Introduction to MongoDB: What is MongoDB, Why MongoDB, Terms used in RDBMS and MongoDB, Data Types in MongoDB, MongoDB Query Language.

3.1 What is MongoDB?

MongoDB is

1. Cross-platform.
2. Open source.
3. Non-relational.
4. Distributed.
5. NoSQL.
6. Document-oriented data store.

3.2 WHY MongoDB?

Few of the major **challenges** with traditional RDBMS are dealing with **large volumes of data**, **rich variety** of data-particularly unstructured data, and **meeting up to the scale needs** of enterprise data.

The need is for a database that can scale out or scale horizontally to meet the scale requirements, has flexibility with respect to schema, is fault tolerant, is consistent and partition tolerant, and can be easily distributed over a multitude of nodes in a cluster.

1. **Full index support**
2. **Rich query language**
3. **Auto sharding**

4. **Document oriented**
5. **High performance**
6. **Fast in-place updates**
7. **Replication**
8. **Easy scalability**
9. **High availability**

3.2.1 Using Java Script Object Notation(JSON)

- JSON is extremely expressive.
- MongoDB actually does not use JSON but BSON – it is Binary JSON. It is an open standard. It is used to store complex data structures.

Let us trace the journey from .csv to XML to JSON:

Let us look at how data is stored in .csv file.

Assume that this data is about the employees of an organization named "XYZ". As we can see below, the column values are separated using commas and the rows are separated by a carriage return.

John, Mathews, +123 4567 8900
Andrews, Symmonds, +4567890 1234
Mable, Mathews, +789 1234 5678

However it can be made slightly more legible by adding column heading.

FirstName, LastName, ContactNo
John, Mathews, +1234567 8900
Andrews, Symmonds, +4567890 1234
Mable. Mathews, +789 12345678

Challenges with CSV Format

1. **Flat Structure**: CSV works best with flat, non-repeating data.

2. **Multiple Values Problem**:

   Some employees have **multiple Office and Home contact numbers**.

   Some have **multiple email addresses** (2, 3, or more).

3. **Merge Complexity**:

   When different departments use different CSV formats, **merging** becomes **tedious** and **error-prone**.

   Field inconsistency and missing/extra columns further complicate consolidation.

Why XML is Not Ideal for Simpler Use Cases

Pros:

- XML supports complex and hierarchical data structures.
- Suitable for highly structured data formats.

Cons:

- **Too verbose and heavy** for simple data exchange.
- Requires definition of data structure using **schemas or DTDs.**
- Overkill for lightweight, frequently changing employee records.

JSON as an Effective Alternative

1. Extensible and lightweight.
2. Handles **arrays/lists** of data naturally (e.g., multiple contacts or emails).
3. Easy to read and write.
4. Excellent for web applications and APIs.

```
{
  "FirstName": "John",
  "LastName": "Mathews",
  "ContactNo": ["+123 45678900", "+123 4444 5555"],
  "Emails": ["john@example.com", "john.mathews@work.com"]
},
{
  "FirstName": "Andrews",
  "LastName": "Symmonds",
  "ContactNo": ["+456 7890 1234", "+456 6666 7777"]
},
{
  "FirstName": "Mable",
  "LastName": "Mathews",
  "ContactNo": ["+789 1234 5678"]
}
```

JSON is very expressive. It provides the much needed ease to store and retrieve documents in their real form. The binary form of JSON is BSON. BSON is an open standard. In most cases it consumes less space as compared to the text-based JSON. There is yet another advantage with BSON. It is much easier and quicker to convert BSON to a programming language's native data format. There are MongoDB drivers available for a number of programming languages such as C, C++, Ruby, PHP, Python, C#, etc., and each works slightly differently. Using the basic binary format enables the native data structures to be built quickly for each language without going through the hassle of first processing JSON.

3.2.2 Creating or generating a Unique key

- Each JSON document should have a unique identifier.
- It is the _id key.
- It is similar to the primary key in relational databases.
- This facilitates search for documents based on the unique identifier.

- An index is automatically built on the unique identifier.
- It is your choice to either provide unique values yourself or have the mongo shell generate the same.

3.2.2.1 Database

It is a collection of collections. In other words, it is like a container for collections. It gets created the first time that your collection makes a reference to it. This can also be created on demand. Each database gets its own set of files on the file system. A single MongoDB server can house several databases.

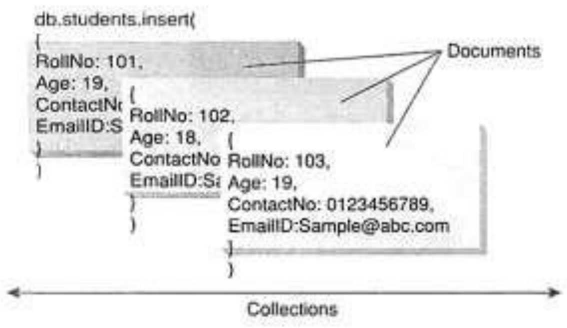
3.2.2.2 Collection

A collection is analogous to a table of RDBMS. A collection is created on demand. It gets created the first time that you attempt to save a document that references it. A collection exists within a single database. A collection holds several MongoDB documents. A collection does not enforce a schema. This implies that documents within a collection can have different fields. Even if the documents within a collection have same fields, the order of the fields can be different.

3.2.2.3 Document

A document is analogous to a row/record/tuple in an RDBMS table. A document has a dynamic schema. This implies that a document in a collection need not necessarily have the same set of fields/key-value pairs.

Shown in Figure below is a collection by the name "students" containing three documents.



3.2.3 Support for Dynamic Queries

MongoDB has extensive support for dynamic queries. This is in keeping with traditional RDBMS wherein we have static data and dynamic queries. CouchDB, another document-oriented, schema-less NoSQL data-base and MongoDB's biggest competitor, works on quite the reverse philosophy. It has support for dynamic data and static queries.

MongoDB: Dynamic Queries, Static Data

MongoDB allows you to build **dynamic queries** using a rich and expressive query language.

You can query on any field, including nested documents and arrays.

This aligns with the traditional **RDBMS approach**, where the **data structure (schema)** is fixed, and the **queries are dynamic**.

Example: You can construct queries at runtime, filter based on various fields, and use advanced operators ($gt, $in, $or, etc.).

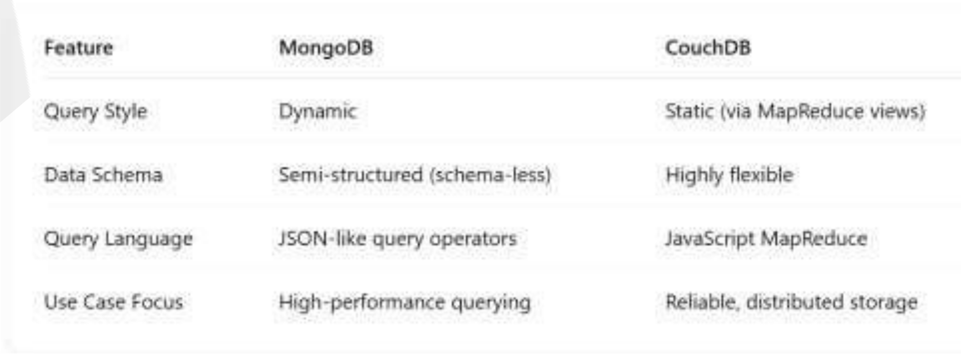
CouchDB: Static Queries, Dynamic Data

CouchDB uses **MapReduce views** to query data.

Once a view (which is essentially a query) is defined, it becomes **static**—you cannot alter it dynamically without redefining the view.

The **data is more flexible**, and each document can have a completely different structure.

This allows more flexibility in storing heterogeneous data but limits ad-hoc querying unless pre-defined.



Feature	MongoDB	CouchDB
Query Style	Dynamic	Static (via MapReduce views)
Data Schema	Semi-structured (schema-less)	Highly flexible
Query Language	JSON-like query operators	JavaScript MapReduce
Use Case Focus	High-performance querying	Reliable, distributed storage

3.2.4 Storing Binary Data

MongoDB provides GridFS to support the storage of binary data. It can store up to 4 MB of data. This usually suffices for photographs (such as a profile picture) or small audio clips. However, if one wishes to store movie clips, MongoDB has another solution.

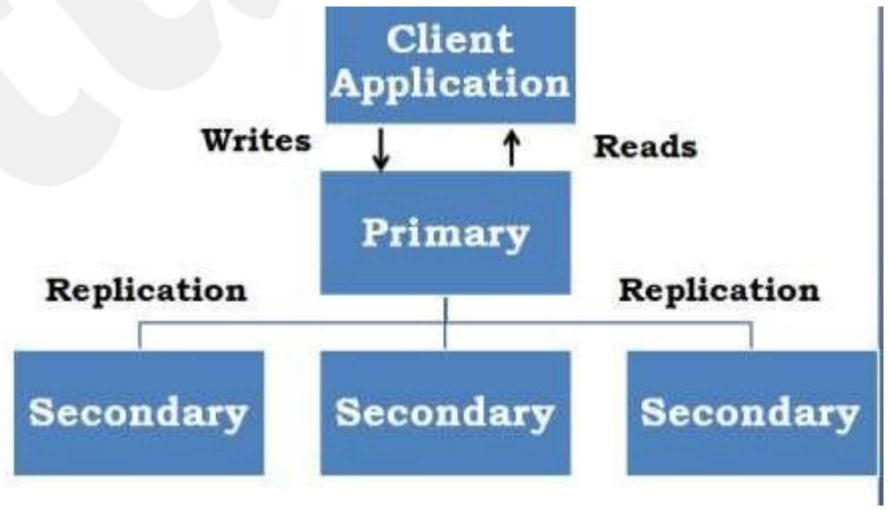
It stores the **metadata** (data about data along with the context information) in a collection called **"file".** It then breaks the data into **small pieces** called chunks and stores it in the **"chunks"** collection. This process takes care about the need for easy scalability.

3.2.5 Replication

Why replication?

It provides **data redundancy and high availability**. It helps to recover from hardware failure and service interruptions. In MongoDB, the replica set has a single primary and several secondaries. Each write request from the client is directed to the primary. The primary logs all write requests into its Oplog (operations log). The Oplog is then used by the secondary replica members to synchronize their data. This way there is strict adherence to consistency

Refer Figure. The clients usually read from the primary. However, the client can also specify a read preference that will then direct the read operations to the secondary.

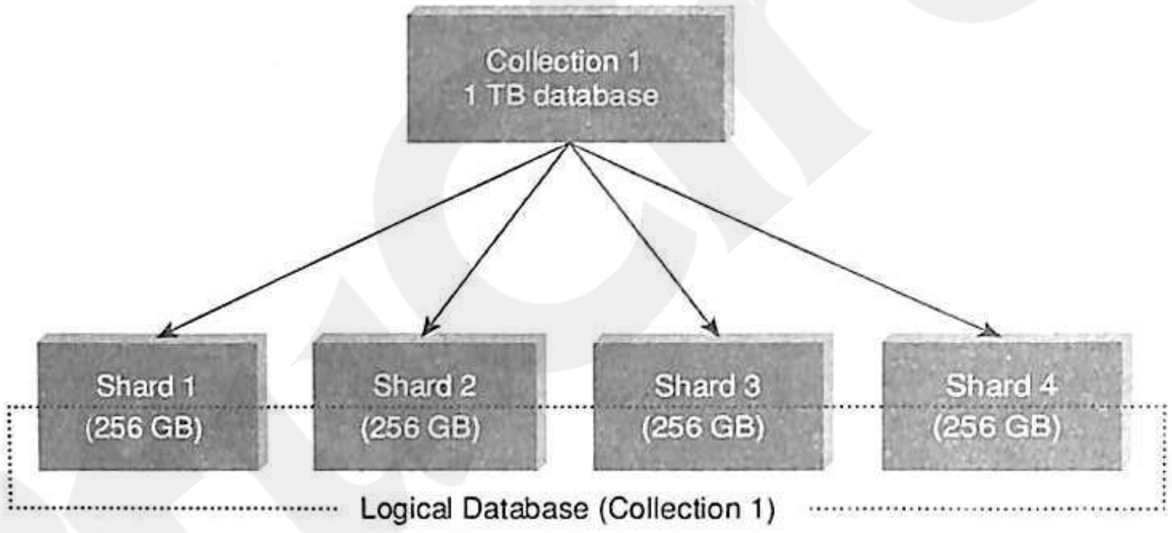


3.2.6 Sharding

Sharding is akin to horizontal scaling. It means that the large dataset is divided and distributed over multiple servers or shards. Each shard is an independent database and collectively they would constitute a logical database.

The prime advantages of sharding are as follows:

1. Sharding reduces the amount of data that each shard needs to store and manage. For example, if the dataset was 1 TB in size and we were to distribute this over four shards, each shard would house just 256 GB data. As the cluster grows, the amount of data that each shard will store and manage will decrease.
2. Sharding reduces the number of operations that each shard handles. For example, if we were to insert data, the application needs to access only that shard which houses that data.



3.2.7 Updating Information In-Place

MongoDB updates the information in-place. This implies that it updates the data wherever it is available. It does not allocate separate space and the indexes remain unaltered.

MongoDB is all for lazy-writes. It writes to the disk once every second. Reading and writing to disk is a slow operation as compared to reading and writing from memory. The fewer the reads and writes that we perform to the disk, the better is the performance. This makes MongoDB faster than its other competitors who write almost immediately to the disk. However, there is a tradeoff. MongoDB makes no guarantee that data will be stored safely on the disk.

3.3 Terms used in RDBMS and MongoDB

Structural Differences

RDBMS Terms	MogoDB Terms	Description
Database	Database	A logical grouping of data. Both RDBMS and MongoDB use this term.
Table	Collection	A group of related records. In MongoDB, a collection stores multiple documents instead of rows
Row	Document	In MongoDB, a document is a JSON-like object (BSON) that holds data in key-value pairs
Column	Field	A specific attribute of data in RDBMS is called a field in MongoDB.
Primary Key	_id Field	Every MongoDB document has a unique _id field, which acts as the primary key.

Relationships and Joins

RDBMS Terms	MongoDB Equivalent	Description
Foreign Key	Reference (Manual or $lookup)	In MongoDB, relationships can be represented by storing references to documents in other collections
JOIN Operation	Embedding or $lookup	RDBMS uses JOIN to fetch related data, while MongoDB supports embedding related documents or using $lookup for similar functionality

Indexing and Querying

RDBMS Terms	MongoDB Equivalent	Description
Index	Index	Both RDBMS and MongoDB use indexes to speed up queries. MongoDB supports single-field, compound, text, and geospatial indexes.
SQL Query (SELECT, WHERE, etc.)	Find Query (db.collection.find())	MongoDB uses a JavaScript-like query syntax instead of SQL.
Aggregate Functions (SUM, AVG,	Aggregation Pipeline	MongoDB's aggregation framework replaces SQL functions with operations like

COUNT)		**$group, $sum, $avg, etc.**

Transactions and Consistency

RDBMS Terms	MongoDB Equivalent	Description
ACID Transactions	**Multi-document Transactions**	**RDBMS supports ACID compliance by default, while MongoDB introduced multi-document transactions in version 4.0 for similar behavior.**
Commit & Rollback	**Session-Based Transactions**	**In MongoDB, transactions work using session-based operations.**

Session-based operation refers to processes where a user's interactions are tracked across multiple requests using a unique session ID. It allows continuity, like maintaining login state or a shopping cart, throughout a user's visit.

Performance and Scaling

RDBMS Terms	MongoDB Equivalent	Description
Vertical Scaling	**Horizontal Scaling**	**RDBMS typically scales vertically by upgrading hardware, whereas MongoDB scales horizontally using sharding.**

Partitioning	**Sharding**	**Data distribution across multiple servers in MongoDB is handled by sharding**
Replication (Master-Slave)	**Replication (Replica Sets)**	**Both databases support replication, but MongoDB uses Replica Sets for automatic failover and redundancy.**

Backup and Recovery

RDBMS Terms	**MongoDB Equivalent**	**Description**
Backup & Restore (mysqldump, pg_dump)	**mongodump & mongorestore**	**MongoDB provides tools like mongodump and mongorestore for backups.**
Point-in-Time Recovery	**Oplog Replay**	**In MongoDB, oplog (operations log) enables point-in-time recovery for replica sets**

Data Integrity and Constraints

RDBMS Terms	**MongoDB Equivalent**	**Description**
Constraints (NOT NULL, UNIQUE, CHECK, FOREIGN KEY)	**Schema Validation & Document Validation**	**MongoDB allows schema validation rules but is not as strict as RDBMS constraints.**

Triggers	**Change Streams**	**MongoDB supports change streams, which notify applications of data changes in real time.**

Security and Authentication

RDBMS Terms	**MongoDB Equivalent**	**Description**
User Roles & Permissions	**Role-Based Access Control (RBAC)**	**MongoDB uses RBAC for fine-grained security.**
Authentication (LDAP, Kerberos, etc.)	**Authentication with LDAP, Kerberos, SCRAM**	**MongoDB supports authentication methods like LDAP, Kerberos, and SCRAM-SHA**
Encryption	**Encryption at Rest & In Transit**	**MongoDB supports TLS/SSL for encrypted connections and AES-256 for data encryption at rest**

3.3.1 Create Database

Creating a Database

Syntax:

```
use DATABASE_Name
```

Example:

To create a database named myDB, use:

```
use myDB
```

Output:

```
switched to db myDB
```

Verifying the Current Database

To check which database you are currently using:

```
db
```

Output:

```
myDB
```

Listing All Databases

To list all existing databases:

```
show dbs
```

Output (example):

```
admin  (empty)
local    0.078GB
test   0.078GB
```

- The newly created database (e.g., myDB) does **not appear** in the list from show dbs **until it contains at least one document**.
- The **default database** in MongoDB is test.

