Class 6: Aggregation Operators
Experiment 5

Introduction

● Aggregation is a way of processing a large number of documents

In a collection by means of passing them through different ways.

● For example min , max , avg etc

Syntax

db.collection.aggregate(<AGGREGATE OPERATION>

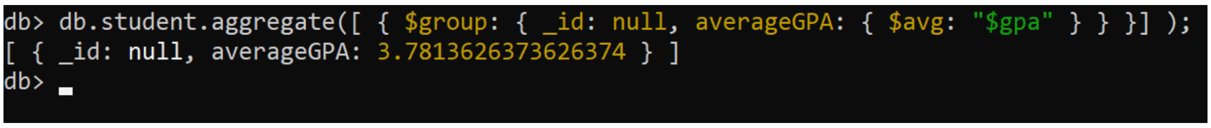
Average GPA of All Students

To find average gpa of all the students

Example:

```
db.student.aggregate([
   { $group: {_id : null , averageGPA : {$avg: “$gpa”}}}
  ]);
```

Output:



```
db> db.student.aggregate([ { $group: { _id: null, averageGPA: { $avg: "$gpa" } } }] );
[ { _id: null, averageGPA: 3.7813626373626374 } ]
db> 
```

Explanation:

● $group: Groups all documents together.

○ _id: null: Sets the group identifier to null .

○ averageGPA: Calculates the average value of the "gpa" using the $avg operator.

Minimum and Maximum Age:

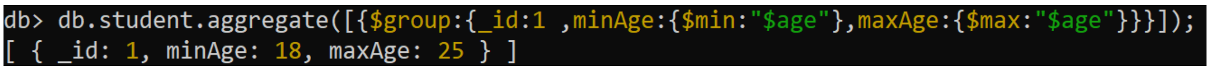
Example:

db.student.aggregate([

{ $group: {_id:1 , minAge :{$min : “$age”} , maxAge :

{$max: “$age”}}}]);

Output :



```
db> db.student.aggregate([{$group:{_id:1 ,minAge:{$min:"$age"},maxAge:{$max:"$age"}}}]);
[ { _id: 1, minAge: 18, maxAge: 25 } ]
```

Explanation:

● Similar to the previous example, it uses $group to group all documents.

● minAge: Uses the $min operator to find the minimum value in the "age" field.

● maxAge: Uses the $max operator to find the maximum value in the "age" field.

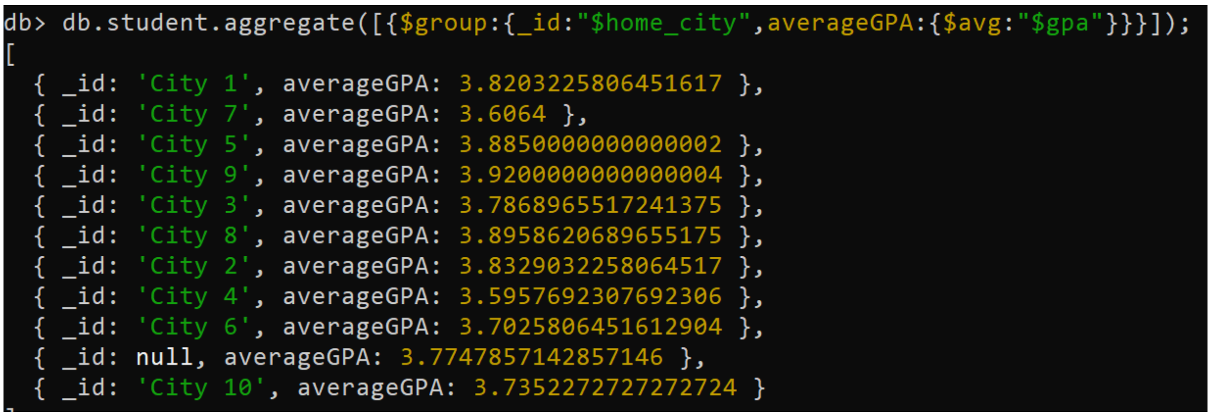
To calculate Average GPA for all home cities

