MODULE 7

MONGODB PROGRAMMING IN PYTHON

INTRODUCTION TO MONGODB

Terminology

> A *document* is the basic unit of data for MongoDB and is roughly equivalent to a row in a relational database management system (but much more expressive).

> A *collection* can be thought of as a table with a dynamic schema.

> A single instance of MongoDB can host multiple independent *databases*, each of which can have its own collections.

> Every document has a special key, "_id", that is unique within a collection.

> MongoDB comes with a simple but powerful JavaScript *shell*, which is useful for the administration of MongoDB instances and data manipulation.

3

Key-Value Documents

> An ordered set of keys with associated values.

> The representation of a document varies by programming language, but most languages have a data structure that is a natural fit, such as a map, hash, or dictionary.

> In JavaScript, for example, documents are represented as objects:

```
{"greeting" : "Hello, world!"}
```

> multiple key/value pairs:

```
{"greeting" : "Hello, world!",
 "foo" : 3}
```

4

Keys in Documents

> The keys in a document are strings.

> Any UTF-8 character is allowed in a key, with a few notable exceptions:

> Keys must not contain the character **\0** (the null character).

– This character is used to signify the end of a key.

> The **.** and **$** characters have some special properties and should be used only in certain circumstances.

– In general, they should be considered reserved, and drivers will complain if they are used inappropriately.

Type-sensitive and Case-sensitive

> MongoDB is type-sensitive and case-sensitive.

> For example, these documents are distinct:

```
{"foo" : 3}
{"foo" : "3"}
```

> as are as these:

```
{"foo" : 3}
{"Foo" : 3}
```

Duplicate Keys

> MongoDB cannot contain duplicate keys.

> For example, the following is not a legal document:

```
{
  "greeting" : "Hello, world!",
  "greeting" : "Hello, MongoDB!"
}
```

Ordered Key-Value Pairs

> Key/value pairs in documents are ordered:

```
{"x" : 1, "y" : 2}
```

> is not the same as

```
{"y" : 2, "x" : 1}
```

> Field order does not usually matter and you should not design your schema to depend on a certain ordering of fields (MongoDB may reorder them).

MongoDB Architecture

MongoDB Architecture Components

> **Client Application**
  – The application (front-end or back-end) that communicates with the MongoDB database via drivers or APIs.

> **MongoDB Server** (**mongod**):
  – The server component of MongoDB. This is where data storage, retrieval, and management happen.
  – The server manages databases, collections, and documents.

MongoDB Architecture Components

> **Sharding**

– MongoDB uses a sharded cluster to distribute data across multiple machines. It consists of shards, config servers, and a query router (mongos).

- **Shards**: These store data. Each shard holds a subset of the data.
- **Config Servers**: Manage the metadata and configuration for the cluster.
- **Query Router** (**mongos**): Routes client requests to the appropriate shard.

MongoDB Architecture Components

> **Replication** (Replica Set)

– A replica set is a group of MongoDB instances that maintain the same data, providing high availability and redundancy.

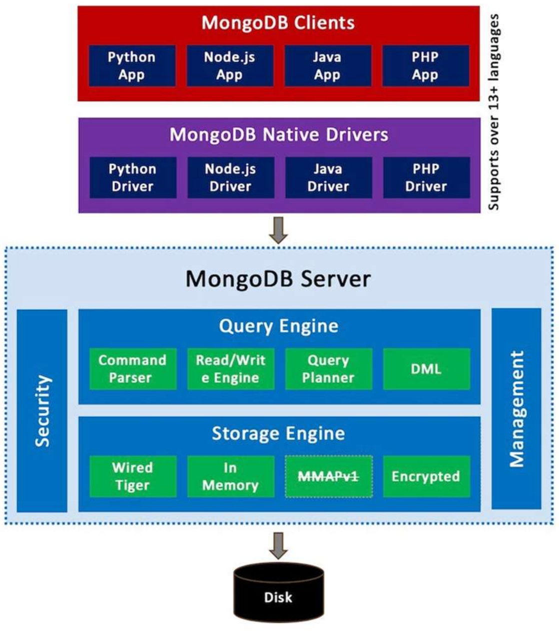
- **Primary Node**: Handles all write operations.
- **Secondary Nodes**: Replicate the primary node's data for backup and failover purposes.

Read/Write Operations

> **Writes**
  - Writes are directed to the primary node in a replica set.
  - In a sharded cluster, mongos ensures that the write goes to the correct shard.

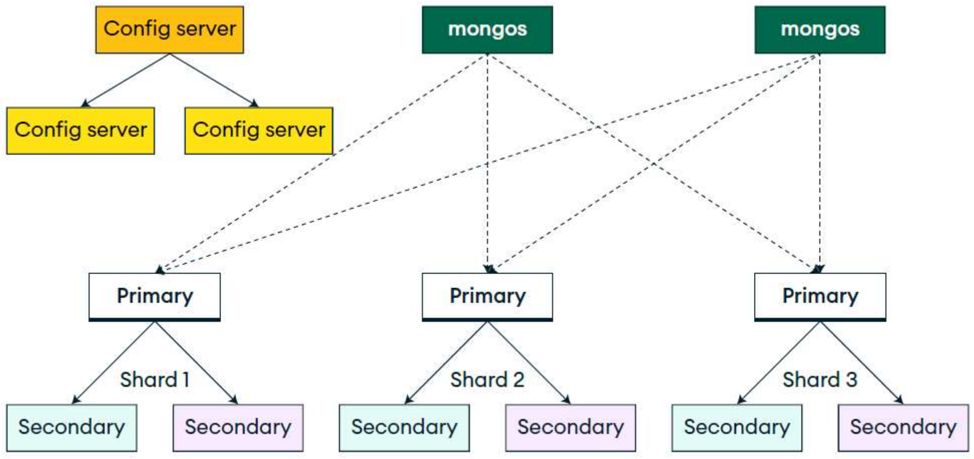
> **Reads**
  - Reads can be performed from the primary or secondary nodes, depending on the read preference settings.

Layered Architecture



A Sharded Cluster

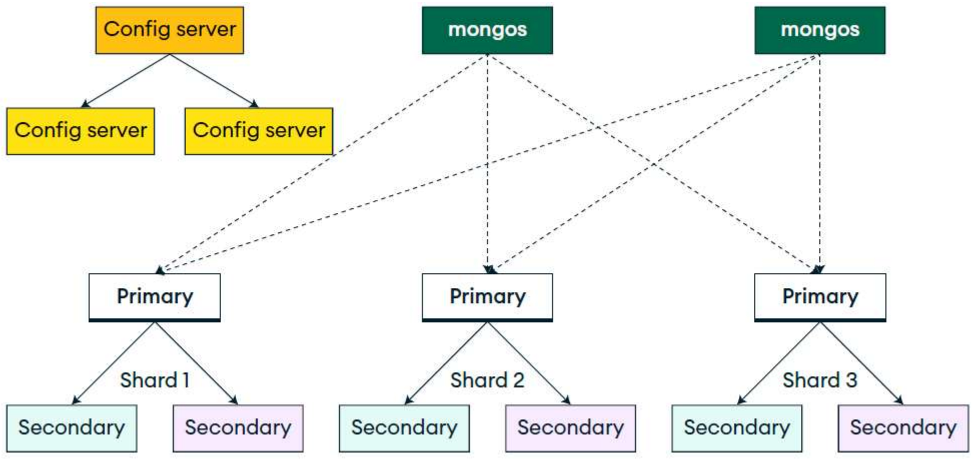
> In a MongoDB cluster, the architecture is designed to distribute data and operations across multiple machines to ensure high availability, horizontal scaling, and fault tolerance.



