 'blue',

brand: 'toddler tyke',

}

])

db.girlDiapers.insert([

{

size: 1,

color: 'pink',

brand: 'little angel',

}

])

Note:

* We use the format db.(collection).insert to add the document.
* We use the brackets [], which indicates an array, so that we can add more than one document at a time.

Add some more data:

db.boyDiapers.insert([

{

size: 2,

color: 'white',

brand: 'boy large white'

}

])

db.girlDiapers.insert([

{

size: 2,

color: 'while',

brand: 'girl large'

}

])

## Selecting Data

If you use **find** with no arguments it lists all documents. Use **pretty** to display the results in easy-to-read indented JSON format:

> db.girlDiapers.find().pretty()

{

"\_id" : ObjectId("59d1e9d5ccf50b62c5a7af55"),

"size" : 1,

"color" : "pink",

"brand" : "little angel"

}

{

"\_id" : ObjectId("59d1f022ccf50b62c5a7af57"),

"size" : 1,

"color" : "white",

"brand" : "girl large"

}

{

"\_id" : ObjectId("59d1f565ccf50b62c5a7af59"),

"size" : 2,

"color" : "white",

"brand" : "girl large"

}

To find all girl diapers of size 2, add arguments to the find statement:

db.girlDiapers.find({"size":2})

{ "\_id" : ObjectId("59d1f565ccf50b62c5a7af59"), "size" : 2, "color" : "white", "brand" : "girl large" }

Now, you could not search both boy’s and girl’s diapers collections at the same time. MongoDB does not do that. Instead you have to program that in your application that you would code using some driver (PyMongo *later*).

## **What is “Namespace” in MongoDB**

MongoDB stores BSON (Binary Interchange and Structure Object Notation) objects in the collection. The concatenation of the collection name and database name is called a namespace.

The canonical name for a collection or index in MongoDB. The namespace is a combination of the database name and the name of the collection or index, like so: [database-name].[collection-or-index-name]. All documents belong to a namespace.

## Things to Remember

Max size of a MongoDB document is 16MB. The maximum BSON document size is 16 megabytes. The maximum document size helps ensure that a single document cannot use excessive amount of RAM or, during transmission, excessive amount of bandwidth

The name "BSON" is based on the term JSON and stands for "Binary JSON". It is a binary form for representing simple or complex data structures including associative arrays (also known as name-value pairs), integer indexed arrays, and a suite of fundamental scalar types.

The total size of an index entry, which can include structural overhead depending on the BSON type, must be less than 1024 bytes. A single collection can have no more than 64 indexes.

A collection and a document in MongoDB is equivalent to which of the SQL concepts respectively?: Table and Row. The way SQL databases stores data rows in a table, MongoDB stores documents inside collections.

By default, the MongoDB cursor in mongo shell is configured to return how many documents? To get the next set of documents, which command is used?: it, 20

# Install MySQL

<https://dev.mysql.com/doc/refman/8.0/en/windows-installation.html>

* Go to <https://dev.mysql.com/downloads/installer/>
* Select “**Windows (x86, 32-bit), MSI Installer**”, no the “web” installer.
* Register or login with Oracle.
* Download and run the installer.
* Install MySQL as “Developer Default”.
  + Do not install as “Server only”.

Usage: <https://dev.mysql.com/doc/refman/8.0/en/tutorial.html>

$> mysql --help

$> mysql -h *host* -u *user* -p

Enter password: \*\*\*\*\*\*\*\*

Welcome to the MySQL monitor. Commands end with ; or \g.

Your MySQL connection id is 25338 to server version: 8.0.27-standard

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>

If you are logging in on the same machine that MySQL is running on, you can omit the host, and simply use the following:

$> mysql -u user -p

Some MySQL installations permit users to connect as the anonymous (unnamed) user to the server running on the local host. If this is the case on your machine, you should be able to connect to that server by invoking **[mysql](https://dev.mysql.com/doc/refman/8.0/en/mysql.html" \o "4.5.1 mysql — The MySQL Command-Line Client)** without any options:

$> mysql

After you have connected successfully, you can disconnect any time by typing QUIT (or \q) at the mysql> prompt:

mysql> QUIT

Bye

mysql> SELECT VERSION(), CURRENT\_DATE;

mysql> select version(), current\_date;

mysql> SeLeCt vErSiOn(), current\_DATE;

mysql> SELECT VERSION(); SELECT NOW();

mysql> SELECT

-> USER()

-> ,

-> CURRENT\_DATE;

mysql> SHOW DATABASES;

mysql> USE test

Database changed

mysql> CREATE DATABASE menagerie;

mysql> USE menagerie

Database changed

$> mysql -h *host* -u *user* -p menagerie

Enter password: \*\*\*\*\*\*\*\*

mysql> SHOW TABLES;

mysql> CREATE TABLE pet (name VARCHAR(20), owner VARCHAR(20),

species VARCHAR(20), sex CHAR(1), birth DATE, death DATE);

mysql> DESCRIBE pet;

## Loading data into a table

After creating your table, you need to populate it. The LOAD DATA and INSERT statements are useful for this.

Suppose that your pet records can be described as shown here. (Observe that MySQL expects dates in '***YYYY-MM-DD***' format; this may differ from what you are used to.)

| **name** | **owner** | **species** | **sex** | **birth** | **death** |
| --- | --- | --- | --- | --- | --- |
| Fluffy | Harold | cat | f | 1993-02-04 |  |
| Claws | Gwen | cat | m | 1994-03-17 |  |
| Buffy | Harold | dog | f | 1989-05-13 |  |
| Fang | Benny | dog | m | 1990-08-27 |  |
| Bowser | Diane | dog | m | 1979-08-31 | 1995-07-29 |
| Chirpy | Gwen | bird | f | 1998-09-11 |  |
| Whistler | Gwen | bird |  | 1997-12-09 |  |
| Slim | Benny | snake | m | 1996-04-29 |  |

Because you are beginning with an empty table, an easy way to populate it is to create a text file containing a row for each of your animals, then load the contents of the file into the table with a single statement.

You could create a text file pet.txt containing one record per line, with values separated by tabs, and given in the order in which the columns were listed in the [CREATE TABLE](https://dev.mysql.com/doc/refman/8.0/en/create-table.html) statement. For missing values (such as unknown sexes or death dates for animals that are still living), you can use NULL values. To represent these in your text file, use \N (backslash, capital-N). For example, the record for Whistler the bird would look like this (where the whitespace between values is a single tab character):

Whistler Gwen bird \N 1997-12-09 \N

Start MySQL as follows or else “LOAD DATA” will not work:

mysql --local-infile=1 -u root -p

To load the text file pet.txt into the pet table, use this statement:

mysql> LOAD DATA LOCAL INFILE '/path/pet.txt' INTO TABLE pet;

If you created the file on Windows with an editor that uses \r\n as a line terminator, you should use this statement instead:

mysql> LOAD DATA LOCAL INFILE '/path/pet.txt' INTO TABLE pet

LINES TERMINATED BY '\r\n';

When you want to add new records one at a time, the [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statement is useful. In its simplest form, you supply values for each column, in the order in which the columns were listed in the [CREATE TABLE](https://dev.mysql.com/doc/refman/8.0/en/create-table.html) statement. Suppose that Diane gets a new hamster named “Puffball.” You could add a new record using an [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html) statement like this:

mysql> INSERT INTO pet

VALUES ('Puffball','Diane','hamster','f','1999-03-30',NULL);

String and date values are specified as quoted strings here. Also, with [INSERT](https://dev.mysql.com/doc/refman/8.0/en/insert.html), you can insert NULL directly to represent a missing value. You do not use \N like you do with [LOAD DATA](https://dev.mysql.com/doc/refman/8.0/en/load-data.html).

## Sample Data for Demos

<https://github.com/AjaySingala/BigDataTutorials.git/MongoDB/GenratedData>

## SQL Tryouts

<https://www.w3schools.com/sql/trysql.asp?filename=trysql_asc>

# SQL vs NoSQL

<https://www.geeksforgeeks.org/difference-between-sql-and-nosql/>

<https://www.mongodb.com/nosql-explained/nosql-vs-sql>

<https://www.guru99.com/sql-vs-nosql.html>

## What is SQL?

Structured Query language ([SQL](https://www.guru99.com/what-is-sql.html)) **pronounced as “S-Q-L” or sometimes as “See-Quel**” is the standard language for dealing with Relational Databases. A relational database defines relationships in the form of tables.

SQL programming can be effectively used to insert, search, update, delete database records.

That doesn’t mean SQL cannot do things beyond that. It can do a lot of things including, but not limited to, optimizing and maintenance of databases.

Relational databases like MySQL Database, Oracle, Ms SQL Server, Sybase, etc. use SQL.

## What is NoSQL?

[NoSQL](https://www.guru99.com/nosql-tutorial.html) is a non-relational DMS, that does not require a fixed schema, avoids joins, and is easy to scale. NoSQL database is used for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps. For example companies like Twitter, Facebook, Google that collect terabytes of user data every single day.

NoSQL database stands for “Not Only SQL” or “Not SQL.” Though a better term would NoREL NoSQL caught on. Carl Strozz introduced the NoSQL concept in 1998.

Traditional RDBMS uses SQL syntax to store and retrieve data for further insights. Instead, a NoSQL database system encompasses a wide range of database technologies that can store structured, semi-structured, unstructured and polymorphic data.

Next, we will discuss the key diff between SQL and NoSQL.

## KEY DIFFERENCE

**SQL** pronounced as “S-Q-L” or as “See-Quel” is primarily called RDBMS or Relational Databases whereas **NoSQL** is a Non-relational or Distributed Database.

Comparing SQL vs NoSQL database, SQL databases are table based databases whereas NoSQL databases can be document based, key-value pairs, graph databases.

SQL databases are vertically scalable while NoSQL databases are horizontally scalable.

SQL databases have a predefined schema whereas NoSQL databases use dynamic schema for unstructured data.

Comparing NoSQL vs SQL performance, SQL requires specialized DB hardware for better performance while NoSQL uses commodity hardware.

## Difference between SQL and NoSQL

1. SQL is used to work on structured data in a relational format. NoSQL is used to work on structured and non-structured data.
2. SQL is schema-based. NoSQL is not (no fixed schema).
3. SQL suppports Structured Query Language. NoSQL is more of a declarative query language.
4. SQL table-based data. NoSQL is document-oriented.
5. SQL is vertically scalable. NoSQL is horizontally scalable.
6. SQL is more for transactional data. NoSQL is more for Big Data (distributed).
7. SQL is relational. NoSQL is non-relational.
8. SQL does not support hierarchical data. NonSQL supports hierarchical data.
9. Features: **SQL**: ACID (Atomicity, Consistency, Isolation, Durability). **NoSQL**: CAP (Consistency, Availability, Partition Tolerance) and BASE (Basically Available, Soft State, Eventual Consistency)

Below is the main difference between NoSQL and SQL:

| **Parameter** | **SQL** | **NOSQL** |
| --- | --- | --- |
| Definition | SQL databases are primarily called RDBMS or Relational Databases | NoSQL databases are primarily called as Non-relational or distributed database |
| Design for | Traditional RDBMS uses SQL syntax and queries to analyze and get the data for further insights. They are used for OLAP systems. | NoSQL database system consists of various kind of database technologies. These databases were developed in response to the demands presented for the development of the modern application. |
| Query Language | Structured query language (SQL) | No declarative query language |
| Type | SQL databases are table based databases | NoSQL databases can be document based, key-value pairs, graph databases |
| Schema | SQL databases have a predefined schema | NoSQL databases use dynamic schema for unstructured data. |
| Ability to scale | SQL databases are vertically scalable | NoSQL databases are horizontally scalable |
| Examples | Oracle, Postgres, and MS-SQL. | [MongoDB](https://www.guru99.com/mongodb-tutorials.html), Redis, Neo4j, Cassandra, Hbase. |
| Best suited for | An ideal choice for the complex query intensive environment. | It is not good fit complex queries. |
| Hierarchical data storage | SQL databases are not suitable for hierarchical data storage. | More suitable for the hierarchical data store as it supports key-value pair method. |
| Variations | One type with minor variations. | Many different types which include key-value stores, document databases, and graph databases. |
| Development Year | It was developed in the 1970s to deal with issues with flat file storage | Developed in the late 2000s to overcome issues and limitations of SQL databases. |
| Open-source | A mix of open-source like Postgres & MySQL, and commercial like Oracle Database. | Open-source |
| Consistency | It should be configured for strong consistency. | It depends on DBMS as some offers strong consistency like MongoDB, whereas others offer only offers eventual consistency, like [Cassandra](https://www.guru99.com/cassandra-tutorial.html). |
| Best Used for | RDBMS database is the right option for solving ACID problems. | NoSQL is a best used for solving data availability problems |
| Importance | It should be used when data validity is super important | Use when it’s more important to have fast data than correct data |
| Best option | When you need to support dynamic queries | Use when you need to scale based on changing requirements |
| Hardware | Specialized DB hardware (Oracle Exadata, etc.) | Commodity hardware |
| Network | Highly available network (Infiniband, Fabric Path, etc.) | Commodity network (Ethernet, etc.) |
| Storage Type | Highly Available Storage (SAN, RAID, etc.) | Commodity drives storage (standard HDDs, JBOD) |
| Best features | Cross-platform support, Secure and free | Easy to use, High performance, and Flexible tool. |
| Top Companies Using | Hootsuite, CircleCI, Gauges | Airbnb, Uber, Kickstarter |
| Average salary | The average salary for any professional SQL Developer is $84,328 per year in the U.S.A. | The average salary for “NoSQL developer” ranges from approximately $72,174 per year |
| ACID vs. BASE Model | [ACID](https://www.guru99.com/dbms-transaction-management.html)( Atomicity, Consistency, Isolation, and Durability) is a standard for RDBMS | Base ( Basically Available, Soft state, Eventually Consistent) is a model of many NoSQL systems |

Difference between ACID vs BASE in DBMS:

Diagram

Description automatically generated

### AICD

* Atomicity
* Consistency
* Isolation
* Durability

Together, ACID is a set of guiding principles that ensure database transactions are processed reliably. A database transaction is any operation performed within a database, such as creating a new record or updating data within one.

Changes made within a database need to be performed with care to ensure the data within doesn’t become corrupted. Applying the ACID properties to each modification of a database is the best way to maintain the accuracy and reliability of a database.

Let’s look at each component of ACID.

* Atomicity:
  + Commits finish an entire operation successfully or the database rolls back to its prior state.
  + Essentially, an atomic transaction ensures that any commit you make finishes the entire operation successfully.  Or, in cases of a lost connection in the middle of an operation, the database is rolled back to its state prior to the commit being initiated.
* Consistency:
  + Any change maintains data integrity or is cancelled completely.
  + A consistent transaction will not violate integrity constraints placed on the data by the database rules. Enforcing consistency ensures that if a database enters into an illegal state (if a violation of data integrity constraints occurs) the process will be aborted and changes rolled back to their previous, legal state.
  + An example of a declarative constraint might be that all customer accounts must have a positive balance. If a transaction would bring a customer account into a negative balance, that transaction would be rolled back. This ensures changes are successful at maintaining data integrity or they are canceled completely.
* Isolation:
  + Any read or write will not be impacted by other reads or writes of separate transactions on the same database.
  + Isolated transactions are considered to be “serializable”, meaning each transaction happens in a distinct order without any transactions occurring in tandem.
  + Importantly, this doesn’t mean two operations can’t happen at the same time. Multiple transactions can occur as long as those transactions have no possibility of impacting the other transactions occurring at the same time.
* Durability:
  + Successful commits will survive permanently.
  + Durability ensures that changes made to the database (transactions) that are successfully committed will survive permanently, even in the case of system failures. This ensures that the data within the database will not be corrupted by:
    - Service outages
    - Crashes
    - Other cases of failure

### BASE

* Basically Available:
  + The system is guaranteed to be available in event of failure.
* Soft State:
  + The state of the data could change without application interactions due to eventual consistency.
* Eventual Consistency:
  + The system will be eventually consistent after the application input. The data will be replicated to different nodes and will eventually reach a consistent state. But the consistency is not guaranteed at a transaction level.

## When to use SQL?

* SQL is the easiest language used to communicate with the RDBMS
* Analyzing behavioral related and customized sessions
* Building custom dashboards
* It allows you to store and gets data from the database quickly
* Preferred when you want to use joins and execute complex queries

## When to use NoSQL?

The below image shows the Google trends for NoSQL vs SQL:

* When ACID support is not needed
* When Traditional RDBMS model is not enough
* Data which need a flexible schema
* Constraints and validations logic not required to be implemented in database
* Logging data from distributed sources
* It should be used to store temporary data like shopping carts, wish list and session data

# Introduction to MongoDB

*MongoDB* is an Open Source database written in C++.

Drivers and client libraries are typically written in their respective languages, although some drivers use C extensions for better performance.

If the load increases, by adding more nodes (such as a computer), the performance can be retained.

It can be used to store data for very high-performance applications (for example Foursquare is using it in production).

MongoDB does not support SQL It supports a rich, ad-hoc query language of its own.

MongoDB stores data as documents. So, it is a *document oriented database*.

FirstName="Arun", Address="St. Xavier's Road", Spouse=[{Name:"Kiran"}], Children=[{Name:"Rihit", Age:8}]. FirstName="Sameer",Address="8 Gandhi Road".

Notice there are two different documents (separated by "."). Storing data in this fashion is called as *document-oriented database*. *MongoDB* is a document-oriented database.

## Key Features

Since MongoDB offers a Document oriented storage, It is simple and easily programmable.

* You can set an index on any attribute of a MongoDB record (as FirstName="Sameer",Address="8 Gandhi Road"), with respect to which, a record can be sort quickly and ordered.
* You can set mirror across local as well as wide area networks, which makes it easily scalable.
* If load increases (more storage space, more processing power), it can be distributed to other nodes across computer networks. This is called as sharding.

**Sharding**:

* + It is a method of distributing data across multiple servers.
  + You may have large sets:
    - for e.g., you have 2 million documents in a collection
    - run a query: it will have to scan through all 2 mil docs.
    - Solution: create "shards"
      * based a criteria, data is split into shards. This is the "shard key".
      * Each shard is on a different server (horizontal scaling)
      * For e.g., qty 1-50. qty > 50
        + 2 shards: 1-5 750k. >50 1.25mil
      * when you run a query with a condition like {'qty' = 10}
        + it will only scan thru that particular "shard"
      * when you run a query with a condition like {'qty' > 75}
* MongoDB supports rich query to fetch data from the database.
* MongoDB supports replacing an entire document (database) or some specific fields with it's update() command.
* MongoDB supports Map/Reduce framework for the batch processing of data and aggregation operation. Here is brief of how Map/Reduce works :
* Map : A master node takes an input. Splits it into smaller sections. Sends it to the associated nodes.
* These nodes may perform the same operation in turn to send those smaller section of input to other nodes. It processes the problem (taken as input) and sends it back to the Master Node.
* Reduce : The master node aggregates those results to find the output.
* GridFS specification of MongoDB supports storage of very large files.
* MongoDB supports various programming languages like C, C# and .NET, C++, Erlang, Haskell, Java, Javascript, Perl, PHP, Python, Ruby, Scala (via Casbah).
* It supports Server-side JavaScript execution. Which allows a developer to use a single programming language for both client and server side code.
* MongoDB is easily installable.

## MongoDB Fundamentals: Frequently Asked Questions

What kind of database is MongoDB?

MongoDB is a document-oriented DBMS. Think of MySQL but with JSON-like objects comprising the data mode, rather than RDBMS tables. MongoDB supports neither joins nor transactions.

Does MongoDB database store its data in tables?

A MongoDB database stores its data in collections instead of tables, which are the rough equivalent of RDBMS tables.

Do MongoDB databases have schemas?

MongoDB uses dynamic schemas. Without defining the structure, you can create collections, i.e. the fields or the types of their values, of the documents in the collection. You can change the structure of documents simply by adding new fields or deleting existing ones.

Which programming languages can be used to work with MongoDB?

MongoDB client drivers exist for all the most popular programming languages.

Does MongoDB support SQL?

No.

What are typical uses for MongoDB?

Content management systems, mobile applications, gaming, e-commerce, analytics, archiving, and logging.

Does MongoDB support transactions?

No. However, MongoDB does provide atomic operations on a single document.

Does MongoDB require a lot of RAM?

Not necessarily. It's certainly possible to run MongoDB on a machine with a small amount of free RAM. MongoDB automatically uses all free memory on the machine as its cache.

Does MongoDB handle caching?

Yes. MongoDB keeps all the most recently used data in RAM. If you have created indexes for your queries and your working data set fits in RAM, MongoDB serves all queries from memory.

What are the limitations of 32-bit versions of MongoDB?

Changed in version 3.0: Commercial support is no longer provided for MongoDB on 32-bit platforms (Linux and Windows).

## History

Development of MongoDB began in October 2007 by 10gen. The first public release was in February 2009.

## MongoDB: Databases, Schemas, and Tables

**Databases:** MongoDB is a document-oriented DBMS, with JSON-like objects comprising the data model, rather than RDBMS tables. MongoDB does not support joins nor transactions. However, it features secondary indexes, an expressive query language, atomic writes on a per-document level, and fully consistent reads. MongoDB uses BSON, a binary object format similar to, but more expressive than JSON.

**Schemas:** MongoDB uses dynamic schemas. We can create collections without defining the structure, i.e., the fields or the types of their values, of the documents. You can change the structure of documents simply by adding new fields or deleting existing ones. Documents in a collection need unique set of fields.

**Tables:** MongoDB database stores its data in collections not in tables The collections are the rough equivalent of RDBMS tables. A collection holds one or more documents, which corresponds to a record or a row in a relational database table, and each document has one or more fields, which corresponds to a column in a relational database table.

## MongoDB and ACID transactions

MongoDB does not support multi-document transactions but provides atomic operations on a single document. Often these document-level atomic operations are sufficient to solve problems that would require ACID transactions in a relational database.

In MongoDB, you can embed related data in nested arrays or nested documents within a single document and update the entire document in a single atomic operation. Relational databases might represent the same kind of data with multiple tables and rows, which would require transaction support to update the data atomically.

MongoDB allows clients to read documents inserted or modified before it commits these modifications to disk, regardless of write concern level or journaling configuration. Applications may observe two classes of behaviors:

* MongoDB will allow clients to read the results of a write operation before the write operation returns for systems with multiple concurrent readers and writers
* If the MongoDB terminates before the journal commits, even if a write returns successfully, queries may have read data that will not exist after the MongoDB restarts.

Other database systems refer to this isolation semantics as read uncommitted. For all inserts and updates, MongoDB modifies

* each document in isolation
* clients never see documents in intermediate states

For multi-document operations, MongoDB does not provide any multi-document transactions or isolation. When MongoDB returns a successful journaled write concern, the data is fully committed to disk and will be available after MongoDB restarts. For replica sets, write operations are durable only after a write replicates and commits to the journal of a majority of the voting members of the set.

## MongoDB and caching

MongoDB has no configurable cache. MongoDB uses all free memory on the system automatically by way of memory-mapped files. Operating systems use the same approach with their file system caches. MongoDB keeps all the most recently used data in RAM. If you have created indexes for your queries and your working data set fits in RAM, MongoDB serves all queries from memory.

MongoDB does not implement a query cache; it serves all queries directly from the indexes and/or data files.

## Tools

There are several tools available for managing MongoDB.

### Monitoring

Network and System monitoring tool Munin has a plugin available for MongoDB.