- If no database is explicitly created, any collections inserted will be stored in the test database.

3.3.2 Drop Database

Syntax to Drop a Database

db.dropDatabase();

Steps to Drop a Specific Database (e.g., "myDB")

1. First, switch to the database you want to drop:
2. use myDB;
3. Then execute the drop command:
4. db.dropDatabase();

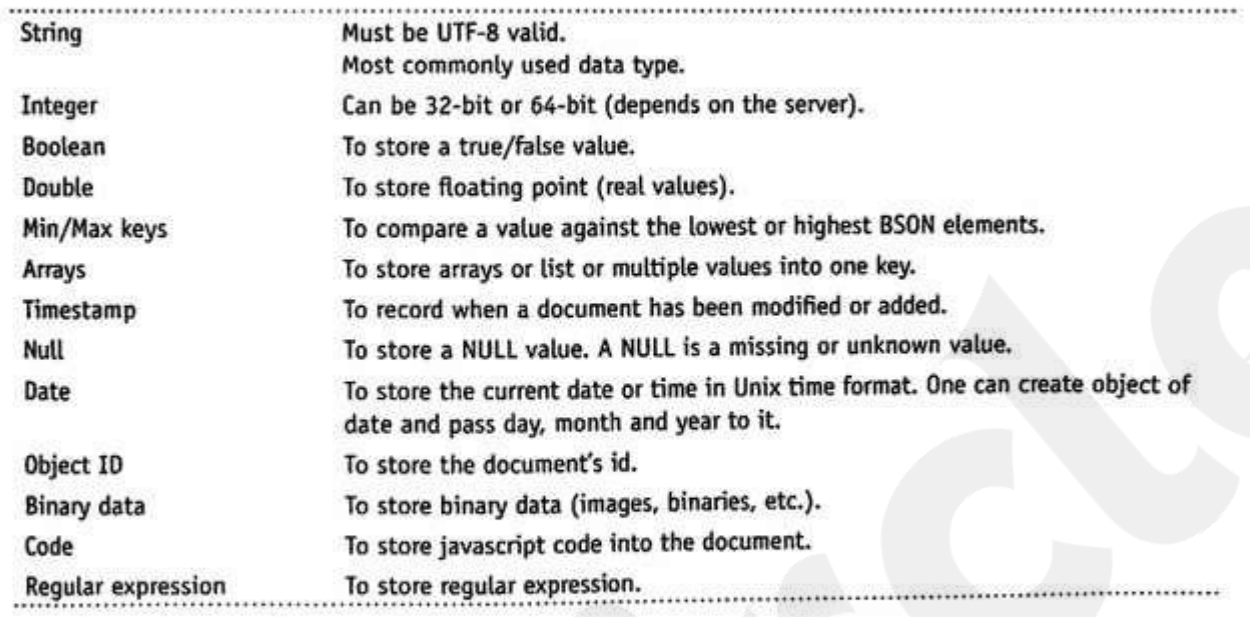
Confirmation Message

After running the command, MongoDB returns:

```
{ "dropped" : "myDB", "ok" : 1 }
```

- Always ensure you are in the correct database before executing db.dropDatabase().
- If **no database is selected**, MongoDB will drop the **default database test**.

Data Types in MongoDB



String	Must be UTF-8 valid. Most commonly used data type.
Integer	Can be 32-bit or 64-bit (depends on the server).
Boolean	To store a true/false value.
Double	To store floating point (real values).
Min/Max keys	To compare a value against the lowest or highest BSON elements.
Arrays	To store arrays or list or multiple values into one key.
Timestamp	To record when a document has been modified or added.
Null	To store a NULL value. A NULL is a missing or unknown value.
Date	To store the current date or time in Unix time format. One can create object of date and pass day, month and year to it.
Object ID	To store the document's id.
Binary data	To store binary data (images, binaries, etc.).
Code	To store javascript code into the document.
Regular expression	To store regular expression.

1. String

In MongoDB, strings must be UTF-8 encoded.

Ex: { "name": "John Doe" }

2. Integer

MongoDB differentiate between 32-bit(Int32) and 64-bit(Int64) integer

Ex: { "age": 25 }

3. Double

Ex: {“Salary” : 1900.54}

4. Boolean

Ex: {“isActive : true”}

5. Array

Stores multiple values in a single field

Ex: { "skills": ["JavaScript", "Python", "MongoDB"] }

6. Object(Embedded document or subdocument)

Stores a document inside another document.

Ex : { "name": "Alice", "address": { "city": "New York", "zip": "10001" }}

7. ObjectId

A 12-byte identifier (timestamp, machine ID, process ID, and counter).

Ex: { "_id": ObjectId("507f1f77bcf86cd799439011") }

8. Date

Default format: ISODate("YYYY-MM-DDTHH:MM:SSZ").

Ex: { "createdAt": ISODate("2024-03-25T10:30:00Z") }

9. Null

Ex: { "deletedAt": null }

10. Binary Data

Stores binary data such as images, audio, or encrypted data.

Ex : { "profilePicture": BinData(0, "base64EncodedData") }

11. Regular Expression

Stores and queries strings using regex patterns.

Ex: { "pattern": /mongodb/i }

12. JavaScript Code

Ex: { "script": function() { return "Hello MongoDB"; } }

13. JavaScript with Scope

Similar to javascript, but allows defining scope (variables) for the script.

Ex: { "script": { "$code": "function(x) { return x * 2; }", "$scope": { "x": 10 } }}

14. Timestamp

Stores a high-precision timestamp (used for internal MongoDB operations).

Ex: { "createdAt": Timestamp(1618928492, 1) }

15. Decimal128

High-precision 128-bit decimal numbers (useful for financial calculations).

Ex: { "amount": NumberDecimal("1234.5678") }

16. MinKey & MaxKey

MinKey: Represents the lowest possible value in MongoDB (useful for sorting).

MaxKey: Represents the highest possible value.

Ex : { "lowestValue": MinKey(), "highestValue": MaxKey() }

Useful MongoDB Shell Commands

To report the name of the current database:

db

Example Output:

test

To display the list of all databases:

```
show dbs
```

Example Output:

```
admin    (empty)
local    0.078GB
myDB1    0.078GB
```

To switch to a new database (e.g., myDB1):

```
use myDB1
```

Output:

```
switched to db myDB1
```

To display the list of collections (tables) in the current database:

```
show collections
```

Example Output:

```
system.indexes
system.js
```

To display the current version of the MongoDB server:

```
db.version()
```

Example Output:

```
2.6.1
```

Consider a table “Students” with the following columns:

1. StudRoll No

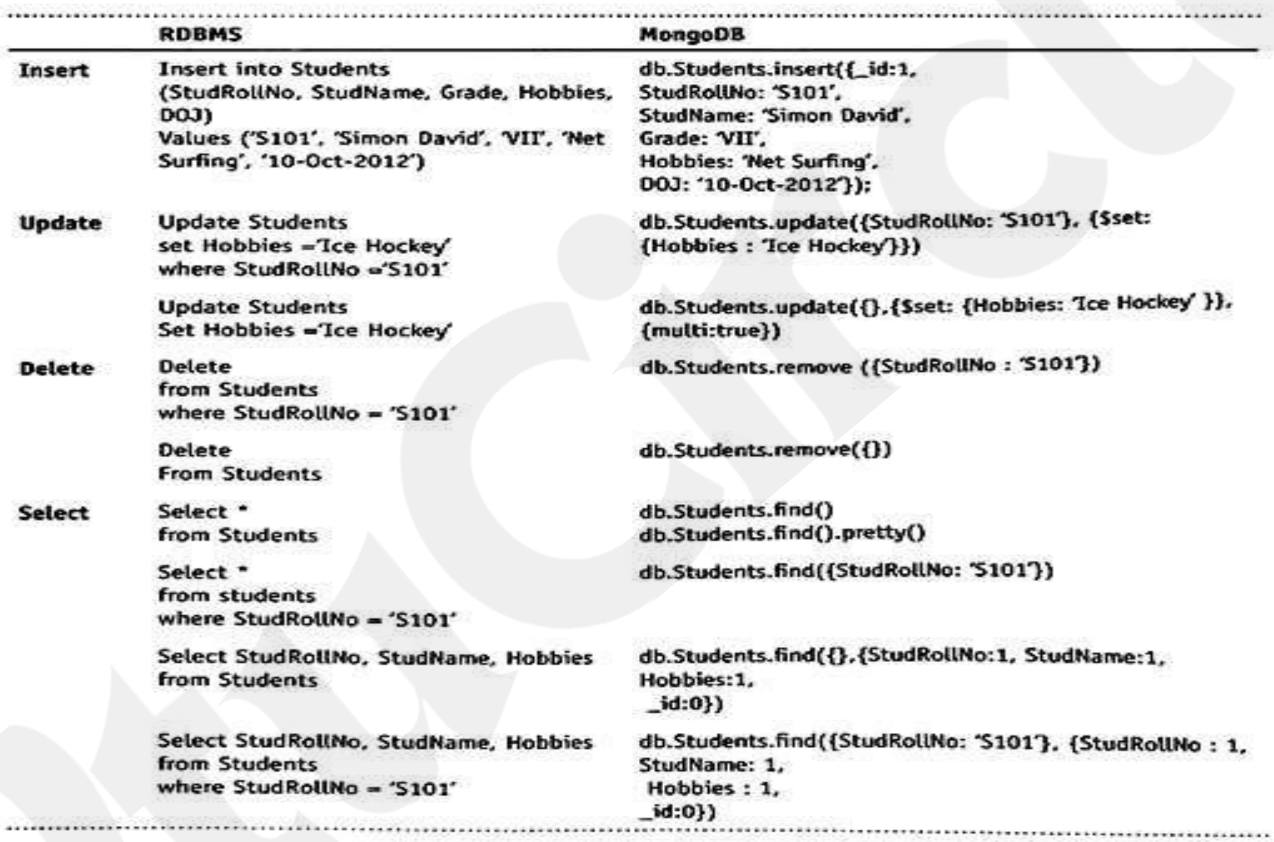
2. StudName

3. Grade

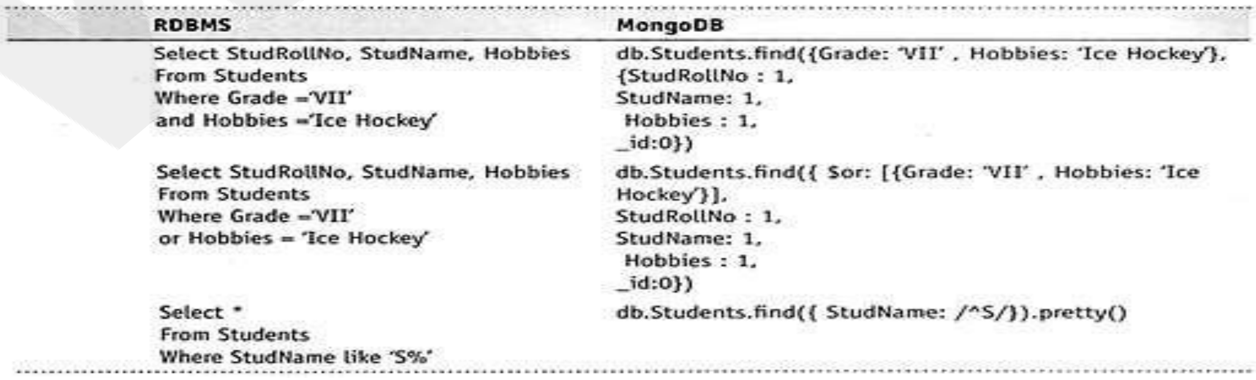
4. Hobbies

5. DOJ

Before we get into the details of CRUD operations in MongoDB, let us look at how the statements are written in RDBMS and MongoDB.



	RDBMS	MongoDB
Insert	Insert into Students (StudRollNo, StudName, Grade, Hobbies, DOJ) Values ('S101', 'Simon David', 'VII', 'Net Surfing', '10-Oct-2012')	db.Students.insert({_id:1, StudRollNo: 'S101', StudName: 'Simon David', Grade: 'VII', Hobbies: 'Net Surfing', DOJ: '10-Oct-2012'});
Update	Update Students set Hobbies ='Ice Hockey' where StudRollNo ='S101'	db.Students.update({StudRollNo: 'S101'}, {$set: {Hobbies : 'Ice Hockey'}})
	Update Students Set Hobbies ='Ice Hockey'	db.Students.update({},{$set: {Hobbies: 'Ice Hockey' }}, {multi:true})
Delete	Delete from Students where StudRollNo = 'S101'	db.Students.remove ({StudRollNo : 'S101'})
	Delete From Students	db.Students.remove({})
Select	Select * from Students	db.Students.find() db.Students.find().pretty()
	Select * from students where StudRollNo = 'S101'	db.Students.find({StudRollNo: 'S101'})
	Select StudRollNo, StudName, Hobbies from Students	db.Students.find({},{StudRollNo:1, StudName:1, Hobbies:1, _id:0})
	Select StudRollNo, StudName, Hobbies from Students where StudRollNo = 'S101'	db.Students.find({StudRollNo: 'S101'}, {StudRollNo : 1, StudName: 1, Hobbies : 1, _id:0})



RDBMS	MongoDB
Select StudRollNo, StudName, Hobbies From Students Where Grade ='VII' and Hobbies ='Ice Hockey'	db.Students.find({Grade: 'VII' , Hobbies: 'Ice Hockey'}, {StudRollNo : 1, StudName: 1, Hobbies : 1, _id:0})
Select StudRollNo, StudName, Hobbies From Students Where Grade ='VII' or Hobbies = 'Ice Hockey'	db.Students.find({ $or: [{Grade: 'VII' , Hobbies: 'Ice Hockey'}], StudRollNo : 1, StudName: 1, Hobbies : 1, _id:0})
Select * From Students Where StudName like 'S%'	db.Students.find({ StudName: /^S/}).pretty()

3.5 MongoDB Query Language

CRUD (Create, Read Update, and Delete) operations in MongoDB

Create → Creation of data is done using insert() or update() or save() method.

Read → Reading the data is performed using the find() method.

Update → Update to data is accomplished using the update() method with UPSERT set to false.

Delete → a document is Deleted using the remove() method.

Creating and Dropping Collections

Creating a Collection

Objective: Create a collection named "Person".

Step 1 – View existing collections:

```
show collections
```

Example output:

```
Students
food
system.indexes
system.js
```

Step 2 – Create the new collection:

```
db.createCollection("Person")
```

Output:

```
{ "ok" : 1 }
```

Outcome – View updated collections:

```
show collections
```

Example output after creation:

```
Person
```

```
Students
food
system.indexes
system.js
```

Dropping a Collection

Objective: Drop the collection named "food".

Step 1 – Check current collections:

```
show collections
```

Example output:

```
Person
Students
food
system.indexes
system.js
```

Step 2 – Drop the collection:

```
db.food.drop()
```

Output:

```
true
```

Outcome – View updated collections:

```
show collections
```

Example output after dropping:

```
Person
Students
system.indexes
system.js
```

3.5.1 Insert Method

1. Create Collection and Insert Document

Objective: Create a collection named Students and insert a document.

Check existing collections:

show collections

Insert first document:

db.Students.insert({_id:1, StudName:"Michelle Jacintha", Grade:"VII", Hobbies:"Internet Surfing"})

Verify insertion:

db.Students.find().pretty()

2. Insert Another Document

Insert second document:

db.Students.insert({_id:2, StudName:"Mabel Mathews", Grade:"VII", Hobbies:"Baseball"})

Verify with pretty():

db.Students.find().pretty()

3. Conditional Insert/Update with upsert

Objective: Insert Aryan David only if not already in the collection. If present, update his hobbies.

Check current documents:

db.Students.find().pretty()

Insert using upsert:

db.Students.update({_id:3}, {StudName:"Aryan David", Grade:"VII", $set:{Hobbies:"Skating"}}, {upsert:true})

Confirm insertion:

db.Students.find().pretty()

4. Update Existing Document

Objective: Update Aryan David's hobbies from "Skating" to "Chess".

Update using upsert (will update if exists, insert otherwise):

db.Students.update({_id:3}, {StudName:"Aryan David", Grade:"VII", $set:{Hobbies:"Chess"}}, {upsert:true})

5. Insert Document Using save()

Objective: Insert Vamsi Bapat without specifying _id.

Save document:

db.Students.save({StudName:"Vamsi Bapat", Grade:"VII"})

Check final documents:

db.Students.find().pretty()

3.5.2 save() method

Inserts a new document if no document with the specified _id exists. If the document exists, it replaces the existing one.