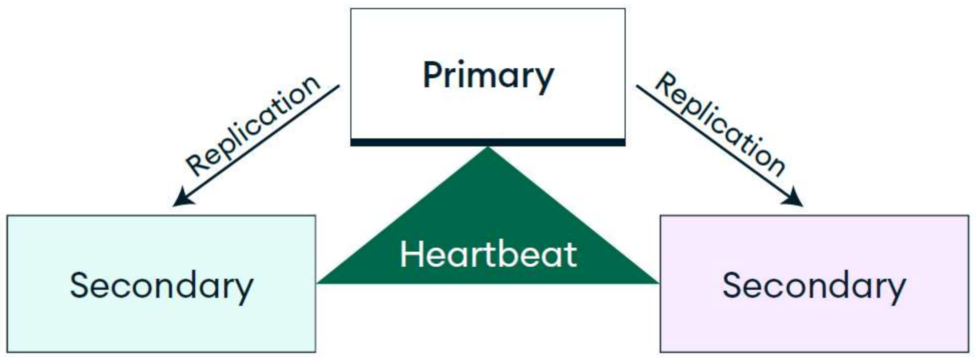
A Sharded Cluster

> Load Distribution: The client doesn’t interact directly with the shards.

> Instead, it connects to a mongos, which is responsible for routing the query to the correct shard(s) based on the shard key.

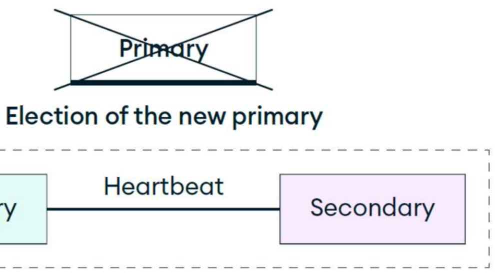


Replication



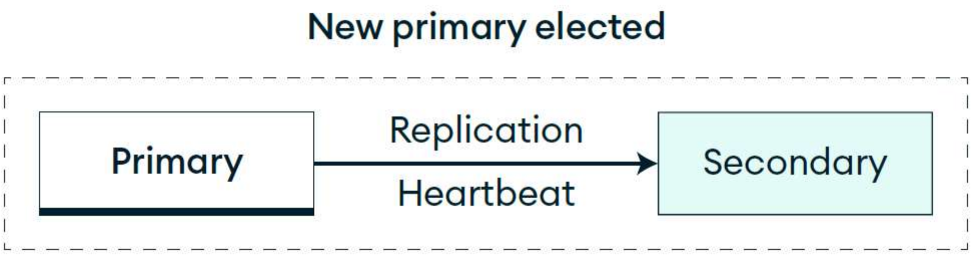
17

Replication



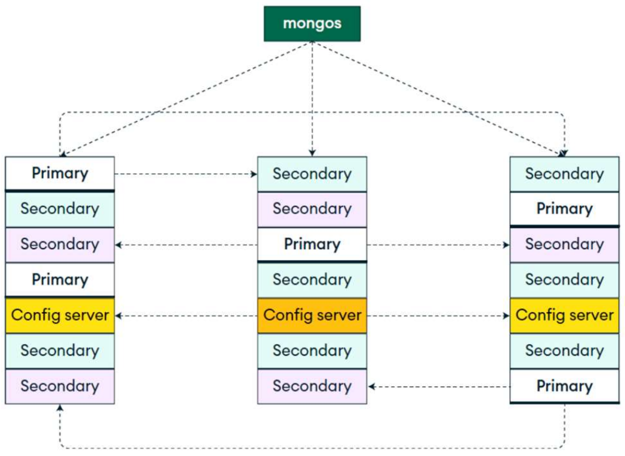
18

Replication



19

Microsharding Cluster



20

Document Structure and Embedding vs. Referencing

> Embed data within a single document when it is closely related and typically accessed together.
  – This improves read performance because you avoid joins and can fetch all necessary data in one query
> Use references (i.e., separate documents with foreign keys) when data is large or frequently updated independently.
  – This helps prevent large documents and facilitates scalability.

Sharding and Schema Design

> MongoDB uses sharding to distribute data across multiple machines in a cluster.
> When designing your schema:
  – **Sharding Key Selection**
    • Choose a sharding key that ensures an even distribution of data across shards.
    • It should prevent hotspots, where most of the operations end up going to a single shard.
    • Avoid monotonically increasing keys (e.g., timestamps or IDs) for this reason.

Sharding and Schema Design

> When designing your schema:

– **Cardinality**

- High-cardinality fields (i.e., fields with a large number of unique values) are better for sharding.
- Low-cardinality fields may not distribute data evenly.

Sharding and Schema Design

> When designing your schema:

– **Query Patterns**

- Ensure the sharding key aligns with your most common query patterns.
- For example, if you frequently query by **`product_id`**, using that field as the shard key could improve performance.

Schema Optimization for Read and Write

> **Write-heavy Workloads**

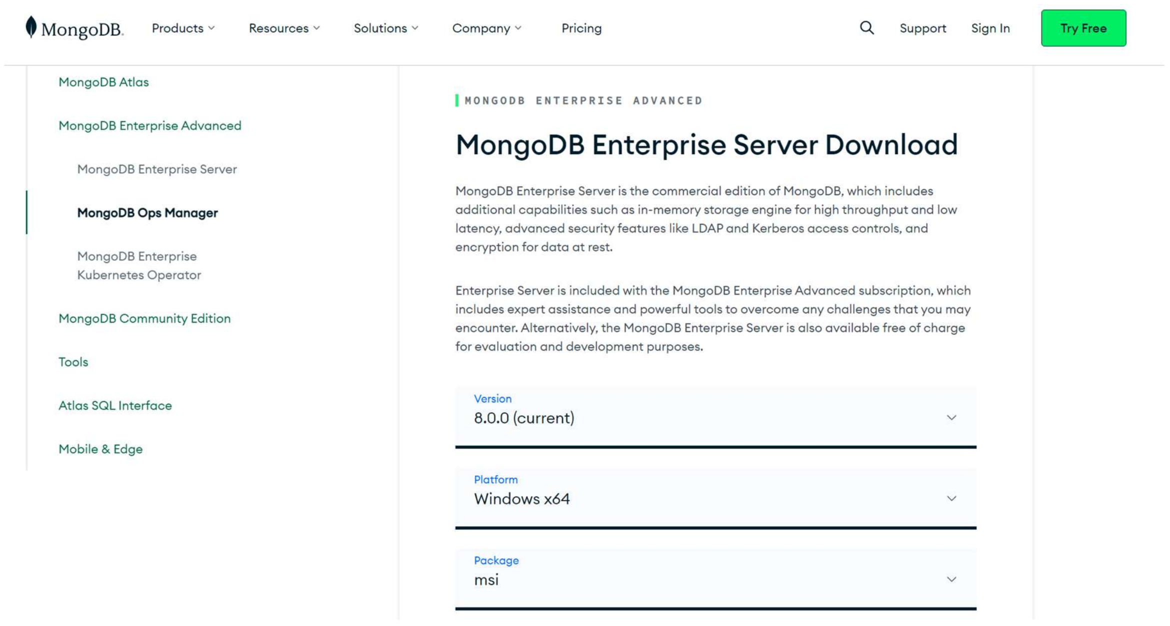
– If you have write-heavy operations, prefer denormalized data structures where data is embedded or duplicated to avoid joins and speed up writes.