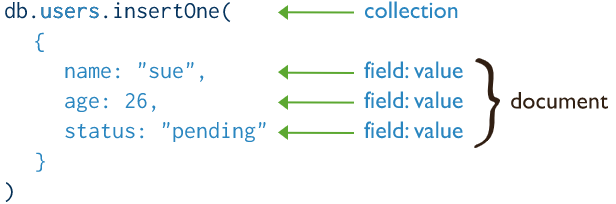
Distributed high-performance system monitoring tool Gangila has a plugin available for MongoDB.

Open-source web based graphic tool Cacti, used to graph CPU load, network bandwidth utilization has a plugin available for MongoDB.

# MongoDB CRUD Operations

## Create Operations

* [db.collection.insertOne()](https://docs.mongodb.com/manual/reference/method/db.collection.insertOne/#mongodb-method-db.collection.insertOne) *New in version 3.2*
* [db.collection.insertMany()](https://docs.mongodb.com/manual/reference/method/db.collection.insertMany/#mongodb-method-db.collection.insertMany) *New in version 3.2*



db.inventory.insertMany([

{ item: "journal", qty: 25, status: "A", size: { h: 14, w: 21, uom: "cm" }, tags: [ "blank", "red" ] },

{ item: "notebook", qty: 50, status: "A", size: { h: 8.5, w: 11, uom: "in" }, tags: [ "red", "blank" ] },

{ item: "paper", qty: 10, status: "D", size: { h: 8.5, w: 11, uom: "in" }, tags: [ "red", "blank", "plain" ] },

{ item: "planner", qty: 0, status: "D", size: { h: 22.85, w: 30, uom: "cm" }, tags: [ "blank", "red" ] },

{ item: "postcard", qty: 45, status: "A", size: { h: 10, w: 15.25, uom: "cm" }, tags: [ "blue" ] }

]);

*// MongoDB adds an \_id field with an ObjectId value if the field is not present in the document*

### Read the inventory data

db.inventory.find({})

To format the results, append the .pretty() to the find operation:

|  |
| --- |
| db.inventory.find({}).pretty() |

### Specify Equality Matches

For an equality match (i.e. <field> equals <value>), specify <field>: <value> in the query filter document and pass to the [db.collection.find()](https://docs.mongodb.com/manual/reference/method/db.collection.find/" \l "mongodb-method-db.collection.find) method.

* In the shell, copy and paste the following to return documents where status field equals "D":

|  |
| --- |
| db.inventory.find( { status: **"D"** } ); |

* In the shell, copy and paste the following to return document where qty field equals 0:

|  |
| --- |
| db.inventory.find( { qty: 0 } ); |

* In the shell, copy and paste the following to return document where qty field equals 0 and status field equals "D":

|  |
| --- |
| db.inventory.find( { qty: 0, status: **"D"** } ); |

* In the shell, copy and paste the following to return document where the uom field, nested inside the size document, equals "in":

|  |
| --- |
| db.inventory.find( { **"size.uom"**: **"in"** } ) |

* In the shell, copy and paste the following to return document where the size field equals the document { h: 14, w: 21, uom: "cm" }:

|  |
| --- |
| db.inventory.find( { size: { h: 14, w: 21, uom: **"cm"** } } ) |

Equality matches on the embedded document require an exact match, including the field order.

* In the shell, copy and paste the following to return documents where the tags array contains "red" as one of its elements:

|  |
| --- |
| db.inventory.find( { tags: **"red"** } ) |

If the tags field is a string instead of an array, then the query is just an equality match.

* In the shell, copy and paste the following to return documents where the tags field matches the specified array exactly, including the order:

|  |
| --- |
| db.inventory.find( { tags: [ **"red"**, **"blank"** ] } ) |

### Specify Fields to Return (Projection)

To specify fields to return, pass a projection document to the [db.collection.find(<query document>, <projection document>)](https://docs.mongodb.com/manual/reference/method/db.collection.find/" \l "mongodb-method-db.collection.find) method. In the projection document, specify:

* <field>: 1 to include a field in the returned documents
* <field>: 0 to exclude a field in the returned documents

In the shell, copy and paste the following to return the \_id, item, and the status fields from all documents in the inventory collection:

|  |
| --- |
| db.inventory.find( { }, { item: 1, status: 1 } ); |

You do not have to specify the \_id field to return the field. It returns by default. To exclude the field, set it to 0 in the projection document. For example, copy and paste the following to return only the item, and the status fields in the matching documents:

|  |
| --- |
| db.inventory.find( {}, { \_id: 0, item: 1, status: 1 } ); |

## Read Operations

* [db.collection.find()](https://docs.mongodb.com/manual/reference/method/db.collection.find/#mongodb-method-db.collection.find)



**Sample data**:

{\_id: ObjectId("5effaa5662679b5af2c58829"),

email: “email@example.com”,

name: {**given**: “Jesse”, family: “Xiao”},

age: 31,

addresses: [{label: “home”,

street: “101 Elm Street”,

city: “Springfield”,

**state**: “CA”,

zip: “90000”,

country: “US”},

{label: “mom”,

street: “555 Main Street”,

city: “Jonestown”,

province: “Ontario”,

country: “CA”}]

}

#### Count Documents in a Collection

> use my\_database;

> db.users.count()

#### Find the First Document in a Collection

> db.users.findOne()

{

"\_id": ObjectId("5ce45d7606444f199acfba1e"),

"name": {**given**: "Alex", family: "Smith"},

"email": "email@example.com"

"age": 27

}

>

#### Find a Document by ID

> db.users.findOne({\_id: ObjectId("5ce45d7606444f199acfba1e")})

{

"\_id": ObjectId("5ce45d7606444f199acfba1e"),

"name": {**given**: "Alex", family: "Smith"},

"email": "email@example.com",

"age": 27

}

>

#### Find a Limited Number of Results

> db.users.find().limit(10)

…

>

#### Find Users by Family name

> db.users.find({"name.family": "Smith"}).count()

1

>

Note that we enclose “name.family” in quotes, because it has a dot in the middle.

#### Query Documents by Numeric Ranges

// All posts having “likes” field with numeric value greater than one:

> db.post.find({likes: {$gt: 1}})

// All posts having 0 likes

> db.post.find({likes: 0})

// All posts that do NOT have exactly 1 like

> db.post.find({likes: {$ne: 1}})

* Read specific columns (*from the test.inventory collection imported earlier*):
  + Display all fields and all documents

db.inventory.find()

* + Display only the item field

db.inventory.find({}, {"item": 1})

* + Display only the item and qty fields

db.inventory.find({}, {"item": 1, "qty": 1})

* + Display only the item and qty fields, but exclude the \_id field

db.inventory.find({}, {"item": 1, "qty": 1, "\_id": 0})

* + Display only the item and qty fields for all “journal” items.

db.inventory.find({"item": "journal"}, {"item": 1, "qty": 1})

* + Display the first 5 documents and only the item and qty fields for all “journal” items.

db.inventory.find({"item": "journal"}, {"item": 1, "qty": 1}).limit(5)

* + Display the next 5 documents after the first 5 documents and only the item and qty fields for all “journal” items.

db.inventory.find({"item": "journal"}, {"item": 1, "qty": 1}).skip(5).limit(5)

* + Display all documents, all fields where the height is greater than 10.

db.inventory.find({"size.h": {$gt: 10}})

* + Display all documents, all fields where the uom is “in”.

db.inventory.find({"size.uom": {$eq: "size.in"}})

* + Display all documents, all fields where the uom is “cm” and height is greater than 10.

db.inventory.find({$and: [{"size.uom": {$eq: "cm"}}, {"size.h": {$gt: 10}}]})

* + Display all documents, and only the item field where the uom not “in” and height is greater than 10.

db.inventory.find({$and: [{"size.uom": {$ne: "in"}}, {"size.h": {$gt: 10}}]}, {"item": 1})

* + Display all documents, and only the item field where the uom not “in” and height is greater than or equal 10.

db.inventory.find({$and: [{"size.uom": {$ne: "in"}}, {"size.h": {$gte: 10}}]}, {"item": 1})

* + Display all documents, and only the item, height and uom fields where the uom not “in” and height is greater than or equal 10.

db.inventory.find({$and: [{"size.uom": {$ne: "in"}}, {"size.h": {$gte: 10}}]}, {"item": 1, "size.h":1, "size.uom":1})

* + Display all documents, and only the item, height and uom fields where the height is either 10 or 14.

db.inventory.find({$or: [{"size.h": {$eq: 14}}, {"size.h": {$eq: 10}}]}, {"item": 1, "size.h":1, "size.uom":1})

* + Display all documents where the item name starts with “p”.

db.inventory.find({"item": {$regex: /^p/,}})

* + Display all documents where the item name starts with “p” or “P”.

db.inventory.find({"item": {$regex: /^p/i,}})

## Update Operations

* [db.collection.update()](https://docs.mongodb.com/manual/reference/method/db.collection.updateOne/#mongodb-method-db.collection.updateOne)
* [db.collection.updateOne()](https://docs.mongodb.com/manual/reference/method/db.collection.updateOne/#mongodb-method-db.collection.updateOne) *New in version 3.2*
* [db.collection.updateMany()](https://docs.mongodb.com/manual/reference/method/db.collection.updateMany/#mongodb-method-db.collection.updateMany) *New in version 3.2*
* [db.collection.replaceOne()](https://docs.mongodb.com/manual/reference/method/db.collection.replaceOne/#mongodb-method-db.collection.replaceOne) *New in version 3.2*

.update(): by default it will update only a single matching document. Include the option { multi: true } to update all matching documents.

# Single match.

db.users.update(

{ age: {$lt: 18 } },

{ $set: { status: ‘reject’ } }

)

# All matches.

db.users.update(

{ age: {$lt: 18 } },

{ $set: { status: ‘reject’ } },

{ multi: true }

)



Refer to Query Selectors for filters [here](#_Query_and_Projection).

**Sample Data**:

|  |
| --- |
| db.inventory.insertMany( [ |
| { item: **"canvas"**, qty: 100, size: { h: 28, w: 35.5, uom: **"cm"** }, status: **"A"** }, |
| { item: **"journal"**, qty: 25, size: { h: 14, w: 21, uom: **"cm"** }, status: **"A"** }, |
| { item: **"mat"**, qty: 85, size: { h: 27.9, w: 35.5, uom: **"cm"** }, status: **"A"** }, |
| { item: **"mousepad"**, qty: 25, size: { h: 19, w: 22.85, uom: **"cm"** }, status: **"P"** }, |
| { item: **"notebook"**, qty: 50, size: { h: 8.5, w: 11, uom: **"in"** }, status: **"P"** }, |
| { item: **"paper"**, qty: 100, size: { h: 8.5, w: 11, uom: **"in"** }, status: **"D"** }, |
| { item: **"planner"**, qty: 75, size: { h: 22.85, w: 30, uom: **"cm"** }, status: **"D"** }, |
| { item: **"postcard"**, qty: 45, size: { h: 10, w: 15.25, uom: **"cm"** }, status: **"A"** }, |
| { item: **"sketchbook"**, qty: 80, size: { h: 14, w: 21, uom: **"cm"** }, status: **"A"** }, |
| { item: **"sketch pad"**, qty: 95, size: { h: 22.85, w: 30.5, uom: **"cm"** }, status: **"A"** } |
| ] ); |

### Update a Single Document

The following example uses the [db.collection.updateOne()](https://docs.mongodb.com/manual/reference/method/db.collection.updateOne/" \l "mongodb-method-db.collection.updateOne) method on the inventory collection to update the first document where item equals "paper":

|  |
| --- |
| db.inventory.updateOne( |
| { item: **"paper"** }, |
| { |
| $set: { **"size.uom"**: **"cm"**, status: **"P"** }, |
| $currentDate: { lastModified: true } |
| } |
| ) |

The update operation:

* uses the [$set](https://docs.mongodb.com/manual/reference/operator/update/set/#mongodb-update-up.-set) operator to update the value of the size.uom field to "cm" and the value of the status field to "P",
* uses the [$currentDate](https://docs.mongodb.com/manual/reference/operator/update/currentDate/#mongodb-update-up.-currentDate) operator to update the value of the lastModified field to the current date. If lastModified field does not exist, [$currentDate](https://docs.mongodb.com/manual/reference/operator/update/currentDate/#mongodb-update-up.-currentDate) will create the field. See [$currentDate](https://docs.mongodb.com/manual/reference/operator/update/currentDate/#mongodb-update-up.-currentDate) for details.

You can also do an Upsert with .updateOne() (*will work with updateMany() too*):

db.girlDiapers.updateOne(

{"size":5},

{$set: {"size":5, "color":"white", "brand":"girl xx-large"}},

{upsert: true}

)

### Update Multiple Documents

|  |
| --- |
| db.inventory.updateMany( |
| { **"qty"**: { $lt: 50 } }, |
| { |
| $set: { **"size.uom"**: **"in"**, status: **"P"** }, |
| $currentDate: { lastModified: true } |
| } |
| ) |

The update operation:

* uses the [$set](https://docs.mongodb.com/manual/reference/operator/update/set/#mongodb-update-up.-set) operator to update the value of the size.uom field to "in" and the value of the status field to "P",
* uses the [$currentDate](https://docs.mongodb.com/manual/reference/operator/update/currentDate/#mongodb-update-up.-currentDate) operator to update the value of the lastModified field to the current date. If lastModified field does not exist, [$currentDate](https://docs.mongodb.com/manual/reference/operator/update/currentDate/#mongodb-update-up.-currentDate) will create the field. See [$currentDate](https://docs.mongodb.com/manual/reference/operator/update/currentDate/#mongodb-update-up.-currentDate) for details.

### Replace Single Document

|  |
| --- |
| db.inventory.replaceOne( |
| { item: **"paper"** }, |
| { item: **"paper"**, instock: [ { warehouse: **"A"**, qty: 60 }, { warehouse: **"B"**, qty: 40 } ] } |
| ) |

Another example:

|  |
| --- |
| { **"\_id"** : 1, **"name"** : **"Central Perk Cafe"**, **"Borough"** : **"Manhattan"** }, |
| { **"\_id"** : 2, **"name"** : **"Rock A Feller Bar and Grill"**, **"Borough"** : **"Queens"**, **"violations"** : 2 }, |
| { **"\_id"** : 3, **"name"** : **"Empire State Pub"**, **"Borough"** : **"Brooklyn"**, **"violations"** : 0 } |

The following operation replaces a single document where name: "Central Perk Cafe":

|  |
| --- |
| db.restaurant.replaceOne( |
| { **"name"** : **"Central Perk Cafe"** }, |
| { **"name"** : **"Central Pork Cafe"**, **"Borough"** : **"Manhattan"** } |
| ); |

The operation returns:

|  |
| --- |
| { **"acknowledged"** : true, **"matchedCount"** : 1, **"modifiedCount"** : 1 } |

If no matches were found, the operation instead returns:

|  |
| --- |
| { **"acknowledged"** : true, **"matchedCount"** : 0, **"modifiedCount"** : 0 } |

Setting upsert: true would insert the document if no match was found.

### Replace with Upsert

The following operation attempts to replace the document with name : "Pizza Rat's Pizzaria", with upsert : true:

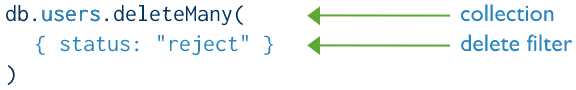
|  |
| --- |
| db.restaurant.replaceOne( |
| { **"name"** : **"Pizza Rat's Pizzaria"** }, |
| { **"\_id"**: 4, **"name"** : **"Pizza Rat's Pizzaria"**, **"Borough"** : **"Manhattan"**, **"violations"** : 8 }, |
| { upsert: true } |
| ); |

Since upsert : true the document is inserted based on the replacement document. The operation returns:

|  |
| --- |
| { |
| **"acknowledged"** : true, |
| **"matchedCount"** : 0, |
| **"modifiedCount"** : 0, |
| **"upsertedId"** : 4 |
| } |

## Delete Operations

* [db.collection.deleteOne()](https://docs.mongodb.com/manual/reference/method/db.collection.deleteOne/#mongodb-method-db.collection.deleteOne) *New in version 3.2*
* [db.collection.deleteMany()](https://docs.mongodb.com/manual/reference/method/db.collection.deleteMany/#mongodb-method-db.collection.deleteMany) *New in version 3.2*



The following operation deletes the order with \_id: ObjectId("563237a41a4d68582c2509da") :

|  |
| --- |
| try { |
| db.orders.deleteOne( { **"\_id"** : ObjectId(**"563237a41a4d68582c2509da"**) } ); |
| } catch (e) { |
| print(e); |
| } |

The operation returns:

|  |
| --- |
| { **"acknowledged"** : true, **"deletedCount"** : 1 } |

The following operation deletes all documents where client : "Crude Traders Inc.":

|  |
| --- |
| try { |
| db.orders.deleteMany( { **"client"** : **"Crude Traders Inc."** } ); |
| } catch (e) { |
| print (e); |
| } |

The operation returns:

|  |
| --- |
| { **"acknowledged"** : true, **"deletedCount"** : 10 } |

The following operation deletes all documents where stock : "Brent Crude Futures" and limit is greater than 48.88:

|  |
| --- |
| try { |
| db.orders.deleteMany( { **"stock"** : **"Brent Crude Futures"**, **"limit"** : { $gt : 48.88 } } ); |
| } catch (e) { |
| print (e); |
| } |

The operation returns:

|  |
| --- |
| { **"acknowledged"** : true, **"deletedCount"** : 8 } |

### Delete All Documents

To delete all documents from a collection, pass an empty filter document {} to the [db.collection.deleteMany()](https://docs.mongodb.com/manual/reference/method/db.collection.deleteMany/" \l "mongodb-method-db.collection.deleteMany) method.

The following example deletes all documents from the inventory collection:

|  |
| --- |
| db.inventory.deleteMany({}) |

The method returns a document with the status of the operation. For more information and examples, see [deleteMany()](https://docs.mongodb.com/manual/reference/method/db.collection.deleteMany/" \l "mongodb-method-db.collection.deleteMany).

### Delete All Documents that Match a Condition

You can specify criteria, or filters, that identify the documents to delete. The filters use the same syntax as read operations.

To specify equality conditions, use <field>:<value> expressions in the query filter document:

|  |
| --- |
| { <field1>: <value1>, ... } |

A query filter document can use the query operators to specify conditions in the following form:

|  |
| --- |
| { <field1>: { <operator1>: <value1> }, ... } |

To delete all documents that match a deletion criteria, pass a filter parameter to the [deleteMany()](https://docs.mongodb.com/manual/reference/method/db.collection.deleteMany/" \l "mongodb-method-db.collection.deleteMany) method.

The following example removes all documents from the inventory collection where the status field equals "A":

|  |
| --- |
| db.inventory.deleteMany({ status : **"A"** }) |

The method returns a document with the status of the operation. For more information and examples, see [deleteMany()](https://docs.mongodb.com/manual/reference/method/db.collection.deleteMany/" \l "mongodb-method-db.collection.deleteMany).

### Delete Only One Document that Matches a Condition

To delete at most a single document that matches a specified filter (even though multiple documents may match the specified filter) use the [db.collection.deleteOne()](https://docs.mongodb.com/manual/reference/method/db.collection.deleteOne/" \l "mongodb-method-db.collection.deleteOne) method.

The following example deletes the first document where status is "D":

|  |
| --- |
| db.inventory.deleteOne( { status: **"D"** } ) |

### Delete Behavior

#### Indexes

Delete operations do not drop indexes, even if deleting all documents from a collection.

#### Atomicity

All write operations in MongoDB are atomic on the level of a single document. For more information on MongoDB and atomicity, see [Atomicity and Transactions](https://docs.mongodb.com/manual/core/write-operations-atomicity/).

### Remove All Documents from a Collection

db.users.remove( { } )

**Note**: To remove all documents from a collection, it may be more efficient to use the [drop()](https://docs.mongodb.com/manual/reference/method/db.collection.drop/#mongodb-method-db.collection.drop) method to drop the entire collection, including the indexes, and then recreate the collection and rebuild the indexes.

### Remove All Documents that Match a Condition

To remove the documents that match a deletion criteria, call the [remove()](https://docs.mongodb.com/manual/reference/method/db.collection.remove/#mongodb-method-db.collection.remove) method with the <query> parameter:

The following operation removes all the documents from the collection products where qty is greater than 20:

|  |
| --- |
| db.products.remove( { qty: { $gt: 20 } } ) |

### Remove a Single Document that Matches a Condition

To remove the first document that match a deletion criteria, call the [remove](https://docs.mongodb.com/manual/reference/method/db.collection.remove/#mongodb-method-db.collection.remove) method with the query criteria and the justOne parameter set to true or 1.

The following operation removes the first document from the collection products where qty is greater than 20:

|  |
| --- |
| db.products.remove( { qty: { $gt: 20 } }, true ) |

## CRUD Operation Examples

**Insert one**:

db.girlDiapers.insertOne(

{

size: 3,

color: 'peach',

brand: 'girl large'

}

)

**Insert many**:

db.girlDiapers.insertMany([

{

size: 2,

color: 'pink',

brand: 'girl medium'

},

{

size: 3,

color: 'pink',

brand: 'girl large'

}

])

**Using try-catch**:

try {

db.boyDiapers.insertMany([

{

size: 2,

color: 'blue',

brand: 'boys medium'

},

{

size: 3,

color: 'green',

brand: 'boys large'

},

{

size: 1,

color: 'black',

brand: 'boys small'

}

]);

} catch(e) {

print(e);

}

**Update One**:

db.girlDiapers.updateOne(

{"size":2, "color":"while"},

{$set: {"color":"white", "brand":"girl medium"}

})

**Upsert**:

db.girlDiapers.updateOne(

{"size":5},

{$set: {"size":5, "color":"white", "brand":"girl xx-large"}},

{upsert: true}

)

## Sorting

**Sample data**:

{\_id: ObjectId("5effaa5662679b5af2c58829"),

email: “email@example.com”,

name: {**given**: “Jesse”, family: “Xiao”},

age: 31,

addresses: [{label: “home”,

street: “101 Elm Street”,

city: “Springfield”,

**state**: “CA”,

zip: “90000”,

country: “US”},

{label: “mom”,

street: “555 Main Street”,

city: “Jonestown”,

province: “Ontario”,

country: “CA”}]

}

## Sort Results by a Field

// order by age, in ascending order (smallest **values** first)

> db.user.find().sort({age: 1})

{

"\_id": ObjectId("5ce45d7606444f199acfba1e"),

"name": {**given**: "Alex", family: "Smith"},

"email": "email@example.com",

"age": 27

}

{

\_id: ObjectId("5effaa5662679b5af2c58829"),

email: “email@example.com”,

name: {**given**: “Jesse”, family: “Xiao”},

age: 31

}

>

// order by age, in descending order (largest **values** first)

> db.user.find().sort({age: -1})

{

\_id: ObjectId("5effaa5662679b5af2c58829"),

email: “email@example.com”,

name: {**given**: “Jesse”, family: “Xiao”},

age: 31

}

{

"\_id": ObjectId("5ce45d7606444f199acfba1e"),

"name": {**given**: "Alex", family: "Smith"},

"email": "email@example.com",

"age": 27

}

>

Examples:

db.zipcodes.find().sort({"city": 1})

db.zipcodes.aggregate( {"$sort" : {"city": -1} } )

## regex

<https://docs.mongodb.com/manual/reference/operator/query/regex/>

Provides regular expression capabilities for pattern matching *strings* in queries.