Objective:

Insert the document of "Hersch Gibbs" into the Students collection using the update() method with the upsert option.

Step 1: Check existing documents in the "Students" collection

Shows the existing documents with their _id, StudName, Grade, and Hobbies.

Step 2: Use update with upsert: false

```
db.Students.update(
  {_id:4, StudName:"Hersch Gibbs", Grade:"VII"},
  {$set: {Hobbies: "Graffiti"}},
  {upsert: false}
);
```

- No document is inserted because a document with _id:4 doesn't exist.
- Result shows nUpserted: 0 meaning no document was inserted.

Step 3: Use update with upsert: true

```
db.Students.update(
  {_id:4, StudName:"Hersch Gibbs", Grade:"VII"},
  {$set: {Hobbies: "Graffiti"}},
  {upsert:  true}
);
```

- A new document with _id:4 is inserted.
- Result shows nUpserted: 1, meaning one document was inserted.

Step 4: Confirm the new document

db.Students.find() now shows the new document of "Hersch Gibbs" with the Hobbies: "Graffiti" included.

3.5.3 Add a new field to an existing document-Update Method

Syntax of update method

```
db.students.update(
  {Age: {$gt: 18}},        // Update Criteria (which documents to update)
  {$set: {Status: "A"}},  // Update Action (what to update/set)
  {multi: true}            // Update Option (update multiple documents)
)
```

Objective:

Add a new field "Location" with value "Newark" to the document with _id:4 in the "Students" collection.

Input:

Check the document with _id:4 before updating:

db.Students.find({_id:4}).pretty();

Output:

```
{
  "_id": 4,
  "Grade": "VII",
  "StudName": "Hersch Gibbs",
  "Hobbies": "Graffiti"
}
```

Act:

Add the new field "Location" with the value "Newark":

```
db.Students.update(
  {_id:4},
  {$set: {Location: "Newark"}}
);
```

Output shows:

```
{
  "nMatched": 1,
  "nUpserted": 0,
  "nModified": 1
}
```

Outcome:

Confirm the new field has been added:

db.Students.find({_id:4}).pretty();

Output:

```
{
  "_id": 4,
  "Grade": "VII",
  "StudName": "Hersch Gibbs",
  "Hobbies": "Graffiti",
  "Location": "Newark"
}
```

3.5.4 Removing an Existing Field from an Existing Document – Remove Method

Objective:

To remove the field "Location" with the value "Newark" from a document with _id: 4 in the Students collection.

Input:

Inspect the current document:

db.Students.find({_id:4}).pretty()

Output:

```
{
  "_id": 4,
  "Grade": "VII",
  "StudName": "Hersch Gibbs",
  "Hobbies": "Graffiti",
  "Location": "Newark"
}
```

Act:

Execute the update command to remove the "Location" field:

db.Students.update({_id:4}, { $unset: { Location: "Newark" } })

This uses:

- update to modify the document,
- $unset to remove the "Location" field (value is ignored, key matters).

Output:

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

Outcome:

Verify the document again:

db.Students.find({_id:4}).pretty()

Result:

```
{
  "_id": 4,
  "Grade": "VII",
  "StudName": "Hersch Gibbs",
  "Hobbies": "Graffiti"
}
```

The "Location" field has been successfully removed.

1. Removing a Document with remove() Method

db.Students.remove({Age: {$gt: 18}})

- Removes all documents in the Students collection where the Age is greater than 18.

2. Removing a Field from a Document with $unset

db.Students.update({_id: 4}, {$unset: {Location: "Newark"}})

- Removes the Location field from the document with _id: 4.

3. Updating Documents with $set

db.Students.update({_id: 4}, {$set: {Grade: "X"}})

- Updates the Grade field of the document with _id: 4 to "X".

4. Adding New Fields

db.Students.update({_id: 4}, {$set: {Sports: "Football"}})

- Adds a new field Sports with the value "Football" to the document with _id: 4.

5. Using multi: true to Update Multiple Documents

```
db.Students.update(
  {Grade: "VII"},
  {$set: {Sports: "Cricket"}},
  {multi: true}
)
```

- Updates all documents where Grade is "VII" by setting Sports to "Cricket".

6. Replacing an Entire Document

```
db.Students.replaceOne(
  {_id: 4},
  {
    _id: 4,
    Grade: "X",
    StudName: "Hersch Gibbs",
    Hobbies: "Graffiti",
    Sports:  "Football"
  }
)
```

- Replaces the entire document with _id: 4.

7. Upsert Operation (update + upsert: true)

```
db.Students.update(
  {_id: 5},
  {$set: {StudName: "Paul Adams", Grade: "VIII"}},
  {upsert: true}
)
```

- If a document with _id: 5 doesn't exist, MongoDB inserts it with the specified fields.

3.5.7 Finding Elements Based on Some Criteria – findOne() Method

Objective:

To retrieve a **single document** from the "Students" collection where a specific field (e.g., Grade) matches a value.

Command:

db.Students.findOne({Grade: "VII"})

- This will return **only one document** (not all matches).
- Equivalent in SQL:

SELECT * FROM Students WHERE Grade = 'VII' LIMIT 1;

Finding Specific Elements – find() with Projections

Objective:

Retrieve only **selected fields** from matching documents.

Command:

db.Students.find({Grade: "VII"}, {StudName: 1, _id: 0})

- This will return only the StudName of students in Grade "VII".
- _id: 0 hides the _id field.
- Equivalent in SQL:

SELECT StudName FROM Students WHERE Grade = 'VII';

Sorting Results – sort() Method

Objective:

Sort documents in ascending or descending order by a specified field.

Commands:

db.Students.find().sort({Grade: 1}) // Ascending

db.Students.find().sort({Grade: -1}) // Descending

- Equivalent in SQL:

SELECT * FROM Students ORDER BY Grade ASC;

SELECT * FROM Students ORDER BY Grade DESC;

Limiting Results – limit() Method

Objective:

Retrieve only a **certain number** of documents.

Command:

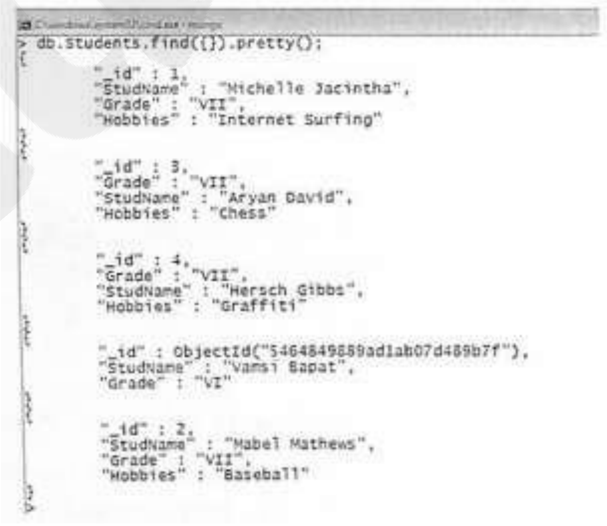
db.Students.find().limit(3)

- Returns the **first 3 documents** from the collection.
- Equivalent in SQL:

SELECT * FROM Students LIMIT 3;

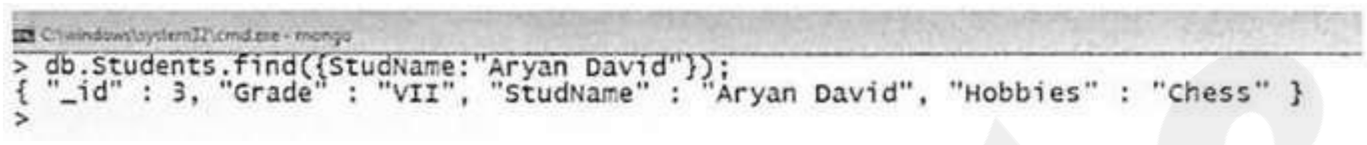
3.5.5 Finding Documents based on Search Criteria - Find Method

Objective: To search for documents from the "Students" collection based on certain search criteria. Input: Check the documents in the "Students" collection before proceeding.



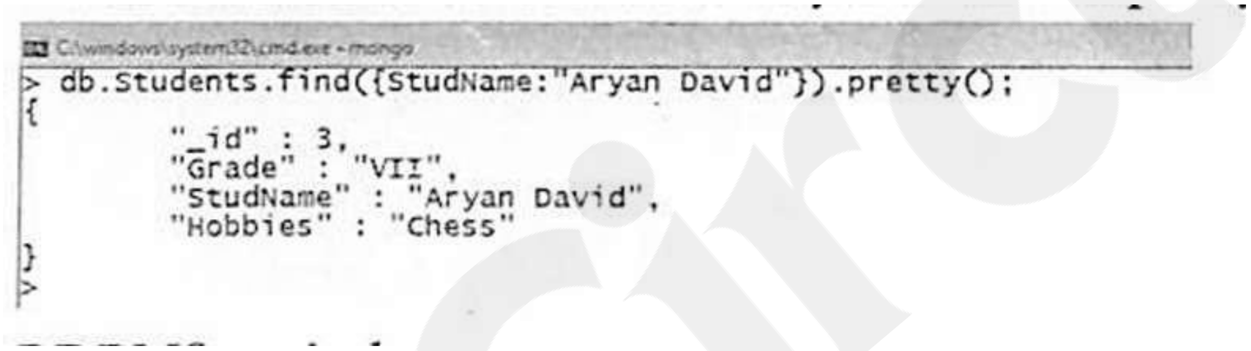
```
> db.students.find({}).pretty();
{
        "_id" : 1,
        "StudName" : "Michelle Jacintha",
        "Grade" : "VII",
        "Hobbies" : "Internet Surfing"
}
{
        "_id" : 3,
        "Grade" : "VII",
        "StudName" : "Aryan David",
        "Hobbies" : "Chess"
}
{
        "_id" : 4,
        "Grade" : "VII",
        "StudName" : "Hersch Gibbs",
        "Hobbies" : "Graffiti"
}
{
        "_id" : ObjectId("5464849889ad1ab07d489b7f"),
        "StudName" : "Vamsi Bapat",
        "Grade" : "VI"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
>
```

Act: Find the document wherein the "StudName" has value "Aryan David”.
db.Students.find({StudName:"Aryan David"});
Outcome:



```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({StudName:"Aryan David"});
{ "_id" : 3, "Grade" : "VII", "StudName" : "Aryan David", "Hobbies" : "Chess" }
>
```

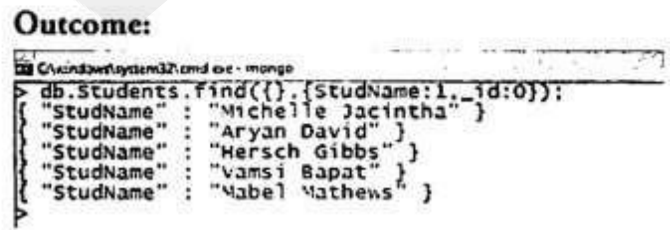
To format the above output, use the pretty() method:
db.Students.find({StudName:"Aryan David"}).pretty();



```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({StudName:"Aryan David"}).pretty();
{
        "_id" : 3,
        "Grade" : "VII",
        "StudName" : "Aryan David",
        "Hobbies" : "Chess"
}
>
```

RDBMS equivalent:
Select *
    From Students
        Where StudName like 'Aryan David';

Objective: To display only the StudName from all the documents of the Student's collection. The identifier "_id" should be suppressed and NOT displayed.
Act:
db.Students.find({}, {StudName: 1,_id:0});
Outcome:



Outcome:

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({},{StudName:1,_id:0});
{ "StudName" : "Michelle Jacintha" }
{ "StudName" : "Aryan David" }
{ "StudName" : "Hersch Gibbs" }
{ "StudName" : "Vamsi Bapat" }
{ "StudName" : "Mabel Mathews" }
>
```

RDBMS equivalent:

Select StudName

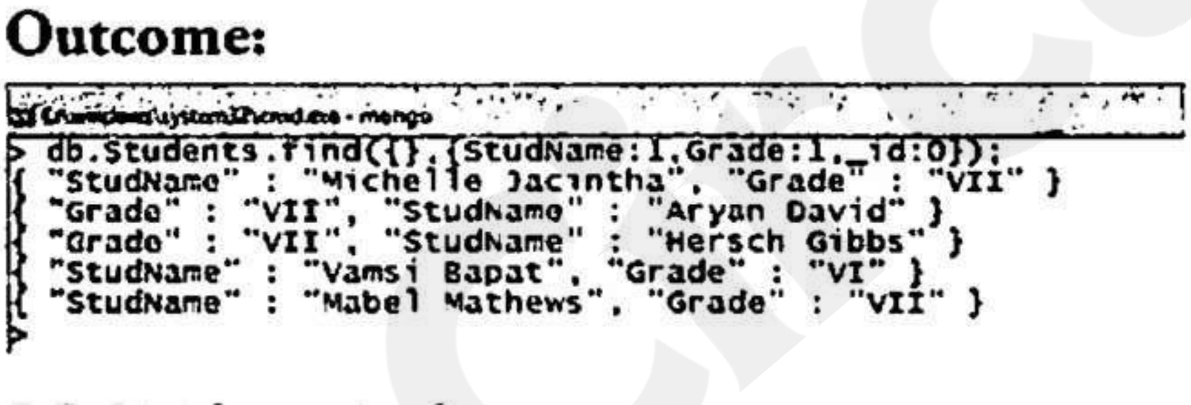
From Students;

Objective: To display only the StudName and Grade from all the documents of the Students collec- tion. The identifier _id should be suppressed and NOT displayed.

Act:

db.Students.find({}, {StudName:1,Grade: 1,_id:0});

Outcome:



Outcome:

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({},{StudName:1,Grade:1,_id:0});
{ "StudName" : "Michelle Jacintha", "Grade" : "VII" }
{ "Grade" : "VII", "StudName" : "Aryan David" }
{ "Grade" : "VII", "StudName" : "Hersch Gibbs" }
{ "StudName" : "Vamsi Bapat", "Grade" : "VI" }
{ "StudName" : "Mabel Mathews", "Grade" : "VII" }
>
```

RDBMS equivalent:

Select StudName, Grade

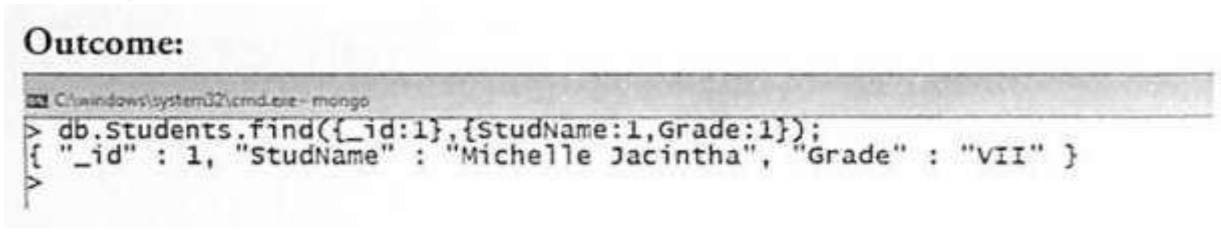
From Students;

Objective: To display the StudName, Grade as well the identifier, id from the document of the Students collection where the _id column is 1.