> **Read-heavy Workloads**

– For read-heavy systems, optimize your schema by creating efficient indexes and using projections to limit the amount of data returned in queries.

INSTALLATION AND TOOLS

https://www.mongodb.com/try/download/ops-manager


MongoDB Products Resources Solutions Company Pricing Support Sign In Try Free

MongoDB Atlas
MongoDB Enterprise Advanced
MongoDB Enterprise Server
MongoDB Ops Manager
MongoDB Enterprise Kubernetes Operator
MongoDB Community Edition
Tools
Atlas SQL Interface
Mobile & Edge

MONGODB ENTERPRISE ADVANCED

MongoDB Enterprise Server Download

MongoDB Enterprise Server is the commercial edition of MongoDB, which includes additional capabilities such as in-memory storage engine for high throughput and low latency, advanced security features like LDAP and Kerberos access controls, and encryption for data at rest.

Enterprise Server is included with the MongoDB Enterprise Advanced subscription, which includes expert assistance and powerful tools to overcome any challenges that you may encounter. Alternatively, the MongoDB Enterprise Server is also available free of charge for evaluation and development purposes.

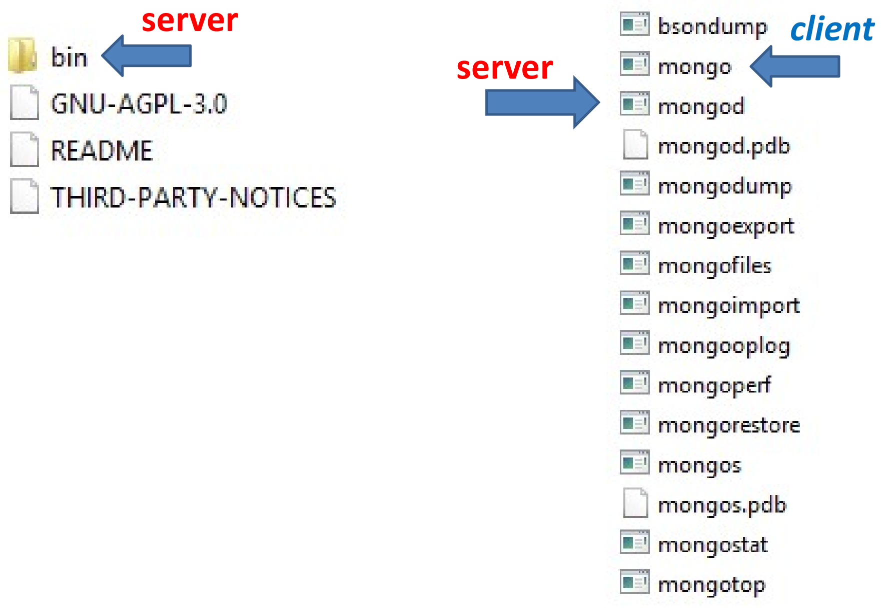
Version: 8.0.0 (current)

Platform: Windows x64

Package: msi

27

Installation Folder

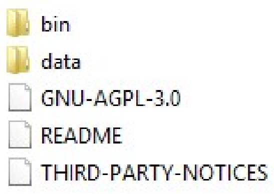


28

Data folder

> Create a data folder

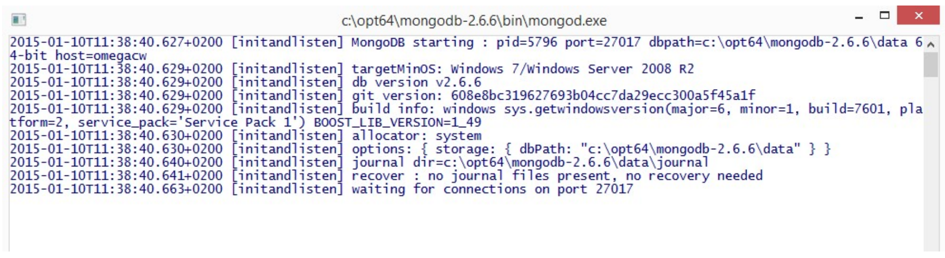
```
data\db
```


bin
data
GNU-AGPL-3.0
README
THIRD-PARTY-NOTICES

29

Running the server

```
set MONGO_HOME=c:\opt64\mongodb-7.0.14
set PATH=%MONGO_HOME%\bin;%PATH%
start mongod --dbpath=%MONGO_HOME%\data
```



```
2015-01-10T11:38:40.627+0200 [initandlisten] MongoDB starting : pid=5796 port=27017 dbpath=c:\opt64\mongodb-2.6.6\data 6
4-bit host=omegacw
2015-01-10T11:38:40.629+0200 [initandlisten] targetMinOS: Windows 7/Windows Server 2008 R2
2015-01-10T11:38:40.629+0200 [initandlisten] db version v2.6.6
2015-01-10T11:38:40.629+0200 [initandlisten] git version: 608e8bc319627693b04cc7da29ecc300a5f45a1f
2015-01-10T11:38:40.629+0200 [initandlisten] build info: windows sys.getwindowsversion(major=6, minor=1, build=7601, pla
tform=2, service_pack='Service Pack 1') BOOST_LIB_VERSION=1_49
2015-01-10T11:38:40.630+0200 [initandlisten] allocator: system
2015-01-10T11:38:40.630+0200 [initandlisten] options: { storage: { dbPath: "c:\opt64\mongodb-2.6.6\data" } }
2015-01-10T11:38:40.640+0200 [initandlisten] journal dir=c:\opt64\mongodb-2.6.6\data\journal
2015-01-10T11:38:40.641+0200 [initandlisten] recover : no journal files present, no recovery needed
2015-01-10T11:38:40.663+0200 [initandlisten] waiting for connections on port 27017
```

30

MongoDB Shell

> The MongoDB Shell (mongosh) is a fully functional JavaScript and Node.js-based REPL environment used to interact with MongoDB deployments.

> **`mongosh`** is a powerful and versatile command-line interface that serves as an interpreter.

> You can use **`mongosh`** to talk to the database directly and perform
  – CRUD operations : create, read, update, delete
  – administrative operations
  – aggregations
  – Indexing

Connecting to a MongoDB deployment using mongosh

> mongosh mongodb://127.0.0.1:27017/world

> mongosh mongodb://localhost:27017/world -f exercise03.js

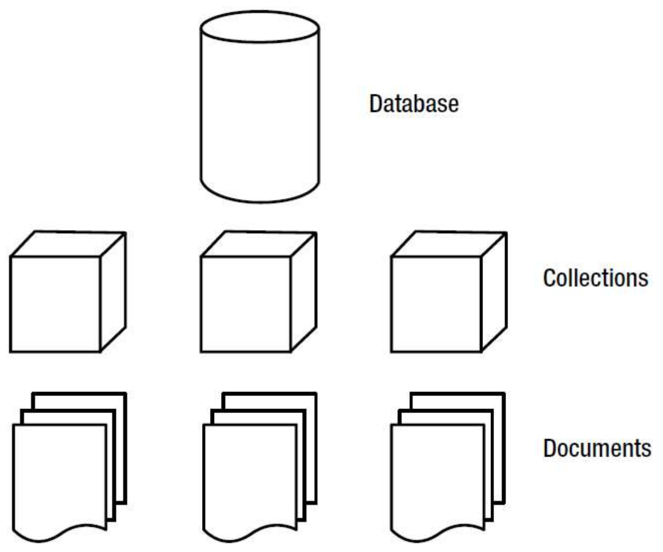
MongoDB Shell

```
> Math.sin(Math.PI/4)
0.7071067811865475
> Math.tan(Math.PI/4)
0.9999999999999999
> function fact(n){
... if (n==0 || n==1) return 1;
... return n * fact(n-1);
... }
> fact(5)
120
```