To use [$regex](https://docs.mongodb.com/manual/reference/operator/query/regex/#mongodb-query-op.-regex), use one of the following syntaxes:

|  |
| --- |
| { <field>: { $regex: /pattern/, $options: **'<options>'** } } |
| { <field>: { $regex: **'pattern'**, $options: **'<options>'** } } |
| { <field>: { $regex: /pattern/<options> } } |

### $options

The following <options> are available for use with regular expression.

| **Option** | **Description** | **Syntax Restrictions** |
| --- | --- | --- |
| i | Case insensitivity to match upper and lower cases. For an example, see [Perform Case-Insensitive Regular Expression Match](https://docs.mongodb.com/manual/reference/operator/query/regex/#std-label-regex-case-insensitive). |  |
| m | For patterns that include anchors (i.e. ^ for the start, $ for the end), match at the beginning or end of each line for strings with multiline values. Without this option, these anchors match at beginning or end of the string. For an example, see [Multiline Match for Lines Starting with Specified Pattern](https://docs.mongodb.com/manual/reference/operator/query/regex/#std-label-regex-multiline-match).  If the pattern contains no anchors or if the string value has no newline characters (e.g. \n), the m option has no effect. |  |
| x | "Extended" capability to ignore all white space characters in the [$regex](https://docs.mongodb.com/manual/reference/operator/query/regex/#mongodb-query-op.-regex) pattern unless escaped or included in a character class.  Additionally, it ignores characters in-between and including an un-escaped hash/pound (#) character and the next new line, so that you may include comments in complicated patterns. This only applies to data characters; white space characters may never appear within special character sequences in a pattern.  The x option does not affect the handling of the VT character (i.e. code 11). | Requires $regex with $options syntax |
| s | Allows the dot character (i.e. .) to match all characters including newline characters. For an example, see [Use the . Dot Character to Match New Line](https://docs.mongodb.com/manual/reference/operator/query/regex/#std-label-regex-dot-new-line). | Requires $regex with $options syntax |

**NOTE**

The [$regex](https://docs.mongodb.com/manual/reference/operator/query/regex/#mongodb-query-op.-regex) operator does not support the global search modifier g.

### $in Expressions

To include a regular expression in an $in query expression, you can only use JavaScript regular expression objects (i.e. /pattern/ ). For example:

|  |
| --- |
| { name: { $in: [ /^acme/i, /^ack/ ] } } |

You cannot use [$regex](https://docs.mongodb.com/manual/reference/operator/query/regex/#mongodb-query-op.-regex) operator expressions inside an [$in](https://docs.mongodb.com/manual/reference/operator/query/in/#mongodb-query-op.-in).

### Implicit AND Conditions for the Field

To include a regular expression in a comma-separated list of query conditions for the field, use the [$regex](https://docs.mongodb.com/manual/reference/operator/query/regex/#mongodb-query-op.-regex) operator. For example:

|  |
| --- |
| { name: { $regex: /acme.\*corp/i, $nin: [ **'acmeblahcorp'** ] } } |
| { name: { $regex: /acme.\*corp/, $options: **'i'**, $nin: [ **'acmeblahcorp'** ] } } |
| { name: { $regex: **'acme.\*corp'**, $options: **'i'**, $nin: [ **'acmeblahcorp'** ] } } |

### x and s Options

To use either the x option or s options, you must use the [$regex](https://docs.mongodb.com/manual/reference/operator/query/regex/#mongodb-query-op.-regex) operator expression with the [$options](https://docs.mongodb.com/manual/reference/operator/query/regex/#mongodb-query-op.-options) operator. For example, to specify the i and the s options, you must use [$options](https://docs.mongodb.com/manual/reference/operator/query/regex/#mongodb-query-op.-options) for both:

|  |
| --- |
| { name: { $regex: /acme.\*corp/, $options: **"si"** } } |
| { name: { $regex: **'acme.\*corp'**, $options: **"si"** } } |

### $regex and $not

Starting in 4.0.7, [$not](https://docs.mongodb.com/manual/reference/operator/query/not/#mongodb-query-op.-not) operator can perform logical NOT operation on both:

* regular expression objects (i.e. /pattern/)

For example:

|  |
| --- |
| db.inventory.find( { item: { $not: /^p.\*/ } } ) |

* [$regex](https://docs.mongodb.com/manual/reference/operator/query/regex/#mongodb-query-op.-regex) operator expressions (starting in MongoDB 4.0.7).

For example:

|  |
| --- |
| db.inventory.find( { item: { $not: { $regex: **"^p.\*"** } } } ) |
| db.inventory.find( { item: { $not: { $regex: /^p.\*/ } } } ) |

In 4.0.6 and earlier, you could use [$not](https://docs.mongodb.com/manual/reference/operator/query/not/#mongodb-query-op.-not) operator with regular expression objects (i.e. /pattern/) but not with [$regex](https://docs.mongodb.com/manual/reference/operator/query/regex/#mongodb-query-op.-regex) operator expressions.

### Examples

The following examples use a collection products with the following documents:

|  |
| --- |
| { **"\_id"** : 100, **"sku"** : **"abc123"**, **"description"** : **"Single line description."** } |
| { **"\_id"** : 101, **"sku"** : **"abc789"**, **"description"** : **"First line\nSecond line"** } |
| { **"\_id"** : 102, **"sku"** : **"xyz456"**, **"description"** : **"Many spaces before line"** } |
| { **"\_id"** : 103, **"sku"** : **"xyz789"**, **"description"** : **"Multiple\nline description"** } |

#### Perform a LIKE Match

The following example matches all documents where the sku field is like "%789":

|  |
| --- |
| db.products.find( { sku: { $regex: /789$/ } } ) |

The example is analogous to the following SQL LIKE statement:

|  |
| --- |
| SELECT \* FROM products |
| WHERE sku like "%789"; |

#### Perform Case-Insensitive Regular Expression Match

The following example uses the i option perform a case-insensitive match for documents with sku value that starts with ABC.

|  |
| --- |
| db.products.find( { sku: { $regex: /^ABC/i } } ) |

The query matches the following documents:

|  |
| --- |
| { **"\_id"** : 100, **"sku"** : **"abc123"**, **"description"** : **"Single line description."** } |
| { **"\_id"** : 101, **"sku"** : **"abc789"**, **"description"** : **"First line\nSecond line"** } |

#### Multiline Match for Lines Starting with Specified Pattern

The following example uses the m option to match lines starting with the letter S for multiline strings:

|  |
| --- |
| db.products.find( { description: { $regex: /^S/, $options: **'m'** } } ) |

The query matches the following documents:

|  |
| --- |
| { **"\_id"** : 100, **"sku"** : **"abc123"**, **"description"** : **"Single line description."** } |
| { **"\_id"** : 101, **"sku"** : **"abc789"**, **"description"** : **"First line\nSecond line"** } |

Without the m option, the query would match just the following document:

|  |
| --- |
| { **"\_id"** : 100, **"sku"** : **"abc123"**, **"description"** : **"Single line description."** } |

If the [$regex](https://docs.mongodb.com/manual/reference/operator/query/regex/#mongodb-query-op.-regex) pattern does not contain an anchor, the pattern matches against the string as a whole, as in the following example:

|  |
| --- |
| db.products.find( { description: { $regex: /S/ } } ) |

Then, the [$regex](https://docs.mongodb.com/manual/reference/operator/query/regex/#mongodb-query-op.-regex) would match both documents:

|  |
| --- |
| { **"\_id"** : 100, **"sku"** : **"abc123"**, **"description"** : **"Single line description."** } |
| { **"\_id"** : 101, **"sku"** : **"abc789"**, **"description"** : **"First line\nSecond line"** } |

#### Use the . Dot Character to Match New Line

The following example uses the s option to allow the dot character (i.e. .) to match all characters including new line as well as the i option to perform a case-insensitive match:

|  |
| --- |
| db.products.find( { description: { $regex: /m.\*line/, $options: **'si'** } } ) |

The query matches the following documents:

|  |
| --- |
| { **"\_id"** : 102, **"sku"** : **"xyz456"**, **"description"** : **"Many spaces before line"** } |
| { **"\_id"** : 103, **"sku"** : **"xyz789"**, **"description"** : **"Multiple\nline description"** } |

Without the s option, the query would have matched only the following document:

|  |
| --- |
| { **"\_id"** : 102, **"sku"** : **"xyz456"**, **"description"** : **"Many spaces before line"** } |

#### Ignore White Spaces in Pattern

The following example uses the x option ignore white spaces and the comments, denoted by the # and ending with the \n in the matching pattern:

|  |
| --- |
| var pattern = **"abc #category code\n123 #item number"** |
| db.products.find( { sku: { $regex: pattern, $options: **"x"** } } ) |

The query matches the following document:

|  |
| --- |
| { **"\_id"** : 100, **"sku"** : **"abc123"**, **"description"** : **"Single line description."** } |

# Aggregation

Aggregation operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. MongoDB provides three ways to perform aggregation: the [aggregation pipeline](https://docs.mongodb.com/manual/aggregation/#std-label-aggregation-framework), the [map-reduce function](https://docs.mongodb.com/manual/aggregation/#std-label-aggregation-map-reduce), and [single purpose aggregation methods](https://docs.mongodb.com/manual/aggregation/#std-label-single-purpose-agg-operations).

### Aggregation Pipeline

MongoDB's aggregation framework is modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms the documents into an aggregated result. For example:

|  |
| --- |
| db.orders.aggregate([ |
| { $match: { status: **"A"** } }, |
| { $group: { \_id: **"$cust\_id"**, total: { $sum: **"$amount"** } } } |
| ]) |

**First Stage**: The [$match](https://docs.mongodb.com/manual/reference/operator/aggregation/match/#mongodb-pipeline-pipe.-match) stage filters the documents by the status field and passes to the next stage those documents that have status equal to "A".

**Second Stage**: The [$group](https://docs.mongodb.com/manual/reference/operator/aggregation/group/#mongodb-pipeline-pipe.-group) stage groups the documents by the cust\_id field to calculate the sum of the amount for each unique cust\_id.

The most basic pipeline stages provide filters that operate like queries and document transformations that modify the form of the output document.

Other pipeline operations provide tools for grouping and sorting documents by specific field or fields as well as tools for aggregating the contents of arrays, including arrays of documents. In addition, pipeline stages can use operators for tasks such as calculating the average or concatenating a string.

The pipeline provides efficient data aggregation using native operations within MongoDB, and is the preferred method for data aggregation in MongoDB.

The aggregation pipeline can operate on a sharded collection.

The aggregation pipeline can use indexes to improve its performance during some of its stages. In addition, the aggregation pipeline has an internal optimization phase. See [Pipeline Operators and Indexes](https://docs.mongodb.com/manual/core/aggregation-pipeline/#std-label-aggregation-pipeline-operators-and-performance) and [Aggregation Pipeline Optimization](https://docs.mongodb.com/manual/core/aggregation-pipeline-optimization/) for details.

### Single Purpose Aggregation Operations

MongoDB also provides [db.collection.estimatedDocumentCount()](https://docs.mongodb.com/manual/reference/method/db.collection.estimatedDocumentCount/#mongodb-method-db.collection.estimatedDocumentCount), [db.collection.count()](https://docs.mongodb.com/manual/reference/method/db.collection.count/#mongodb-method-db.collection.count) and [db.collection.distinct()](https://docs.mongodb.com/manual/reference/method/db.collection.distinct/#mongodb-method-db.collection.distinct).

All of these operations aggregate documents from a single collection. While these operations provide simple access to common aggregation processes, they lack the flexibility and capabilities of an aggregation pipeline.

### Map-Reduce[**¶**](https://docs.mongodb.com/manual/aggregation/#map-reduce)

As of MongoDB 5.0 the map-reduce operation is deprecated.

An aggregation pipeline provides better performance and usability than a map-reduce operation.

Map-reduce operations can be rewritten using aggregation pipeline operators, such as [$group](https://docs.mongodb.com/manual/reference/operator/aggregation/group/#mongodb-pipeline-pipe.-group), [$merge](https://docs.mongodb.com/manual/reference/operator/aggregation/merge/#mongodb-pipeline-pipe.-merge), and others.

For map-reduce operations that require custom functionality, MongoDB provides the [$accumulator](https://docs.mongodb.com/manual/reference/operator/aggregation/accumulator/#mongodb-group-grp.-accumulator) and [$function](https://docs.mongodb.com/manual/reference/operator/aggregation/function/#mongodb-expression-exp.-function) aggregation operators starting in version 4.4. Use these operators to define custom aggregation expressions in JavaScript.

## Filtered top Subset

### [Scenario](https://www.practical-mongodb-aggregations.com/examples/foundational/filtered-top-subset.html#scenario)

You want to query a collection of people to find the three youngest people who have a job in engineering, sorted by the youngest person first.

This example is the only one in the book that you can also achieve entirely using MQL and serves as a helpful comparison between MQL and Aggregation Pipelines.

### [Sample Data Population](https://www.practical-mongodb-aggregations.com/examples/foundational/filtered-top-subset.html#sample-data-population)

Drop any old version of the database (if it exists) and then populate a new persons collection with 5 person documents:

use book-filtered-top-subset;

db.dropDatabase();

// Create an index for a persons collection

db.persons.createIndex({"vocation": 1, "dateofbirth": 1});

// Insert records into the persons collection

db.persons.insertMany([

{

"person\_id": "6392529400",

"firstname": "Elise",

"lastname": "Smith",

"dateofbirth": ISODate("1972-01-13T09:32:07Z"),

"vocation": "ENGINEER",

"address": {

"number": 5625,

"street": "Tipa Circle",

"city": "Wojzinmoj",

},

},

{

"person\_id": "1723338115",

"firstname": "Olive",

"lastname": "Ranieri",

"dateofbirth": ISODate("1985-05-12T23:14:30Z"),

"gender": "FEMALE",

"vocation": "ENGINEER",

"address": {

"number": 9303,

"street": "Mele Circle",

"city": "Tobihbo",

},

},

{

"person\_id": "8732762874",

"firstname": "Toni",

"lastname": "Jones",

"dateofbirth": ISODate("1991-11-23T16:53:56Z"),

"vocation": "POLITICIAN",

"address": {

"number": 1,

"street": "High Street",

"city": "Upper Abbeywoodington",

},

},

{

"person\_id": "7363629563",

"firstname": "Bert",

"lastname": "Gooding",

"dateofbirth": ISODate("1941-04-07T22:11:52Z"),

"vocation": "FLORIST",

"address": {

"number": 13,

"street": "Upper Bold Road",

"city": "Redringtonville",

},

},

{

"person\_id": "1029648329",

"firstname": "Sophie",

"lastname": "Celements",

"dateofbirth": ISODate("1959-07-06T17:35:45Z"),

"vocation": "ENGINEER",

"address": {

"number": 5,

"street": "Innings Close",

"city": "Basilbridge",

},

},

{

"person\_id": "7363626383",

"firstname": "Carl",

"lastname": "Simmons",

"dateofbirth": ISODate("1998-12-26T13:13:55Z"),

"vocation": "ENGINEER",

"address": {

"number": 187,

"street": "Hillside Road",

"city": "Kenningford",

},

},

]);

### [Aggregation Pipeline](https://www.practical-mongodb-aggregations.com/examples/foundational/filtered-top-subset.html#aggregation-pipeline)

Define a single pipeline ready to perform the aggregation:

var pipeline = [

// Match engineers only

{"$match": {

"vocation": "ENGINEER",

}},

// Sort by youngest person first

{"$sort": {

"dateofbirth": -1,

}},

// Only include the first 3 youngest people

{"$limit": 3},

// Exclude unrequired fields from each person record

{"$unset": [

"\_id",

"vocation",

"address",

]},

];

### [Execution](https://www.practical-mongodb-aggregations.com/examples/foundational/filtered-top-subset.html#execution)

Execute the aggregation using the defined pipeline and also view its explain plan:

db.persons.aggregate(pipeline);

db.persons.explain("executionStats").aggregate(pipeline);

### [Expected Results](https://www.practical-mongodb-aggregations.com/examples/foundational/filtered-top-subset.html#expected-results)

Three documents should be returned, representing the three youngest people who are engineers (ordered by youngest first), omitting the \_id or address attributes of each person, as shown below:

[

{

person\_id: '7363626383',

firstname: 'Carl',

lastname: 'Simmons',

dateofbirth: ISODate('1998-12-26T13:13:55.000Z')

},

{

person\_id: '1723338115',

firstname: 'Olive',

lastname: 'Ranieri',

dateofbirth: ISODate('1985-05-12T23:14:30.000Z'),

gender: 'FEMALE'

},

{

person\_id: '6392529400',

firstname: 'Elise',

lastname: 'Smith',

dateofbirth: ISODate('1972-01-13T09:32:07.000Z')

}

]

### [Observations](https://www.practical-mongodb-aggregations.com/examples/foundational/filtered-top-subset.html#observations)

* **Index Use.** A basic aggregation pipeline, where if many records belong to the collection, a compound index for vocation + dateofbirth should exist to enable the database to fully optimise the execution of the pipeline combining the filter of the $match stage with the sort from the sort stage and the limit of the limit stage.
* **Unset Use.** An $unset stage is used rather than a $project stage. This enables the pipeline to avoid being verbose. More importantly, it means the pipeline does not have to be modified if a new field appears in documents added in the future (for example, see the gender field that appears in only Olive's record).
* **MQL Similarity.** For reference, the MQL equivalent for you to achieve the same result is shown below (you can try this in the Shell):

db.persons.find(

{"vocation": "ENGINEER"},

{"\_id": 0, "vocation": 0, "address": 0},

).sort(

{"dateofbirth": -1}

).limit(3);

## Group & Total

### [Scenario](https://www.practical-mongodb-aggregations.com/examples/foundational/group-and-total.html#scenario)

You want to generate a report to show what each shop customer purchased in 2020. You will group the individual order records by customer, capturing each customer's first purchase date, the number of orders they made, the total value of all their orders and a list of their order items sorted by date.

### [Sample Data Population](https://www.practical-mongodb-aggregations.com/examples/foundational/group-and-total.html#sample-data-population)

Drop any old version of the database (if it exists) and then populate a new orders collection with 9 order documents spanning 2019-2021, for 3 different unique customers:

use book-group-and-total;

db.dropDatabase();

// Create index for an orders collection

db.orders.createIndex({"orderdate": -1});

// Insert records into the orders collection

db.orders.insertMany([

{

"customer\_id": "elise\_smith@myemail.com",

"orderdate": ISODate("2020-05-30T08:35:52Z"),

"value": NumberDecimal("231.43"),

},

{

"customer\_id": "elise\_smith@myemail.com",

"orderdate": ISODate("2020-01-13T09:32:07Z"),

"value": NumberDecimal("99.99"),

},

{

"customer\_id": "oranieri@warmmail.com",

"orderdate": ISODate("2020-01-01T08:25:37Z"),

"value": NumberDecimal("63.13"),

},

{

"customer\_id": "tj@wheresmyemail.com",

"orderdate": ISODate("2019-05-28T19:13:32Z"),

"value": NumberDecimal("2.01"),

},

{

"customer\_id": "tj@wheresmyemail.com",

"orderdate": ISODate("2020-11-23T22:56:53Z"),

"value": NumberDecimal("187.99"),

},

{

"customer\_id": "tj@wheresmyemail.com",

"orderdate": ISODate("2020-08-18T23:04:48Z"),

"value": NumberDecimal("4.59"),

},

{

"customer\_id": "elise\_smith@myemail.com",

"orderdate": ISODate("2020-12-26T08:55:46Z"),

"value": NumberDecimal("48.50"),

},

{

"customer\_id": "tj@wheresmyemail.com",

"orderdate": ISODate("2021-02-29T07:49:32Z"),

"value": NumberDecimal("1024.89"),

},

{

"customer\_id": "elise\_smith@myemail.com",

"orderdate": ISODate("2020-10-03T13:49:44Z"),

"value": NumberDecimal("102.24"),

},

]);

### [Aggregation Pipeline](https://www.practical-mongodb-aggregations.com/examples/foundational/group-and-total.html#aggregation-pipeline)

Define a single pipeline ready to perform the aggregation:

var pipeline = [

// Match only orders made in 2020

{"$match": {

"orderdate": {

"$gte": ISODate("2020-01-01T00:00:00Z"),

"$lt": ISODate("2021-01-01T00:00:00Z"),

},

}},

// Sort by order date ascending (required to pick out 'first\_purchase\_date' below)

{"$sort": {

"orderdate": 1,

}},

// Group by customer

{"$group": {

"\_id": "$customer\_id",

"first\_purchase\_date": {"$first": "$orderdate"},

"total\_value": {"$sum": "$value"},

"total\_orders": {"$sum": 1},

"orders": {"$push": {"orderdate": "$orderdate", "value": "$value"}},

}},

// Sort by each customer's first purchase date

{"$sort": {

"first\_purchase\_date": 1,

}},

// Set customer's ID to be value of the field that was grouped on

{"$set": {

"customer\_id": "$\_id",

}},

// Omit unwanted fields

{"$unset": [

"\_id",

]},

];

### [Execution](https://www.practical-mongodb-aggregations.com/examples/foundational/group-and-total.html#execution)

Execute the aggregation using the defined pipeline and also view its explain plan:

db.orders.aggregate(pipeline);

db.orders.explain("executionStats").aggregate(pipeline);

### [Expected Results](https://www.practical-mongodb-aggregations.com/examples/foundational/group-and-total.html#expected-results)

Three documents should be returned, representing the three customers, each showing the customer's first purchase date, the total value of all their orders, the number of orders they made and a list of each order's detail, for 2020 only, as shown below:

[

{

customer\_id: 'oranieri@warmmail.com',

first\_purchase\_date: ISODate('2020-01-01T08:25:37.000Z'),

total\_value: NumberDecimal('63.13'),

total\_orders: 1,

orders: [

{orderdate: ISODate('2020-01-01T08:25:37.000Z'), value: NumberDecimal('63.13')}

]

},

{

customer\_id: 'elise\_smith@myemail.com',

first\_purchase\_date: ISODate('2020-01-13T09:32:07.000Z'),

total\_value: NumberDecimal('482.16'),

total\_orders: 4,

orders: [

{orderdate: ISODate('2020-01-13T09:32:07.000Z'), value: NumberDecimal('99.99')},

{orderdate: ISODate('2020-05-30T08:35:52.000Z'), value: NumberDecimal('231.43')},

{orderdate: ISODate('2020-10-03T13:49:44.000Z'), value: NumberDecimal('102.24')},

{orderdate: ISODate('2020-12-26T08:55:46.000Z'), value: NumberDecimal('48.50')}

]

},

{

customer\_id: 'tj@wheresmyemail.com',

first\_purchase\_date: ISODate('2020-08-18T23:04:48.000Z'),

total\_value: NumberDecimal('192.58'),

total\_orders: 2,

orders: [

{orderdate: ISODate('2020-08-18T23:04:48.000Z'), value: NumberDecimal('4.59')},

{orderdate: ISODate('2020-11-23T22:56:53.000Z'), value: NumberDecimal('187.99')}

]

}

]

Note, the order of fields shown for each document may vary.

### [Observations](https://www.practical-mongodb-aggregations.com/examples/foundational/group-and-total.html#observations)

* **Double Sort Use.** It is necessary to perform a $sort on the order date both before and after the $group stage. The $sort before the $group is required because the $group stage uses a $first group accumulator to capture just the first order's orderdate value for each grouped customer. The $sort after the $group is required because the act of having just grouped on customer ID will mean that the records are no longer sorted by purchase date for the records coming out of the $group stage.
* **Renaming Group.** Towards the end of the pipeline, you will see what is a typical pattern for pipelines that use $group, consisting of a combination of $set+$unset stages, to essentially take the group's key (which is always called \_id) and substitute it with a more meaningful name (customer\_id).
* **Lossless Decimals.** You may notice the pipeline uses a NumberDecimal() function to ensure the order amounts in the inserted records are using a lossless decimal type, [IEEE 754 decimal128](https://docs.mongodb.com/manual/tutorial/model-monetary-data/). In this example, if you use a JSON float or double type instead, the order totals will suffer from a loss of precision. For instance, for the customer elise\_smith@myemail.com, if you use a double type, the total\_value result will have the value shown in the second line below, rather than the first line:

// Desired result achieved by using decimal128 types

total\_value: NumberDecimal('482.16')

// Result that occurs if using float or double types instead

total\_value: 482.15999999999997

## [Unpack Arrays & Group Differently](https://www.practical-mongodb-aggregations.com/examples/foundational/unpack-array-group-differently.html#unpack-arrays--group-differently)

### [Scenario](https://www.practical-mongodb-aggregations.com/examples/foundational/unpack-array-group-differently.html#scenario)

You want to generate a retail report to list the total value and quantity of expensive products sold (valued over 15 dollars). The source data is a list of shop orders, where each order contains the set of products purchased as part of the order.