Example:

db.student.aggregate([

{$group:{_id: “$home_city” , averageGPA :{$avg : $gpa } } }]);

Output:



```
db> db.student.aggregate([{$group:{_id:"$home_city",averageGPA:{$avg:"$gpa"}}}]);
[
  { _id: 'City 1', averageGPA: 3.8203225806451617 },
  { _id: 'City 7', averageGPA: 3.6064 },
  { _id: 'City 5', averageGPA: 3.8850000000000002 },
  { _id: 'City 9', averageGPA: 3.9200000000000004 },
  { _id: 'City 3', averageGPA: 3.7868965517241375 },
  { _id: 'City 8', averageGPA: 3.8958620689655175 },
  { _id: 'City 2', averageGPA: 3.8329032258064517 },
  { _id: 'City 4', averageGPA: 3.5957692307692306 },
  { _id: 'City 6', averageGPA: 3.7025806451612904 },
  { _id: null, averageGPA: 3.7747857142857146 },
  { _id: 'City 10', averageGPA: 3.7352272727272724 }
```

Collect Unique Courses Offered (Using $addToSet):

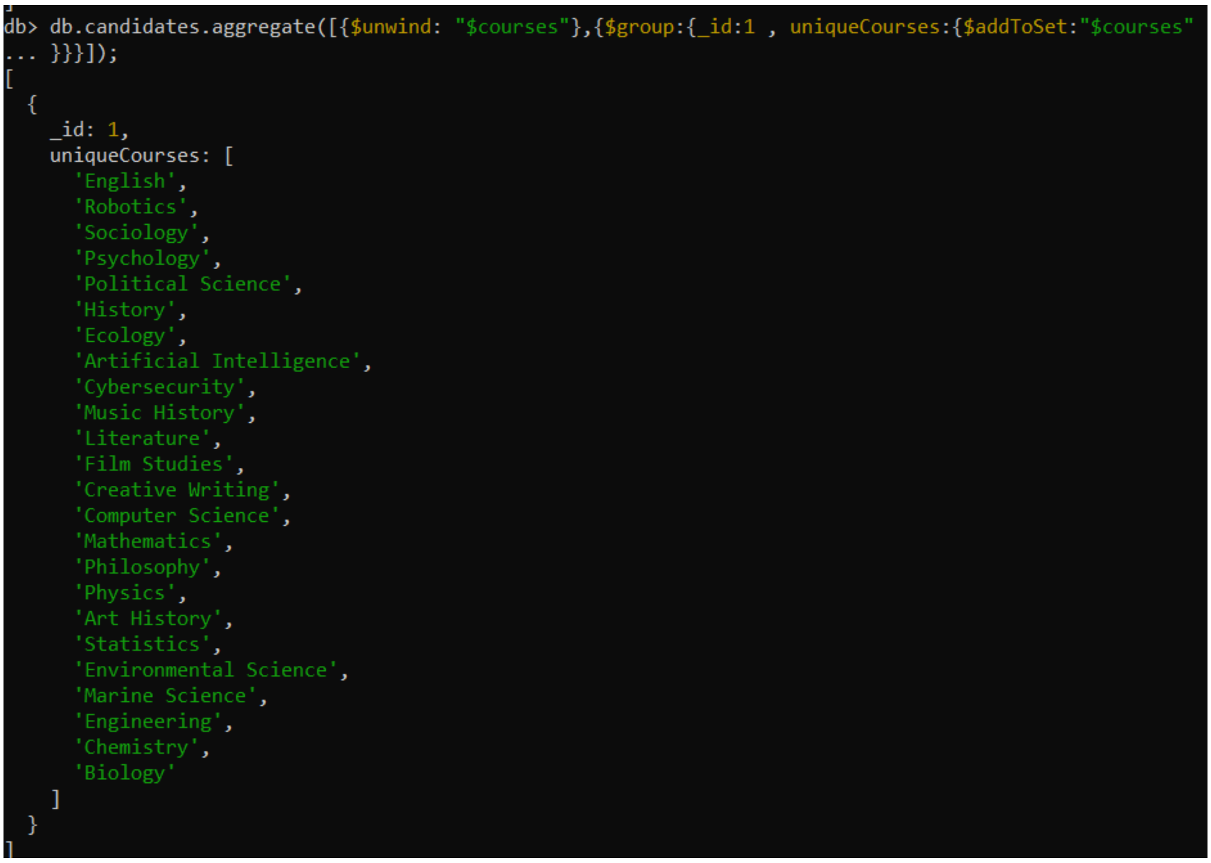
Example:

db.candidates.aggregate([

{ $unwind: "$courses" },

{ $group: { _id: 1 , uniqueCourses: { $addToSet: "$courses" } } }]);

Output:



```
db> db.candidates.aggregate([{$unwind: "$courses"},{$group:{_id:1 , uniqueCourses:{$addToSet:"$courses"
... }}}]);
[
  {
    _id: 1,
    uniqueCourses: [
      'English',
      'Robotics',
      'Sociology',
      'Psychology',
      'Political Science',
      'History',
      'Ecology',
      'Artificial Intelligence',
      'Cybersecurity',
      'Music History',
      'Literature',
      'Film Studies',
      'Creative Writing',
      'Computer Science',
      'Mathematics',
      'Philosophy',
      'Physics',
      'Art History',
      'Statistics',
      'Environmental Science',
      'Marine Science',
      'Engineering',
      'Chemistry',
      'Biology'
    ]
  }
]
```

Class 7: Aggregation pipeline

Experiment 6

Introduction

Aggregation Pipeline and its operators run with the

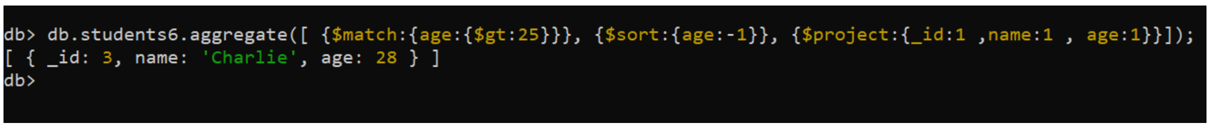
db.collection.aggregate() method do not modify

documents in a collection , the pipeline contains a $group, $sort, $project , $merge etc stages.

A. Finding students with age greater than 25 , sorted by age in descending order, and only return name and age.

```
db.students6.aggregate([
{$match:{age:{$gt:25}}},
{$sort:{age:-1}},
{$project :{_id:1 , name:1 , age:1}}])
```

Output:

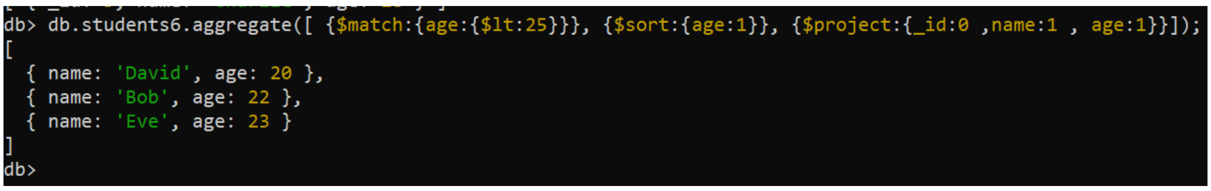


```
db> db.students6.aggregate([ {$match:{age:{$gt:25}}}, {$sort:{age:-1}}, {$project:{_id:1 ,name:1 , age:1}}]);
[ { _id: 3, name: 'Charlie', age: 28 } ]
db>
```

B. Find students with age less than 20, sorted by name in ascending order, and only return name and score

```
db.students6.aggregate([
{$match:{age:{$lt:23}}},
{$sort:{age:1}},
{$project :{_id:0, name:1 , age:1}}])
```