MongoDB Applications

bsondump	Reads contents of BSON-formatted rollback files
mongo	The database shell
mongod	The core database server
mongodump	Database backup utility
mongoexport	Export utility (JSON, CSV, TSV), not reliable for backup
mongofiles	Manipulates files in GridFS objects
mongoimport	Import utility (JSON, CSV, TSV), not reliable for recoveries
mongooplog	Pulls oplog entries from another mongod instance
mongoperf	Check disk I/O performance
mongorestore	Database backup restore utility
mongos	Mongodb sharding routerprocess
mongosniff	Sniff/traces MongoDB database activity in real time, Unix-like systems only
mongostat	Returns counters of database operation
mongotop	Tracks/reports MongoDB read/write activities
mongorestore	Restore/import utility

The MongoDB database model



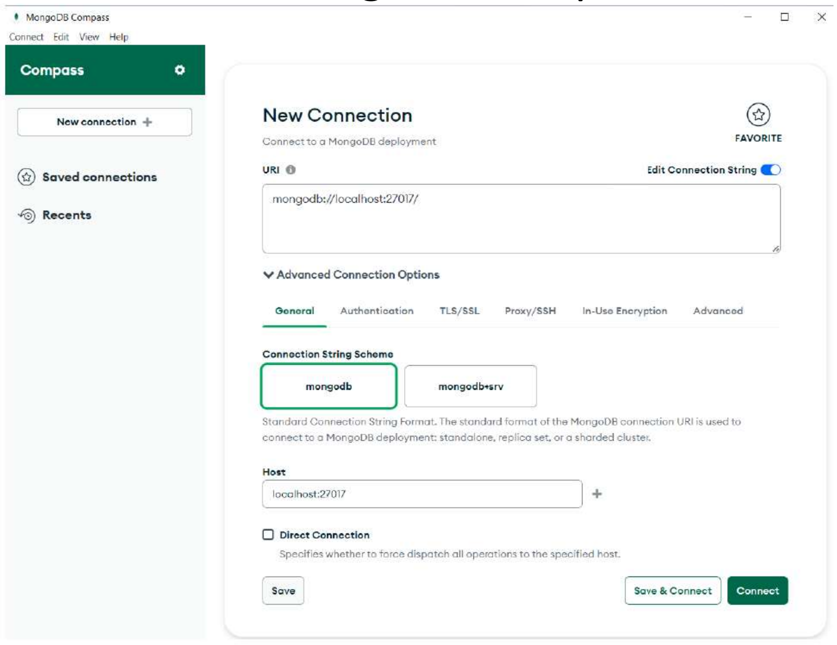
Basic Commands within the MongoDB Shell

Command	Function
`show dbs`	Shows the names of the available databases.
`show collections`	Shows the collections in the current database.
`show users`	Shows the users in the current database.
`use <db name>`	Sets the current database to <db name>.

MongoDB Compass

- MongoDB Compass is a powerful and intuitive GUI tool designed to simplify database management, query building, and data visualization for MongoDB.
- It offers a user-friendly alternative to the command-line interface by providing a visual representation of MongoDB data and collections.
- Compass is a free interactive tool.

MongoDB Compass



New Connection

Connect to a MongoDB deployment

URI — Edit Connection String

mongodb://localhost:27017/

Advanced Connection Options

Connection String Scheme: mongodb | mongodb+srv

Standard Connection String Format. The standard format of the MongoDB connection URI is used to connect to a MongoDB deployment: standalone, replica set, or a sharded cluster.

Host: localhost:27017

Direct Connection

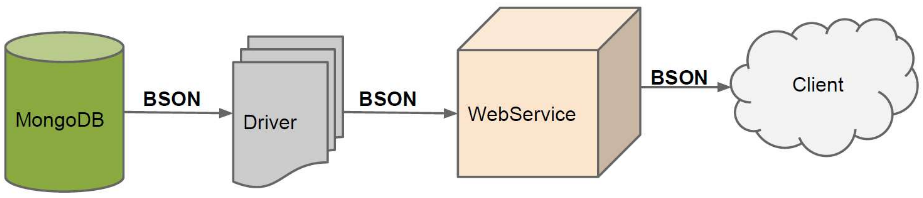
Specifies whether to force dispatch all operations to the specified host.

mongoDB

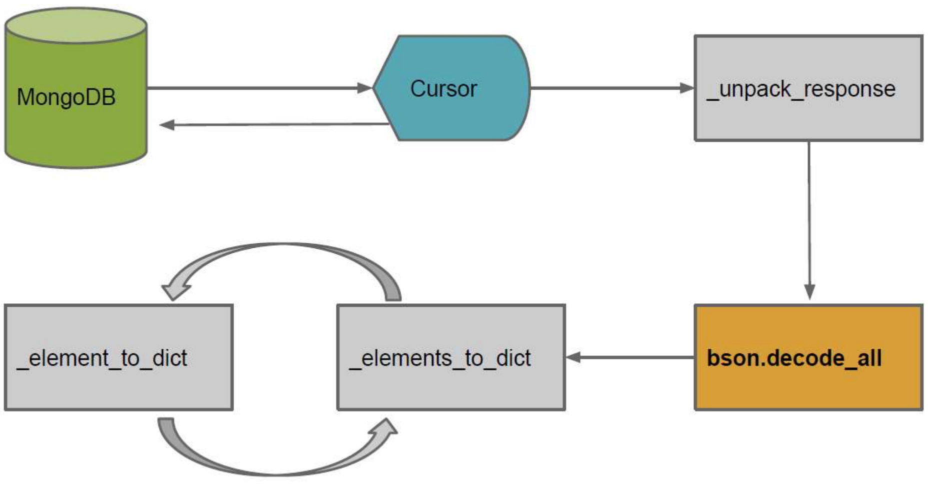
PyMongo

MongoDB Flow

- BSON is the serialization format used by mongodb to talk its clients
- Involves decoding BSON and then re-encoding JSON



The Python Driver



Exercises #1

```
from pymongo import MongoClient
import pprint

client = MongoClient("mongodb://localhost:27017")
db = client["world"]
countries1 = db.countries1

for country in countries1.find():
    pprint.pprint(country)
```

Exercises #2

```
from pymongo import MongoClient
import pprint

client = MongoClient("mongodb://localhost:27017")
db = client["world"]
countries1 = db.countries1

continents = countries1.distinct("continent")
pprint.pprint(continents)
```

Exercises #3

```
from pymongo import MongoClient
import pprint

client = MongoClient("mongodb://localhost:27017")
db = client["imdb"]
movies1 = db.movies1

aggregates = movies1.aggregate([ {"$project": {"genres" : 1} } ,
                                 {"$unwind" : "$genres"}, \
                             {"$group" : { "_id": "$genres.name", \
                                           "total": {"$sum": 1}}} \
                             ])
for agg in aggregates:
    pprint.pprint(agg)
```

Exercises #4

```
import pprint
import pymongo

from pymongo import MongoClient

client = MongoClient()
client = MongoClient('mongodb://localhost:27017')
db = client['imdb'];
movies1 = db.movies1

movies = movies1.find({"year" : { "$gte" : 1970, "$lt": 1980 }},\
          {"title": 1, "year":2 , "_id": 0})\
                  .sort([("year", pymongo.DESCENDING),\
                         ("title",pymongo.ASCENDING)])

for movie in movies:
    pprint.pprint(movie)
```