### [Sample Data Population](https://www.practical-mongodb-aggregations.com/examples/foundational/unpack-array-group-differently.html#sample-data-population)

Drop any old version of the database (if it exists) and then populate a new orders collection where each document contains an array of products purchased:

use book-unpack-array-group-differently;

db.dropDatabase();

// Insert 4 records into the orders collection each with 1+ product items

db.orders.insertMany([

{

"order\_id": 6363763262239,

"products": [

{

"prod\_id": "abc12345",

"name": "Asus Laptop",

"price": NumberDecimal("431.43"),

},

{

"prod\_id": "def45678",

"name": "Karcher Hose Set",

"price": NumberDecimal("22.13"),

},

],

},

{

"order\_id": 1197372932325,

"products": [

{

"prod\_id": "abc12345",

"name": "Asus Laptop",

"price": NumberDecimal("429.99"),

},

],

},

{

"order\_id": 9812343774839,

"products": [

{

"prod\_id": "pqr88223",

"name": "Morphy Richardds Food Mixer",

"price": NumberDecimal("431.43"),

},

{

"prod\_id": "def45678",

"name": "Karcher Hose Set",

"price": NumberDecimal("21.78"),

},

],

},

{

"order\_id": 4433997244387,

"products": [

{

"prod\_id": "def45678",

"name": "Karcher Hose Set",

"price": NumberDecimal("23.43"),

},

{

"prod\_id": "jkl77336",

"name": "Picky Pencil Sharpener",

"price": NumberDecimal("0.67"),

},

{

"prod\_id": "xyz11228",

"name": "Russell Hobbs Chrome Kettle",

"price": NumberDecimal("15.76"),

},

],

},

]);

## Using $unwind (aggregation)

Deconstructs an array field from the input documents to output a document for each element. Each output document is the input document with the value of the array field replaced by the element.

If you specify a path for a field that does not exist in an input document or the field is an empty array, [$unwind](https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/#mongodb-pipeline-pipe.-unwind), by default, ignores the input document and will not output documents for that input document.

### Unwind Array

In [mongosh](https://docs.mongodb.com/mongodb-shell/" \l "mongodb-binary-bin.mongosh), create a sample collection named inventory with the following document:

|  |
| --- |
| db.inventory.insertOne({ **"\_id"** : 1, **"item"** : **"ABC1"**, sizes: [ **"S"**, **"M"**, **"L"**] }) |

The following aggregation uses the [$unwind](https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/#mongodb-pipeline-pipe.-unwind) stage to output a document for each element in the sizes array:

|  |
| --- |
| db.inventory.aggregate( [ { $unwind : **"$sizes"** } ] ) |

The operation returns the following results:

|  |
| --- |
| { **"\_id"** : 1, **"item"** : **"ABC1"**, **"sizes"** : **"S"** } |
| { **"\_id"** : 1, **"item"** : **"ABC1"**, **"sizes"** : **"M"** } |
| { **"\_id"** : 1, **"item"** : **"ABC1"**, **"sizes"** : **"L"** } |

Each document is identical to the input document except for the value of the sizes field which now holds a value from the original sizes array.

### includeArrayIndex and preserveNullAndEmptyArrays

New in version 3.2.

In [mongosh](https://docs.mongodb.com/mongodb-shell/" \l "mongodb-binary-bin.mongosh), create a sample collection named inventory2 with the following documents:

|  |
| --- |
| db.inventory2.insertMany([ |
| { **"\_id"** : 1, **"item"** : **"ABC"**, price: NumberDecimal(**"80"**), **"sizes"**: [ **"S"**, **"M"**, **"L"**] }, |
| { **"\_id"** : 2, **"item"** : **"EFG"**, price: NumberDecimal(**"120"**), **"sizes"** : [ ] }, |
| { **"\_id"** : 3, **"item"** : **"IJK"**, price: NumberDecimal(**"160"**), **"sizes"**: **"M"** }, |
| { **"\_id"** : 4, **"item"** : **"LMN"** , price: NumberDecimal(**"10"**) }, |
| { **"\_id"** : 5, **"item"** : **"XYZ"**, price: NumberDecimal(**"5.75"**), **"sizes"** : null } |
| ]) |

The following [$unwind](https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/#mongodb-pipeline-pipe.-unwind) operations are equivalent and return a document for each element in the sizes field. If the sizes field does not resolve to an array but is not missing, null, or an empty array, [$unwind](https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/#mongodb-pipeline-pipe.-unwind) treats the non-array operand as a single element array.

|  |
| --- |
| db.inventory2.aggregate( [ { $unwind: **"$sizes"** } ] ) |
| db.inventory2.aggregate( [ { $unwind: { path: **"$sizes"** } } ] ) |

The operation returns the following documents:

|  |
| --- |
| { **"\_id"** : 1, **"item"** : **"ABC"**, **"price"** : NumberDecimal(**"80"**), **"sizes"** : **"S"** } |
| { **"\_id"** : 1, **"item"** : **"ABC"**, **"price"** : NumberDecimal(**"80"**), **"sizes"** : **"M"** } |
| { **"\_id"** : 1, **"item"** : **"ABC"**, **"price"** : NumberDecimal(**"80"**), **"sizes"** : **"L"** } |
| { **"\_id"** : 3, **"item"** : **"IJK"**, **"price"** : NumberDecimal(**"160"**), **"sizes"** : **"M"** } |

#### includeArrayIndex

The following [$unwind](https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/#mongodb-pipeline-pipe.-unwind) operation uses the includeArrayIndex option to include the array index in the output.

|  |
| --- |
| db.inventory2.aggregate( [ |
| { |
| $unwind: |
| { |
| path: **"$sizes"**, |
| includeArrayIndex: **"arrayIndex"** |
| } |
| }]) |

The operation unwinds the sizes array and includes the array index of the array index in the new arrayIndex field. If the sizes field does not resolve to an array but is not missing, null, or an empty array, the arrayIndex field is null.

|  |
| --- |
| { **"\_id"** : 1, **"item"** : **"ABC"**, **"price"** : NumberDecimal(**"80"**), **"sizes"** : **"S"**, **"arrayIndex"** : NumberLong(0) } |
| { **"\_id"** : 1, **"item"** : **"ABC"**, **"price"** : NumberDecimal(**"80"**), **"sizes"** : **"M"**, **"arrayIndex"** : NumberLong(1) } |
| { **"\_id"** : 1, **"item"** : **"ABC"**, **"price"** : NumberDecimal(**"80"**), **"sizes"** : **"L"**, **"arrayIndex"** : NumberLong(2) } |
| { **"\_id"** : 3, **"item"** : **"IJK"**, **"price"** : NumberDecimal(**"160"**), **"sizes"** : **"M"**, **"arrayIndex"** : null } |

#### preserveNullAndEmptyArrays

The following [$unwind](https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/#mongodb-pipeline-pipe.-unwind) operation uses the preserveNullAndEmptyArrays option to include documents whose sizes field is null, missing, or an empty array.

|  |
| --- |
| db.inventory2.aggregate( [ |
| { $unwind: { path: **"$sizes"**, preserveNullAndEmptyArrays: true } } |
| ] ) |

The output includes those documents where the sizes field is null, missing, or an empty array:

|  |
| --- |
| { **"\_id"** : 1, **"item"** : **"ABC"**, **"price"** : NumberDecimal(**"80"**), **"sizes"** : **"S"** } |
| { **"\_id"** : 1, **"item"** : **"ABC"**, **"price"** : NumberDecimal(**"80"**), **"sizes"** : **"M"** } |
| { **"\_id"** : 1, **"item"** : **"ABC"**, **"price"** : NumberDecimal(**"80"**), **"sizes"** : **"L"** } |
| { **"\_id"** : 2, **"item"** : **"EFG"**, **"price"** : NumberDecimal(**"120"**) } |
| { **"\_id"** : 3, **"item"** : **"IJK"**, **"price"** : NumberDecimal(**"160"**), **"sizes"** : **"M"** } |
| { **"\_id"** : 4, **"item"** : **"LMN"**, **"price"** : NumberDecimal(**"10"**) } |
| { **"\_id"** : 5, **"item"** : **"XYZ"**, **"price"** : NumberDecimal(**"5.75"**), **"sizes"** : null } |

### Group by Unwound Values

In [mongosh](https://docs.mongodb.com/mongodb-shell/" \l "mongodb-binary-bin.mongosh), create a sample collection named inventory2 with the following documents:

|  |
| --- |
| db.inventory2.insertMany([ |
| { **"\_id"** : 1, **"item"** : **"ABC"**, price: NumberDecimal(**"80"**), **"sizes"**: [ **"S"**, **"M"**, **"L"**] }, |
| { **"\_id"** : 2, **"item"** : **"EFG"**, price: NumberDecimal(**"120"**), **"sizes"** : [ ] }, |
| { **"\_id"** : 3, **"item"** : **"IJK"**, price: NumberDecimal(**"160"**), **"sizes"**: **"M"** }, |
| { **"\_id"** : 4, **"item"** : **"LMN"** , price: NumberDecimal(**"10"**) }, |
| { **"\_id"** : 5, **"item"** : **"XYZ"**, price: NumberDecimal(**"5.75"**), **"sizes"** : null } |
| ]) |

The following pipeline unwinds the sizes array and groups the resulting documents by the unwound size values:

|  |
| --- |
| db.inventory2.aggregate( [ |
| *// First Stage* |
| { |
| $unwind: { path: **"$sizes"**, preserveNullAndEmptyArrays: true } |
| }, |
| *// Second Stage* |
| { |
| $group: |
| { |
| \_id: **"$sizes"**, |
| averagePrice: { $avg: **"$price"** } |
| } |
| }, |
| *// Third Stage* |
| { |
| $sort: { **"averagePrice"**: -1 } |
| } |
| ] ) |

#### First Stage:

The [$unwind](https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/#mongodb-pipeline-pipe.-unwind) stage outputs a new document for each element in the sizes array. The stage uses the preserveNullAndEmptyArrays option to include in the output those documents where sizes field is missing, null or an empty array. This stage passes the following documents to the next stage:

|  |
| --- |
| { **"\_id"** : 1, **"item"** : **"ABC"**, **"price"** : NumberDecimal(**"80"**), **"sizes"** : **"S"** } |
| { **"\_id"** : 1, **"item"** : **"ABC"**, **"price"** : NumberDecimal(**"80"**), **"sizes"** : **"M"** } |
| { **"\_id"** : 1, **"item"** : **"ABC"**, **"price"** : NumberDecimal(**"80"**), **"sizes"** : **"L"** } |
| { **"\_id"** : 2, **"item"** : **"EFG"**, **"price"** : NumberDecimal(**"120"**) } |
| { **"\_id"** : 3, **"item"** : **"IJK"**, **"price"** : NumberDecimal(**"160"**), **"sizes"** : **"M"** } |
| { **"\_id"** : 4, **"item"** : **"LMN"**, **"price"** : NumberDecimal(**"10"**) } |
| { **"\_id"** : 5, **"item"** : **"XYZ"**, **"price"** : NumberDecimal(**"5.75"**), **"sizes"** : null } |

#### Second Stage:

The [$group](https://docs.mongodb.com/manual/reference/operator/aggregation/group/#mongodb-pipeline-pipe.-group) stage groups the documents by sizes and calculates the average price of each size. This stage passes the following documents to the next stage:

|  |
| --- |
| { **"\_id"** : **"S"**, **"averagePrice"** : NumberDecimal(**"80"**) } |
| { **"\_id"** : **"L"**, **"averagePrice"** : NumberDecimal(**"80"**) } |
| { **"\_id"** : **"M"**, **"averagePrice"** : NumberDecimal(**"120"**) } |
| { **"\_id"** : null, **"averagePrice"** : NumberDecimal(**"45.25"**) } |

#### Third Stage:

The [$sort](https://docs.mongodb.com/manual/reference/operator/aggregation/sort/#mongodb-pipeline-pipe.-sort) stage sorts the documents by averagePrice in descending order. The operation returns the following result:

|  |
| --- |
| { **"\_id"** : **"M"**, **"averagePrice"** : NumberDecimal(**"120"**) } |
| { **"\_id"** : **"L"**, **"averagePrice"** : NumberDecimal(**"80"**) } |
| { **"\_id"** : **"S"**, **"averagePrice"** : NumberDecimal(**"80"**) } |
| { **"\_id"** : null, **"averagePrice"** : NumberDecimal(**"45.25"**) } |

### Unwind Embedded Arrays

In [mongosh](https://docs.mongodb.com/mongodb-shell/" \l "mongodb-binary-bin.mongosh), create a sample collection named sales with the following documents:

|  |
| --- |
| db.sales.insertMany([ |
| { |
| \_id: **"1"**, |
| **"items"** : [ |
| { |
| **"name"** : **"pens"**, |
| **"tags"** : [ **"writing"**, **"office"**, **"school"**, **"stationary"** ], |
| **"price"** : NumberDecimal(**"12.00"**), |
| **"quantity"** : NumberInt(**"5"**) |
| }, |
| { |
| **"name"** : **"envelopes"**, |
| **"tags"** : [ **"stationary"**, **"office"** ], |
| **"price"** : NumberDecimal(**"1.95"**), |
| **"quantity"** : NumberInt(**"8"**) |
| } |
| ] |
| }, |
| { |
| \_id: **"2"**, |
| **"items"** : [ |
| { |
| **"name"** : **"laptop"**, |
| **"tags"** : [ **"office"**, **"electronics"** ], |
| **"price"** : NumberDecimal(**"800.00"**), |
| **"quantity"** : NumberInt(**"1"**) |
| }, |
| { |
| **"name"** : **"notepad"**, |
| **"tags"** : [ **"stationary"**, **"school"** ], |
| **"price"** : NumberDecimal(**"14.95"**), |
| **"quantity"** : NumberInt(**"3"**) |
| } |
| ] |
| } |
| ]) |

The following operation groups the items sold by their tags and calculates the total sales amount per each tag.

|  |
| --- |
| db.sales.aggregate([ |
| *// First Stage* |
| { $unwind: **"$items"** }, |
|  |
| *// Second Stage* |
| { $unwind: **"$items.tags"** }, |
|  |
| *// Third Stage* |
| { |
| $group: |
| { |
| \_id: **"$items.tags"**, |
| totalSalesAmount: |
| { |
| $sum: { $multiply: [ **"$items.price"**, **"$items.quantity"** ] } |
| } |
| } |
| } |
| ]) |

#### First Stage

The first [$unwind](https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/#mongodb-pipeline-pipe.-unwind) stage outputs a new document for each element in the items array:

{ "\_id" : "1", "items" : { "name" : "pens", "tags" : [ "writing", "office", "school", "stationary" ], "price" : NumberDecimal("12.00"), "quantity" : 5 } }

{ "\_id" : "1", "items" : { "name" : "envelopes", "tags" : [ "stationary", "office" ], "price" : NumberDecimal("19.95"), "quantity" : 8 } }

{ "\_id" : "2", "items" : { "name" : "laptop", "tags" : [ "office", "electronics" ], "price" : NumberDecimal("800.00"), "quantity" : 1 } }

{ "\_id" : "2", "items" : { "name" : "notepad", "tags" : [ "stationary", "school" ], "price" : NumberDecimal("14.95"), "quantity" : 3 } }

#### Second Stage

The second [$unwind](https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/#mongodb-pipeline-pipe.-unwind) stage outputs a new document for each element in the items.tags arrays:

{ "\_id" : "1", "items" : { "name" : "pens", "tags" : "writing", "price" : NumberDecimal("12.00"), "quantity" : 5 } }

{ "\_id" : "1", "items" : { "name" : "pens", "tags" : "office", "price" : NumberDecimal("12.00"), "quantity" : 5 } }

{ "\_id" : "1", "items" : { "name" : "pens", "tags" : "school", "price" : NumberDecimal("12.00"), "quantity" : 5 } }

{ "\_id" : "1", "items" : { "name" : "pens", "tags" : "stationary", "price" : NumberDecimal("12.00"), "quantity" : 5 } }

{ "\_id" : "1", "items" : { "name" : "envelopes", "tags" : "stationary", "price" : NumberDecimal("19.95"), "quantity" : 8 } }

{ "\_id" : "1", "items" : { "name" : "envelopes", "tags" : "office", "price" : NumberDecimal("19.95"), "quantity" : 8 } }

{ "\_id" : "2", "items" : { "name" : "laptop", "tags" : "office", "price" : NumberDecimal("800.00"), "quantity" : 1 } }

{ "\_id" : "2", "items" : { "name" : "laptop", "tags" : "electronics", "price" : NumberDecimal("800.00"), "quantity" : 1 } }

{ "\_id" : "2", "items" : { "name" : "notepad", "tags" : "stationary", "price" : NumberDecimal("14.95"), "quantity" : 3 } }

{ "\_id" : "2", "items" : { "name" : "notepad", "tags" : "school", "price" : NumberDecimal("14.95"), "quantity" : 3 } }

#### Third Stage

The [$group](https://docs.mongodb.com/manual/reference/operator/aggregation/group/#mongodb-pipeline-pipe.-group) stage groups the documents by the tag and calculates the total sales amount of items with each tag:

|  |
| --- |
| { **"\_id"** : **"writing"**, **"totalSalesAmount"** : NumberDecimal(**"60.00"**) } |
| { **"\_id"** : **"stationary"**, **"totalSalesAmount"** : NumberDecimal(**"264.45"**) } |
| { **"\_id"** : **"electronics"**, **"totalSalesAmount"** : NumberDecimal(**"800.00"**) } |
| { **"\_id"** : **"school"**, **"totalSalesAmount"** : NumberDecimal(**"104.85"**) } |
| { **"\_id"** : **"office"**, **"totalSalesAmount"** : NumberDecimal(**"1019.60"**) } |

### [Aggregation Pipeline](https://www.practical-mongodb-aggregations.com/examples/foundational/unpack-array-group-differently.html#aggregation-pipeline)

Define a single pipeline ready to perform the aggregation:

var pipeline = [

// Unpack each product from the each order's product as a new separate record

{"$unwind": {

"path": "$products",

}},

// Match only products valued greater than 15.00

{"$match": {

"products.price": {

"$gt": NumberDecimal("15.00"),

},

}},

// Group by product type, capturing each product's total value + quantity

{"$group": {

"\_id": "$products.prod\_id",

"product": {"$first": "$products.name"},

"total\_value": {"$sum": "$products.price"},

"quantity": {"$sum": 1},

}},

// Set product id to be the value of the field that was grouped on

{"$set": {

"product\_id": "$\_id",

}},

// Omit unwanted fields

{"$unset": [

"\_id",

]},

];

### [Execution](https://www.practical-mongodb-aggregations.com/examples/foundational/unpack-array-group-differently.html#execution)

Execute the aggregation using the defined pipeline and also view its explain plan:

db.orders.aggregate(pipeline);

db.orders.explain("executionStats").aggregate(pipeline);

### [Expected Results](https://www.practical-mongodb-aggregations.com/examples/foundational/unpack-array-group-differently.html#expected-results)

Four documents should be returned, representing only the four expensive products that were referenced multiple times in the customer orders, each showing the product's total order value and amount sold as shown below:

[

{

product\_id: 'pqr88223',

product: 'Morphy Richardds Food Mixer',

total\_value: NumberDecimal('431.43'),

quantity: 1

},

{

product\_id: 'abc12345',

product: 'Asus Laptop',

total\_value: NumberDecimal('861.42'),

quantity: 2

},

{

product\_id: 'def45678',

product: 'Karcher Hose Set',

total\_value: NumberDecimal('67.34'),

quantity: 3

},

{

product\_id: 'xyz11228',

product: 'Russell Hobbs Chrome Kettle',

total\_value: NumberDecimal('15.76'),

quantity: 1

}

]

Note, the order of fields shown for each document may vary.

### [Observations](https://www.practical-mongodb-aggregations.com/examples/foundational/unpack-array-group-differently.html#observations)

* **Unwinding Arrays.** The $unwind stage is a powerful concept, although often unfamiliar to many developers initially. Distilled down, it does one simple thing: it generates a new record for each element in an array field of every input document. If a source collection has 3 documents and each document contains an array of 4 elements, then performing an $unwind on each record's array field produces 12 records (3 x 4).
* **Introducing A Partial Match**. The current example pipeline scans all documents in the collection and then filters out unpacked products where price > 15.00. If the pipeline executed this filter as the first stage, it would incorrectly produce some result product records with a value of 15 dollars or less. This would be the case for an order composed of both inexpensive and expensive products. However, you can still improve the pipeline by including an additional "partial match" filter at the start of the pipeline for products valued at over 15 dollars. The aggregation could leverage an index (on products.price), resulting in a partial rather than full collection scan. This extra filter stage is beneficial if the input data set is large and many customer orders are for inexpensive items only. This approach is described in the chapter [Pipeline Performance Considerations](https://www.practical-mongodb-aggregations.com/guides/performance.html#explore-if-bringing-forward-a-partial-match-is-possible).

## [Distinct List Of Values](https://www.practical-mongodb-aggregations.com/examples/foundational/distinct-values.html#distinct-list-of-values)

### [Scenario](https://www.practical-mongodb-aggregations.com/examples/foundational/distinct-values.html#scenario)

You want to query a collection of persons where each document contains data on one or more languages spoken by the person. The query result should be an alphabetically sorted list of unique languages that a developer can subsequently use to populate a list of values in a user interface's "drop-down" widget.

This example is the equivalent of a SELECT DISTINCT statement in [SQL](https://en.wikipedia.org/wiki/SQL).