Output:

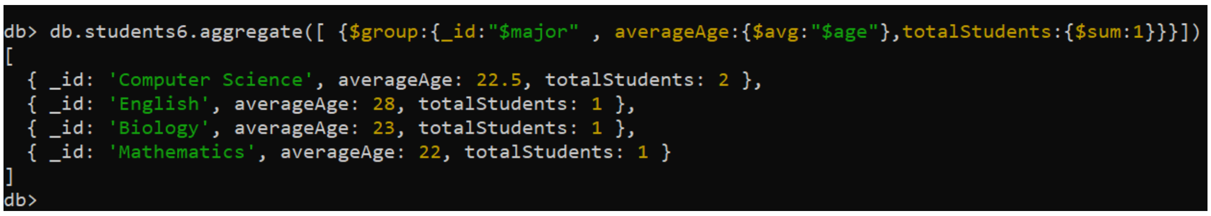


```
db> db.students6.aggregate([ {$match:{age:{$lt:25}}}, {$sort:{age:1}}, {$project:{_id:0 ,name:1 , age:1}}]);
[
  { name: 'David', age: 20 },
  { name: 'Bob', age: 22 },
  { name: 'Eve', age: 23 }
]
db>
```

Grouping students by major, calculating average age and total number of students in each major:

```
db.students6.aggregate([
{$group:{_id: “$major” , averageAge :{$avg: “age”},
totalStudents:{$sum :1}}}
]);
```

Output:



```
db> db.students6.aggregate([ {$group:{_id:"$major" , averageAge:{$avg:"$age"},totalStudents:{$sum:1}}}])
[
  { _id: 'Computer Science', averageAge: 22.5, totalStudents: 2 },
  { _id: 'English', averageAge: 28, totalStudents: 1 },
  { _id: 'Biology', averageAge: 23, totalStudents: 1 },
  { _id: 'Mathematics', averageAge: 22, totalStudents: 1 }
]
db>
```

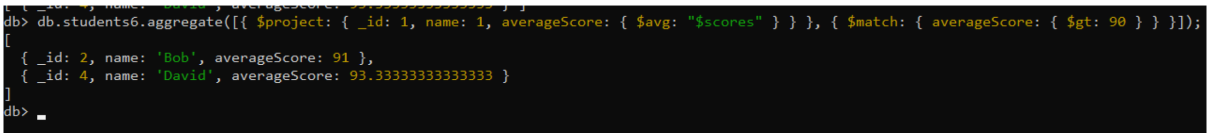
Finding students with an average above 90 .

db.students6.aggregate([

{$project:{_id:1 , name:1, averageScore :{$avg:"$scores}

}},{$match:{averageScore:{$gt:90}}}]);

Output:



```
db> db.students6.aggregate([{ $project: { _id: 1, name: 1, averageScore: { $avg: "$scores" } } }, { $match: { averageScore: { $gt: 90 } } }]);
[
  { _id: 2, name: 'Bob', averageScore: 91 },
  { _id: 4, name: 'David', averageScore: 93.33333333333333 }
]
db> _
```

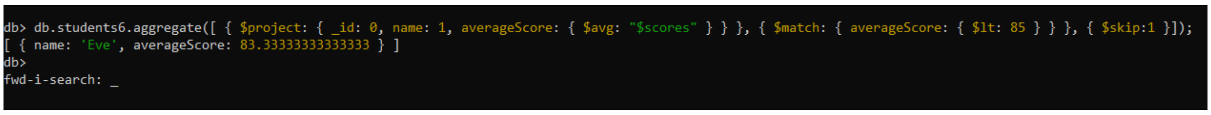
Finding students with an average score below 80 and skip the first document .

db.students6.aggregate([

{$project:{_id:1 , name:1, averageScore :{$avg:"$scores}

}},{$match:{averageScore:{$lt:85}}},{skip:1}]);

Output:



```
db> db.students6.aggregate([ { $project: { _id: 0, name: 1, averageScore: { $avg: "$scores" } } }, { $match: { averageScore: { $lt: 85 } } }, { $skip:1 }]);
[ { name: 'Eve', averageScore: 83.33333333333333 } ]
db>
fwd-i-search: _
```

Class 8: ACID & Indexes
Experiment 8

INTRO DUCTION

● Acids?