Act:

db.Students.find({_id:1},{StudName:1,Grade:1});

Outcome:



Outcome:

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({_id:1},{StudName:1,Grade:1});
{ "_id" : 1, "StudName" : "Michelle Jacintha", "Grade" : "VII" }
>
```

RDBMS equivalent:

Select StudRoll No, StudName, Grade

From Students

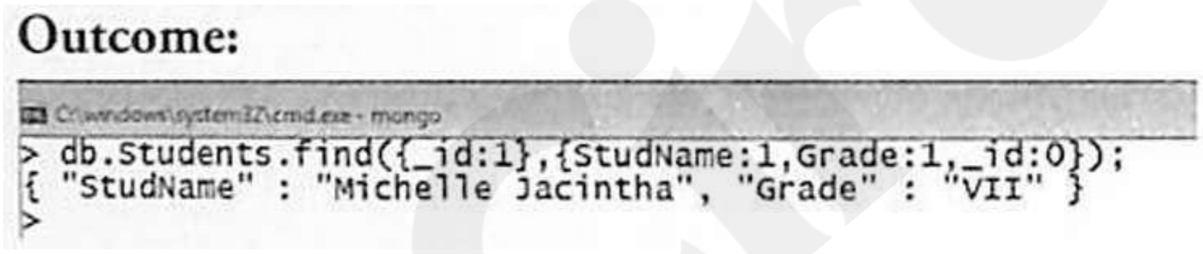
Where StudRollNo = '1';

Objective: To display the StudName and Grade from the document of the Students collection where the_id column is 1. The id field should NOT be displayed.

Act:

db.Students.find({_id:1}, {StudName:1,Grade:1,_id:0});

Outcome:



Outcome:

C:\windows\system32\cmd.exe - mongo

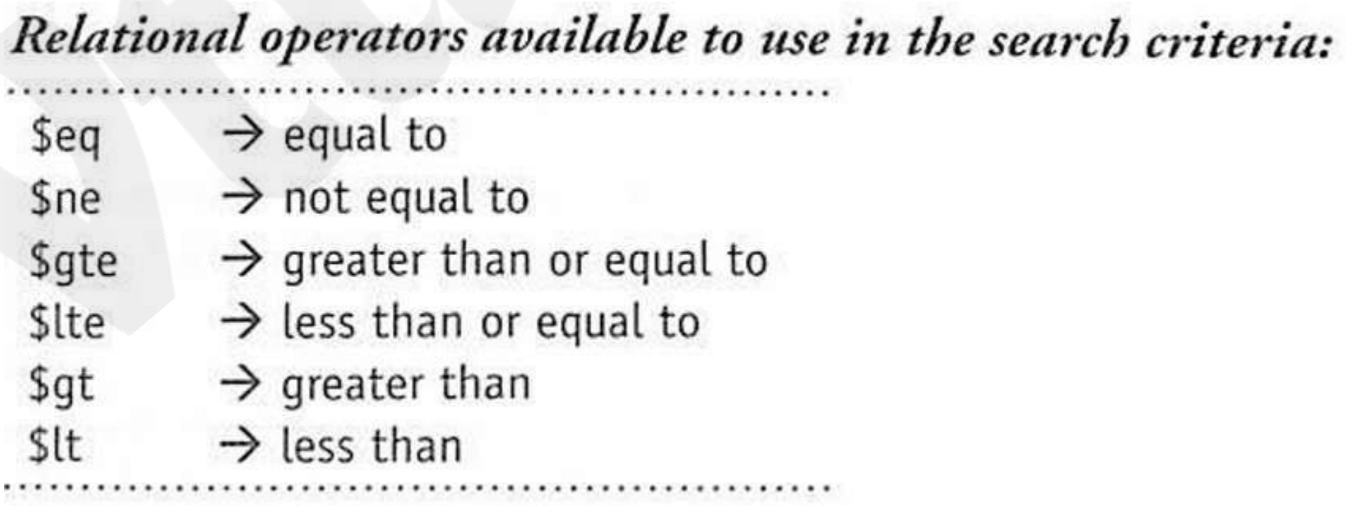
```
> db.Students.find({_id:1},{StudName:1,Grade:1,_id:0});
{ "StudName" : "Michelle Jacintha", "Grade" : "VII" }
>
```

RDBMS equivalent:

Select StudName, Grade

From Students

Where StudRollNo like '1';



Relational operators available to use in the search criteria:

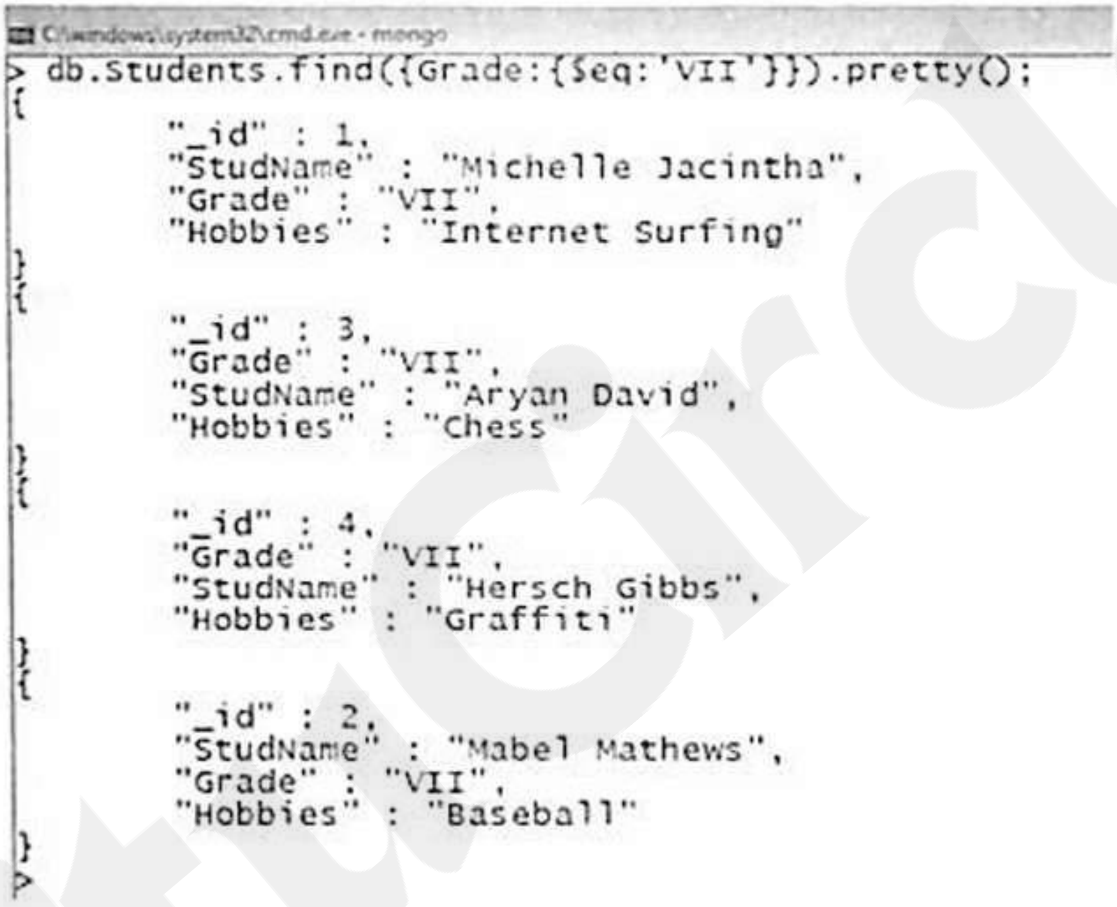
$eq	→ equal to
$ne	→ not equal to
$gte	→ greater than or equal to
$lte	→ less than or equal to
$gt	→ greater than
$lt	→ less than

Objective: To find those documents where the Grade is set to 'VII'.

Act:

db.Students.find({Grade: {$eq:'VII'}}).pretty();

Outcome:



```
> db.Students.find({Grade:{$eq:'VII'}}).pretty();
{
        "_id" : 1,
        "StudName" : "Michelle Jacintha",
        "Grade" : "VII",
        "Hobbies" : "Internet Surfing"
}
{
        "_id" : 3,
        "Grade" : "VII",
        "StudName" : "Aryan David",
        "Hobbies" : "Chess"
}
{
        "_id" : 4,
        "Grade" : "VII",
        "StudName" : "Hersch Gibbs",
        "Hobbies" : "Graffiti"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
>
```

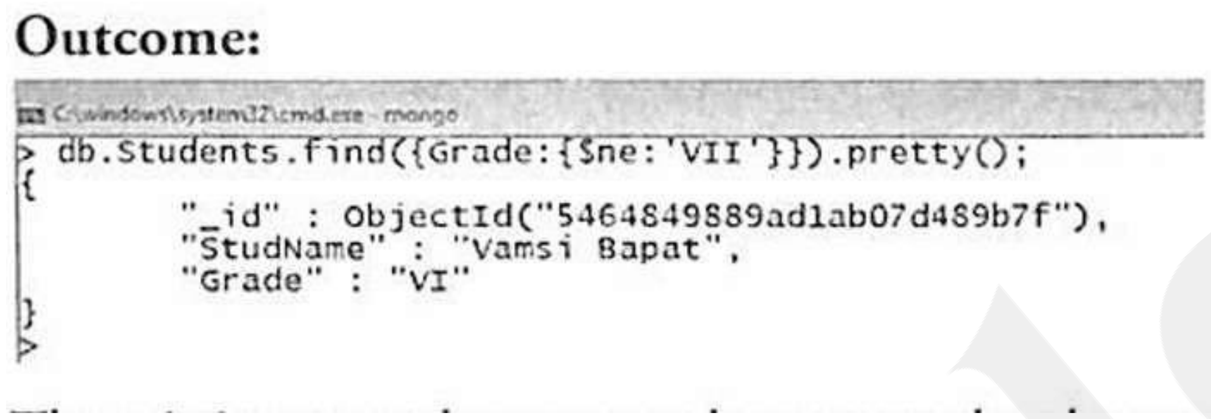
RDBMS Equivalent:

```
Select *
        From Students
                Where Grade like 'VII';
```

Objective: To find those documents where the Grade is NOT set to 'VII'.

Act:

db.Students.find({Grade: {$ne: 'VII'}}).pretty();


Outcome:

C:\windows\system32\cmd.exe - mongo

```
> db.Students.find({Grade:{$ne:'VII'}}).pretty();
{
        "_id" : ObjectId("5464849889ad1ab07d489b7f"),
        "StudName" : "Vamsi Bapat",
        "Grade" : "VI"
}
>
```

RDBMS Equivalent:

Select *

From Students

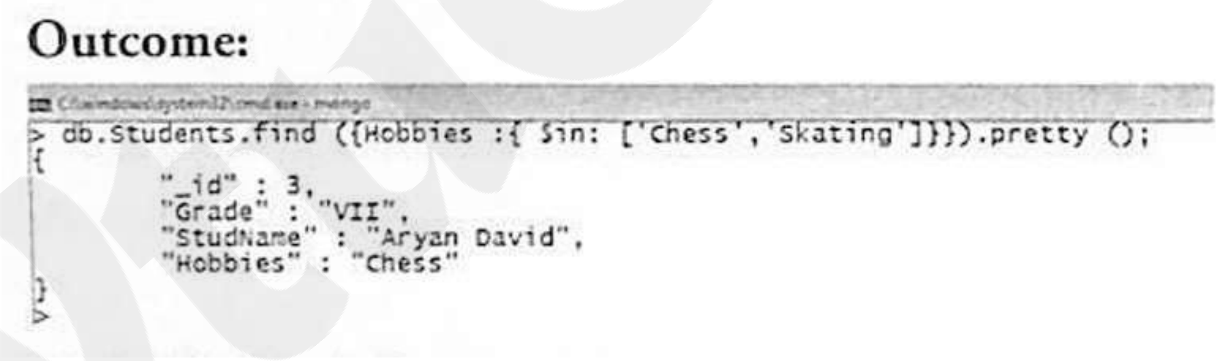
Where Grade <> 'VII';

Objective: To find those documents from the Students collection where the Hobbies is set to either 'Chess' or is set to 'Skating'.

Act:

db.Students.find ({Hobbies :{ $in: ['Chess', 'Skating']}}).pretty ();

Outcome:


Outcome:

C:\windows\system32\cmd.exe - mongo

```
> db.Students.find ({Hobbies :{ $in: ['Chess','Skating']}}).pretty ();
{
        "_id" : 3,
        "Grade" : "VII",
        "StudName" : "Aryan David",
        "Hobbies" : "Chess"
}
>
```

RDBMS Equivalent:

Select *

From Students

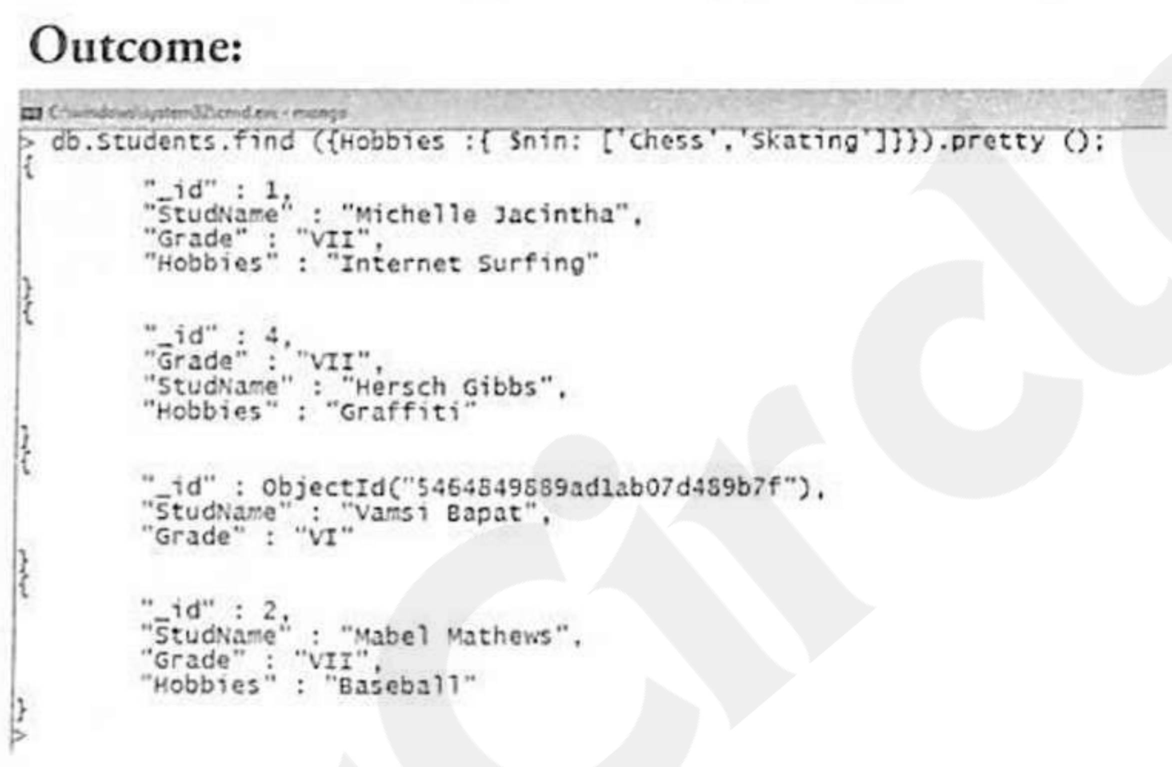
Where Hobbies in ('Chess', 'Skating');

Objective: To find those documents from the Students collection where the Hobbies is set neither to 'Chess' nor is set to 'Skating'.

Act:

db.Students.find({Hobbies :{ $nin: ['Chess','Skating']}}).pretty ();

Outcome:



Outcome:

```
> db.Students.find ({Hobbies :{ $nin: ['Chess','Skating']}}).pretty ();
{
        "_id" : 1,
        "StudName" : "Michelle Jacintha",
        "Grade" : "VII",
        "Hobbies" : "Internet Surfing"
}
{
        "_id" : 4,
        "Grade" : "VII",
        "StudName" : "Hersch Gibbs",
        "Hobbies" : "Graffiti"
}
{
        "_id" : ObjectId("5464849889ad1ab07d489b7f"),
        "StudName" : "Vamsi Bapat",
        "Grade" : "VI"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
>
```

RDBMS Equivalent:

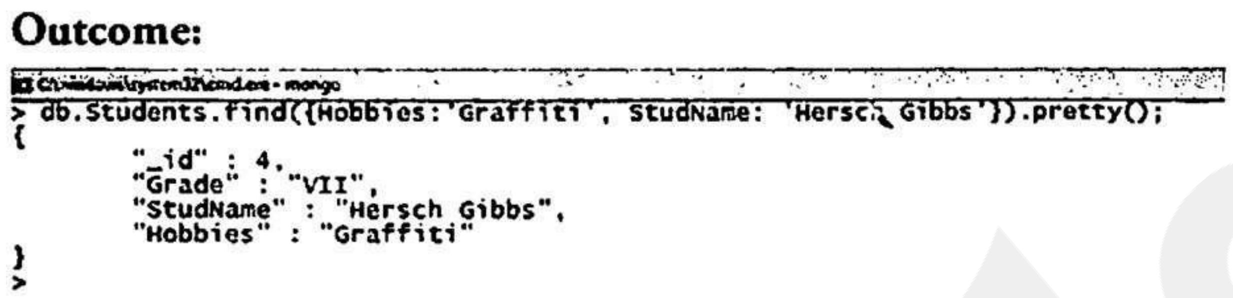
```
Select *
    From Students
        Where Hobbies not in ('Chess', 'Skating');
```

Objective: To find those documents from the Students collection where the Hobbies is set to 'Graffiti' and the StudName is set to 'Hersch Gibbs' (AND condition).

Act:

db.Students.find({Hobbies:'Graffiti', StudName: 'Hersch Gibbs'}).pretty();

Outcome:


Outcome:

```
> db.Students.find({Hobbies:'Graffiti', StudName: 'Hersch Gibbs'}).pretty();
{
        "_id" : 4,
        "Grade" : "VII",
        "StudName" : "Hersch Gibbs",
        "Hobbies" : "Graffiti"
}
>
```

RDBMS Equivalent:

Select *

From Students

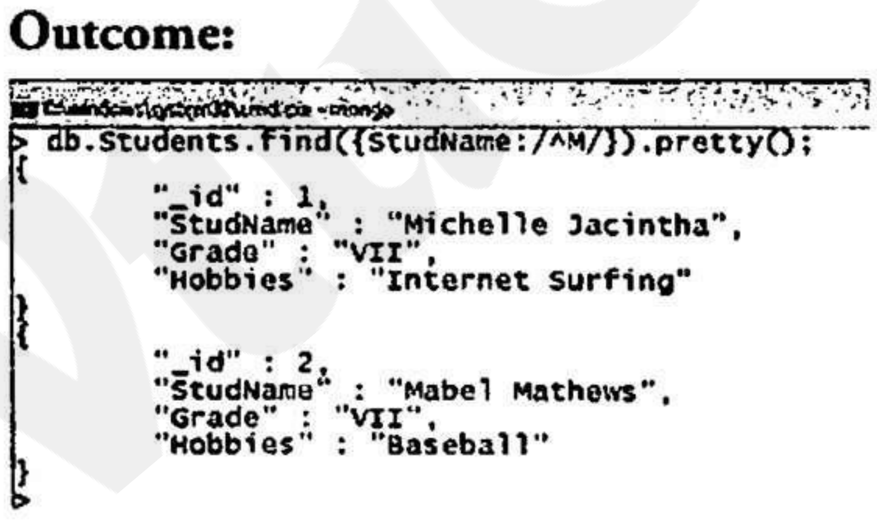
Where Hobbies like 'Graffiti' and StudName like 'Hersch Gibbs';

Objective: To find documents from the Students collection where the StudName begins with "M".

Act:

db.Students.find({StudName:/^M/}).pretty();

Outcome:


Outcome:

```
> db.Students.find({StudName:/^M/}).pretty();
{
        "_id" : 1,
        "StudName" : "Michelle Jacintha",
        "Grade" : "VII",
        "Hobbies" : "Internet Surfing"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
>
```

RDBMS Equivalent:

Select *

From Students

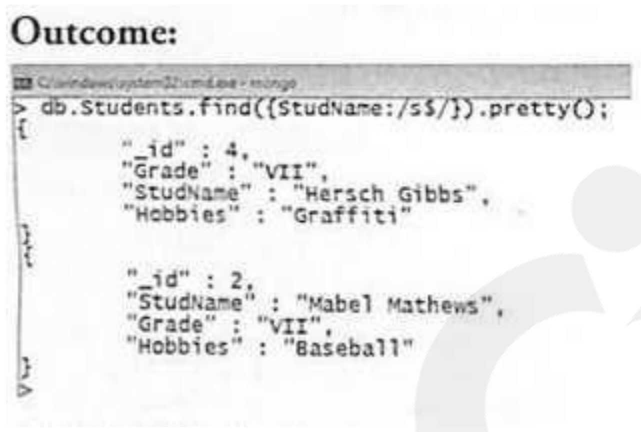
Where StudName like 'M%';

Objective: To find documents from the Students collection where the StudName ends in "s".

Act:

db.Students.find({StudName:/s$/}).pretty();

Outcome:



Outcome:

```
> db.Students.find({StudName:/s$/}).pretty();
{
        "_id" : 4,
        "Grade" : "VII",
        "StudName" : "Hersch Gibbs",
        "Hobbies" : "Graffiti"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
>
```

RDBMS Equivalent:

Select *

From Students

Where StudName like '%s';

Objective: To find documents from the Students collection where the StudName has an "e" in any position.

Act:

db.Students.find({StudName:/e/}).pretty();

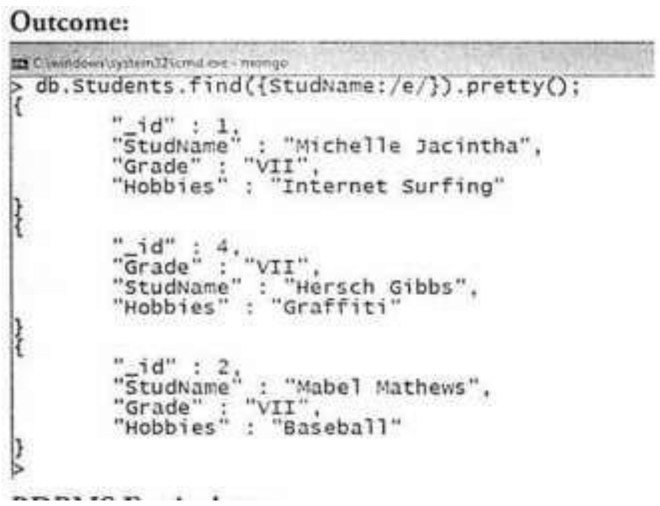
OR

db.Students.find({StudName:/.*e.*/}).pretty();

OR

db.Students.find({StudName: {$regex:"e"}}).pretty();

Outcome:



Outcome:

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({StudName:/e/}).pretty();
{
        "_id" : 1,
        "StudName" : "Michelle Jacintha",
        "Grade" : "VII",
        "Hobbies" : "Internet Surfing"
}
{
        "_id" : 4,
        "Grade" : "VII",
        "StudName" : "Hersch Gibbs",
        "Hobbies" : "Graffiti"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
>
```

RDBMS Equivalent:

Select *

From Students

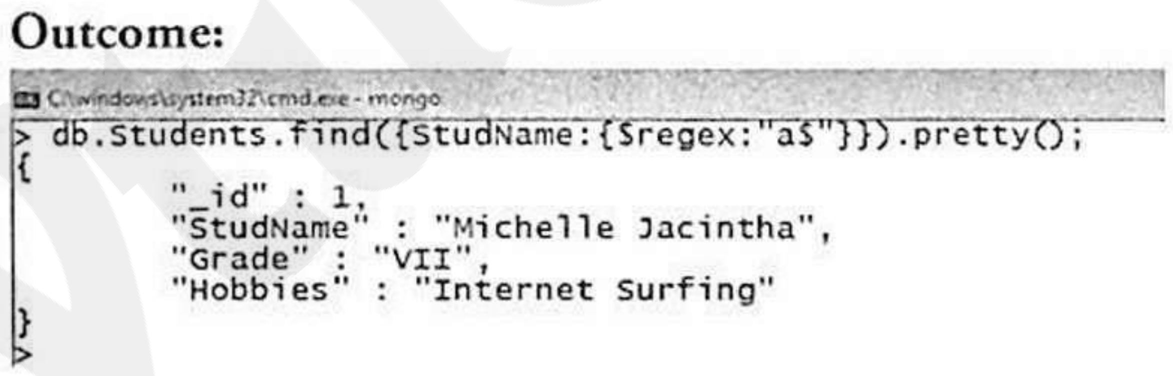
Where StudName like '%e%';

Objective: To find documents from the Students collection where the StudName ends in "a".

Act:

db.Students.find({StudName: {$regex:"a$"}}).pretty();

Outcome:



Outcome:

```
C:\windows\system32\cmd.exe - mongo
> db.Students.find({StudName:{$regex:"a$"}}).pretty();
{
        "_id" : 1,
        "StudName" : "Michelle Jacintha",
        "Grade" : "VII",
        "Hobbies" : "Internet Surfing"
}
>
```

RDBMS Equivalent:

Select *

From Students

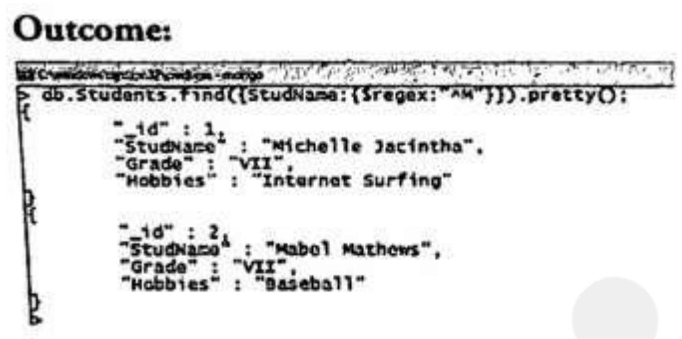
Where StudName like "%a";

Objective: To find documents from the Students collection where the StudName begins with "M".

Act:

db.Students.find({StudName:{$regex:"^M"}}).pretty();

Outcome:



Outcome:

```
> db.Students.find({StudName:{$regex:"^M"}}).pretty();
{
        "_id" : 1,
        "StudName" : "Michelle Jacintha",
        "Grade" : "VII",
        "Hobbies" : "Internet Surfing"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
>
```

RDBMS Equivalent:

```
Select *
        From Students
                Where StudName like 'M%';
```

3.5.6 Dealing with NULL Values

Objective:

To add or manage a field (Location) with a NULL value in documents of the Students collection.

- NULL indicates a missing or unknown value.
- This is useful when we don't know the value at the moment but may update it later.

Step 1: Viewing Existing Documents

To view specific documents before updating:

db.Students.find({$or: [{_id: 3}, {_id: 4}]})

Step 2: Adding NULL Values

To insert a NULL value in the "Location" field:

db.Students.update({_id: 3}, {$set: {Location: null}});

db.Students.update({_id: 4}, {$set: {Location: null}});

RDBMS Equivalent:

UPDATE Students SET Location = NULL WHERE StudRollNo IN (3, 4);

Step 3: Searching for NULL Values

To find documents where Location is NULL or does not exist:

db.Students.find({Location: {$eq: null}});

RDBMS Equivalent:

SELECT * FROM Students WHERE Location IS NULL;

Step 4: Removing Fields with NULL Values

To **remove the Location field** where it's NULL:

db.Students.update({_id: 3}, {$unset: {Location: null}});

db.Students.update({_id: 4}, {$unset: {Location: null}});

Step 5: Confirming the Change

To verify that the fields have been removed:

db.Students.find()

- NULL values can be assigned using $set.
- NULL fields can be removed using $unset.
- Documents with NULL or missing fields can be queried with $eq: null.

3.5.7 Count, Limit, Sort, and Skip

Objective: To find the number of documents in the Students collection.

Act:

db.Students.count()

Objective: To find the number of documents in the Students collection wherein the Grade is VII.

Act:

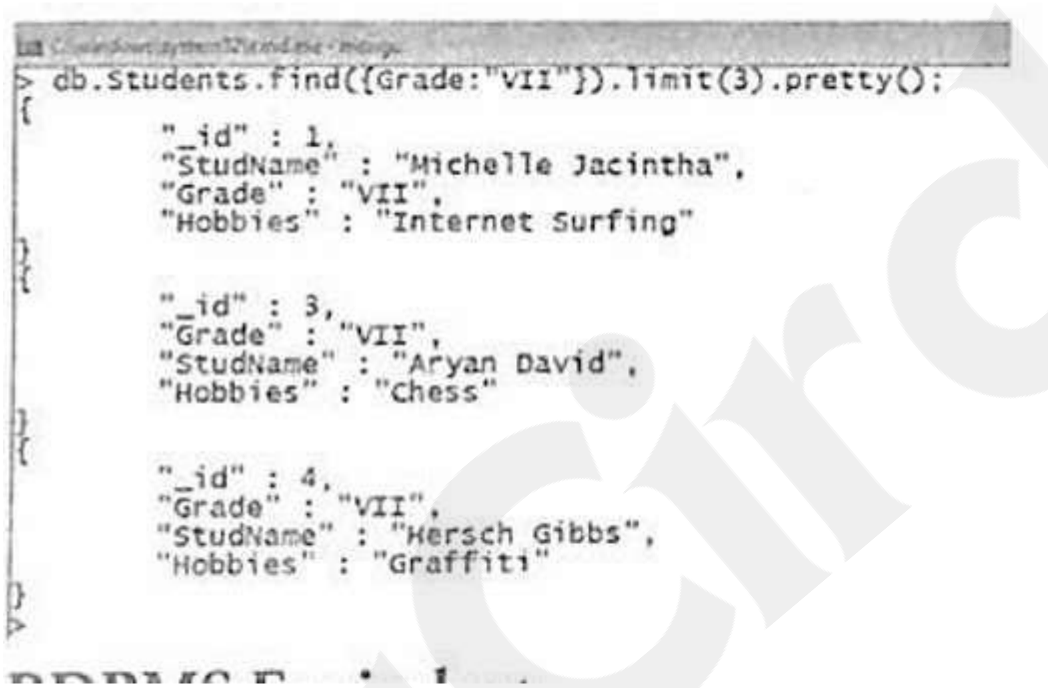
db.Students.count({Grade:"VII"});

Objective: To retrieve the first 3 documents from the Students collection wherein the Grade is VII.

Act:

db.Students.find({Grade:"VII"}).limit(3).pretty();

Outcome:



```
> db.Students.find({Grade:"VII"}).limit(3).pretty();
{
        "_id" : 1,
        "StudName" : "Michelle Jacintha",
        "Grade" : "VII",
        "Hobbies" : "Internet Surfing"
}
{
        "_id" : 3,
        "Grade" : "VII",
        "StudName" : "Aryan David",
        "Hobbies" : "Chess"
}
{
        "_id" : 4,
        "Grade" : "VII",
        "StudName" : "Hersch Gibbs",
        "Hobbies" : "Graffiti"
}
>
```

RDBMS Equivalent:

Select *

From Students

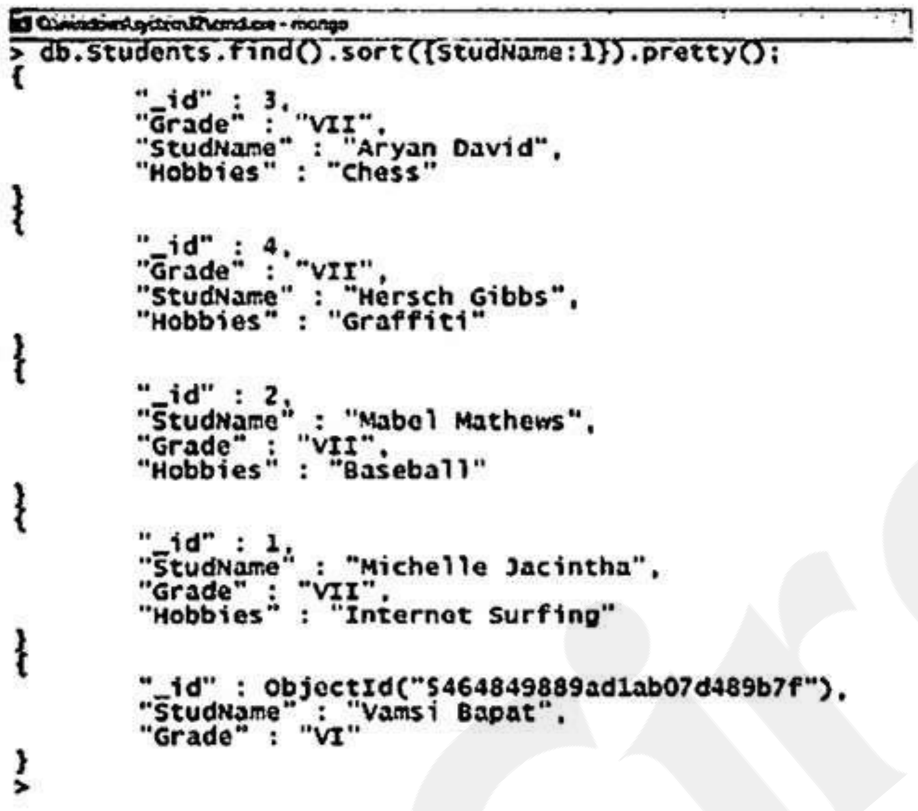
Where Grade like "VII' and rownum <4;

Objective: To sort the documents from the Students collection in the ascending order of StudName.

Act:

db.Students.find().sort({StudName:1}).pretty();

Outcome:



```
> db.Students.find().sort({StudName:1}).pretty();
{
        "_id" : 3,
        "Grade" : "VII",
        "StudName" : "Aryan David",
        "Hobbies" : "Chess"
}
{
        "_id" : 4,
        "Grade" : "VII",
        "StudName" : "Hersch Gibbs",
        "Hobbies" : "Graffiti"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
{
        "_id" : 1,
        "StudName" : "Michelle Jacintha",
        "Grade" : "VII",
        "Hobbies" : "Internet Surfing"
}
{
        "_id" : ObjectId("5464849889ad1ab07d489b7f"),
        "StudName" : "Vamsi Bapat",
        "Grade" : "VI"
}
>
```

RDBMS Equivalent:

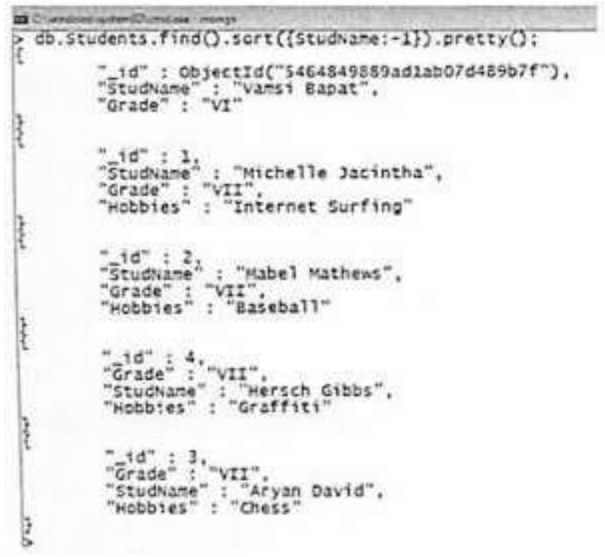
```
Select *
        From Students
                Order by StudName asc;
```

Objective: To sort the documents from the Students collection in the descending order of StudName.