### [Sample Data Population](https://www.practical-mongodb-aggregations.com/examples/foundational/distinct-values.html#sample-data-population)

Drop any old version of the database (if it exists) and then populate a new persons collection with 9 documents:

use book-distinct-values;

db.dropDatabase();

// Insert records into the persons collection

db.persons.insertMany([

{

"firstname": "Elise",

"lastname": "Smith",

"vocation": "ENGINEER",

"language": "English",

},

{

"firstname": "Olive",

"lastname": "Ranieri",

"vocation": "ENGINEER",

"language": ["Italian", "English"],

},

{

"firstname": "Toni",

"lastname": "Jones",

"vocation": "POLITICIAN",

"language": ["English", "Welsh"],

},

{

"firstname": "Bert",

"lastname": "Gooding",

"vocation": "FLORIST",

"language": "English",

},

{

"firstname": "Sophie",

"lastname": "Celements",

"vocation": "ENGINEER",

"language": ["Gaelic", "English"],

},

{

"firstname": "Carl",

"lastname": "Simmons",

"vocation": "ENGINEER",

"language": "English",

},

{

"firstname": "Diego",

"lastname": "Lopez",

"vocation": "CHEF",

"language": "Spanish",

},

{

"firstname": "Helmut",

"lastname": "Schneider",

"vocation": "NURSE",

"language": "German",

},

{

"firstname": "Valerie",

"lastname": "Dubois",

"vocation": "SCIENTIST",

"language": "French",

},

]);

### [Aggregation Pipeline](https://www.practical-mongodb-aggregations.com/examples/foundational/distinct-values.html#aggregation-pipeline)

Define a single pipeline ready to perform the aggregation:

var pipeline = [

// Unpack each language field which may be an array or a single value

{"$unwind": {

"path": "$language",

}},

// Group by language

{"$group": {

"\_id": "$language",

}},

// Sort languages alphabetically

{"$sort": {

"\_id": 1,

}},

// Change \_id field's name to 'language'

{"$set": {

"language": "$\_id",

"\_id": "$$REMOVE",

}},

];

### [Execution](https://www.practical-mongodb-aggregations.com/examples/foundational/distinct-values.html#execution)

Execute the aggregation using the defined pipeline and also view its explain plan:

db.persons.aggregate(pipeline);

db.persons.explain("executionStats").aggregate(pipeline);

### [Expected Results](https://www.practical-mongodb-aggregations.com/examples/foundational/distinct-values.html#expected-results)

Seven unique language names should be returned sorted in alphabetical order, as shown below:

[

{language: 'English'},

{language: 'French'},

{language: 'Gaelic'},

{language: 'German'},

{language: 'Italian'},

{language: 'Spanish'},

{language: 'Welsh'}

]

### [Observations](https://www.practical-mongodb-aggregations.com/examples/foundational/distinct-values.html#observations)

* **Unwinding Non-Arrays.** In some of the example's documents, the language field is an array, whilst in others, the field is a simple string value. The $unwind stage can seamlessly deal with both field types and does not throw an error if it encounters a non-array value. Instead, if the field is not an array, the stage outputs a single record using the field's string value in the same way it would if the field was an array containing just one element. If you are sure the field in every document will only ever be a simple field rather than an array, you can omit this first stage ($unwind) from the pipeline.
* **Group ID Provides Unique Values.** By grouping on a single field and not accumulating other fields such as total or count, the output of a $group stage is just every unique group's ID, which in this case is every unique language.
* **Unset Alternative.** For the pipeline to be consistent with earlier examples in this book, it could have included an additional $unset stage to exclude the \_id field. However, partly to show another way, the example pipeline used here marks the \_id field for exclusion in the $set stage by being assigned the $$REMOVE variable.

## Aggregation with the Zip Code Data Set

**Import Data**:

Source: <https://media.mongodb.org/zips.json>

From the Unix shell prompt (*not mongo*):

Path to mongoimport: “C:\Program Files\MongoDB\Server\mongodb-database-tools\bin\mongoimport”

mongoimport --db products --collection zipcodes --type json --file zips.json

**Verify**:

mongo

use products

db.zipcodes.find()

db.zipcodes.find({“\_id”: “10280”})

**Return States with Populations above 10 Million**:

db.zipcodes.aggregate( [

{ $group: { \_id: "$state", totalPop: { $sum: "$pop" } } },

{ $match: { totalPop: { $gte: 10\*1000\*1000 } } }

] )

In this example, the aggregation pipeline consists of the $group stage followed by the $match stage:

* The $group stage groups the documents of the zipcode collection by the state field, calculates the totalPop field for each state, and outputs a document for each unique state.
  + The new per-state documents have two fields: the \_id field and the totalPop field.
  + The \_id field contains the value of the state; i.e. the group by field.
  + The totalPop field is a calculated field that contains the total population of each state.
  + To calculate the value, $group uses the $sum operator to add the population field (pop) for each state.
* After the $group stage, the documents in the pipeline resemble the following:

{

"\_id" : "AK",

"totalPop" : 550043

}

* The $match stage filters these grouped documents to output only those documents whose totalPop value is greater than or equal to 10 million. The $match stage does not alter the matching documents but outputs the matching documents unmodified.

The equivalent SQL for this aggregation operation is:

SELECT state, SUM(pop) AS totalPop

FROM zipcodes

GROUP BY state

HAVING totalPop >= (10\*1000\*1000)

**Return Average City Population by State**

The following aggregation operation returns the average populations for cities in each state:

db.zipcodes.aggregate( [

{$group: {\_id: {state: "$state", city: "$city"}, pop: {$sum: "$pop"}}},

{ $group: { \_id: "$\_id.state", avgCityPop: { $avg: "$pop" } } }

] )

In this example, the aggregation pipeline consists of the $group stage followed by another $group stage:

* The first $group stage groups the documents by the combination of city and state, uses the $sum expression to calculate the population for each combination, and outputs a document for each city and state combination.
* After this stage in the pipeline, the documents resemble the following:

{

"\_id" : {

"state" : "CO",

"city" : "EDGEWATER"

},

"pop" : 13154

}

* A second $group stage groups the documents in the pipeline by the \_id.state field (i.e. the state field inside the \_id document), uses the $avg expression to calculate the average city population (avgCityPop) for each state, and outputs a document for each state.
* The documents that result from this aggregation operation resembles the following:

{

"\_id" : "MN",

"avgCityPop" : 5335

}

**Return Largest and Smallest Cities by State**

The following aggregation operation returns the smallest and largest cities by population for each state:

db.zipcodes.aggregate( [

{ $group:

{

\_id: { state: "$state", city: "$city" },

pop: { $sum: "$pop" }

}

},

// 1 = ASC. -1 = DESC.

{ $sort: { pop: 1 } },

{ $group:

{

\_id : "$\_id.state",

biggestCity: { $last: "$\_id.city" },

biggestPop: { $last: "$pop" },

smallestCity: { $first: "$\_id.city" },

smallestPop: { $first: "$pop" }

}

},

// the following $project is optional, and

// modifies the output format.

{ $project:

{ \_id: 0,

state: "$\_id",

biggestCity: { name: "$biggestCity", pop: "$biggestPop" },

smallestCity: { name: "$smallestCity", pop: "$smallestPop" }

}

}

] )

In this example, the aggregation pipeline consists of a $group stage, a $sort stage, another $group stage, and a $project stage:

* The first [$group](https://docs.mongodb.com/manual/reference/operator/aggregation/group/#mongodb-pipeline-pipe.-group) stage groups the documents by the combination of the city and state, calculates the [sum](https://docs.mongodb.com/manual/reference/operator/aggregation/sum/#mongodb-group-grp.-sum) of the pop values for each combination, and outputs a document for each city and state combination.
* At this stage in the pipeline, the documents resemble the following:

|  |
| --- |
| { |
| "\_id" : { |
| "state" : "CO", |
| "city" : "EDGEWATER" |
| }, |
| "pop" : 13154 |
| } |

* The $sort stage orders the documents in the pipeline by the pop field value, from smallest to largest; i.e. by increasing order. This operation does not alter the documents.
* The next $group stage groups the now-sorted documents by the \_id.state field (i.e. the state field inside the \_id document) and outputs a document for each state.
  + The stage also calculates the following four fields for each state. Using the $last expression, the $group operator creates the biggestCity and biggestPop fields that store the city with the largest population and smallest population. Using the [$first](https://docs.mongodb.com/manual/reference/operator/aggregation/first/#mongodb-group-grp.-first) expression, the $group operator creates the smallestCity and smallestPop fields that store the city with the smallest population and largest population.
* The documents, at this stage in the pipeline, resemble the following:

|  |
| --- |
| { |
| "\_id" : "WA", |
| "biggestCity" : "SEATTLE", |
| "biggestPop" : 520096, |
| "smallestCity" : "BENGE", |
| "smallestPop" : 2 |
| } |

* The final $project stage renames the \_id field to state and moves the biggestCity, biggestPop, smallestCity, and smallestPop into biggestCity and smallestCity embedded documents.
* The output documents of this aggregation operation resemble the following:

|  |
| --- |
| { |
| "state" : "RI", |
| "biggestCity" : { |
| "name" : "CRANSTON", |
| "pop" : 176404 |
| }, |
| "smallestCity" : { |
| "name" : "CLAYVILLE", |
| "pop" : 45 |
| } |
| } |

## Aggregation with User Preference Data (*Exercise*)

**Data Model**

Consider a hypothetical sports club with a database that contains a users collection that tracks the user's join dates, sport preferences, and stores these data in documents that resemble the following:

|  |
| --- |
| { |
| \_id : "jane", |
| joined : ISODate("2011-03-02"), |
| likes : ["golf", "racquetball"] |
| } |
| { |
| \_id : "joe", |
| joined : ISODate("2012-07-02"), |
| likes : ["tennis", "golf", "swimming"] |
| } |

**Normalize and Sort Documents**

The following operation returns user names in upper case and in alphabetical order. The aggregation includes user names for all documents in the users collection. You might do this to normalize user names for processing.

|  |
| --- |
| db.users.aggregate( |
| [ |
| { $project : { name:{$toUpper:"$\_id"} , \_id:0 } }, |
| { $sort : { name : 1 } } |
| ] |
| ) |

All documents from the users collection pass through the pipeline, which consists of the following operations:

* The [$project](https://docs.mongodb.com/manual/reference/operator/aggregation/project/#mongodb-pipeline-pipe.-project) operator:
  + creates a new field called name.
  + converts the value of the \_id to upper case, with the [$toUpper](https://docs.mongodb.com/manual/reference/operator/aggregation/toUpper/#mongodb-expression-exp.-toUpper) operator. Then the [$project](https://docs.mongodb.com/manual/reference/operator/aggregation/project/#mongodb-pipeline-pipe.-project) creates a new field, named name to hold this value.
  + suppresses the id field. [$project](https://docs.mongodb.com/manual/reference/operator/aggregation/project/#mongodb-pipeline-pipe.-project) will pass the \_id field by default, unless explicitly suppressed.
* The [$sort](https://docs.mongodb.com/manual/reference/operator/aggregation/sort/#mongodb-pipeline-pipe.-sort) operator orders the results by the name field.

The results of the aggregation would resemble the following:

|  |
| --- |
| { |
| "name" : "JANE" |
| }, |
| { |
| "name" : "JILL" |
| }, |
| { |
| "name" : "JOE" |
| } |

**Return Usernames Ordered by Join Month**

The following aggregation operation returns user names sorted by the month they joined. This kind of aggregation could help generate membership renewal notices.

|  |
| --- |
| db.users.aggregate( |
| [ |
| { $project : |
| { |
| month\_joined : { $month : "$joined" }, |
| name : "$\_id", |
| \_id : 0 |
| } |
| }, |
| { $sort : { month\_joined : 1 } } |
| ] |
| ) |

The pipeline passes all documents in the users collection through the following operations:

* The [$project](https://docs.mongodb.com/manual/reference/operator/aggregation/project/#mongodb-pipeline-pipe.-project) operator:
  + Creates two new fields: month\_joined and name.
  + Suppresses the id from the results. The [aggregate()](https://docs.mongodb.com/manual/reference/method/db.collection.aggregate/#mongodb-method-db.collection.aggregate) method includes the \_id, unless explicitly suppressed.
* The [$month](https://docs.mongodb.com/manual/reference/operator/aggregation/month/#mongodb-expression-exp.-month) operator converts the values of the joined field to integer representations of the month. Then the [$project](https://docs.mongodb.com/manual/reference/operator/aggregation/project/#mongodb-pipeline-pipe.-project) operator assigns those values to the month\_joined field.
* The [$sort](https://docs.mongodb.com/manual/reference/operator/aggregation/sort/#mongodb-pipeline-pipe.-sort) operator sorts the results by the month\_joined field.

The operation returns results that resemble the following:

|  |
| --- |
| { |
| "month\_joined" : 1, |
| "name" : "ruth" |
| }, |
| { |
| "month\_joined" : 1, |
| "name" : "harold" |
| }, |
| { |
| "month\_joined" : 1, |
| "name" : "kate" |
| } |
| { |
| "month\_joined" : 2, |
| "name" : "jill" |
| } |

**Return Total Number of Joins per Month**

The following operation shows how many people joined each month of the year. You might use this aggregated data for recruiting and marketing strategies.

|  |
| --- |
| db.users.aggregate( |
| [ |
| { $project : { month\_joined : { $month : "$joined" } } } , |
| {$group: { \_id : {month\_joined:"$month\_joined"} , number : { $sum : 1 }}}, |
| { $sort : { "\_id.month\_joined" : 1 } } |
| ] |
| ) |

The pipeline passes all documents in the users collection through the following operations:

* The [$project](https://docs.mongodb.com/manual/reference/operator/aggregation/project/#mongodb-pipeline-pipe.-project) operator creates a new field called month\_joined.
* The [$month](https://docs.mongodb.com/manual/reference/operator/aggregation/month/#mongodb-expression-exp.-month) operator converts the values of the joined field to integer representations of the month. Then the [$project](https://docs.mongodb.com/manual/reference/operator/aggregation/project/#mongodb-pipeline-pipe.-project) operator assigns the values to the month\_joined field.
* The [$group](https://docs.mongodb.com/manual/reference/operator/aggregation/group/#mongodb-pipeline-pipe.-group) operator collects all documents with a given month\_joined value and counts how many documents there are for that value. Specifically, for each unique value, [$group](https://docs.mongodb.com/manual/reference/operator/aggregation/group/#mongodb-pipeline-pipe.-group) creates a new "per-month" document with two fields:
  + \_id, which contains a nested document with the month\_joined field and its value.
  + number, which is a generated field. The [$sum](https://docs.mongodb.com/manual/reference/operator/aggregation/sum/#mongodb-group-grp.-sum) operator increments this field by 1 for every document containing the given month\_joined value.
* The [$sort](https://docs.mongodb.com/manual/reference/operator/aggregation/sort/#mongodb-pipeline-pipe.-sort) operator sorts the documents created by [$group](https://docs.mongodb.com/manual/reference/operator/aggregation/group/#mongodb-pipeline-pipe.-group) according to the contents of the month\_joined field.

The result of this aggregation operation would resemble the following:

|  |
| --- |
| { |
| "\_id" : { |
| "month\_joined" : 1 |
| }, |
| "number" : 3 |
| }, |
| { |
| "\_id" : { |
| "month\_joined" : 2 |
| }, |
| "number" : 9 |
| }, |
| { |
| "\_id" : { |
| "month\_joined" : 3 |
| }, |
| "number" : 5 |
| } |

**Return the Five Most Common "Likes"**

The following aggregation collects top five most "liked" activities in the data set. This type of analysis could help inform planning and future development.

|  |
| --- |
| db.users.aggregate( |
| [ |
| { $unwind : "$likes" }, |
| { $group : { \_id : "$likes" , number : { $sum : 1 } } }, |
| { $sort : { number : -1 } }, |
| { $limit : 5 } |
| ] |
| ) |

The pipeline begins with all documents in the users collection, and passes these documents through the following operations:

* The [$unwind](https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/#mongodb-pipeline-pipe.-unwind) operator separates each value in the likes array, and creates a new version of the source document for every element in the array.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **EXAMPLE:**  Given the following document from the users collection:   |  | | --- | | { | | \_id : "jane", | | joined : ISODate("2011-03-02"), | | likes : ["golf", "racquetball"] | | } |   The [$unwind](https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/#mongodb-pipeline-pipe.-unwind) operator would create the following documents:   |  | | --- | | { | | \_id : "jane", | | joined : ISODate("2011-03-02"), | | likes : "golf" | | } | | { | | \_id : "jane", | | joined : ISODate("2011-03-02"), | | likes : "racquetball" | | } | |

* The [$group](https://docs.mongodb.com/manual/reference/operator/aggregation/group/#mongodb-pipeline-pipe.-group) operator collects all documents with the same value for the likes field and counts each grouping. With this information, [$group](https://docs.mongodb.com/manual/reference/operator/aggregation/group/#mongodb-pipeline-pipe.-group) creates a new document with two fields:
  + \_id, which contains the likes value.
  + number, which is a generated field. The [$sum](https://docs.mongodb.com/manual/reference/operator/aggregation/sum/#mongodb-group-grp.-sum) operator increments this field by 1 for every document containing the given likes value.
* The [$sort](https://docs.mongodb.com/manual/reference/operator/aggregation/sort/#mongodb-pipeline-pipe.-sort) operator sorts these documents by the number field in reverse order.
* The [$limit](https://docs.mongodb.com/manual/reference/operator/aggregation/limit/#mongodb-pipeline-pipe.-limit) operator only includes the first 5 result documents.

The results of aggregation would resemble the following:

|  |
| --- |
| { |
| "\_id" : "golf", |
| "number" : 33 |
| }, |
| { |
| "\_id" : "racquetball", |
| "number" : 31 |
| }, |
| { |
| "\_id" : "swimming", |
| "number" : 24 |
| }, |
| { |
| "\_id" : "handball", |
| "number" : 19 |
| }, |
| { |
| "\_id" : "tennis", |
| "number" : 18 |
| } |

## Normalized Documents

We just said that in MongoDB there is no normalization because storage is cheap and computational power expensive. But you can create normalize documents.

For example we can create a sales record for each size 2 girl large document like this with the **diaper**field pointing to the diaper object. That might make more sense in this case as you would not want the diaper collection to grow many times larger each time you make a sale.

***NOTE: Use relevant ObjectId of girlDiaper document.***

db.girlDiapers.insert([

{ "diaper" : ObjectId("607ae683212b25b6b16b20aa"),

"price" : 45.2,

"quanity" : 10,

"sku" : "case"

}

])

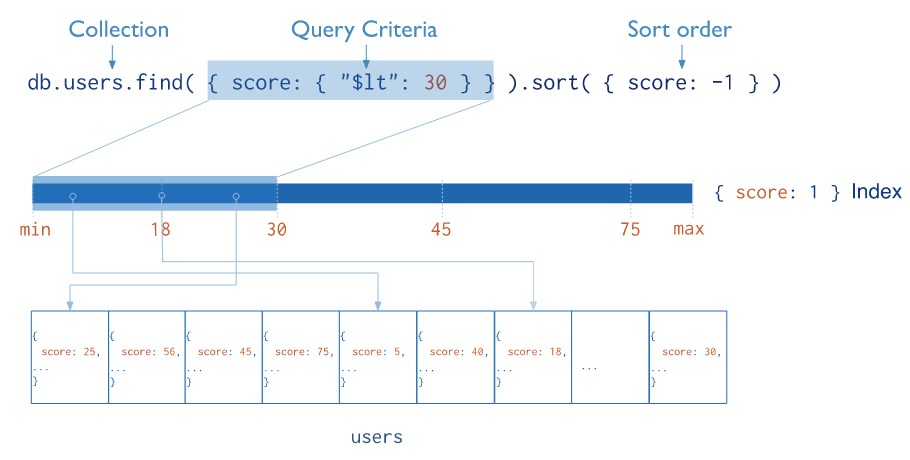
# Indexing

MongoDB allows you to create indexes, even on nested fields in subdocuments, to keep queries performing well even as collections grow very large.

Indexes support the efficient execution of queries in MongoDB. Without indexes, MongoDB must perform a collection scan, i.e. scan every document in a collection, to select those documents that match the query statement. If an appropriate index exists for a query, MongoDB can use the index to limit the number of documents it must inspect.

Indexes are special data structures that store a small portion of the collection's data set in an easy to traverse form. The index stores the value of a specific field or set of fields, ordered by the value of the field. The ordering of the index entries supports efficient equality matches and range-based query operations. In addition, MongoDB can return sorted results by using the ordering in the index.

The following diagram illustrates a query that selects and orders the matching documents using an index:



Fundamentally, indexes in MongoDB are similar to indexes in other database systems. MongoDB defines indexes at the collection level and supports indexes on any field or sub-field of the documents in a MongoDB collection.

## Default \_id Index

MongoDB creates a unique index on the \_id field during the creation of a collection. The \_id index prevents clients from inserting two documents with the same value for the \_id field. You cannot drop this index on the \_id field.

## Create an Index

The following example creates a single key descending index on the name field:

|  |
| --- |
| db.collection.createIndex( { name: -1 } ) |

The [db.collection.createIndex()](https://docs.mongodb.com/manual/reference/method/db.collection.createIndex/" \l "mongodb-method-db.collection.createIndex) method only creates an index if an index of the same specification does not already exist.

## Index Names

The default name for an index is the concatenation of the indexed keys and each key's direction in the index ( i.e. 1 or -1) using underscores as a separator. For example, an index created on { item : 1, quantity: -1 } has the name item\_1\_quantity\_-1.

You can create indexes with a custom name, such as one that is more human-readable than the default. For example, consider an application that frequently queries the products collection to populate data on existing inventory. The following [createIndex()](https://docs.mongodb.com/manual/reference/method/db.collection.createIndex/" \l "mongodb-method-db.collection.createIndex) method creates an index on item and quantity named query for inventory:

|  |
| --- |
| db.products.createIndex( |
| { item: 1, quantity: -1 } , |
| { name: **"query for inventory"** } |
| ) |

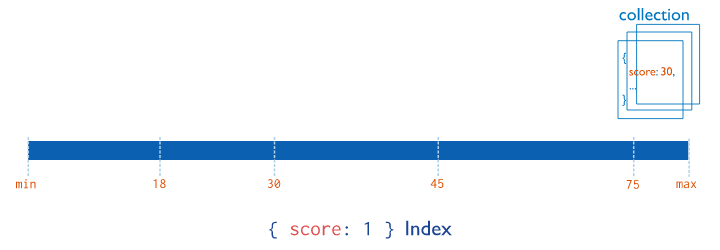
You can view index names using the [db.collection.getIndexes()](https://docs.mongodb.com/manual/reference/method/db.collection.getIndexes/" \l "mongodb-method-db.collection.getIndexes) method. You cannot rename an index once created. Instead, you must drop and re-create the index with a new name.

## Index Types

MongoDB provides a number of different index types to support specific types of data and queries.

### Single Field

In addition to the MongoDB-defined \_id index, MongoDB supports the creation of user-defined ascending/descending indexes on a single field of a document.

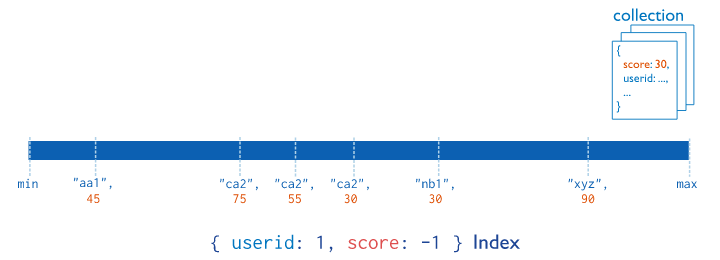


For a single-field index and sort operations, the sort order (i.e. ascending or descending) of the index key does not matter because MongoDB can traverse the index in either direction.

### Compound Index

MongoDB also supports user-defined indexes on multiple fields, i.e. compound indexes.

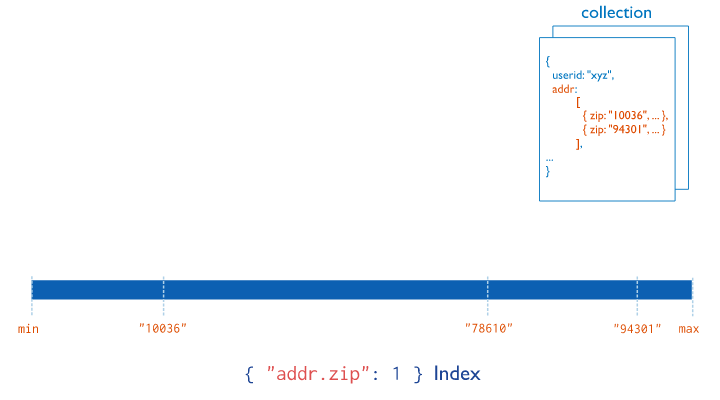
The order of fields listed in a compound index has significance. For instance, if a compound index consists of { userid: 1, score: -1 }, the index sorts first by userid and then, within each userid value, sorts by score.



For compound indexes and sort operations, the sort order (i.e. ascending or descending) of the index keys can determine whether the index can support a sort operation. See Sort Order for more information on the impact of index order on results in compound indexes.