●ACID is an acronym that stands for atomicity, consistency, isolation, and durability. Together, these ACID properties ensure that a set of database operations (grouped together in a transaction) leave the database in a valid state even in the event of unexpected errors.

● Indexes?

● By using an index to limit the number of documents MongoDB scans, queries can be more efficient and therefore return faster(i.e.,unique, sparse, compound and multikey indexes)

● Demonstrating optimization of queries using indexes.

Few concepts

● Atomicity

● Consistency

● Replication

● Sharding

ACIDS

Atomicity

Money needs to both be removed from one account and added to the other, or the transaction will be aborted. Removing money from one account without adding it to another would leave the data in an inconsistent state.

Consistency

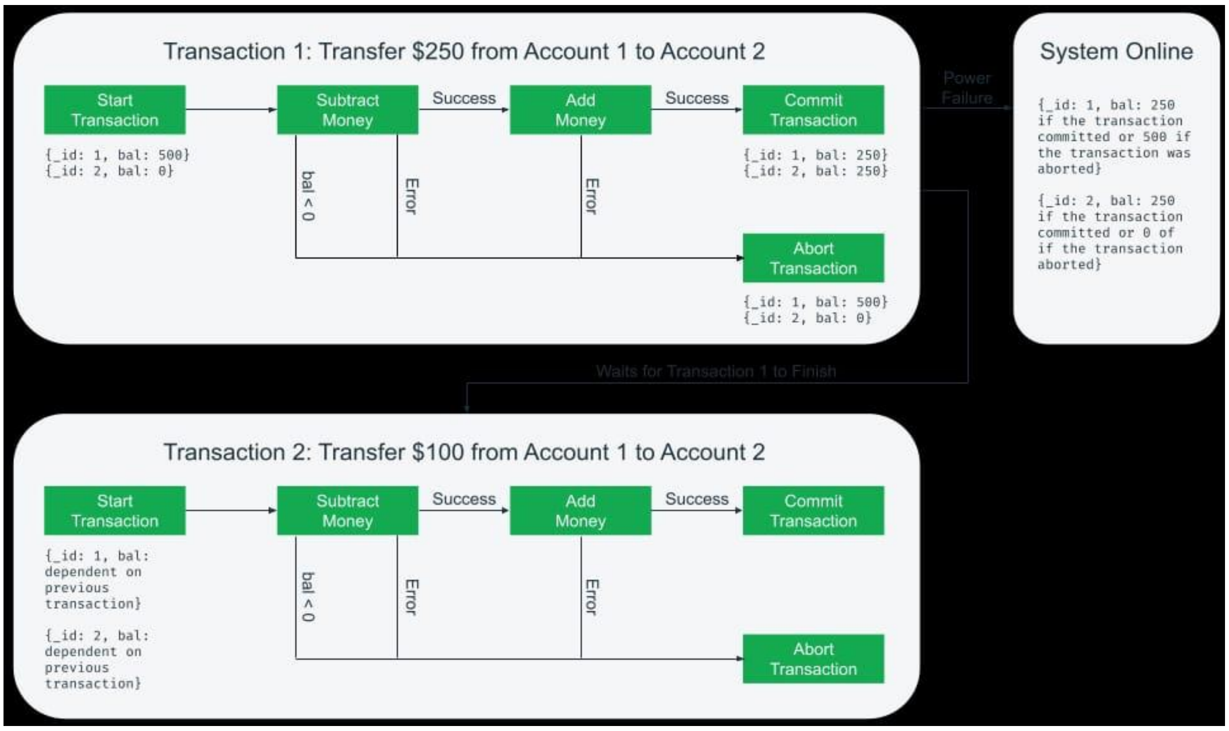
Consider a database constraint that an account balance cannot drop below zero dollars. All updates to an account balance inside of a transaction must leave the account with a valid, non-negative balance, or the transaction should be aborted

Isolation

Consider two concurrent requests to transfer money from the same bank account. The final result of running the transfer requests concurrently should be the same as running the transfer requests sequentially.