Example #5

```
from pymongo import MongoClient
import pprint
import re

client = MongoClient("mongodb://localhost:27017")
db = client["world"]
countries1 = db.countries1

rge_country_name = re.compile("^.{5}$", re.IGNORECASE)

countries = countries1.find({"name": rge_country_name},\
                             {"name": True, "_id": False})

for country in countries:
    pprint.pprint(country)
```

Example #6

```
import pprint
from pymongo import MongoClient

client = MongoClient()
client = MongoClient('mongodb://localhost:27017')
db = client['world'];
countries1 = db.countries1

countryCountsByContinent = countries1.group(\
    key={"continent": 1}, condition={}, initial={"count": 0},\
    reduce="function(c,h){ h.count = h.count +1;}")
pprint.pprint(countryCountsByContinent)
```

Example #7

```
import pprint
from pymongo import MongoClient

client = MongoClient()
client = MongoClient('mongodb://localhost:27017')
db = client['imdb'];
movies1 = db.movies1

movies = movies1.find( \
  { "$where": "this.directors.length > 1" } , \
  {"title": 1, "directors": 1, "_id":0 })

for movie in movies:
    pprint.pprint(movie)
```

Example #8

```
import pprint
from pymongo import MongoClient

client = MongoClient()
client = MongoClient('mongodb://localhost:27017')
db = client['world'];
countries1 = db.countries1

countriesByContinent = countries1.group(key={"continent": 1},\
                    condition={}, initial={"countries": []},\
           reduce="function(c,h){ h.countries.push(c.name)}")
pprint.pprint(countriesByContinent)
```

CRUD OPERATIONS

51

Creating a database

```
MongoDB shell version: 2.6.6
connecting to: test
> show dbs
admin  (empty)
local  0.078GB
> use world
switched to db world
> show dbs
admin  (empty)
local  0.078GB
> db
world
```

52

Inserting to a database

> Insert a new document into the "customers" collection with the following data:
  - _id: 101
  - name: "John Doe“
  - age: 30
  - city: "New York"

Inserting to a database

```
db.customers.insertOne({
   _id: 101,
   name: "John Doe",
   age: 30,
   city: "New York"
});
```

Inserting multiple documents to a database

> Insert multiple documents into the "products" collection with the following data:
  - _id: 101
  - name: "Product A“
  - price: 15
  - category: "Electronics“
  - _id: 102
  - name: "Product B“
  - price: 20
  - category: "Clothing"

Inserting multiple documents to a database

```
db.products.insertMany([
  { _id: 101, name: "Product A", price: 15, category: "Electronics" },
  { _id: 102, name: "Product B", price: 20, category: "Clothing" }
]);
```

Updating to a database

> Update the "age" field of the document with _id: 101 to 35

```
db.customers.updateOne(
  { _id: 101 },
  { $set: { age: 35 } }
);
```

Updating multiple documents

> Update the "price" field of all documents in the "products" collection where the category is "Electronics" to 20.

```
db.products.updateMany(
  { category: "Electronics" },
  { $set: { price: 20 } }
);
```

Find and replace in a document

> Find and replace the first document in the "customers" collection where the name is "John Doe" with the following data:
  - name: "Jane Smith"
  - city: "Los Angeles"

```
db.customers.findOneAndUpdate(
  { name: "John Doe" },
  { $set: { name: "Jane Smith", city: "Los Angeles" } }
);
```

Push a new item

> Push a new item into the "items" array of the document with _id: 1001 in the "orders" collection.

```
db.orders.updateOne(
  { _id: 1001 },
  { $push: {
     items: { product_id: 103, quantity: 3, price: 10 }
    }
  }
);
```

Delete

> Delete the document with _id: 101 from the "customers" collection.

```
db.customers.deleteOne({ _id: 101 });
```

Delete many documents

> Delete all documents in the "products" collection where the price is less than 10.

```
db.products.deleteMany({ price: { $lt: 10 } });
```

SCHEMAS AND DATA MODELING

Basic Data Types

> Documents in MongoDB can be thought of as “JSON-like” in that they are conceptually similar to objects in JavaScript.
> JSON is a simple representation of data
  – the specification lists only six data types.
> MongoDB adds support for a number of additional data types while keeping JSON’s essential key/value pair nature.

Basic Data Types

> null

– Null can be used to represent both a null value and a nonexistent field:

```
{"x" : null}
```

> boolean

– There is a boolean type, which can be used for the values true and false:

```
{"x" : true}
```

Basic Data Types

> number

– The shell defaults to using 64-bit floating point numbers. Thus, these numbers look “normal” in the shell:

```
{"x" : 3.14}
```

or:

```
{"x" : 3}
```

– For integers, use the **`NumberInt`** or **`NumberLong`** classes, which represent 4-byte or 8-byte signed integers.

```
{"x" : NumberInt("3")}
{"x" : NumberLong("3")}
```

Basic Data Types

- string
  - Any string of UTF-8 characters can be represented using the string type:

```
{"x" : "foobar"}
```

- date
  - Dates are stored as milliseconds since the epoch.
  - The time zone is not stored:

```
{"x" : new Date()}
```

- regular expression
  - Queries can use regular expressions using JavaScript’s regular expression syntax:

```
{"x" : /foobar/i}
```

Basic Data Types

- array
  - Sets or lists of values can be represented as arrays:

```
{"x" : ["a", "b", "c"]}
```

  - One of the great things about arrays in documents is that MongoDB “understands” their
  - structure and knows how to reach inside of arrays to perform operations on their contents.
  - This allows us to query on arrays and build indexes using their contents.

Basic Data Types

> embedded document

– Documents can contain entire documents embedded as values in a parent document:

```
{
   "name" : "John Doe",
   "address" : {
     "street" : "123 Park Street",
     "city" : "Anytown",
     "state" : "NY"
    }
}
```

Basic Data Types

> *binary data*

– Binary data is a string of arbitrary bytes.

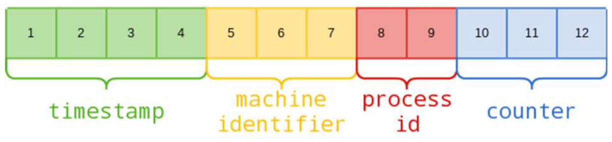
– It cannot be manipulated from the shell.

– Binary data is the only way to save non-UTF-8 strings to the database.

> *code*

– Queries and documents can also contain arbitrary JavaScript code:

```
{"x" : function() { /* ... */ }}
```



OBJECTIDS

71

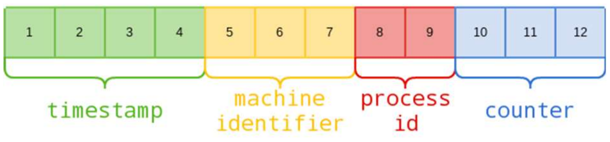
"_id"

> Immutable and unique
  – You cannot change after the document is created
  – Two different documents cannot have the same **_id** attribute value

72

"_id"

> **ObjectId** (12-Byte) (Big-Endian)
– Time the data created (4-Byte)
– Process Id (2-Byte)
– Machine Id (3-Byte)
– Incremental number (3-Byte)



ObjectId

```
> new ObjectId
ObjectId("5392be94a8b5a76657943b7a")
> new ObjectId
ObjectId("5392be95a8b5a76657943b7b")
> new ObjectId
ObjectId("5392be96a8b5a76657943b7c")
```

_id.getTimestamp()

```
> db.world.find()[0]
{
  "_id" : ObjectId("5392b9aba8b5a76657943b78"),
        "code" : "TUR",
        "name" : "Turkey",
        "population" : 77000000
}
> db.world.find()[0]._id
ObjectId("5392b9aba8b5a76657943b78")
> db.world.find()[0]._id.getTimestamp()
ISODate("2014-06-07T07:05:15Z")
```

_id.str

> The hexadecimal string representation of the object.