### Multikey Index

MongoDB uses multikey indexes to index the content stored in arrays. If you index a field that holds an array value, MongoDB creates separate index entries for every element of the array. These multikey indexes allow queries to select documents that contain arrays by matching on element or elements of the arrays. MongoDB automatically determines whether to create a multikey index if the indexed field contains an array value; you do not need to explicitly specify the multikey type.



### Geospatial Index

To support efficient queries of geospatial coordinate data, MongoDB provides two special indexes: [2d indexes](https://docs.mongodb.com/manual/core/2d/) that uses planar geometry when returning results and [2dsphere indexes](https://docs.mongodb.com/manual/core/2dsphere/) that use spherical geometry to return results.

### Text Indexes

MongoDB provides a text index type that supports searching for string content in a collection. These text indexes do not store language-specific stop words (e.g. "the", "a", "or") and stem the words in a collection to only store root words.

### Hashed Indexes

To support [hash based sharding](https://docs.mongodb.com/manual/core/hashed-sharding/#std-label-sharding-hashed-sharding), MongoDB provides a [hashed index](https://docs.mongodb.com/manual/core/index-hashed/) type, which indexes the hash of the value of a field. These indexes have a more random distribution of values along their range, but only support equality matches and cannot support range-based queries.

## Examples

> db.user.createIndex({"name.family": 1})

Create a Unique Index

> db.user.createIndex({email: 1}, {unique: true})

Unique indexes allow you to ensure that there is at most one record in the collection with a given value for that field – very useful with things like email addresses!

### See Indexes on a Collection

> db.user.getIndexes()

[

{

"v" : 2,

"key" : {

"\_id" : 1

},

"name" : "\_id\_",

"ns" : "my\_database.user"

},

{

"v" : 2,

"key" : {

"name.given" : 1

},

"name" : "name.given\_1",

"ns" : "my\_database.user"

}

]

Note that by default, collections always have an index on the \_id field, for easy document retrieval by primary key, so any additional indexes will be listed after that.

### Drop an Index

> **db**.user.dropIndex("**name**.given\_1")

## Single Field Index

MongoDB provides complete support for indexes on any field in a collection of documents. By default, all collections have an index on the \_id field, and applications and users may add additional indexes to support important queries and operations.

### Create an Ascending Index on a Single Field

Consider a collection named records that holds documents that resemble the following sample document:

|  |
| --- |
| { |
| **"\_id"**: ObjectId(**"570c04a4ad233577f97dc459"**), |
| **"score"**: 1034, |
| **"location"**: { state: **"NY"**, city: **"New York"** } |
| } |

The following operation creates an ascending index on the score field of the records collection:

|  |
| --- |
| db.records.createIndex( { score: 1 } ) |

The value of the field in the index specification describes the kind of index for that field. For example, a value of 1 specifies an index that orders items in ascending order. A value of -1 specifies an index that orders items in descending order.

The created index will support queries that select on the field score, such as the following:

|  |
| --- |
| db.records.find( { score: 2 } ) |
| db.records.find( { score: { $gt: 10 } } ) |

### Create an Index on an Embedded Field

You can create indexes on fields within embedded documents, just as you can index top-level fields in documents. Indexes on embedded fields differ from indexes on embedded documents, which include the full content up to the maximum index size of the embedded document in the index. Instead, indexes on embedded fields allow you to use a "dot notation," to introspect into embedded documents.

Consider a collection named records that holds documents that resemble the following sample document:

|  |
| --- |
| { |
| **"\_id"**: ObjectId(**"570c04a4ad233577f97dc459"**), |
| **"score"**: 1034, |
| **"location"**: { state: **"NY"**, city: **"New York"** } |
| } |

The following operation creates an index on the location.state field:

|  |
| --- |
| db.records.createIndex( { **"location.state"**: 1 } ) |

The created index will support queries that select on the field location.state, such as the following:

|  |
| --- |
| db.records.find( { **"location.state"**: **"CA"** } ) |
| db.records.find( { **"location.city"**: **"Albany"**, **"location.state"**: **"NY"** } ) |

### Create an Index on Embedded Document

You can also create indexes on embedded document as a whole.

Consider a collection named records that holds documents that resemble the following sample document:

|  |
| --- |
| { |
| **"\_id"**: ObjectId(**"570c04a4ad233577f97dc459"**), |
| **"score"**: 1034, |
| **"location"**: { state: **"NY"**, city: **"New York"** } |
| } |

The location field is an embedded document, containing the embedded fields city and state. The following command creates an index on the location field as a whole:

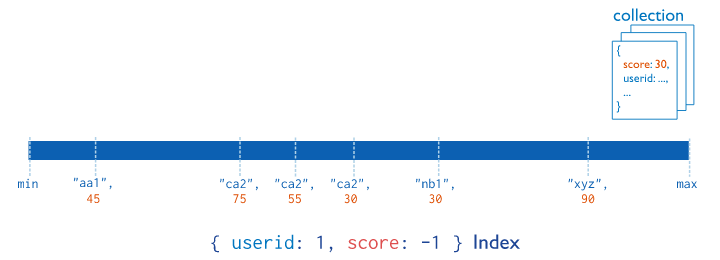
|  |
| --- |
| db.records.createIndex( { location: 1 } ) |

The following query can use the index on the location field:

|  |
| --- |
| db.records.find( { location: { city: **"New York"**, state: **"NY"** } } ) |

## Compound Indexes

MongoDB supports compound indexes, where a single index structure holds references to multiple fields within a collection's documents. The following diagram illustrates an example of a compound index on two fields:



Compound indexes can support queries that match on multiple fields.

### Create a Compound Index

To create a compound index use an operation that resembles the following prototype:

|  |
| --- |
| db.collection.createIndex( { <field1>: <type>, <field2>: <type2>, ... } ) |

The value of the field in the index specification describes the kind of index for that field. For example, a value of 1 specifies an index that orders items in ascending order. A value of -1 specifies an index that orders items in descending order.

Consider a collection named products that holds documents that resemble the following document:

|  |
| --- |
| { |
| **"\_id"**: ObjectId(...), |
| **"item"**: **"Banana"**, |
| **"category"**: [**"food"**, **"produce"**, **"grocery"**], |
| **"location"**: **"4th Street Store"**, |
| **"stock"**: 4, |
| **"type"**: **"cases"** |
| } |

The following operation creates an ascending index on the item and stock fields:

|  |
| --- |
| db.products.createIndex( { **"item"**: 1, **"stock"**: 1 } ) |

The order of the fields listed in a compound index is important. The index will contain references to documents sorted first by the values of the item field and, within each value of the item field, sorted by values of the stock field.

In addition to supporting queries that match on all the index fields, compound indexes can support queries that match on the prefix of the index fields. That is, the index supports queries on the item field as well as both item and stock fields:

|  |
| --- |
| db.products.find( { item: **"Banana"** } ) |
| db.products.find( { item: **"Banana"**, stock: { $gt: 5 } } ) |

### Sort Order

Indexes store references to fields in either ascending (1) or descending (-1) sort order. For single-field indexes, the sort order of keys doesn't matter because MongoDB can traverse the index in either direction. However, for compound indexes, sort order can matter in determining whether the index can support a sort operation.

Consider a collection events that contains documents with the fields username and date. Applications can issue queries that return results sorted first by ascending username values and then by descending (i.e. more recent to last) date values, such as:

|  |
| --- |
| db.events.find().sort( { username: 1, date: -1 } ) |

or queries that return results sorted first by descending username values and then by ascending date values, such as:

|  |
| --- |
| db.events.find().sort( { username: -1, date: 1 } ) |

The following index can support both these sort operations:

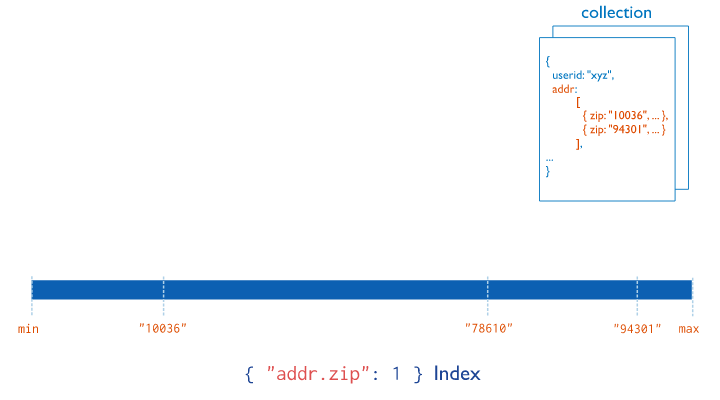
|  |
| --- |
| db.events.createIndex( { **"username"** : 1, **"date"** : -1 } ) |

However, the above index **cannot** support sorting by ascending username values and then by ascending date values, such as the following:

|  |
| --- |
| db.events.find().sort( { username: 1, date: 1 } ) |

## Multikey Indexes

To index a field that holds an array value, MongoDB creates an index key for each element in the array. These multikey indexes support efficient queries against array fields. Multikey indexes can be constructed over arrays that hold both scalar values  (e.g. strings, numbers) and nested documents.



### Index Basic Arrays

Consider a survey collection with the following document:

|  |
| --- |
| { \_id: 1, item: **"ABC"**, ratings: [ 2, 5, 9 ] } |

Create an index on the field ratings:

|  |
| --- |
| db.survey.createIndex( { ratings: 1 } ) |

Since the ratings field contains an array, the index on ratings is multikey. The multikey index contains the following three index keys, each pointing to the same document:

* 2,
* 5, and
* 9.

### Index Arrays with Embedded Documents

You can create multikey indexes on array fields that contain nested objects.

Consider an inventory collection with documents of the following form:

|  |
| --- |
| { |
| \_id: 1, |
| item: **"abc"**, |
| stock: [ |
| { size: **"S"**, color: **"red"**, quantity: 25 }, |
| { size: **"S"**, color: **"blue"**, quantity: 10 }, |
| { size: **"M"**, color: **"blue"**, quantity: 50 } |
| ] |
| } |
| { |
| \_id: 2, |
| item: **"def"**, |
| stock: [ |
| { size: **"S"**, color: **"blue"**, quantity: 20 }, |
| { size: **"M"**, color: **"blue"**, quantity: 5 }, |
| { size: **"M"**, color: **"black"**, quantity: 10 }, |
| { size: **"L"**, color: **"red"**, quantity: 2 } |
| ] |
| } |
| { |
| \_id: 3, |
| item: **"ijk"**, |
| stock: [ |
| { size: **"M"**, color: **"blue"**, quantity: 15 }, |
| { size: **"L"**, color: **"blue"**, quantity: 100 }, |
| { size: **"L"**, color: **"red"**, quantity: 25 } |
| ] |
| } |
|  |
| ... |

The following operation creates a multikey index on the stock.size and stock.quantity fields:

|  |
| --- |
| db.inventory.createIndex( { **"stock.size"**: 1, **"stock.quantity"**: 1 } ) |

The compound multikey index can support queries with predicates that include both indexed fields as well as predicates that include only the index prefix "stock.size", as in the following examples:

|  |
| --- |
| db.inventory.find( { **"stock.size"**: **"M"** } ) |
| db.inventory.find( { **"stock.size"**: **"S"**, **"stock.quantity"**: { $gt: 20 } } ) |

The compound multikey index can also support sort operations, such as the following examples:

|  |
| --- |
| db.inventory.find( ).sort( { **"stock.size"**: 1, **"stock.quantity"**: 1 } ) |
| db.inventory.find( { **"stock.size"**: **"M"** } ).sort( { **"stock.quantity"**: 1 } ) |

## Manage Indexes

### List all Indexes on a Collection

To return a list of all indexes on a collection, use the [db.collection.getIndexes()](https://docs.mongodb.com/manual/reference/method/db.collection.getIndexes/" \l "mongodb-method-db.collection.getIndexes) method or a similar method for your driver.

For example, to view all indexes on the people collection, run the following command:

|  |
| --- |
| db.people.getIndexes() |

### List All Indexes for a Database

To list all the collection indexes in a database, you can use the following operation in [mongosh](https://docs.mongodb.com/mongodb-shell/" \l "mongodb-binary-bin.mongosh):

|  |
| --- |
| db.getCollectionNames().forEach(function(collection) { |
| indexes = db[collection].getIndexes(); |
| print(**"Indexes for "** + collection + **":"**); |
| printjson(indexes); |
| }); |

*Starting in version 3.0, MongoDB deprecates direct access to the system.indexes collection, which had previously been used to list all indexes in a database.*

### Remove Indexes

#### Remove Specific Index

To remove an index, use the [db.collection.dropIndex()](https://docs.mongodb.com/manual/reference/method/db.collection.dropIndex/" \l "mongodb-method-db.collection.dropIndex) method.

For example, the following operation removes an ascending index on the tax-id field in the accounts collection:

|  |
| --- |
| db.accounts.dropIndex( { **"tax-id"**: 1 } ) |

The operation returns a document with the status of the operation:

|  |
| --- |
| { **"nIndexesWas"** : 3, **"ok"** : 1 } |

Where the value of nIndexesWas reflects the number of indexes before removing this index.

#### Remove All Indexes

You can also use the [db.collection.dropIndexes()](https://docs.mongodb.com/manual/reference/method/db.collection.dropIndexes/" \l "mongodb-method-db.collection.dropIndexes) to remove all indexes except for the \_id index from a collection.

For example, the following command removes all indexes from the accounts collection:

|  |
| --- |
| db.accounts.dropIndexes() |

# Schemas and Relations

## One-to-One Relationships with Embedded Documents

Embedding connected data in a single document can reduce the number of read operations required to obtain data. In general, you should structure your schema so your application receives all of its required information in a single read operation.

### Embedded Document Pattern

The following example that maps patron and address relationships. The example illustrates the advantage of embedding over referencing if you need to view one data entity in context of the other. In this one-to-one relationship between patron and address data, the address belongs to the patron.

In the normalized data model, the address document contains a reference to the patron document.

// patron document

{

\_id: "joe",

name: "Joe Bookreader"

}

// address document

{

patron\_id: "joe", // reference to patron document

street: "123 Fake Street",

city: "Faketon",

state: "MA",

zip: "12345"

}

If the address data is frequently retrieved with the name information, then with referencing, your application needs to issue multiple queries to resolve the reference. The better data model would be to embed the address data in the patron data, as in the following document:

{

\_id: "joe",

name: "Joe Bookreader",

address: {

street: "123 Fake Street",

city: "Faketon",

state: "MA",

zip: "12345"

}

}

With the embedded data model, your application can retrieve the complete patron information with one query.

### Subset Pattern[https://docs.mongodb.com/manual/tutorial/model-embedded-one-to-one-relationships-between-documents/ - subset-pattern](https://docs.mongodb.com/manual/tutorial/model-embedded-one-to-one-relationships-between-documents/#subset-pattern)

A potential problem with the embedded document pattern is that it can lead to large documents that contain fields that the application does not need. This unnecessary data can cause extra load on your server and slow down read operations. Instead, you can use the subset pattern to retrieve the subset of data which is accessed the most frequently in a single database call.

Consider an application that shows information on movies. The database contains a movie collection with the following schema:

{

"\_id": 1,

"title": "The Arrival of a Train",

"year": 1896,

"runtime": 1,

"released": ISODate("01-25-1896"),

"poster": "http://ia.media-imdb.com/images/M/MV5BMjEyNDk5MDYzOV5BMl5BanBnXkFtZTgwNjIxMTEwMzE@.\_V1\_SX300.jpg",

"plot": "A group of people are standing in a straight line along the platform of a railway station, waiting for a train, which is seen coming at some distance. When the train stops at the platform, ...",

"fullplot": "A group of people are standing in a straight line along the platform of a railway station, waiting for a train, which is seen coming at some distance. When the train stops at the platform, the line dissolves. The doors of the railway-cars open, and people on the platform help passengers to get off.",

"lastupdated": ISODate("2015-08-15T10:06:53"),

"type": "movie",

"directors": [ "Auguste Lumière", "Louis Lumière" ],

"imdb": {

"rating": 7.3,

"votes": 5043,

"id": 12

},

"countries": [ "France" ],

"genres": [ "Documentary", "Short" ],

"tomatoes": {

"viewer": {

"rating": 3.7,

"numReviews": 59

},

"lastUpdated": ISODate("2020-01-09T00:02:53")

}

}

Here, the movie collection contains several fields that the application does not need to show a simple overview of a movie, such as fullplot and rating information. Instead of storing all of the movie data in a single collection, you can split the collection into two collections:

* The movie collection contains basic information on a movie. This is the data that the application loads by default:

// movie collection

{

"\_id": 1,

"title": "The Arrival of a Train",

"year": 1896,

"runtime": 1,

"released": ISODate("1896-01-25"),

"type": "movie",

"directors": [ "Auguste Lumière", "Louis Lumière" ],

"countries": [ "France" ],

"genres": [ "Documentary", "Short" ],

}

* The movie\_details collection contains additional, less frequently-accessed data for each movie:

// movie\_details collection

{

"\_id": 156,

"movie\_id": 1, // reference to the movie collection

"poster": "http://ia.media-imdb.com/images/M/MV5BMjEyNDk5MDYzOV5BMl5BanBnXkFtZTgwNjIxMTEwMzE@.\_V1\_SX300.jpg",

"plot": "A group of people are standing in a straight line along the platform of a railway station, waiting for a train, which is seen coming at some distance. When the train stops at the platform, ...",

"fullplot": "A group of people are standing in a straight line along the platform of a railway station, waiting for a train, which is seen coming at some distance. When the train stops at the platform, the line dissolves. The doors of the railway-cars open, and people on the platform help passengers to get off.",

"lastupdated": ISODate("2015-08-15T10:06:53"),

"imdb": {

"rating": 7.3,

"votes": 5043,

"id": 12

},

"tomatoes": {

"viewer": {

"rating": 3.7,

"numReviews": 59

},

"lastUpdated": ISODate("2020-01-29T00:02:53")

}

}

This method improves read performance because it requires the application to read less data to fulfill its most common request. The application can make an additional database call to fetch the less-frequently accessed data if needed.

### Trade-Offs of the Subset Pattern

Using smaller documents containing more frequently-accessed data reduces the overall size of the working set. These smaller documents result in improved read performance and make more memory available for the application.

However, it is important to understand your application and the way it loads data. If you split your data into multiple collections improperly, your application will often need to make multiple trips to the database and rely on JOIN operations to retrieve all of the data that it needs.

In addition, splitting your data into many small collections may increase required database maintenance, as it may become difficult to track what data is stored in which collection.

## One-to-Many Relationships with Embedded Documents

### Embedded Document Pattern

The following example that maps patron and multiple address relationships. The example illustrates the advantage of embedding over referencing if you need to view many data entities in context of another. In this one-to-many relationship between patron and address data, the patron has multiple address entities.

In the normalized data model, the address documents contain a reference to the patron document.

// patron document

{

\_id: "joe",

name: "Joe Bookreader"

}

// address documents

{

patron\_id: "joe", // reference to patron document

street: "123 Fake Street",

city: "Faketon",

state: "MA",

zip: "12345"

}

{

patron\_id: "joe",

street: "1 Some Other Street",

city: "Boston",

state: "MA",

zip: "12345"

}

If your application frequently retrieves the address data with the name information, then your application needs to issue multiple queries to resolve the references. A more optimal schema would be to embed the address data entities in the patron data, as in the following document:

{

"\_id": "joe",

"name": "Joe Bookreader",

"addresses": [

{

"street": "123 Fake Street",

"city": "Faketon",

"state": "MA",

"zip": "12345"

},

{

"street": "1 Some Other Street",

"city": "Boston",

"state": "MA",

"zip": "12345"

}

]

}

With the embedded data model, your application can retrieve the complete patron information with one query.

### Subset Pattern

A potential problem with the embedded document pattern is that it can lead to large documents, especially if the embedded field is unbounded. In this case, you can use the subset pattern to only access data which is required by the application, instead of the entire set of embedded data.

Consider an e-commerce site that has a list of reviews for a product:

{

"\_id": 1,

"name": "Super Widget",

"description": "This is the most useful item in your toolbox.",

"price": { "value": NumberDecimal("119.99"), "currency": "USD" },

"reviews": [

{

"review\_id": 786,

"review\_author": "Kristina",

"review\_text": "This is indeed an amazing widget.",

"published\_date": ISODate("2019-02-18")

},

{

"review\_id": 785,

"review\_author": "Trina",

"review\_text": "Nice product. Slow shipping.",

"published\_date": ISODate("2019-02-17")

},

...

{

"review\_id": 1,

"review\_author": "Hans",

"review\_text": "Meh, it's okay.",

"published\_date": ISODate("2017-12-06")

}

]

}

The reviews are sorted in reverse chronological order. When a user visits a product page, the application loads the ten most recent reviews.

Instead of storing all of the reviews with the product, you can split the collection into two collections:

* The product collection stores information on each product, including the product's ten most recent reviews:

// The product collection.

{

"\_id": 1,

"name": "Super Widget",

"description": "This is the most useful item in your toolbox.",

"price": { "value": NumberDecimal("119.99"), "currency": "USD" },

"reviews": [

{

"review\_id": 786,

"review\_author": "Kristina",

"review\_text": "This is indeed an amazing widget.",

"published\_date": ISODate("2019-02-18")

}

...

{

"review\_id": 776,

"review\_author": "Pablo",

"review\_text": "Amazing!",

"published\_date": ISODate("2019-02-16")

}

]

}

* The review collection stores all reviews. Each review contains a reference to the product for which it was written.

// The review collection.

{

"review\_id": 786,

"product\_id": 1,

"review\_author": "Kristina",

"review\_text": "This is indeed an amazing widget.",

"published\_date": ISODate("2019-02-18")

}

{

"review\_id": 785,

"product\_id": 1,

"review\_author": "Trina",

"review\_text": "Nice product. Slow shipping.",

"published\_date": ISODate("2019-02-17")

}

...

{

"review\_id": 1,

"product\_id": 1,

"review\_author": "Hans",

"review\_text": "Meh, it's okay.",

"published\_date": ISODate("2017-12-06")

}

By storing the ten most recent reviews in the product collection, only the required subset of the overall data is returned in the call to the product collection. If a user wants to see additional reviews, the application makes a call to the review collection.

### Trade-Offs of the Subset Pattern

Using smaller documents containing more frequently-accessed data reduces the overall size of the working set. These smaller documents result in improved read performance for the data that the application accesses most frequently.

However, the subset pattern results in data duplication. In the example, reviews are maintained in both the product collection and the reviews collection. Extra steps must be taken to ensure that the reviews are consistent between each collection. For example, when a customer edits their review, the application may need to make two write operations: one to update the product collection and one to update the reviews collection.

You must also implement logic in your application to ensure that the reviews in the product collection are always the ten most recent reviews for that product.

**Other Sample Use Cases**:

In addition to product reviews, the subset pattern can also be a good fit to store:

* Comments on a blog post, when you only want to show the most recent or highest-rated comments by default.
* Cast members in a movie, when you only want to show cast members with the largest roles by default.

## One-to-Many Relationships with Document References

### Pattern

The following example that maps publisher and book relationships. The example illustrates the advantage of referencing over embedding to avoid repetition of the publisher information.

Embedding the publisher document inside the book document would lead to **repetition** of the publisher data, as the following documents show:

{

title: "MongoDB: The Definitive Guide",

author: [ "Kristina Chodorow", "Mike Dirolf" ],

published\_date: ISODate("2010-09-24"),

pages: 216,

language: "English",

publisher: {

name: "O'Reilly Media",

founded: 1980,

location: "CA"

}

}

{

title: "50 Tips and Tricks for MongoDB Developer",

author: "Kristina Chodorow",

published\_date: ISODate("2011-05-06"),

pages: 68,

language: "English",

publisher: {

name: "O'Reilly Media",

founded: 1980,

location: "CA"

}

}

To avoid repetition of the publisher data, use *references* and keep the publisher information in a separate collection from the book collection.