Durability

Consider a power failure immediately after a database has confirmed that money has been transferred from one bank account to another. The database should still hold the updated information even though there was an unexpected failure.


Transaction 1: Transfer $250 from Account 1 to Account 2
Start Transaction
Subtract Money
Success
Add Money
Success
Commit Transaction
{_id: 1, bal: 500}
{_id: 2, bal: 0}
bal < 0
Error
Error
{_id: 1, bal: 250}
{_id: 2, bal: 250}
Abort Transaction
{_id: 1, bal: 500}
{_id: 2, bal: 0}
Power Failure
System Online
{_id: 1, bal: 250 if the transaction committed or 500 if the transaction was aborted}
{_id: 2, bal: 250 if the transaction committed or 0 of if the transaction aborted}
Waits for Transaction 1 to Finish
Transaction 2: Transfer $100 from Account 1 to Account 2
Start Transaction
Subtract Money
Success
Add Money
Success
Commit Transaction
{_id: 1, bal: dependent on previous transaction}
{_id: 2, bal: dependent on previous transaction}
bal < 0
Error
Error
Abort Transaction

The diagram demonstrates how the ACID properties impact the flow of transferring money from one bank account to another.

Replication (Master - Slave)

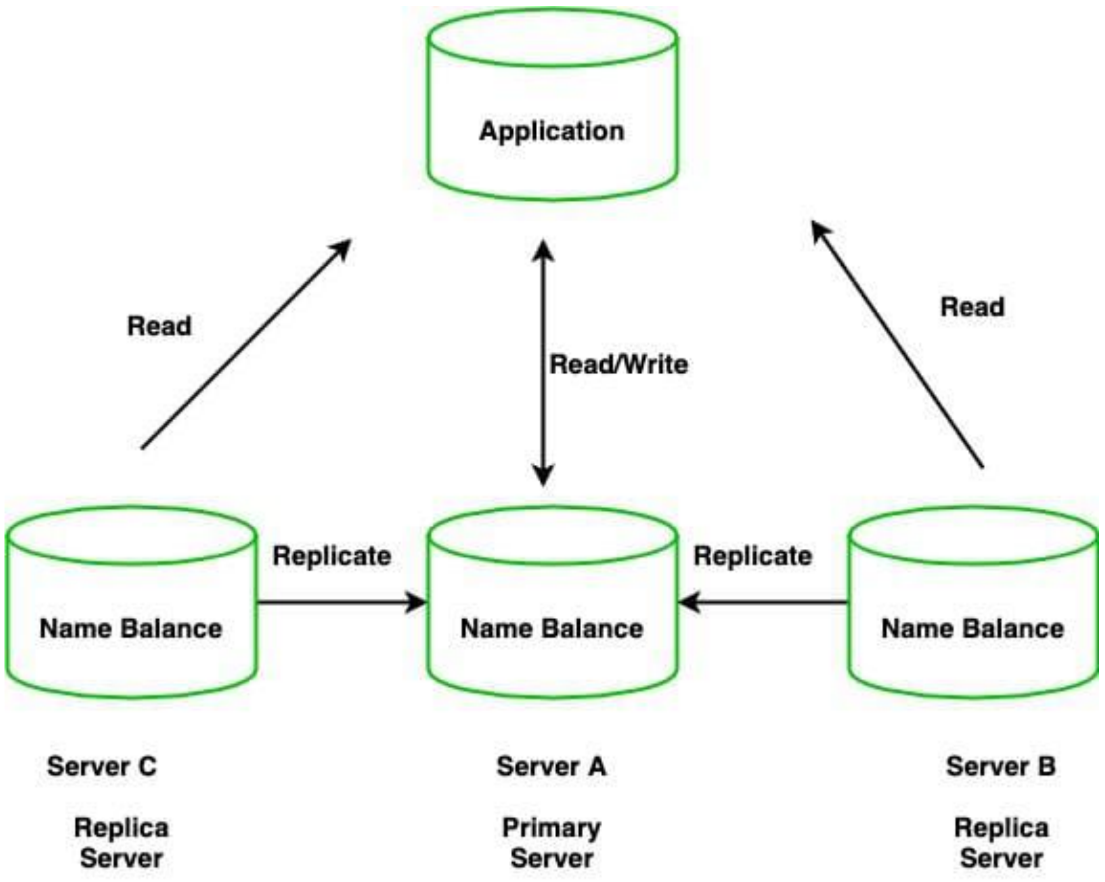
In simple terms, MongoDB replication is the process of creating a copy of the same data set in more than one MongoDB server. This can be achieved by using a Replica Set. A replica set is a group of MongoDB instances that maintain the same data set and pertain to any mongod process.