> **`toString()`**

Returns the JavaScript representation in the form of a string literal

> **`valueOf()`**

– Returns the representation of the object as a hexadecimal string.

– The returned string is the **`str`** attribute.

Relations

> You can choose either to embed information into a document or reference that information from another document

```
|_media
    |_cds
        |_id, artist, title, genre, releasedate
    |_ cd_tracklists
        |_cd_id, songtitle, length


|_media
    |_items
        |_<document>
```

77

Relations

> In the non-relational approach, the document might look like:

```
{
    "Type": "CD",
    "Artist": "Nirvana",
    "Title": "Nevermind",
    "Genre": "Grunge",
    "Releasedate": "1991.09.24",
    "Tracklist": [
        {
        "Track" : "1",
        "Title" : "Smells Like Teen Spirit",
        "Length" : "5:02"
        },
        {
        "Track" : "2",
        "Title" : "In Bloom",
        "Length" : "4:15"
        }
    ]
}
```

78

Relations

> When information is retrieved for a given CD, that information only needs to be loaded from one document into RAM, not from multiple documents.

> Remember that every reference requires another query in the database.

> The rule of thumb

– when using MongoDB is to embed data whenever you can.

– This approach is far more efficient and almost always viable.

79

Importing json

```
mongoimport --db world
       --collection countries1
       --type json
       --quiet
       --file countries_v1.json
       --jsonArray
```

80

db.cities2.stats()

```
{
        "ns" : "world.cities2",
        "count" : 4078,
        "size" : 462880,
        "avgObjSize" : 113,
        "storageSize" : 696320,
        "numExtents" : 4,
        "nindexes" : 1,
        "lastExtentSize" : 524288,
        "paddingFactor" : 1,
        "systemFlags" : 1,
        "userFlags" : 1,
        "totalIndexSize" : 122640,
        "indexSizes" : {
                "_id_" : 122640
        },
        "ok" : 1
}
```

db.cities2.storageSize()

```
> db.cities2.storageSize()
696320
> db.cities2.totalSize()
818960
> db.cities2.totalIndexSize()
122640
```

QUERYING DATA

Queries

> Find all documents in the "customers" collection.

```
db.customers.find();
```

Queries

```
> db.countries.find(
     { population : { $gt : 50000000 } },
     {name: true}
  ).count()
24
```

Queries

> Find all documents in the "products" collection where the price is greater than 20.

```
  db.products.find({ price: { $gt: 20 } });
```

Queries

> Find all documents in the "orders" collection where the order date is between January 1st and March 31st, 2023.

```
db.orders.find({
   orderDate: {
     $gte: ISODate("2023-01-01"),
     $lt: ISODate("2023-04-01")
   }
});
```

87

Queries

```
db.countries.find(
  {
    population : {
       $gt : 50000000 ,
       $lt : 100000000
    }
  },
  {name: true}
)
```

88

Queries

```
db.countries.find(
 {  'cities.name' :
    { $in : [ 'Istanbul', 'New York']}
 },
 { name: true }
)
{ "_id" : "TUR", "name" : "Turkey" }
{ "_id" : "USA", "name" : "United States" }
```

Distinct

```
> db.countries.distinct('continent')
[
        "North America",
        "Asia",
        "Africa",
        "Europe",
        "Oceania",
        "South America"
]
```

Regular Expression

```
db.countries1.find(
 { name : /^....$/},
 { name: 1 , _id: 0}
)
{ "name" : "Cuba" }
{ "name" : "Guam" }
{ "name" : "Iran" }
{ "name" : "Laos" }
{ "name" : "Mali" }
{ "name" : "Niue" }
{ "name" : "Oman" }
{ "name" : "Peru" }
{ "name" : "Chad" }
```

91

Sorting

```
db.countries.find(
   { continent : 'Asia’},
   { name: 1, population:1,_id :0}
).sort({population: -1})
 .limit(10)
{ "name" : "China", "population" : 1277558000 }
{ "name" : "India", "population" : 1013662000 }
{ "name" : "Indonesia", "population" : 212107000 }
{ "name" : "Pakistan", "population" : 156483000 }
{ "name" : "Bangladesh", "population" : 129155000 }
{ "name" : "Japan", "population" : 126714000 }
{ "name" : "Vietnam", "population" : 79832000 }
{ "name" : "Philippines", "population" : 75967000 }
{ "name" : "Iran", "population" : 67702000 }
{ "name" : "Turkey", "population" : 66591000 }
```

92

Pagination

```
db.countries.find(
{ continent : 'Asia'},
{ name: 1, population:1,_id :0})
.sort({population: -1})
.skip(10)
.limit(10)
{ "name" : "Thailand", "population" : 61399000 }
{ "name" : "South Korea", "population" : 46844000 }
{ "name" : "Myanmar", "population" : 45611000 }
{ "name" : "Uzbekistan", "population" : 24318000 }
{ "name" : "North Korea", "population" : 24039000 }
{ "name" : "Nepal", "population" : 23930000 }
{ "name" : "Afghanistan", "population" : 22720000 }
{ "name" : "Taiwan", "population" : 22256000 } ...
```

93

AGGREGATION PIPELINES

94

MongoDB aggregation framework

> The MongoDB aggregation framework allows you to fine tune and process complex data on the server, drastically reducing the amount of data that is transferred to the application for further processing.

> The aggregation framework is an incredibly powerful data processing workhorse that enables you to