When using references, the growth of the relationships determine where to store the reference. If the number of books per publisher is small with limited growth, storing the book reference inside the publisher document may sometimes be useful. Otherwise, if the number of books per publisher is unbounded, this data model would lead to mutable, growing arrays, as in the following example:

// publisher.

{

name: "O'Reilly Media",

founded: 1980,

location: "CA",

books: [123456789, 234567890, ...]

}

// books.

{

\_id: 123456789,

title: "MongoDB: The Definitive Guide",

author: [ "Kristina Chodorow", "Mike Dirolf" ],

published\_date: ISODate("2010-09-24"),

pages: 216,

language: "English"

}

{

\_id: 234567890,

title: "50 Tips and Tricks for MongoDB Developer",

author: "Kristina Chodorow",

published\_date: ISODate("2011-05-06"),

pages: 68,

language: "English"

}

To avoid mutable, growing arrays, store the publisher reference inside the book document:

// publisher.

{

\_id: "oreilly",

name: "O'Reilly Media",

founded: 1980,

location: "CA"

}

// books.

{

\_id: 123456789,

title: "MongoDB: The Definitive Guide",

author: [ "Kristina Chodorow", "Mike Dirolf" ],

published\_date: ISODate("2010-09-24"),

pages: 216,

language: "English",

publisher\_id: "oreilly"

}

{

\_id: 234567890,

title: "50 Tips and Tricks for MongoDB Developer",

author: "Kristina Chodorow",

published\_date: ISODate("2011-05-06"),

pages: 68,

language: "English",

publisher\_id: "oreilly"

}

REF:

<https://docs.mongodb.com/manual/tutorial/model-tree-structures-with-parent-references/>

# MongoDB Drivers - PyMongo

Of course, you probably would not use the command line shell for an application. Instead you would write a program to interact with MongoDB using any of the many drivers available. There are drivers for C++, C#, Java, Node.JS, Scala, Python, and more.

### Pre-requisites for PyMongo

* **Installation of python3**

sudo apt install python3

* **Pip Installation**

sudo apt install python3-pip

* **Add following alias in $HOME/.bash\_aliases in some cases file may be hidden.**

alias pip="/usr/bin/python3 -m pip "

* **Refresh current terminal session.**

. ~/.profile

* **Check pip usage**

pip

* **Check pip version**

pip3 –version

pip --version

### Install PyMongo on Ubuntu

To use Python, execute the following from the O/S prompt:

sudo pip3 install pymongo

### Install PyMongo on Windows

Execute either of these commands:

pip install pymongo

py -m pip install pymongo

python3 -m pip install pymongo

python -m pip install pymongo

Make sure the MonoDB daemon has been started to use pymongo.

Then to query for size 2 diapers across the boy and girl collections from the mongodb shell:

from pymongo import MongoClient

client = MongoClient()

db = client.products

x=db.list\_collection\_names()

for i in range(len(x)):

c = x[i]

d = db.get\_collection(c)

for e in d.find({"size": 2}):

print(e)

Outputs:

{'size': 2.0, 'brand': 'boy large white', 'color': 'white', '\_id': ObjectId('59d1f564ccf50b62c5a7af58')}

{'size': 2.0, 'brand': 'girl large', 'color': 'white', '\_id': ObjectId('59d1f565ccf50b62c5a7af59')}

## Some Useful Commands

### Making a connection to MongoDB

>>> **from** **pymongo** **import** MongoClient

>>>client = MongoClient()

>>> client

MongoClient(host=['localhost:27017'], ..., connect=True)

### Getting a database

>>> db = client.test\_database

>>> db = client[“test\_database”]

>>> db = client[“test-database”]

### Getting a collection

>>> collection = db.test\_collection

>>> collection = db['test-collection']

## Sample Database Operations

In the MongoDB shell, create a database and collection as follows:

> use test\_database

> db.createCollection(“posts”)

Then, switch to the Python shell / terminal and execute the following:

**>>>** post = {"author": "Mike",

**...**  "text": "My first blog post!",

**...**  "tags": ["mongodb", "python", "pymongo"],

**...**  "date": datetime.datetime.utcnow()}

### Inserting a Document

To insert a document into a collection we can use the [insert\_one()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html" \l "pymongo.collection.Collection.insert_one" \o "pymongo.collection.Collection.insert_one) method:

**>>>** posts = db.posts

**>>>** post\_id = posts.insert\_one(post).inserted\_id

**>>>** post\_id

ObjectId('...')

When a document is inserted a special key, "\_id", is automatically added if the document doesn’t already contain an "\_id" key. The value of "\_id" must be unique across the collection. [insert\_one()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html" \l "pymongo.collection.Collection.insert_one" \o "pymongo.collection.Collection.insert_one) returns an instance of [InsertOneResult](https://pymongo.readthedocs.io/en/stable/api/pymongo/results.html" \l "pymongo.results.InsertOneResult" \o "pymongo.results.InsertOneResult).

After inserting the first document, the posts collection has actually been created on the server. We can verify this by listing all of the collections in our database:

**>>>** db.list\_collection\_names()

[u'posts']

### Getting a Single Document With **[find\_one()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html" \l "pymongo.collection.Collection.find_one" \o "pymongo.collection.Collection.find_one)**

The most basic type of query that can be performed in MongoDB is [find\_one()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html" \l "pymongo.collection.Collection.find_one" \o "pymongo.collection.Collection.find_one). This method returns a single document matching a query (or None if there are no matches). It is useful when you know there is only one matching document, or are only interested in the first match. Here we use [find\_one()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html" \l "pymongo.collection.Collection.find_one" \o "pymongo.collection.Collection.find_one) to get the first document from the posts collection:

**>>> import** **pprint**

**>>>** pprint.pprint(posts.find\_one())

{u'\_id': ObjectId('...'),

u'author': u'Mike',

u'date': datetime.datetime(...),

u'tags': [u'mongodb', u'python', u'pymongo'],

u'text': u'My first blog post!'}

The result is a dictionary matching the one that we inserted previously.

Note

 The returned document contains an "\_id", which was automatically added on insert.

[find\_one()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html#pymongo.collection.Collection.find_one) also supports querying on specific elements that the resulting document must match. To limit our results to a document with author “Mike” we do:

**>>>** pprint.pprint(posts.find\_one({"author": "Mike"}))

{u'\_id': ObjectId('...'),

u'author': u'Mike',

u'date': datetime.datetime(...),

u'tags': [u'mongodb', u'python', u'pymongo'],

u'text': u'My first blog post!'}

If we try with a different author, like “Eliot”, we’ll get no result:

**>>>** posts.find\_one({"author": "Eliot"})

>>>

Once you have the document returned, you can extract the attributes as follows:

**>>>** aPost =posts.find\_one({"author": "Eliot"})

**>>>** aPost[“author”]

**>>>** aPost[“text”]

### Querying By ObjectId

We can also find a post by its \_id, which in our example is an ObjectId:

**>>>** post\_id

ObjectId(...)

**>>>** pprint.pprint(posts.find\_one({"\_id": post\_id}))

{u'\_id': ObjectId('...'),

u'author': u'Mike',

u'date': datetime.datetime(...),

u'tags': [u'mongodb', u'python', u'pymongo'],

u'text': u'My first blog post!'}

Note that an ObjectId is not the same as its string representation:

**>>>** post\_id\_as\_str = str(post\_id)

**>>>** posts.find\_one({"\_id": post\_id\_as\_str}) *# No result*

>>>

A common task in web applications is to get an ObjectId from the request URL and find the matching document. It’s necessary in this case to **convert the ObjectId from a string** before passing it to find\_one:

**from** **bson.objectid** **import** ObjectId

*# The web framework gets post\_id from the URL and passes it as a string*

**def** get(post\_id\_str):

*# Convert from string to ObjectId:*

document = client.test\_database.posts.find\_one({'\_id': ObjectId(post\_id\_str)})

print(document)

**A Note On Unicode Strings**

You probably noticed that the regular Python strings we stored earlier look different when retrieved from the server (e.g. u’Mike’ instead of ‘Mike’). A short explanation is in order.

MongoDB stores data in [BSON format](http://bsonspec.org/). BSON strings are UTF-8 encoded so PyMongo must ensure that any strings it stores contain only valid UTF-8 data. Regular strings (<type ‘str’>) are validated and stored unaltered. Unicode strings (<type ‘unicode’>) are encoded UTF-8 first. The reason our example string is represented in the Python shell as u’Mike’ instead of ‘Mike’ is that PyMongo decodes each BSON string to a Python unicode string, not a regular str.

### Bulk Inserts

In order to make querying a little more interesting, let’s insert a few more documents. In addition to inserting a single document, we can also perform bulk insert operations, by passing a list as the first argument to [insert\_many()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html" \l "pymongo.collection.Collection.insert_many" \o "pymongo.collection.Collection.insert_many). This will insert each document in the list, sending only a single command to the server:

**>>>** new\_posts = [{"author": "Mike",

**...**  "text": "Another post!",

**...**  "tags": ["bulk", "insert"],

**...**  "date": datetime.datetime(2009, 11, 12, 11, 14)},

**...**  {"author": "Eliot",

**...**  "title": "MongoDB is fun",

**...**  "text": "and pretty easy too!",

**...**  "date": datetime.datetime(2009, 11, 10, 10, 45)}]

**>>>** new\_posts[0]

**>>>** new\_posts[1]

**>>>** result = posts.insert\_many(new\_posts)

**>>>** result.inserted\_ids

[ObjectId('...'), ObjectId('...')]

There are a couple of interesting things to note about this example:

* The result from [insert\_many()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html" \l "pymongo.collection.Collection.insert_many" \o "pymongo.collection.Collection.insert_many) now returns two [ObjectId](https://pymongo.readthedocs.io/en/stable/api/bson/objectid.html" \l "bson.objectid.ObjectId" \o "bson.objectid.ObjectId) instances, one for each inserted document.
* new\_posts[1] has a different “shape” than the other posts - there is no "tags" field and we’ve added a new field, "title". This is what we mean when we say that MongoDB is schema-free.

### Querying for More Than One Document

To get more than a single document as the result of a query we use the [find()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html#pymongo.collection.Collection.find) method. [find()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html#pymongo.collection.Collection.find) returns a [Cursor](https://pymongo.readthedocs.io/en/stable/api/pymongo/cursor.html#pymongo.cursor.Cursor) instance, which allows us to iterate over all matching documents. For example, we can iterate over every document in the posts collection:

**>>> for** post **in** posts.find():

**...**  pprint.pprint(post)

**...**

{u'\_id': ObjectId('...'),

u'author': u'Mike',

u'date': datetime.datetime(...),

u'tags': [u'mongodb', u'python', u'pymongo'],

u'text': u'My first blog post!'}

{u'\_id': ObjectId('...'),

u'author': u'Mike',

u'date': datetime.datetime(...),

u'tags': [u'bulk', u'insert'],

u'text': u'Another post!'}

{u'\_id': ObjectId('...'),

u'author': u'Eliot',

u'date': datetime.datetime(...),

u'text': u'and pretty easy too!',

u'title': u'MongoDB is fun'}

Just like we did with [find\_one()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html" \l "pymongo.collection.Collection.find_one" \o "pymongo.collection.Collection.find_one), we can pass a document to [find()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html#pymongo.collection.Collection.find) to limit the returned results. Here, we get only those documents whose author is “Mike”:

**>>> for** post **in** posts.find({"author": "Mike"}):

**...**  pprint.pprint(post)

**...**

{u'\_id': ObjectId('...'),

u'author': u'Mike',

u'date': datetime.datetime(...),

u'tags': [u'mongodb', u'python', u'pymongo'],

u'text': u'My first blog post!'}

{u'\_id': ObjectId('...'),

u'author': u'Mike',

u'date': datetime.datetime(...),

u'tags': [u'bulk', u'insert'],

u'text': u'Another post!'}

### Counting

If we just want to know how many documents match a query we can perform a [count\_documents()](https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html" \l "pymongo.collection.Collection.count_documents" \o "pymongo.collection.Collection.count_documents) operation instead of a full query. We can get a count of all of the documents in a collection:

**>>>** posts.count\_documents({})

3

or just of those documents that match a specific query:

**>>>** posts.count\_documents({"author": "Mike"})

2

### Range Queries

MongoDB supports many different types of [advanced queries](http://www.mongodb.org/display/DOCS/Advanced+Queries). As an example, lets perform a query where we limit results to posts older than a certain date, but also sort the results by author:

**>>>** d = datetime.datetime(2009, 11, 12, 12)

**>>> for** post **in** posts.find({"date": {"$lt": d}}).sort("author"):

**...**  pprint.pprint(post)

**...**

{u'\_id': ObjectId('...'),

u'author': u'Eliot',

u'date': datetime.datetime(...),

u'text': u'and pretty easy too!',

u'title': u'MongoDB is fun'}

{u'\_id': ObjectId('...'),

u'author': u'Mike',

u'date': datetime.datetime(...),

u'tags': [u'bulk', u'insert'],

u'text': u'Another post!'}

Here we use the special "$lt" operator to do a range query, and also call [sort()](https://pymongo.readthedocs.io/en/stable/api/pymongo/cursor.html#pymongo.cursor.Cursor.sort) to sort the results by author.

#### ­­Query and Projection Operators

**Query Selectors**

|  |  |
| --- | --- |
| **Name** | **Description** |
| **Comparison** | | |
| [$eq](https://docs.mongodb.com/manual/reference/operator/query/eq/#mongodb-query-op.-eq) | Matches values that are equal to a specified value. |
| [$gt](https://docs.mongodb.com/manual/reference/operator/query/gt/#mongodb-query-op.-gt) | Matches values that are greater than a specified value. |
| [$gte](https://docs.mongodb.com/manual/reference/operator/query/gte/#mongodb-query-op.-gte) | Matches values that are greater than or equal to a specified value. |
| [$in](https://docs.mongodb.com/manual/reference/operator/query/in/#mongodb-query-op.-in) | Matches any of the values specified in an array. |
| [$lt](https://docs.mongodb.com/manual/reference/operator/query/lt/#mongodb-query-op.-lt) | Matches values that are less than a specified value. |
| [$lte](https://docs.mongodb.com/manual/reference/operator/query/lte/#mongodb-query-op.-lte) | Matches values that are less than or equal to a specified value. |
| [$ne](https://docs.mongodb.com/manual/reference/operator/query/ne/#mongodb-query-op.-ne) | Matches all values that are not equal to a specified value. |
| [$nin](https://docs.mongodb.com/manual/reference/operator/query/nin/#mongodb-query-op.-nin) | Matches none of the values specified in an array. |
| **Logical** | | |
| [$and](https://docs.mongodb.com/manual/reference/operator/query/and/#mongodb-query-op.-and) | Joins query clauses with a logical AND returns all documents that match the conditions of both clauses | |
| $not | Inverts the effect of a query expression and returns documents that do not match the query expression. | |
| $nor | Joins query clauses with a logical NOR returns all documents that fail to match both clauses. | |
| $or | Joins query clauses with a logical OR returns all documents that match the conditions of either clause. | |
| **Logical** | | |
| $exists | Matches documents that have the specified field. | |
| $type | Selects documents if a field is of the specified type. | |

### Indexing

Adding indexes can help accelerate certain queries and can also add additional functionality to querying and storing documents. In this example, we’ll demonstrate how to create a [unique index](http://docs.mongodb.org/manual/core/index-unique/) on a key that rejects documents whose value for that key already exists in the index.

First, create a new collection called “profiles” for users.

**>>>** db.createCollection**(**‘profiles’)

First, we’ll need to create the index:

**>>>** from pymongo import ASCENDING, DESCENDING

**>>>** result = db.profiles.create\_index([('user\_id',ASCENDING)], unique=**True**)

**>>>** sorted(list(db.profiles.index\_information()))

[u'\_id\_', u'user\_id\_1']

Notice that we have two indexes now: one is the index on \_id that MongoDB creates automatically, and the other is the index on user\_id we just created.

Now let’s set up some user profiles:

**>>>** user\_profiles = [

**...**  {'user\_id': 211, 'name': 'Luke'},

**...**  {'user\_id': 212, 'name': 'Ziltoid'}]

**>>>** result = db.profiles.insert\_many(user\_profiles)

The index prevents us from inserting a document whose user\_id is already in the collection:

**>>>** new\_profile = {'user\_id': 213, 'name': 'Drew'}

**>>>** duplicate\_profile = {'user\_id': 212, 'name': 'Tommy'}

**>>>** result = db.profiles.insert\_one(new\_profile) *# This is fine.*

**>>>** result = db.profiles.insert\_one(duplicate\_profile)

Traceback (most recent call last):

DuplicateKeyError: E11000 duplicate key error index: test\_database.profiles.$user\_id\_1 dup key: { : 212 }

### Delete Data

Delete a single document:

**>>>** for profile in db.profiles.find():

**...**  print(profile)

**>>>** result = db.profiles.delete\_one({“name”: “Drew”})

**>>>** print(result.deleted\_count)

**>>>** for profile in db.profiles.find():

**...**  print(profile)

Delete multiple documents:

**>>>** # First, add some documents to test many deletes.

**>>>** temp\_profiles = [

**...**  {'user\_id': 213, 'name': ‘Starlord’},

**...**  {'user\_id': 214, 'name': ‘Starlord’]

**>>>** result = db.profiles.insert\_many(temp\_profiles)

**>>>** for profile in db.profiles.find():

**...**  print(profile)

**>>>** db.profiles.delete\_many({“name”: “Starlord”})

**>>>** print(result.deleted\_count)

**>>>** for profile in db.profiles.find():

**...**  print(profile)

### Update Data

# Use the test\_collection

**>>>** testcoll = db.test\_collection

# Setup some data

**>>>** test\_data = [{‘x’:1},{‘x’:1},{‘x’:1},{‘x’:1},{‘x’:1}]

**>>>** result = testcoll.insert\_many(test\_data)

**>>>** print(result.inserted\_ids)

**>>>** for d in testcoll.find():

**...**  print(d)

# Update single document

**>>>** result = testcoll.update\_one({‘x’:1}, {‘$inc’:{‘x’:3}})

**>>>** print(result.matched\_count)

**>>>** print(result.modified\_count)

**>>>** for d in testcoll.find():

**...**  print(d)

# Update multiple documents

**>>>** result = testcoll.update\_many({‘x’:1}, {‘$inc’:{‘x’:3}})

**>>>** print(result.matched\_count)

**>>>** print(result.modified\_count)

**>>>** for d in testcoll.find():

**...**  print(d)

### Replace Data

**>>>** for d in testcoll.find():

**...**  print(d)

# Replace single document.

**>>>** result = testcoll.replace\_one({'x': 5}, {'y': 1})

**>>>** result.matched\_count

1

**>>>** result.modified\_count

1

**>>> for** doc **in** testcoll.find({}):

**...**  print(doc)

# Upsert: Update if exists. Insert if it doesn’t.

# Set 3rd param of replace\_one as True.

**>>>** result = testcoll.replace\_one({'x': 1}, {'x': 1})

**>>>** result.matched\_count

1

**>>>** result.modified\_count

1

**>>>** result.upserted\_id

ObjectId('54f11e5c8891e756a6e1abd4')

**>>> for** doc **in** testcoll.find({}):

**...**  print(doc)

### Limit and Projection

**>>> for** doc **in** testcoll.find():

**...**  print(doc)

# Limit the no. of results returned.

**>>> for** doc **in** testcoll.find(limit=2):

**...**  print(doc)

# Limit which columns are returned.

# This will print all columns except the “\_id” column.

**>>> for** doc **in** testcoll.find(projection={'\_id': False}):

**...**  print(doc)

# This will print only the “\_id” column.

**>>> for** doc **in** testcoll.find(projection={'\_id': True}):

**...**  print(doc)

# This will not print the “\_id” and name columns.

**>>> for** doc **in** testcoll.find(projection={'\_id': True}):

**...**  print(doc)

# This will not return only 1 document with the “\_id” and name columns.

**>>> for** doc **in** testcoll.find(projection={'\_id': True}, limit=1):

**...**  print(doc)

## Group By

from pymongo import MongoClient

# creation of MongoClient

client=MongoClient()

# Connect with the portnumber and host

client = MongoClient("mongodb://localhost:27017/")

# Access database

mydatabase = client['test']

# Access collection of the database

mycollection=mydatabase['writers']

# writer\_profiles = [

#     {"\_id":1, "user":"Ajay", "title":"Python", "comments":5},

#     {"\_id":2, "user":"John",  "title":"JavaScript", "comments":15},

#     {"\_id":3, "user":"Ajay",  "title":"C#", "comments":6},

#     {"\_id":4, "user":"John",  "title":"MongoDB", "comments":2},

#     {"\_id":5, "user":"Mary",  "title":"MongoDB", "comments":9}]

# mycollection.insert\_many(writer\_profiles)

# Group By Users.

print("No. of Articles Grouped by Writer...")

agg\_result= mycollection.aggregate(

    [{

    "$group" :

        {"\_id" : "$user",

         "num\_tutorial" : {"$sum" : 1}

         }}

    ])

for i in agg\_result:

    print(i)

# Group By Title.

print("No. of Articles Grouped by Title...")

agg\_result= mycollection.aggregate(

    [{

    "$group" :

        {"\_id" : "$title",

         "total" : {"$sum" : 1}

         }}

    ])

for i in agg\_result:

    print(i)

### Another Example of Group By

**db.teams.insertOne({team: "Mavs", position: "Guard", points: 31})**

**db.teams.insertOne({team: "Spurs", position: "Guard", points: 22})**

**db.teams.insertOne({team: "Rockets", position: "Center", points: 19})**

**db.teams.insertOne({team: "Warriors", position: "Forward", points: 26})**

**db.teams.insertOne({team: "Cavs", position: "Guard", points: 33})**

**Example 1: Group By and Count**

We can use the following code to group by the ‘position’ field and count the occurrences of each position.

**db.teams.aggregate([**

**{$group : {\_id:"$position", count:{$sum:1}}}**

**])**

This returns the following results:

**{ \_id: 'Forward', count: 1 }**

**{ \_id: 'Guard', count: 3 }**

**{ \_id: 'Center', count: 1 }**

**Example 2: Group By and Count (Then Sort)**

We can use the following code to count the occurrences of each position and automatically sort the results in **ascending order**:

**db.teams.aggregate([**

**{$group : {\_id:"$position", count:{$sum:1}}},**

**{$sort: {count:1}}**

**])**

This returns the following results:

**{ \_id: 'Forward', count: 1 }**

**{ \_id: 'Center', count: 1 }**

**{ \_id: 'Guard', count: 3 }**

We can use **-1** in the count argument to instead sort the results in **descending order**:

**db.teams.aggregate([**

**{$group : {\_id:"$position", count:{$sum:1}}},**

**{$sort: {count:-1}}**

**])**

This returns the following results:

**{ \_id: 'Guard', count: 3 }**

**{ \_id: 'Forward', count: 1 }**

**{ \_id: 'Center', count: 1 }**

## Group By - Having

#It's the equivalent of the following SQL instruction:

# SELECT COUNT(\*) FROM Table

# GROUP BY your\_field

# HAVING COUNT(\*) > N

query = db.collection.aggregate([

{

"$group": { "\_id": "$your\_field", #GROUP BY your\_field

"count": {"$sum":1} } #COUNT(\*)

},

{ "$match": { "count": { "$gt": N } } } #HAVING COUNT(\*) > N

])

# MongoDB Drivers – Scala

* <https://mongodb.github.io/mongo-scala-driver/2.8/getting-started/>
* <https://mongodb.github.io/mongo-scala-driver/2.8/getting-started/installation-guide/>
* <https://mongodb.github.io/mongo-scala-driver/2.8/getting-started/quick-tour/>
* <https://mongodb.github.io/mongo-scala-driver/2.8/getting-started/quick-tour-case-classes/>
* <https://mongodb.github.io/mongo-scala-driver/2.8/reference/connecting/>
* <https://mongodb.github.io/mongo-scala-driver/2.8/reference/crud/>
* <https://github.com/mongodb/mongo-scala-driver/blob/master/examples/src/test/scala/tour/QuickTour.scala>

## The Observable API

The Scala driver is free of dependencies on any third-party frameworks for asynchronous programming. To achieve this the Scala API makes use of an custom implementation of the Observer pattern which comprises three traits:

1. Observable
2. Observer
3. Subscription

An Observable is a provider of a potentially unbounded number of sequenced elements, published according to the demand received from it’s Subscription.