Sharding

Sharding is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations. Database systems with large data sets or high throughput applications can challenge the capacity of a single server.

Replication and Sharding

Replication and Sharding are two important features for scalability and data availability in MongoDB. Replication enhances data availability by creating duplicate copies of the dataset, whereas sharding helps in horizontal scaling by partitioning the large collection (dataset) into smaller discrete parts called shards.



INDEXES

Types Of Indexes :

- **Single Field Index**

  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.

  Example : `db. collection.createIndex({ field :1})`

- **Compound Field 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.

  Example : `db.collection.createIndex({ field1 :1, field2 : -1})`

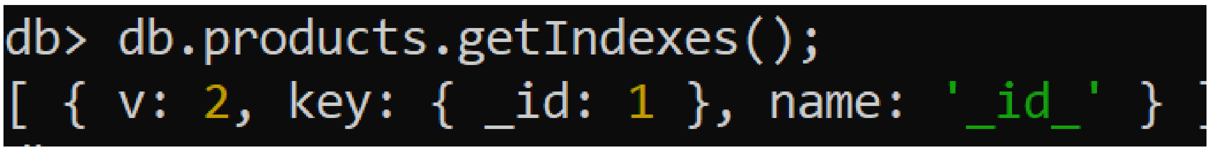
- **Multikey Index**

  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.

  Example : `db.collection.createIndex({ arrayField :1})`

Hands on :

```
db.products.getIndexes();
```



```
db> db.products.getIndexes();
[ { v: 2, key: { _id: 1 }, name: '_id_' } ]
```

Now ,

db.products.insertMany([

{ _id: 1, name: "Product A", category: "Electronics", price: 99.99, tags: ["electronics", "gadget"] },

{ _id: 2, name: "Product B", category: "Clothing", price: 49.99, tags: ["clothing", "fashion"] },

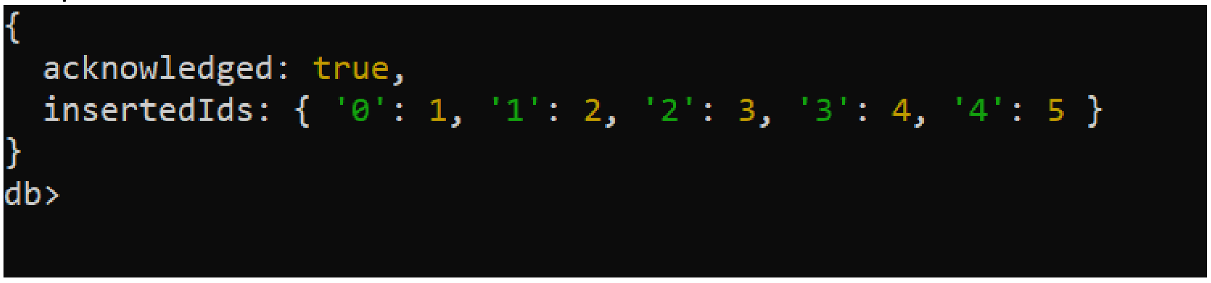
{ _id: 3, name: "Product C", category: "Electronics", price: 199.99, tags: ["electronics", "gadget"] },