– Handle custom data presentations through views

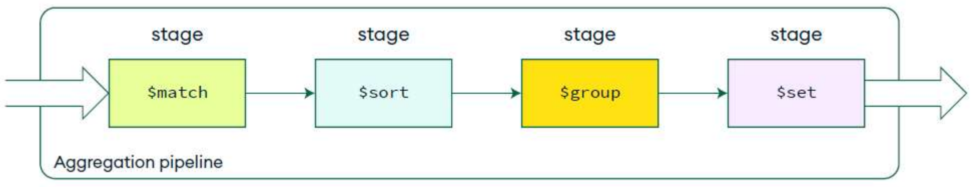
– Join data from different collections

– Perform data science tasks, such as data wrangling and analysis

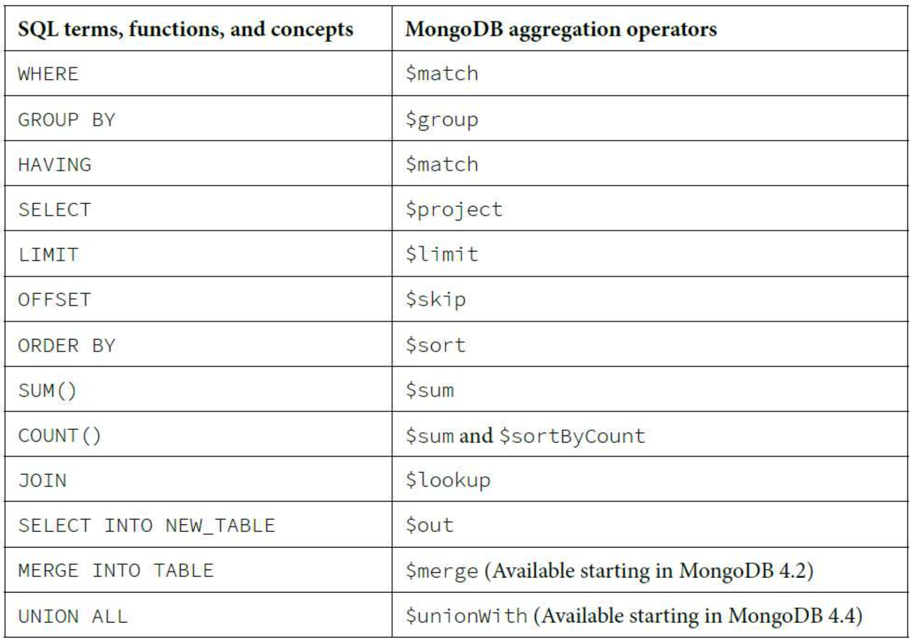
– Handle big data

– Run real-time analytics, monitor data, and create dashboards

The aggregation pipeline structure



SQL and MongoDB terminology and concepts



SQL terms, functions, and concepts	MongoDB aggregation operators
`WHERE`	`$match`
`GROUP BY`	`$group`
`HAVING`	`$match`
`SELECT`	`$project`
`LIMIT`	`$limit`
`OFFSET`	`$skip`
`ORDER BY`	`$sort`
`SUM()`	`$sum`
`COUNT()`	`$sum` and `$sortByCount`
`JOIN`	`$lookup`
`SELECT INTO NEW_TABLE`	`$out`
`MERGE INTO TABLE`	`$merge` (Available starting in MongoDB 4.2)
`UNION ALL`	`$unionWith` (Available starting in MongoDB 4.4)

Aggregation API

> From a strictly formal standpoint, the aggregation framework is a Turing-complete, proper domain-specific language (DSL), implying its capability to tackle any business problem.

> With over 150 operators and expressions, the MongoDB ecosystem offers a vast range of possibilities, enabling you to perform incredibly complex and useful data operations through relatively simple steps.

> All MongoDB language-specific drivers (JavaScript, Python, etc.) fully support the aggregation framework, making it convenient for you to write aggregations in any language.

Exercise 1: Basic Aggregation

> Using the orders collection below, write an aggregation query to calculate the total quantity ordered for each product.

```
[
  { "_id": 1, "product": "Pen", "quantity": 10 },
  { "_id": 2, "product": "Notebook", "quantity": 5 },
  { "_id": 3, "product": "Pen", "quantity": 3 },
  { "_id": 4, "product": "Notebook", "quantity": 8 }
]
```

Answer

```
db.orders.aggregate([
  {
    $group: {
      _id: "$product",
      totalQuantity: { $sum: "$quantity" }
    }
  }
])
```

Exercise 2: Filtering Data with `$match`

> From the sales collection, write an aggregation query to find the total sales only for products sold in 2023.

```
[
  { "_id": 1, "product": "Laptop", "amount": 1000, "year": 2022 },
  { "_id": 2, "product": "Phone", "amount": 600, "year": 2023 },
  { "_id": 3, "product": "Laptop", "amount": 1100, "year": 2023 },
  { "_id": 4, "product": "Tablet", "amount": 300, "year": 2023 }
]
```

Answer

```
db.sales.aggregate([
  { $match: { year: 2023 } },
  {
    $group: {
      _id: "$product",
      totalSales: { $sum: "$amount" }
    }
  }
])
```

Exercise 3: Using $project for Field Manipulation

> Given the **employees** collection below, write an aggregation query to calculate the total salary of each employee by multiplying the **baseSalary** and **monthsWorked** fields.

```
[
  { "_id": 1, "name": "Alice", "baseSalary": 3000, "monthsWorked": 12 },
  { "_id": 2, "name": "Bob", "baseSalary": 2500, "monthsWorked": 10 },
  { "_id": 3, "name": "Charlie", "baseSalary": 3500, "monthsWorked": 12 }
]
```

Answer

```
db.employees.aggregate([
  {
    $project: {
      name: 1,
      totalSalary: {
         $multiply: ["$baseSalary", "$monthsWorked"]
      }
    }
  }
])
```

Exercise 4: Using $unwind

> Using the **`students`** collection, write an aggregation query to list each student along with each subject they are taking, using the **`$unwind`** operator.

```
[
 { "_id": 1, "name": "John", "subjects": ["Math", "Physics"] },
 { "_id": 2, "name": "Jane", "subjects": ["Chemistry", "Biology"] }
]
```

Answer

```
db.students.aggregate(
 [
  {
    $unwind: "$subjects"
  }
 ]
)
```

Exercise 5: Sorting Results

> Using the products collection, write an aggregation query to find the total quantity sold per product and sort the results by total quantity in descending order.

```
[
  { "_id": 1, "product": "Pen", "quantity": 10 },
  { "_id": 2, "product": "Notebook", "quantity": 5 },
  { "_id": 3, "product": "Pen", "quantity": 3 },
  { "_id": 4, "product": "Notebook", "quantity": 8 }
]
```

Answer

```
db.products.aggregate([
  {
    $group: {
      _id: "$product",
      totalQuantity: { $sum: "$quantity" }
    }
  },
  { $sort: { totalQuantity: -1 } }
])
```

Exercise 6: Joining Collections with $lookup

> Given the **orders** and **customers** collections, write an aggregation query to join the two collections and display each order with the customer details.

> **orders** collection:

```
[
  { "_id": 1, "customerId": 101, "product": "Laptop", "quantity": 2 },
  { "_id": 2, "customerId": 102, "product": "Phone", "quantity": 1 }
]
```

> **customers** collection:

```
[
  { "_id": 101, "name": "John Doe" },
  { "_id": 102, "name": "Jane Smith" }
]
```

Answer

```
db.orders.aggregate([
  {
    $lookup: {
      from: "customers",
      localField: "customerId",
      foreignField: "_id",
      as: "customerDetails"
    }
  }
])
```

Exercise 7: Using `$facet` for Multiple Aggregations

> Write an aggregation query on the **`products`** collection that performs the following in parallel:
  – Find the total number of documents.
  – Calculate the average quantity sold.

```
[
  { "_id": 1, "product": "Pen", "quantity": 10 },
  { "_id": 2, "product": "Notebook", "quantity": 5 },
  { "_id": 3, "product": "Pen", "quantity": 3 },
  { "_id": 4, "product": "Notebook", "quantity": 8 }
]
```

Answer

```
db.products.aggregate( [ {
    $facet: {
      totalDocuments: [{ $count: "count" }],
      averageQuantity: [ {
          $group: { _id: null,
                 avgQuantity: { $avg: "$quantity" }
          }
       }]
    }
  }
])
```

Exercise 8: Bucketizing Data with $bucket

> Write an aggregation query to group students into three categories based on their score: 0-50 (Low), 51-80 (Medium), and 81-100 (High).

```
[
  { "_id": 1, "name": "Alice", "score": 45 },
  { "_id": 2, "name": "Bob", "score": 78 },
  { "_id": 3, "name": "Charlie", "score": 89 }
]
```

Answer

```
db.students.aggregate([{
    $bucket: {
      groupBy: "$score",
      boundaries: [0, 51, 81, 101],
      default: "Unknown",
      output: {
        count: { $sum: 1 }
      }
    }
  }
])
```

Exercise 9

> Find the top 5 customers who have spent the most.

Answer

```
db.orders.aggregate([
   { $unwind: "$items" },
   { $group: {
       _id: "$customer_id",
       totalSpent: {
         $sum: { $multiply: [ "$items.price", "$items.quantity" ] } }
       }
   },
   { $sort: { totalSpent: -1 } },
   { $limit: 5 }
])
```

Exercise 10

> Find the most popular product category based on the total quantity sold.