In response to a call to Observable.subscribe(Subscriber) the possible invocation sequences for methods on the Observer are given by the following protocol:

onSubscribe onNext\* (onError | onComplete)?

This means that onSubscribe is always signalled, followed by a possibly unbounded number of onNext signals (as requested by the Subscription) followed by an onError signal if there is a failure, or an onComplete signal when no more elements are available - as long as the Subscription is not cancelled.

### Observable

The [Observable](https://mongodb.github.io/mongo-scala-driver/2.8/scaladoc/org/mongodb/scala/Observable.html) is a trait wrapping the Java interface and where appropriate implementations of the trait extend it to make a fluent API.

**NOTE:**

All Observables returned from the API are cold, meaning that no I/O happens until they are subscribed to. As such an observer is guaranteed to see the whole sequence from the beginning. So just creating an Observable won’t cause any network IO, and it’s not until Subscriber.request() is called that the driver executes the operation.

Each Subscription to an Observable relates to a single MongoDB operation and its “Subscriber” will receive it’s own specific set of results.

## Helpers used in the Sample

For the Quick Tour we use custom implicit helpers defined in [Helpers.scala](https://github.com/mongodb/mongo-scala-driver/blob/2.8.x/examples/src/test/scala/tour/Helpers.scala" \t "_blank). These helpers get and print results and although this is an artificial scenario for asynchronous code we block on the results of one example before starting the next, so as to ensure the state of the database. The Helpers object provides the following methods:

* results(): Blocks until the Observable is completed and returns the collected results
* headResult(): Blocks until the first result of the Observable can be returned
* printResults(): Blocks until the Observable is completed, and prints out each result.
* printHeadResult(): Blocks until the first result of the Observable is available and then prints it.

**Source:**

* scalademos/src/main/scala/example/mongodb.scala
* scalademos/src/main/scala/example/Helpers.scala

**First add the following in build.sbt**:

libraryDependencies += "org.mongodb.scala" %% "mongo-scala-driver" % "2.8.0"

// mongodb.scala

package example;

// import scala.collection.JavaConverters.\_

// import org.mongodb.scala.connection.ClusterSettings

import org.mongodb.scala.\_

import example.Helpers.\_

object MongoDemo {

  def main(args: Array[String]) {

    println("Starting MongoDB - Scala Demo...")

    /\*\* \*\* differet ways to connec to MongoDB: // To directly connect to the

      \* default server localhost on port 27017 val mongoClient: MongoClient =

      \* MongoClient()

      \*

      \* // Use a Connection String val mongoClient: MongoClient =

      \* MongoClient("mongodb://localhost")

      \*

      \* // or provide custom MongoClientSettings val settings:

      \* MongoClientSettings = MongoClientSettings.builder()

      \* .applyToClusterSettings(b => b.hosts(List(new

      \* ServerAddress("localhost")).asJava)) .build() val mongoClient:

      \* MongoClient = MongoClient(settings)

      \*

      \* val database: MongoDatabase = mongoClient.getDatabase("mydb")

      \*

      \* val collection: MongoCollection[Document] =

      \* database.getCollection("test");

      \*/

    //

    val client: MongoClient = MongoClient()

    val database: MongoDatabase = client.getDatabase("test")

    // Get a Collection.

    val collection: MongoCollection[Document] = database.getCollection("test")

    // // insert a document

    // val document: Document = Document("\_id" -> 1, "x" -> 1)

    // val insertObservable: Observable[Completed] = collection.insertOne(document)

    // insertObservable.subscribe(new Observer[Completed] {

    //   override def onNext(result: Completed): Unit = println(s"onNext: $result")

    //   override def onError(e: Throwable): Unit = println(s"onError: $e")

    //   override def onComplete(): Unit = println("onComplete")

    // })

    // Insert a document.

    // Sample:

    // {

    //   "name" : "MongoDB",

    //   "type" : "database",

    //   "count" : 1,

    //   "info" : {

    //               x : 203,

    //               y : 102

    //             }

    // }

    val doc: Document = Document(

      "\_id" -> 0,

      "name" -> "MongoDB",

      "type" -> "database",

      "count" -> 1,

      "info" -> Document("x" -> 203, "y" -> 102)

    )

    //println(doc)

    println("Printing results...")

    collection.insertOne(doc).results()

    // In the API all methods returning a Observables are “cold” streams meaning

    // that nothing happens until they are Subscribed to.

    val observable: Observable[Completed] = collection.insertOne(doc)

    // Only when an Observable is subscribed to and data requested will the operation happen.

    // Explictly subscribe:

    observable.subscribe(new Observer[Completed] {

      override def onNext(result: Completed): Unit = println("Inserted")

      override def onError(e: Throwable): Unit = println("Failed")

      override def onComplete(): Unit = println("Completed")

    })

    // Once the document has been inserted the onNext method will be called

    // and it will print “Inserted!” followed by the onComplete method which will print “Completed”.

    // If there was an error for any reason the onError method would print “Failed”.

    // Add multiple documents.

    // The following example will add multiple documents of the form: { "i" : value }

    val documents = (1 to 100) map { i: Int => Document("i" -> i) }

    val insertObservable = collection.insertMany(documents)

    // As we haven’t subscribed yet no documents have been inserted, lets chain together two operations, inserting and counting.

    // Count Documents in A Collection.

    // Lets chain the two operations together using a for comprehension.

    // The following code should insert the documents then count the number of documents and print the results.

    val insertAndCount = for {

      insertResult <- insertObservable

      countResult <- collection.count()

    } yield countResult

    // println("total using head()...")

    // println(s"(1) total # of documents after inserting 100 small ones (should be 101):  ${insertAndCount.head()}")

    println("total using headResult()...")

    println(

      s"(2) total # of documents after inserting 100 small ones (should be 101):  ${insertAndCount.headResult()}"

    )

    // Find the 1st doc in the collection.

    println("First doc in collection with head()...")

    collection.find().first().head()

    println("First doc in collection with printHeadResult()...")

    collection.find().first().printHeadResult()

    // Find All Documents in a Collection.

    collection.find().printResults()

    // Get A Single Document with a Query Filter.

    import org.mongodb.scala.model.Filters.\_

    println("Get A Single Document with a Query Filter...")

    collection.find(equal("i", 71)).first().printHeadResult()

    // Get a Set of Documents with a Query.

    println("Get a Set of Documents with a Query...")

    collection.find(gt("i", 50)).printResults()

    // We could also get a range, say 50 < i <= 100:

    println("Get a Set of Documents with a Range Query...")

    collection.find(and(gt("i", 50), lte("i", 100))).printResults()

    // Sorting documents.

    import org.mongodb.scala.model.Sorts.\_

    println("Sorting in DESC order...")

    collection.find(exists("i")).sort(descending("i")).printResults()

    // Projecting fields.

    import org.mongodb.scala.model.Projections.\_

    println("Projecting...All fields....")

    collection.find().first().printHeadResult()

    println("Projecting...Exclude Id....")

    collection.find().projection(excludeId()).first().printHeadResult()

    // Aggregations.

    import org.mongodb.scala.model.Aggregates.\_

    println("Aggregations...multiply i by 10")

    collection

      .aggregate(

        Seq(

          filter(gt("i", 0)),

          project(Document("""{ITimes10: {$multiply: ["$i", 10]}}"""))

        )

      )

      .printResults()

    // Updating documents.

    import org.mongodb.scala.model.Updates.\_

    println("Updating doc...BEFORE...")

    collection.find(equal("i", 1)).printResults()

    println("Updating doc...")

    collection

      .updateOne(equal("i", 10), set("i", 110))

      .printHeadResult("Update Result: ")

    println("Updating doc...AFTER...")

    collection.find(equal("i", 10)).printResults()

    collection.find(equal("i", 110)).printResults()

    // Update ALL docs matching a filter.

    println("Update many docs...BEFORE...")

    collection.find(lt("i", 100)).printResults()

    // Here we increment the value of i by 100 where i is less than 100.

    collection

      .updateMany(lt("i", 100), inc("i", 100))

      .printHeadResult("Update Result: ")

    println("Update many docs...AFTER...")

    collection.find().printResults()

    // Deleting docs.

    println("Deleting docs...BEFORE...")

    collection.find(and(gt("i", 100), lte("i", 115))).printResults()

    collection.deleteOne(equal("i", 110)).printHeadResult("Delete Result: ")

    println("Deleting docs...AFTER...")

    collection.find(and(gt("i", 100), lte("i", 115))).printResults()

    // Delete multiple docs.

    println("Deleting multiple docs BEFORE...")

    collection.find(gt("i", 95)).printResults()

    collection.deleteMany(gte("i", 101)).printHeadResult("Delete Result: ")

    println("Deleting multiple docs AFTER...")

    collection.find(gt("i", 95)).printResults()

    // Bulk operations.

    import org.mongodb.scala.model.\_

    val writes: List[WriteModel[\_ <: Document]] = List(

      InsertOneModel(Document("\_id" -> 4)),

      InsertOneModel(Document("\_id" -> 5)),

      InsertOneModel(Document("\_id" -> 6)),

      UpdateOneModel(Document("\_id" -> 1), set("x", 2)),

      DeleteOneModel(Document("\_id" -> 2)),

      ReplaceOneModel(Document("\_id" -> 3), Document("\_id" -> 3, "x" -> 4))

    )

    // // 1. Ordered bulk operation - order is guaranteed

    // collection.bulkWrite(writes).printHeadResult("Bulk write results: ")

    // 2. Unordered bulk operation - no guarantee of order of operation

    collection

      .bulkWrite(writes, BulkWriteOptions().ordered(false))

      .printHeadResult("Bulk write results (unordered): ")

    println("After bulk Write...")

    collection.find().printResults()

  }

}

## Aggregation

<https://mongodb.github.io/mongo-java-driver/4.1/driver-scala/tutorials/aggregation/>

The [aggregation pipeline](http://docs.mongodb.org/manual/core/aggregation-pipeline) is a framework for data aggregation, modeled on the concept of data processing pipelines.

### Download and setup the Restaurants Dataset

The example below requires a restaurants collection in the test database. To create and populate the collection, follow the directions in [github](https://github.com/mongodb/docs-assets/tree/drivers) (<https://github.com/mongodb/docs-assets/tree/drivers>).

**Note**: This dataset is different from the Datatset used in the [Exercises](#_Structure_of_'restaurants') section.

The [restaurants.json](https://raw.githubusercontent.com/mongodb/docs-assets/drivers/restaurants.json) dataset contains 10 documents in the following structure:

{

"\_id" : <ObjectId>,

"name" : <string>,

"contact" : {

"phone" : <string>

"email" : <string>

"location" : [ <longitude>, <latitude> ]

},

"stars" : int,

"categories" : <array of strings>

"grades" : <array of integers>,

}

1. Download and save the restaurants.json dataset from here: <https://raw.githubusercontent.com/mongodb/docs-assets/drivers/restaurants.json>
2. In the terminal shell or command prompt, use mongoimport (or mongoimport.exe on Windows) to insert the documents. For example, the following mongoimport connects to a mongod instance running on localhost on port number 27017.

mongoimport --db test --collection restaurants --drop --file <path to saved file>

Now, include the following import statements in your Scala program **mongodb.scala**:

// mongodb.scala

**import** org.mongodb.scala.\_

**import** org.mongodb.scala.model.Aggregates.\_

**import** org.mongodb.scala.model.Accumulators.\_

**import** org.mongodb.scala.model.Filters.\_

**import** org.mongodb.scala.model.Projections.\_

### Connect to a MongoDB Deployment

Connect to a MongoDB deployment and declare and define a MongoDatabase and a MongoCollection instances.

For example, include the following code to connect to a standalone MongoDB deployment running on localhost on port 27017 and define database to refer to the test database and collection to refer to the restaurants collection.

**val** mongoClient: MongoClient = MongoClient()

**val** database: MongoDatabase = mongoClient.getDatabase(**"test"**)

**val** collection: MongoCollection[Document] = database.getCollection(**"restaurants"**)

### Perform Aggregation

To perform aggregation, pass a list of [aggregation stages](http://docs.mongodb.org/manual/meta/aggregation-quick-reference) to the [MongoCollection.aggregate()](https://mongodb.github.io/mongo-java-driver/4.1/apidocs/mongo-scala-driver/org/mongodb/scala/MongoCollection.html" \l "aggregate[C](pipeline:Seq[org.mongodb.scala.bson.conversions.Bson])(implicite:org.mongodb.scala.bson.DefaultHelper.DefaultsTo[C,TResult],implicitct:scala.reflect.ClassTag[C]):org.mongodb.scala.AggregateObservable[C]) method. The Scala driver provides the [Aggregates](https://mongodb.github.io/mongo-java-driver/4.1/apidocs/mongo-scala-driver/org/mongodb/scala/model/Aggregates$.html) helper class that contains builders for aggregation stages.

In the following example, the aggregation pipeline

* First uses a [$match](http://docs.mongodb.org/manual/reference/operator/aggregation/match/) stage to filter for documents whose categories array field contains the element Bakery. The example uses [Aggregates.filter](https://mongodb.github.io/mongo-java-driver/4.1/builders/aggregation/" \l "match) to build the $match stage.
* Then, uses a [$group](http://docs.mongodb.org/manual/reference/operator/aggregation/group/) stage to group the matching documents by the stars field, accumulating a count of documents for each distinct value of stars. The example uses [Aggregates.group](https://mongodb.github.io/mongo-java-driver/4.1/builders/aggregation/" \l "group) to build the $group stage and [Accumulators.sum](https://mongodb.github.io/mongo-java-driver/4.1/apidocs/mongo-scala-driver/org/mongodb/scala/model/Accumulators$.html" \l "sum[TExpression](fieldName:String,expression:TExpression):org.mongodb.scala.model.BsonField) to build the [accumulator expression](http://docs.mongodb.org/manual/reference/operator/aggregation/group/#accumulator-operator). For the [accumulator expressions](http://docs.mongodb.org/manual/reference/operator/aggregation-group/) for use within the [$group](http://docs.mongodb.org/manual/reference/operator/aggregation/group/) stage, the Scala driver provides [Accumulators](https://mongodb.github.io/mongo-java-driver/4.1/apidocs/mongodb-driver-core/com/mongodb/client/model/Accumulators.html) helper class.

collection.aggregate(Seq(

Aggregates.filter(Filters.equal(**"categories"**, **"Bakery"**)),

Aggregates.group(**"$stars"**, Accumulators.sum(**"count"**, 1))

)).printResults()

### Use Aggregation Expressions

For [$group accumulator expressions](http://docs.mongodb.org/manual/reference/operator/aggregation-group/), the Scala driver provides [Accumulators](https://mongodb.github.io/mongo-java-driver/4.1/apidocs/mongo-scala-driver/org/mongodb/scala/model/Accumulators$.html) helper class. For other [aggregation expressions](http://docs.mongodb.org/manual/meta/aggregation-quick-reference/#aggregation-expressions), manually build the expression Document.

In the following example, the aggregation pipeline uses a [$project](http://docs.mongodb.org/manual/reference/operator/aggregation/project/) stage to return only the name field and the calculated field firstCategory whose value is the first element in the categories array. The example uses [Aggregates.project](https://mongodb.github.io/mongo-java-driver/4.1/builders/aggregation/" \l "project) and various [Projections](https://mongodb.github.io/mongo-java-driver/4.1/apidocs/mongo-scala-driver/org/mongodb/scala/model/Projections$.html) methods to build the $project stage.

collection.aggregate(

Seq(

Aggregates.project(

Projections.fields(

Projections.excludeId(),

Projections.include(**"name"**),

Projections.computed(

**"firstCategory"**,

Document(**"$arrayElemAt"**-> Seq(**"$categories"**, 0))

)

)

)

)

).printResults()

# Exercises

<https://www.w3resource.com/mongodb-exercises/?passed=passed>

## **Structure of 'restaurants' collection:**

{

"address": {

"building": "1007",

"coord": [ -73.856077, 40.848447 ],

"street": "Morris Park Ave",

"zipcode": "10462"

},

"borough": "Bronx",

"cuisine": "Bakery",

"grades": [

{ "date": { "$date": 1393804800000 }, "grade": "A", "score": 2 },

{ "date": { "$date": 1378857600000 }, "grade": "A", "score": 6 },

{ "date": { "$date": 1358985600000 }, "grade": "A", "score": 10 },

{ "date": { "$date": 1322006400000 }, "grade": "A", "score": 9 },

{ "date": { "$date": 1299715200000 }, "grade": "B", "score": 14 }

],

"name": "Morris Park Bake Shop",

"restaurant\_id": "30075445"

}

[You may download the compressed file and uncompress it to find the collection used in our exercises. The collection comprises of 3772 documents.](https://www.w3resource.com/mongodb-exercises/restaurants.zip) Download from here: <https://www.w3resource.com/mongodb-exercises/restaurants.zip>

### Import data using mongoimport

"C:\Program Files\MongoDB\Server\mongodb-database-tools\bin\mongoimport" --file .\restaurants.json --db test --collection restaurants

### Read data from python

# read\_restaurant.py

import json

file = open('restaurants.json', 'r')

for oneRec in file:

    print (oneRec)

    z = json.loads(oneRec)

    # print(z)

    print(type(oneRec))

    print(type(z))

    print(z["address"])

    print(z["address"]["building"])

    print(z["address"]["street"])

    print(z["grades"][0]["date"])

    print(z["grades"][0]["date"]["$date"])

    for date in z["grades"]:

        print(date)

        ms = date["date"]["$date"]

        dt = datetime(1970, 1, 1) + timedelta(milliseconds=ms)

        print("dt:", dt)

**Convert milliseconds to date in the MongoDB shell**:

dt = new Date(1393804800000)

**Output**:

ISODate("2014-03-03T00:00:00Z")

## Tasks

1. Write a MongoDB query to display all the documents in the collection restaurants. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-1.php)

2. Write a MongoDB query to display the fields restaurant\_id, name, borough and cuisine for all the documents in the collection restaurant. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-2.php)

3. Write a MongoDB query to display the fields restaurant\_id, name, borough and cuisine, but exclude the field \_id for all the documents in the collection restaurant. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-3.php)

4. Write a MongoDB query to display the fields restaurant\_id, name, borough and zip code, but exclude the field \_id for all the documents in the collection restaurant. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-4.php)

5. Write a MongoDB query to display all the restaurant which is in the borough Bronx. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-5.php)

6. Write a MongoDB query to display the first 5 restaurant which is in the borough Bronx. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-6.php)

7.Write a MongoDB query to display the next 5 restaurants after skipping first 5 which are in the borough Bronx. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-7.php)

8. Write a MongoDB query to find the restaurants who achieved a score more than 90. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-8.php)

9. Write a MongoDB query to find the restaurants that achieved a score, more than 80 but less than 100. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-9.php)

10. Write a MongoDB query to find the restaurants which locate in latitude value less than -95.754168.[Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-10.php)

11. Write a MongoDB query to find the restaurants that do not prepare any cuisine of 'American' and their grade score more than 70 and latitude less than -65.754168. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-11.php)

12. Write a MongoDB query to find the restaurants which do not prepare any cuisine of 'American' and achieved a score more than 70 and located in the longitude less than -65.754168.  
Note : Do this query without using $and operator. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-12.php)

13. Write a MongoDB query to find the restaurants which do not prepare any cuisine of 'American ' and achieved a grade point 'A' not belongs to the borough Brooklyn. The document must be displayed according to the cuisine in descending order. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-13.php)

14. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which contain 'Wil' as first three letters for its name. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-14.php)

15. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which contain 'ces' as last three letters for its name. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-15.php)

16. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which contain 'Reg' as three letters somewhere in its name. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-16.php)

17. Write a MongoDB query to find the restaurants which belong to the borough Bronx and prepared either American or Chinese dish. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-17.php)

18. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which belong to the borough Staten Island or Queens or Bronxor Brooklyn. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-18.php)

19. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which are not belonging to the borough Staten Island or Queens or Bronxor Brooklyn. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-19.php)

20. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which achieved a score which is not more than 10. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-20.php)

21. Write a MongoDB query to find the restaurant Id, name, borough and cuisine for those restaurants which prepared dish except 'American' and 'Chinees' or restaurant's name begins with letter 'Wil'. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-21.php)

22. Write a MongoDB query to find the restaurant Id, name, and grades for those restaurants which achieved a grade of "A" and scored 11 on an ISODate "2014-08-11T00:00:00Z" among many of survey dates.. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-22.php)

23. Write a MongoDB query to find the restaurant Id, name and grades for those restaurants where the 2nd element of grades array contains a grade of "A" and score 9 on an ISODate "2014-08-11T00:00:00Z". [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-23.php)

24. Write a MongoDB query to find the restaurant Id, name, address and geographical location for those restaurants where 2nd element of coord array contains a value which is more than 42 and upto 52.. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-24.php)

25. Write a MongoDB query to arrange the name of the restaurants in ascending order along with all the columns. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-25.php)

26. Write a MongoDB query to arrange the name of the restaurants in descending along with all the columns. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-26.php)

27. Write a MongoDB query to arranged the name of the cuisine in ascending order and for that same cuisine borough should be in descending order. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-27.php)

28. Write a MongoDB query to know whether all the addresses contains the street or not. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-28.php)

29. Write a MongoDB query which will select all documents in the restaurants collection where the coord field value is Double. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-29.php)

30. Write a MongoDB query which will select the restaurant Id, name and grades for those restaurants which returns 0 as a remainder after dividing the score by 7. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-30.php)

31. Write a MongoDB query to find the restaurant name, borough, longitude and attitude and cuisine for those restaurants which contains 'mon' as three letters somewhere in its name. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-31.php)

32. Write a MongoDB query to find the restaurant name, borough, longitude and latitude and cuisine for those restaurants which contain 'Mad' as first three letters of its name. [Go to the editor](https://www.w3resource.com/mongodb-exercises/#PracticeOnline)  
[Click me to see the solution](https://www.w3resource.com/mongodb-exercises/mongodb-exercise-32.php)

## More Exercises

<https://mkyong.com/mongodb/mongodb-aggregate-and-group-example/>

# References

<https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/>

<https://docs.mongodb.com/manual/tutorial/install-mongodb-on-windows/>

<https://www.bmc.com/blogs/mongodb-overview-getting-started-with-mongodb/>

<https://www.bmc.com/blogs/mongodb-cheat-sheet/>

<https://www.bmc.com/blogs/mongodb-indexes/>

<https://www.tutorialspoint.com/mongodb/index.htm>

<https://pymongo.readthedocs.io/en/stable/api/pymongo/collection.html#pymongo.collection.Collection.update_one>

<https://realpython.com/introduction-to-mongodb-and-python/>

## Sample Datasets

<https://docs.atlas.mongodb.com/sample-data/available-sample-datasets/>