Act:

db.Students.find().sort((StudName:-1}).pretty();

Outcome:



```
> db.Students.find().sort({StudName:-1}).pretty();
{
        "_id" : ObjectId("5464849889ad1ab07d489b7f"),
        "StudName" : "Vamsi Bapat",
        "Grade" : "VI"
}
{
        "_id" : 1,
        "StudName" : "Michelle Jacintha",
        "Grade" : "VII",
        "Hobbies" : "Internet Surfing"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
{
        "_id" : 4,
        "Grade" : "VII",
        "StudName" : "Hersch Gibbs",
        "Hobbies" : "Graffiti"
}
{
        "_id" : 3,
        "Grade" : "VII",
        "StudName" : "Aryan David",
        "Hobbies" : "Chess"
}
>
```

RDBMS Equivalent:

Select *

From Students

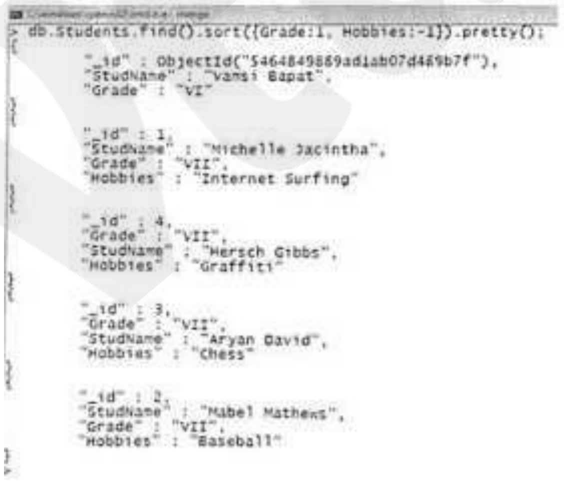
Order by StudName desc;

Objective: To sort the documents from the Students collection first on Grade in ascending order and then on Hobbies in descending order.

Act:

db.Students.find().sort((Grade:1,   Hobbies:-1)).pretty();

Outcome:



```
> db.Students.find().sort({Grade:1, Hobbies:-1}).pretty();
{
        "_id" : ObjectId("5464849889ad1ab07d489b7f"),
        "StudName" : "Vamsi Bapat",
        "Grade" : "VI"
}
{
        "_id" : 1,
        "StudName" : "Michelle Jacintha",
        "Grade" : "VII",
        "Hobbies" : "Internet Surfing"
}
{
        "_id" : 4,
        "Grade" : "VII",
        "StudName" : "Hersch Gibbs",
        "Hobbies" : "Graffiti"
}
{
        "_id" : 3,
        "Grade" : "VII",
        "StudName" : "Aryan David",
        "Hobbies" : "Chess"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
>
```

RDBMS Equivalent:

Select *

From Students

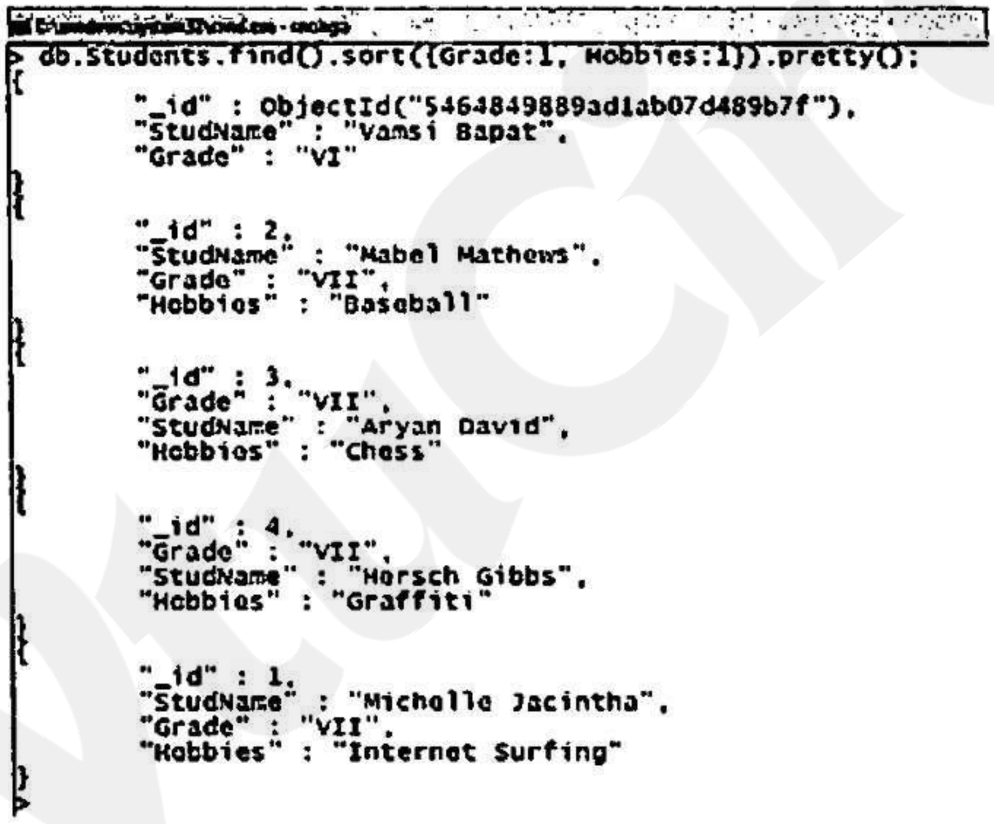
Order by Grade asc, hobbies desc;

Objective: To sort the documents from the Students collection first on Grade in ascending order and then on Hobbies in ascending order.

Act:

db.Students.find().sort((Grade:1,   Hobbies:1}).pretty();

Outcome:



```
> db.Students.find().sort({Grade:1, Hobbies:1}).pretty();
{
        "_id" : ObjectId("5464849889ad1ab07d489b7f"),
        "StudName" : "Vamsi Bapat",
        "Grade" : "VI"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
{
        "_id" : 3,
        "Grade" : "VII",
        "StudName" : "Aryan David",
        "Hobbies" : "Chess"
}
{
        "_id" : 4,
        "Grade" : "VII",
        "StudName" : "Hersch Gibbs",
        "Hobbies" : "Graffiti"
}
{
        "_id" : 1,
        "StudName" : "Michelle Jacintha",
        "Grade" : "VII",
        "Hobbies" : "Internet Surfing"
}
>
```

RDBMS Equivalent:

Select *

From Students

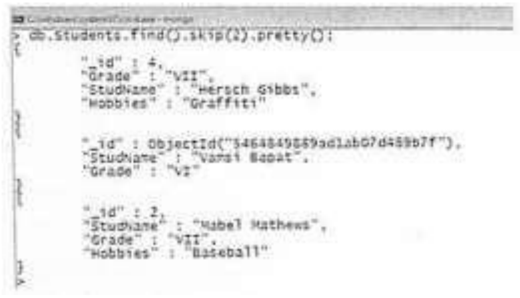
Order by Grade asc, Hobbies asc;

Objective: To skip the first 2 documents from the Students collection.

Act:

db.Students.find().skip (2).pretty();

Outcome:



```
> db.Students.find().skip(2).pretty();
{
        "_id" : 4,
        "Grade" : "VII",
        "StudName" : "Hersch Gibbs",
        "Hobbies" : "Graffiti"
}
{
        "_id" : ObjectId("5464849589ad1ab07d489b7f"),
        "StudName" : "Vamsi Bapat",
        "Grade" : "VI"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
>
```

RDBMS Equivalent:

Select StudRollNo, StudName, Grade, Hobbies

From (Select StudRollNo, StudName, Grade, Hobbies, RowNum as

TheRowNum From Students)

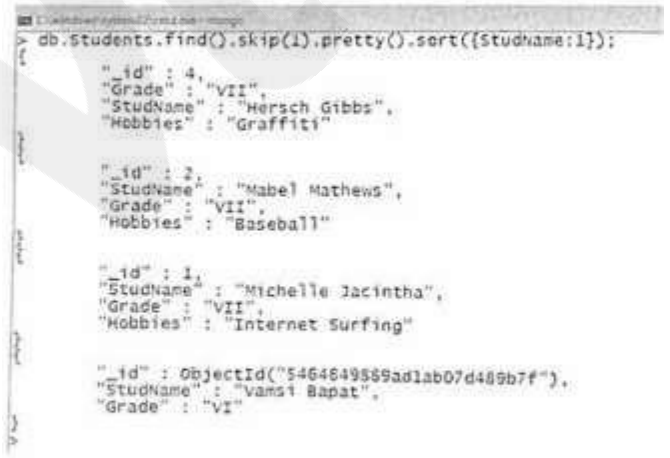
Where TheRowNum > 2;

Objective: To sort the documents from the Students collection and skip the first document from the output.

Act:

db.Students.find().skip (1).pretty().sort({StudName:1});

Outcome:



```
> db.Students.find().skip(1).pretty().sort({StudName:1});
{
        "_id" : 4,
        "Grade" : "VII",
        "StudName" : "Hersch Gibbs",
        "Hobbies" : "Graffiti"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
{
        "_id" : 1,
        "StudName" : "Michelle Jacintha",
        "Grade" : "VII",
        "Hobbies" : "Internet Surfing"
}
{
        "_id" : ObjectId("5464849589ad1ab07d489b7f"),
        "StudName" : "Vamsi Bapat",
        "Grade" : "VI"
}
>
```

RDBMS Equivalent:

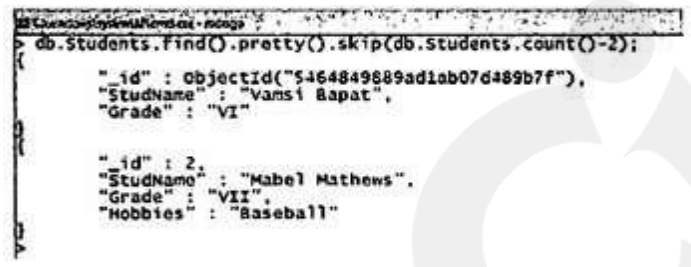
```
Select StudRollNo, StudName, Grade, Hobbies
        From (Select Stud RollNo, StudName, Grade, Hobbies, RowNum as
                TheRowNum From Students)
                        Where TheRowNum > 1
                                Order by StudName;
```

Objective: To display the last 2 records from the Students collection.

Act:

db.Students.find().pretty().skip(db.Students.count()-2);

Outcome:



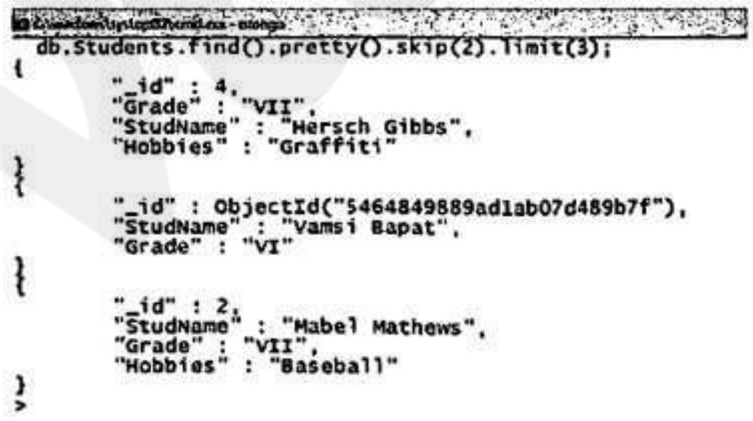
```
> db.Students.find().pretty().skip(db.Students.count()-2);
{
        "_id" : ObjectId("5464849889ad1ab07d489b7f"),
        "StudName" : "Vamsi Bapat",
        "Grade" : "VI"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
>
```

Objective: To retrieve the third, fourth, and fifth document from the Students collection.

Act:

db.Students.find().pretty().skip(2).limit(3);

Outcome:



```
> db.Students.find().pretty().skip(2).limit(3);
{
        "_id" : 4,
        "Grade" : "VII",
        "StudName" : "Hersch Gibbs",
        "Hobbies" : "Graffiti"
}
{
        "_id" : ObjectId("5464849889ad1ab07d489b7f"),
        "StudName" : "Vamsi Bapat",
        "Grade" : "VI"
}
{
        "_id" : 2,
        "StudName" : "Mabel Mathews",
        "Grade" : "VII",
        "Hobbies" : "Baseball"
}
>
```

3.5.8 Arrays

Objective: To create a collection by the name "food" and then insert documents into the "food" collection. Each document should have a "fruits" array.

Act:

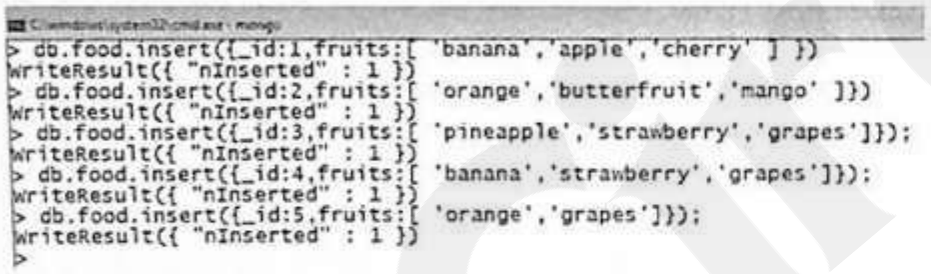
db.food.insert({_id:1,fruits:['banana','apple', 'cherry'] })

db.food.insert({_id:2,fruits:['orange','butterfruit','mango']})

db.food.insert({_id:3,fruits:['pineapple', 'strawberry','grapes']});

db.food.insert({_id:4,fruits:['banana', 'strawberry','grapes']});

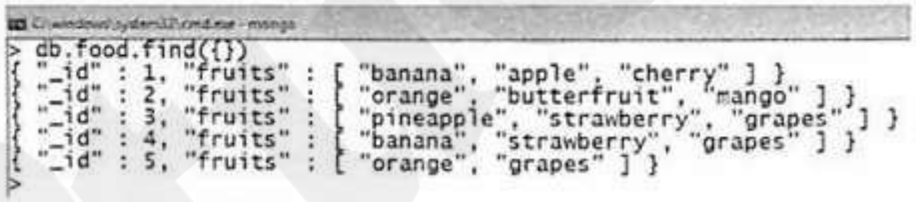
db.food.insert((_id:5,fruits: ['orange','grapes']});



```
> db.food.insert({_id:1,fruits:[ 'banana','apple','cherry' ] })
WriteResult({ "nInserted" : 1 })
> db.food.insert({_id:2,fruits:[ 'orange','butterfruit','mango' ]})
WriteResult({ "nInserted" : 1 })
> db.food.insert({_id:3,fruits:[ 'pineapple','strawberry','grapes']});
WriteResult({ "nInserted" : 1 })
> db.food.insert({_id:4,fruits:[ 'banana','strawberry','grapes']});
WriteResult({ "nInserted" : 1 })
> db.food.insert({_id:5,fruits:[ 'orange','grapes']});
WriteResult({ "nInserted" : 1 })
>
```

Outcome: Let us check if these documents are now in the "food" collection.

db.food.find({})



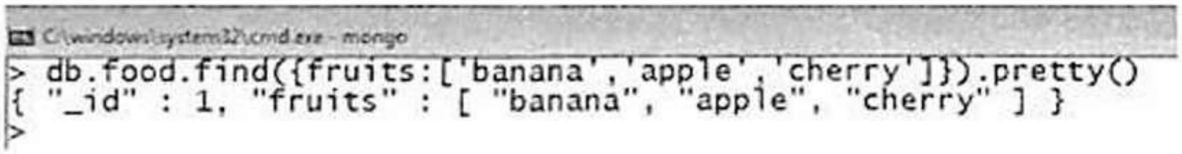
```
> db.food.find({})
{ "_id" : 1, "fruits" : [ "banana", "apple", "cherry" ] }
{ "_id" : 2, "fruits" : [ "orange", "butterfruit", "mango" ] }
{ "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
{ "_id" : 4, "fruits" : [ "banana", "strawberry", "grapes" ] }
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
>
```

Objective: To find those documents from the "food" collection which has the "fruits array" constituted of "banana", "apple" and "cherry".

Act:

db.food.find({fruits: ['banana','apple', 'cherry']}).pretty()

Outcome:



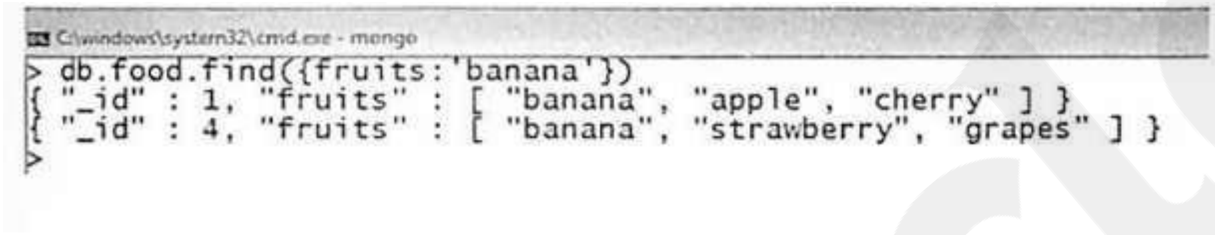
```
> db.food.find({fruits:['banana','apple','cherry']}).pretty()
{ "_id" : 1, "fruits" : [ "banana", "apple", "cherry" ] }
>
```

Objective: To find those documents from the "food" collection which has the "fruits" array having "banana", as an element.