{ _id: 4, name: "Product D", category: "Books", price: 29.99 }, // No tags

{ _id: 5, name: "Product E", category: "Electronics", price: 149.99, tags: ["electronics"] }

]);

Output:



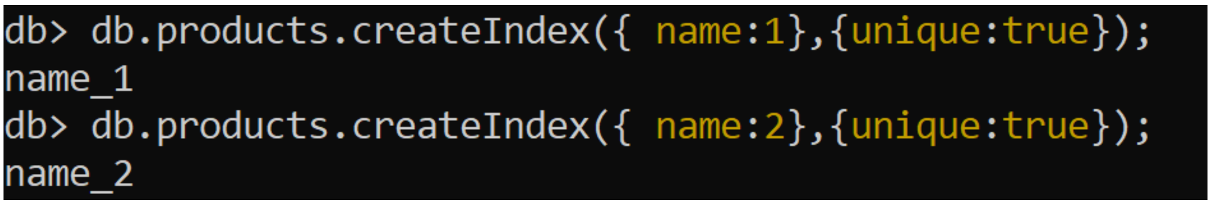
```
{
  acknowledged: true,
  insertedIds: { '0': 1, '1': 2, '2': 3, '3': 4, '4': 5 }
}
db>
```

///Creating Different Types Of Indexes:

1. Unique Index:

db.products.createIndex({name:1},{unique:true});

Output:

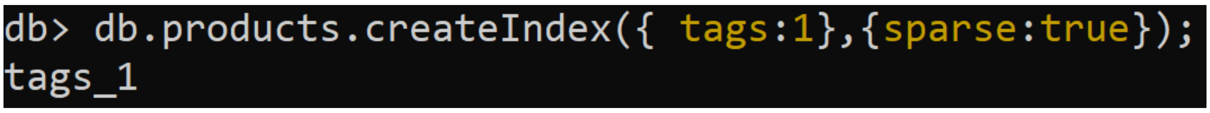


```
db> db.products.createIndex({ name:1},{unique:true});
name_1
db> db.products.createIndex({ name:2},{unique:true});
name_2
```

2.Sparse Index:

db.products.createIndex({tags:1},{sparse:true});

Output:

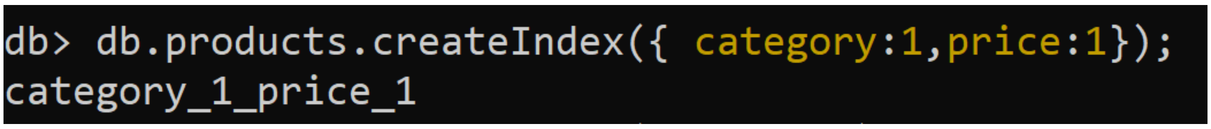


```
db> db.products.createIndex({ tags:1},{sparse:true});
tags_1
```

3.Compound Index:

db.products.createIndex({category:1,price:1});

Output:

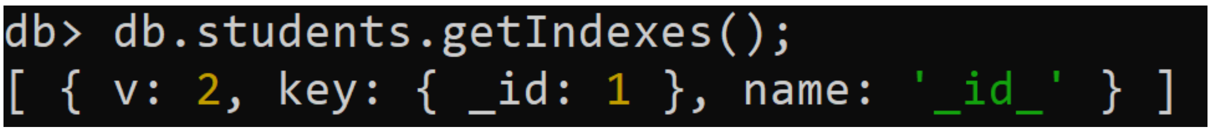


```
db> db.products.createIndex({ category:1,price:1});
category_1_price_1
```

To verify if the indexes have been created :

db.products.getIndexes();

Output:



```
db> db.students.getIndexes();
[ { v: 2, key: { _id: 1 }, name: '_id_' } ]
```