Answer

```
db.orders.aggregate([
   { $unwind: "$items" },
   {
     $group: {
        _id: "$items.category",
        totalQuantity: { $sum: "$items.quantity" }
     }
   },
   { $sort: { totalQuantity: -1 } },
   { $limit: 1 }
])
```

Exercise 11

> Calculate the average price of products in each category.

Answer

```
db.products.aggregate([
    { $group: { _id: "$category", avgPrice: { $avg: "$price" } } }
])
```

Exercise 12

> Find customers who have placed an order in the last 30 days.

Answer

```
db.orders.aggregate([
  {
    $match: {
      orderDate:
      {$gte: ISODate(new Date().getTime() - 30*24*60*60*1000) }
    }
   },
   {
     $group: { _id: "$customer_id" }
   }
])
```

Exercise 13

> Find the total revenue generated by each salesperson in the month of April 2023.

Answer

```
db.orders.aggregate([
   { $match: {
       orderDate: {
         $gte: ISODate("2023-04-01"),$lt: ISODate("2023-05-01") }
      }
   },
   { $unwind: "$items" },
   { $group: { _id: "$salesperson", totalRevenue: {
      $sum: { $multiply: ["$items.price", "$items.quantity"] }}}
   }
])
```

Exercise 14

> Find the number of orders placed by each customer who has spent more than $1000.

Answer

```
db.orders.aggregate([
  { $unwind: "$items" },
  {
    $group: {
      _id: "$customer_id",
      totalSpent: { $sum: { $multiply: ["$items.price", "$items.quantity"] } } }
  },
  { $match: { totalSpent: { $gt: 1000 } } },
  { $lookup: { from: "orders",
               localField: "_id",
               foreignField: "customer_id", as: "customerOrders" }
  },
  { $project: { _id: 1, totalOrders: { $size: "$customerOrders" } } }
])
```

Exercise 15

> Find the top 3 product categories based on the total revenue generated.

Answer

```
db.orders.aggregate([  { $unwind: "$items" },  { $group: { _id: "$items.category",
totalRevenue: { $sum: { $multiply: ["$items.price", "$items.quantity"] } } } },  {
$sort: { totalRevenue: -1 } },  { $limit: 3 }])
```

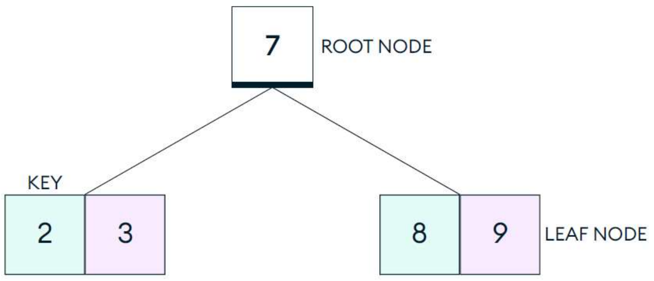
INDEXING

MODULE 8

129

Introduction to indexes

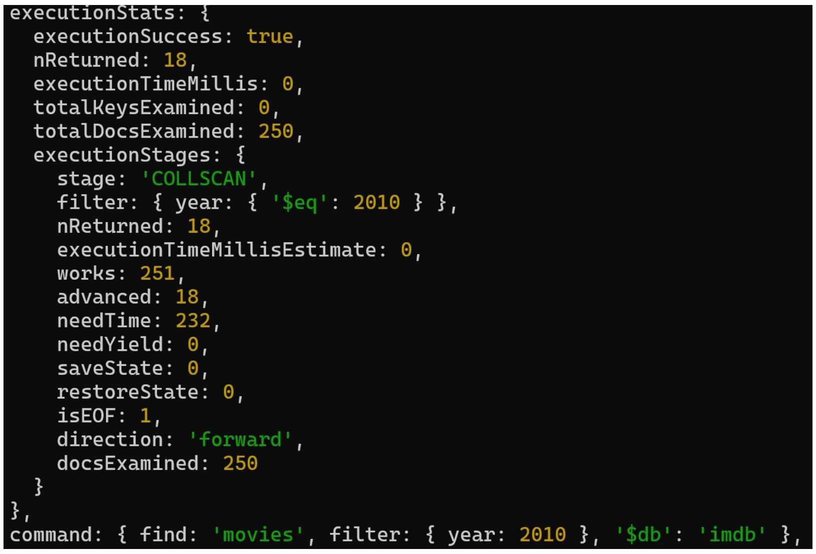
> An index is a special data structure that enables faster access to data in a collection
> MongoDB indexes use a data structure known as B-tree
  – A self-balancing tree data structure that maintains sorted data, and allows sequential access, searches, insertions, and deletions in logarithmic time.
> The B-tree structure is used to store index values and reduce the number of comparisons needed when performing a search.



130

Creating an index

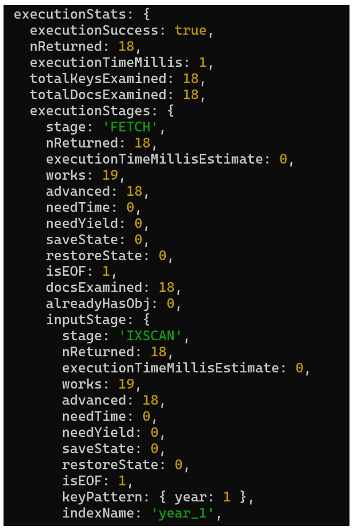
```
db.movies.explain("executionStats").find({"year": 2010});
```



```
executionStats: {
  executionSuccess: true,
  nReturned: 18,
  executionTimeMillis: 0,
  totalKeysExamined: 0,
  totalDocsExamined: 250,
  executionStages: {
    stage: 'COLLSCAN',
    filter: { year: { '$eq': 2010 } },
    nReturned: 18,
    executionTimeMillisEstimate: 0,
    works: 251,
    advanced: 18,
    needTime: 232,
    needYield: 0,
    saveState: 0,
    restoreState: 0,
    isEOF: 1,
    direction: 'forward',
    docsExamined: 250
  }
},
command: { find: 'movies', filter: { year: 2010 }, '$db': 'imdb' },
```

Creating an index

```
db.movies.ensureIndex({ year: 1})
db.movies.explain("executionStats").find({"year": 2010});
```



```
executionStats: {
  executionSuccess: true,
  nReturned: 18,
  executionTimeMillis: 1,
  totalKeysExamined: 18,
  totalDocsExamined: 18,
  executionStages: {
    stage: 'FETCH',
    nReturned: 18,
    executionTimeMillisEstimate: 0,
    works: 19,
    advanced: 18,
    needTime: 0,
    needYield: 0,
    saveState: 0,
    restoreState: 0,
    isEOF: 1,
    docsExamined: 18,
    alreadyHasObj: 0,
    inputStage: {
      stage: 'IXSCAN',
      nReturned: 18,
      executionTimeMillisEstimate: 0,
      works: 19,
      advanced: 18,
      needTime: 0,
      needYield: 0,
      saveState: 0,
      restoreState: 0,
      isEOF: 1,
      keyPattern: { year: 1 },
      indexName: 'year_1',
```

Indexes

> **Compound Indexes**
  – Use compound indexes (indexes on multiple fields) when your queries involve filtering by multiple fields.
  – This ensures the query can utilize the index for faster retrieval.

> **Sparse Indexes**
  – For fields that may be missing in many documents, use sparse indexes.
  – This prevents MongoDB from indexing documents that lack the field, reducing storage space.

> **Covered Queries**
  – Design your schema and indexes to make use of "covered queries," where MongoDB can answer the query entirely using the index without needing to examine the actual document.