Act:

db.food.find({fruits:'banana'})

Outcome:



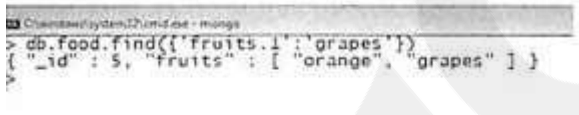
```
C:\windows\system32\cmd.exe - mongo
> db.food.find({fruits:'banana'})
{ "_id" : 1, "fruits" : [ "banana", "apple", "cherry" ] }
{ "_id" : 4, "fruits" : [ "banana", "strawberry", "grapes" ] }
>
```

Objective: To find those documents from the "food" collection which have the "fruits" array having "grapes" in the first index position. The index position begins at 0.

Act:

db.food.find({'fruits. 1':'grapes'})

Outcome:



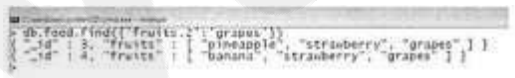
```
> db.food.find({'fruits.1':'grapes'})
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
>
```

Objective: To find those documents from the "food" collection where "grapes" is present in the 2nd index position of the "fruits" array.

Act:

db.food.find({'fruits.2':'grapes'})

Outcome:



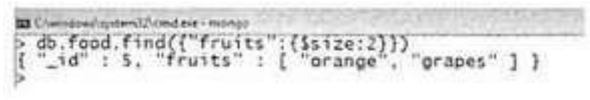
```
> db.food.find({'fruits.2':'grapes'})
{ "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
{ "_id" : 4, "fruits" : [ "banana", "strawberry", "grapes" ] }
>
```

Objective: To find those documents from the "food" collection where the size of the array is two. The size implies that the array holds only 2 values.

Act:

db.food.find({"fruits":{$size:2}})

Outcome:



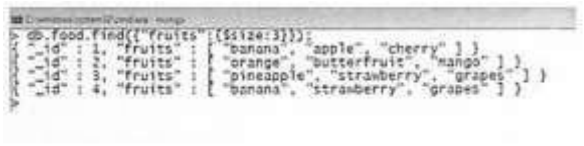
```
> db.food.find({"fruits":{$size:2}})
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
>
```

Objective: To find those documents from the "food" collection where the size of the array is three. The size implies that the array holds only 3 values.

Act:

db.food.find({"fruits":{$size:3}})

Outcome:



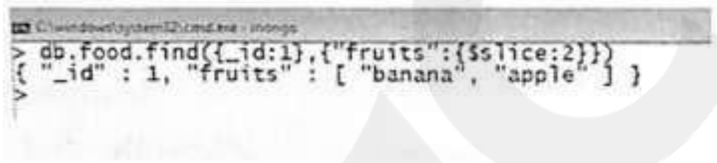
```
> db.food.find({"fruits":{$size:3}});
{ "_id" : 1, "fruits" : [ "banana", "apple", "cherry" ] }
{ "_id" : 2, "fruits" : [ "orange", "butterfruit", "mango" ] }
{ "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
{ "_id" : 4, "fruits" : [ "banana", "strawberry", "grapes" ] }
>
```

Objective: To find the document with (id: 1) from the "food" collection and display the first two elements from the array "fruits".

Act:

db.food.find({_id:1},{"fruits":{$slice:2}})

Outcome:



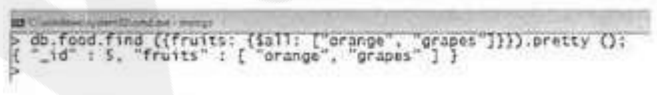
```
> db.food.find({_id:1},{"fruits":{$slice:2}})
{ "_id" : 1, "fruits" : [ "banana", "apple" ] }
>
```

Objective: To find all documents from the "food" collection which have elements "orange" and "grapes" in the array "fruits".

Act:

db.food.find ({fruits: {$all: ["orange", "grapes"]}}).pretty ();

Outcome:



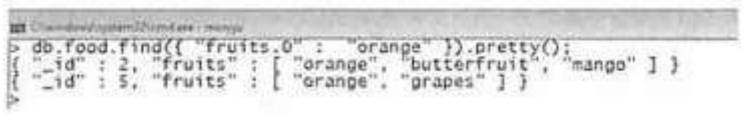
```
> db.food.find ({fruits: {$all: ["orange", "grapes"]}}).pretty ();
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
>
```

Objective: To find those documents from the "food" collection which have the element "orange" in the 0th index position in the array "fruits".

Act:

db.food.find({ "fruits.0" : "orange" }).pretty();

Outcome:



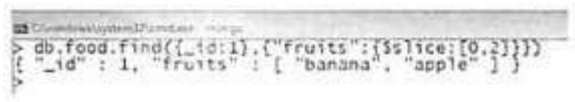
```
> db.food.find({ "fruits.0" : "orange" }).pretty();
{ "_id" : 2, "fruits" : [ "orange", "butterfruit", "mango" ] }
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
>
```

Objective: To find the document with (id: 1) from the "food" collection and display two elements from the array "fruits", starting with the element at 0th index position.

Act:

db.food.find({id:1},{"fruits": {$slice: [0,2]}})

Outcome:



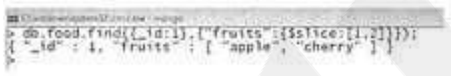
```
> db.food.find({_id:1},{"fruits":{$slice:[0,2]}})
{ "_id" : 1, "fruits" : [ "banana", "apple" ] }
>
```

Objective: To find the document with (id: 1) from the "food" collection and display two elements from the array "fruits", starting with the element at 1" index position.

Act:

db.food.find({_id:1},{"fruits": {$slice:[1,2]}})

Outcome:



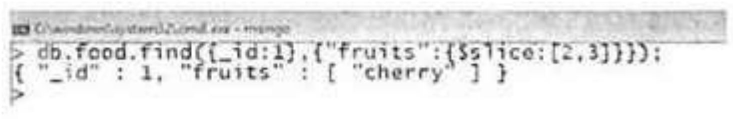
```
> db.food.find({_id:1},{"fruits":{$slice:[1,2]}});
{ "_id" : 1, "fruits" : [ "apple", "cherry" ] }
>
```

Objective: To find the document with (id: 1) from the "food" collection and display three elements from the array "fruits", starting with the element at 2nd index position. Since we have only 3 elements in the array "fruits" for the document with _id:1, it displays only one element, the element at 2nd index position, that is, "cherry".

Act:

db.food.find({_id:1},{"fruits": {$slice: [2,3]}})

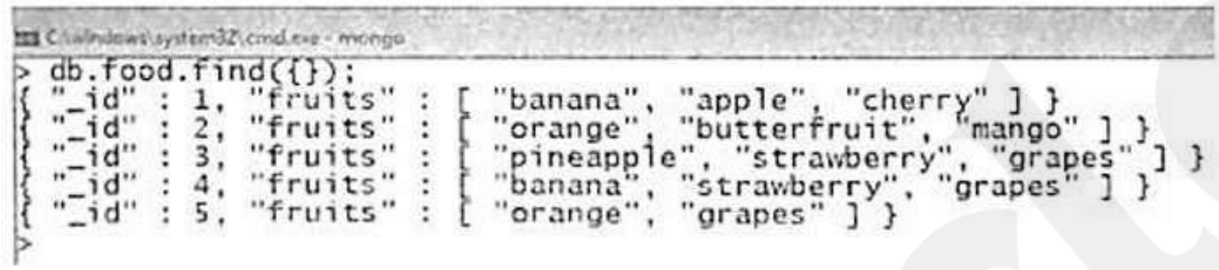
Outcome:



```
> db.food.find({_id:1},{"fruits":{$slice:[2,3]}});
{ "_id" : 1, "fruits" : [ "cherry" ] }
>
```

3.5.8.1 Update on the Array

Before we begin the update operations on the "fruits" array of the documents of "food" collection, let us take a look at the documents that we have in the "food" collection:

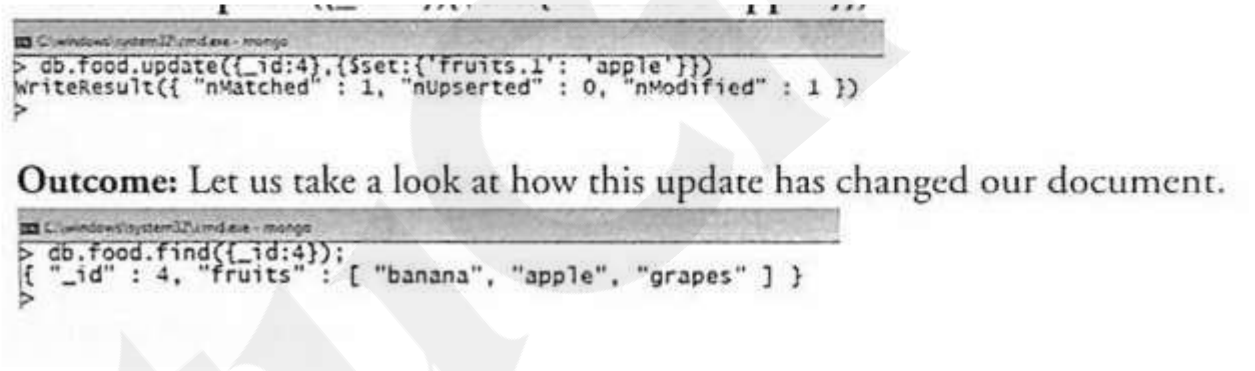


```
C:\windows\system32\cmd.exe - mongo
> db.food.find({});
{ "_id" : 1, "fruits" : [ "banana", "apple", "cherry" ] }
{ "_id" : 2, "fruits" : [ "orange", "butterfruit", "mango" ] }
{ "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
{ "_id" : 4, "fruits" : [ "banana", "strawberry", "grapes" ] }
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
>
```

Objective: To update the document with "_id:4" and replace the element present in the 1st index position of the "fruits" array with "apple".

Act:

db.food.update({_id:4}, {$set:{'fruits.1': 'apple'}})



```
C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:4},{$set:{'fruits.1': 'apple'}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

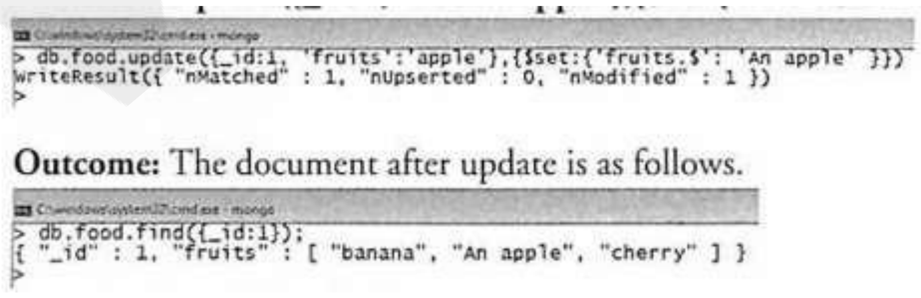
Outcome: Let us take a look at how this update has changed our document.

```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:4});
{ "_id" : 4, "fruits" : [ "banana", "apple", "grapes" ] }
>
```

Objective: To update the document with "_id:1" and replace the element "apple" of the "fruits" array with "An apple".

Act:

db.food.update({_id:1, 'fruits':'apple'}, {$set: {'fruits.$': 'An apple' }})



```
C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:1, 'fruits':'apple'},{$set:{'fruits.$': 'An apple' }})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

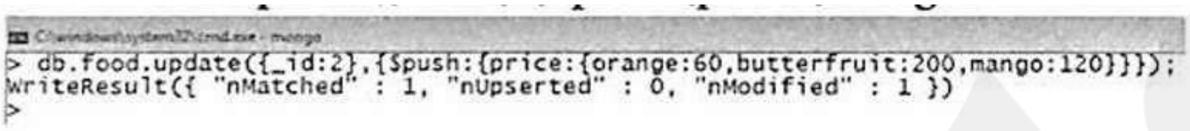
Outcome: The document after update is as follows.

```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:1});
{ "_id" : 1, "fruits" : [ "banana", "An apple", "cherry" ] }
>
```

Objective: To update the document with "_id:2" and push new key value pairs in the "fruits" array.

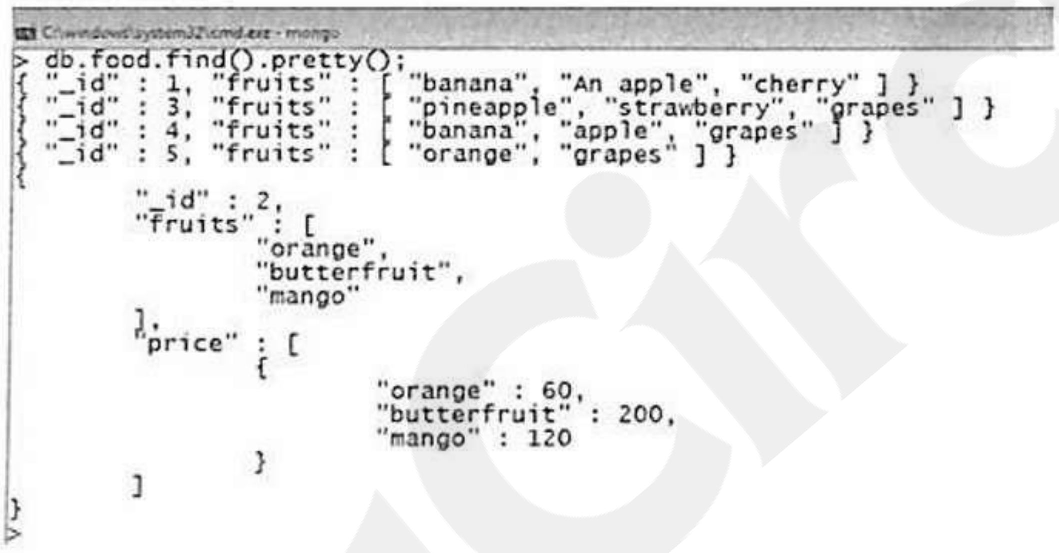
Act:

db.food.update({_id:2},{$push:{price:{orange:60,butterfruit:200,mango: 120}}})



```
> db.food.update({_id:2},{$push:{price:{orange:60,butterfruit:200,mango:120}}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

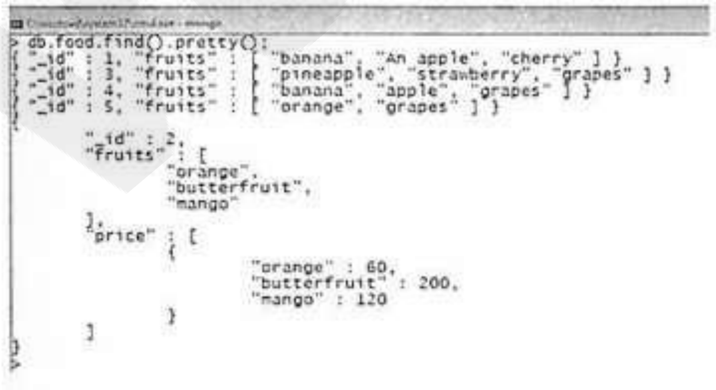
Outcome:



```
> db.food.find().pretty();
{ "_id" : 1, "fruits" : [ "banana", "An apple", "cherry" ] }
{ "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
{ "_id" : 4, "fruits" : [ "banana", "apple", "grapes" ] }
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
{
        "_id" : 2,
        "fruits" : [
                "orange",
                "butterfruit",
                "mango"
        ],
        "price" : [
                {
                        "orange" : 60,
                        "butterfruit" : 200,
                        "mango" : 120
                }
        ]
}
>
```

3.5.8.2 Further Updates to the Array "fruits" ...

Before we do the updates to the documents in the food collection, let us look at the current state:

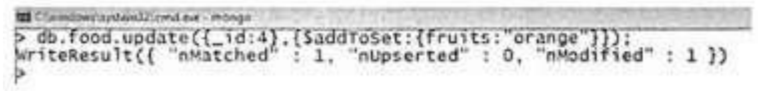


```
> db.food.find().pretty();
{ "_id" : 1, "fruits" : [ "banana", "An apple", "cherry" ] }
{ "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
{ "_id" : 4, "fruits" : [ "banana", "apple", "grapes" ] }
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
{
        "_id" : 2,
        "fruits" : [
                "orange",
                "butterfruit",
                "mango"
        ],
        "price" : [
                {
                        "orange" : 60,
                        "butterfruit" : 200,
                        "mango" : 120
                }
        ]
}
>
```

Objective: To update the document with "_id:4" by adding an element "orange" to the list of elements in the array "fruits".

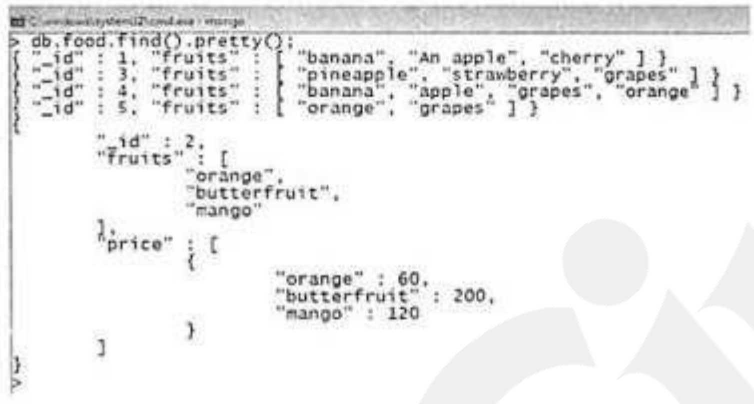
Act:

db.food.update({_id:4}, {$addToSet: {fruits:"orange"}});



```
C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:4},{$addToSet:{fruits:"orange"}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

Outcome: The result after the execution of the statement is as follows.

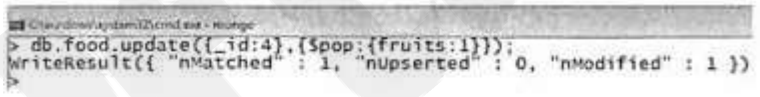


```
C:\windows\system32\cmd.exe - mongo
> db.food.find().pretty();
{ "_id" : 1, "fruits" : [ "banana", "An apple", "cherry" ] }
{ "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
{ "_id" : 4, "fruits" : [ "banana", "apple", "grapes", "orange" ] }
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
{
        "_id" : 2,
        "fruits" : [
                "orange",
                "butterfruit",
                "mango"
        ],
        "price" : [
                {
                        "orange" : 60,
                        "butterfruit" : 200,
                        "mango" : 120
                }
        ]
}
>
```

Objective: To update the document with "_id:4" by popping an element from the list of elements present in the array "fruits". The element popped is the one from the end of the array.

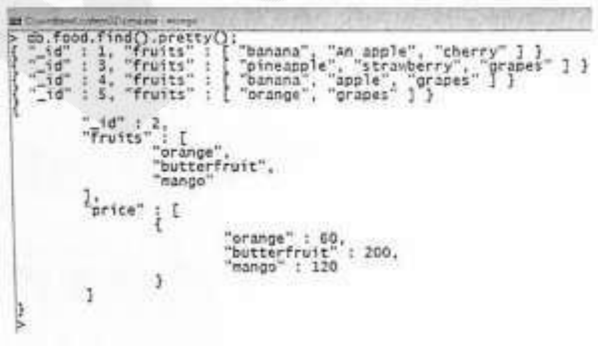
Act:

db.food.update({_id:4},{$pop: {fruits:1}});



```
C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:4},{$pop:{fruits:1}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

Outcome: The "food" collection after the execution of the statement is as follows.

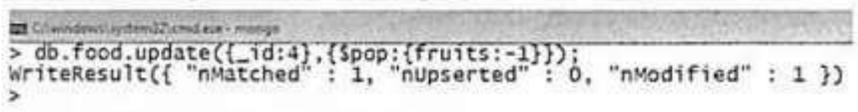


```
C:\windows\system32\cmd.exe - mongo
> db.food.find().pretty();
{ "_id" : 1, "fruits" : [ "banana", "An apple", "cherry" ] }
{ "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
{ "_id" : 4, "fruits" : [ "banana", "apple", "grapes" ] }
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
{
        "_id" : 2,
        "fruits" : [
                "orange",
                "butterfruit",
                "mango"
        ],
        "price" : [
                {
                        "orange" : 60,
                        "butterfruit" : 200,
                        "mango" : 120
                }
        ]
}
>
```

Objective: To update the document with "_id:4" by popping an element from the list of elements present in the array "fruits". The element popped is the one from the beginning of the:

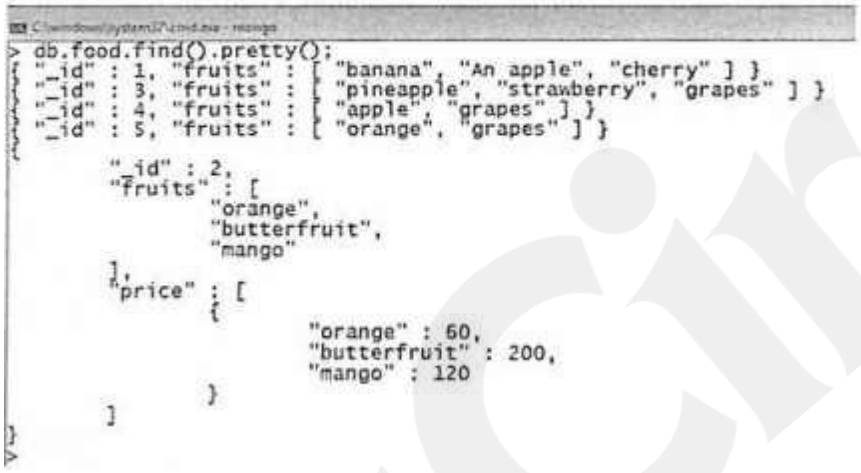
Act:

db.food.update({_id:4}, {$pop:{fruits:-1}});



```
C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:4},{$pop:{fruits:-1}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

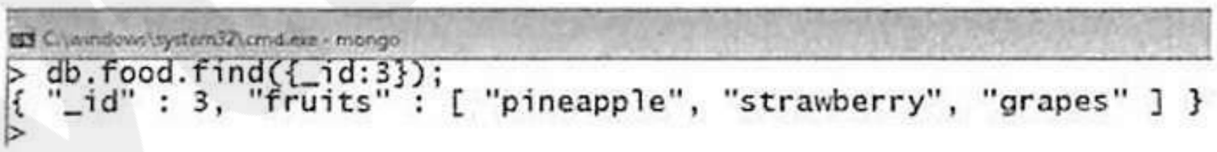
Outcome: The "food" collection after the execution of the above update statement is as follows.



```
C:\windows\system32\cmd.exe - mongo
> db.food.find().pretty();
{ "_id" : 1, "fruits" : [ "banana", "An apple", "cherry" ] }
{ "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
{ "_id" : 4, "fruits" : [ "apple", "grapes" ] }
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
{
        "_id" : 2,
        "fruits" : [
                "orange",
                "butterfruit",
                "mango"
        ],
        "price" : [
                {
                        "orange" : 60,
                        "butterfruit" : 200,
                        "mango" : 120
                }
        ]
}
>
```

Objective: To update the document with "_id:3" by popping two elements from the list of elements present in the array "fruits". The elements popped are "pineapple" and "grapes".

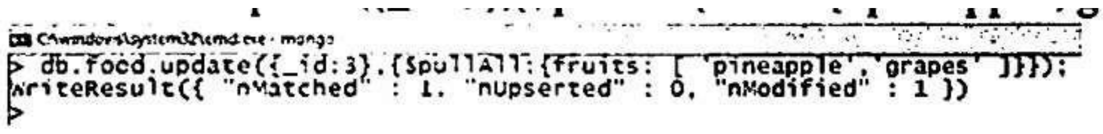
The document with "_id:3" before the update is



```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:3});
{ "_id" : 3, "fruits" : [ "pineapple", "strawberry", "grapes" ] }
>
```

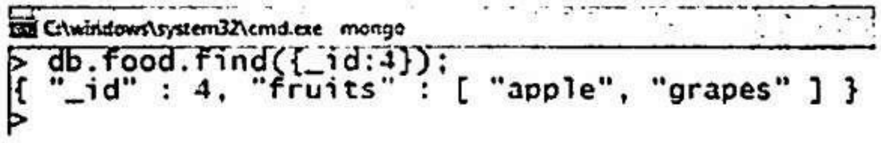
Act:

db.food.update({_id:3},{$pullAll:{fruits: ['pineapple','grapes']}});



```
C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:3},{$pullAll:{fruits: [ 'pineapple','grapes' ]}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

Outcome: The document with "_id:3" after the update is as follows:

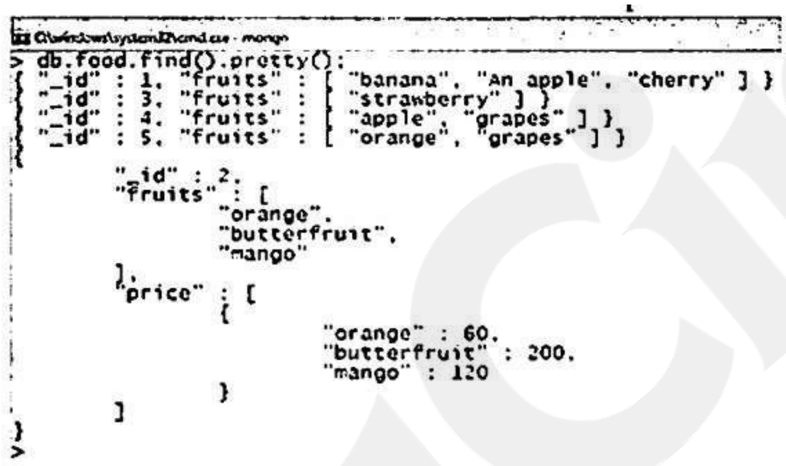


```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:4});
{ "_id" : 4, "fruits" : [ "apple", "grapes" ] }
>
```

Objective: To update the documents having "banana" as an element in the

the element "banana" from those documents.

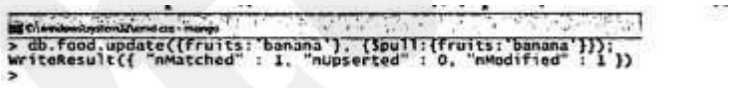
The "food" collection before the update is as follows:



```
C:\windows\system32\cmd.exe - mongo
> db.food.find().pretty();
{ "_id" : 1, "fruits" : [ "banana", "An apple", "cherry" ] }
{ "_id" : 3, "fruits" : [ "strawberry" ] }
{ "_id" : 4, "fruits" : [ "apple", "grapes" ] }
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
{
        "_id" : 2,
        "fruits" : [
                "orange",
                "butterfruit",
                "mango"
        ],
        "price" : [
                {
                        "orange" : 60,
                        "butterfruit" : 200,
                        "mango" : 120
                }
        ]
}
>
```

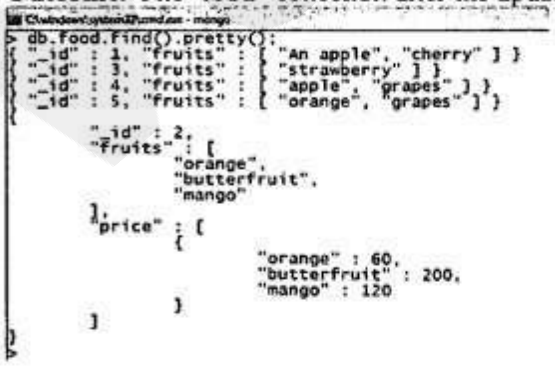
Act:

db.food.update({fruits:'banana'}, {$pull:{fruits:'banana'}})



```
C:\windows\system32\cmd.exe - mongo
> db.food.update({fruits:'banana'}, {$pull:{fruits:'banana'}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

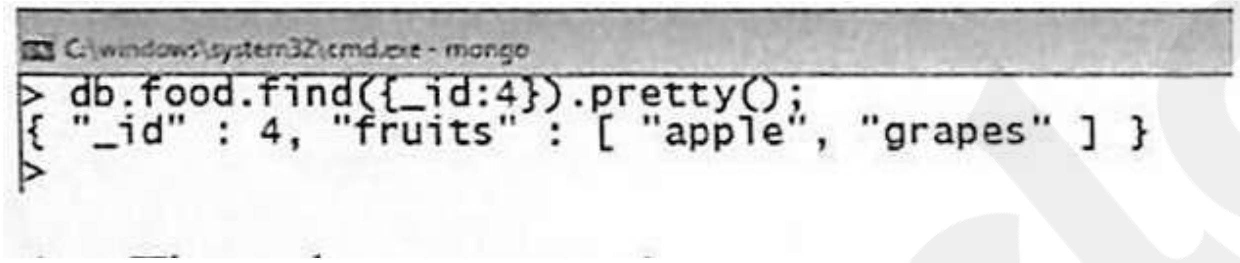
Outcome: The "food" collection after the update is as follows:



```
C:\windows\system32\cmd.exe - mongo
> db.food.find().pretty();
{ "_id" : 1, "fruits" : [ "An apple", "cherry" ] }
{ "_id" : 3, "fruits" : [ "strawberry" ] }
{ "_id" : 4, "fruits" : [ "apple", "grapes" ] }
{ "_id" : 5, "fruits" : [ "orange", "grapes" ] }
{
        "_id" : 2,
        "fruits" : [
                "orange",
                "butterfruit",
                "mango"
        ],
        "price" : [
                {
                        "orange" : 60,
                        "butterfruit" : 200,
                        "mango" : 120
                }
        ]
}
>
```

Objective: To pull out an array element based on index position.

There is no direct way of pulling the array elements by looking up their index numbers. However a workaround is available. The document with "_id:4" in the food collection prior to the update is as follows:

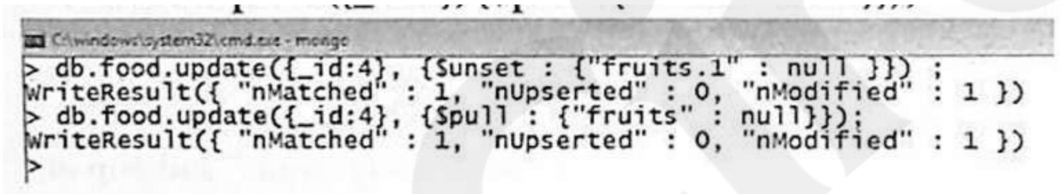


```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:4}).pretty();
{ "_id" : 4, "fruits" : [ "apple", "grapes" ] }
>
```

Act: The update statement is

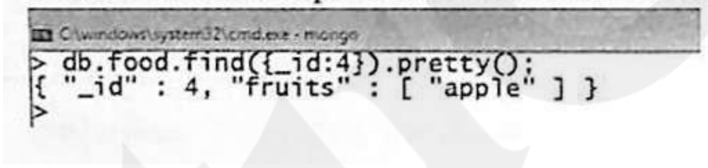
db.food.update({_id:4}, {$unset: {"fruits. 1": null }});

db.food.update({_id:4}, {$pull: {"fruits": null}});



```
C:\windows\system32\cmd.exe - mongo
> db.food.update({_id:4}, {$unset : {"fruits.1" : null }}) ;
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.food.update({_id:4}, {$pull : {"fruits" : null}});
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
>
```

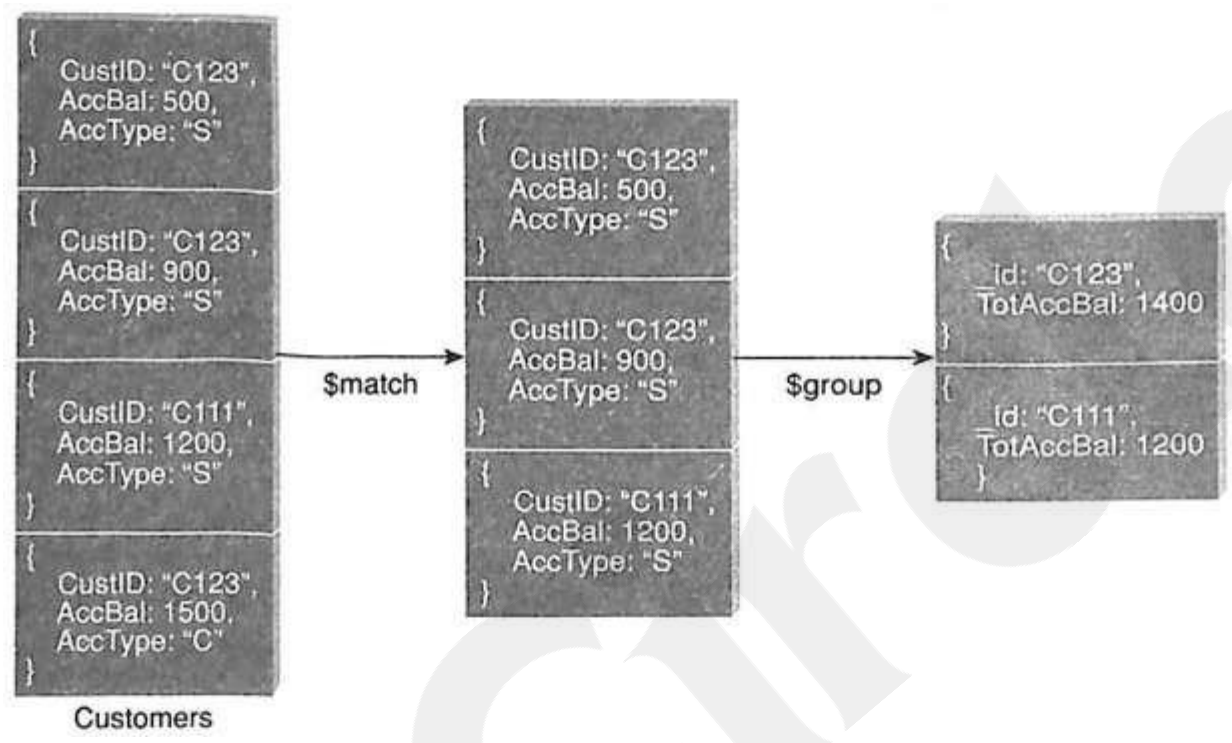
Outcome: After update, the document with _id:4 in the food collection is



```
C:\windows\system32\cmd.exe - mongo
> db.food.find({_id:4}).pretty();
{ "_id" : 4, "fruits" : [ "apple" ] }
>
```

3.5.9 Aggregate Function

Objective: Consider the collection “Customers" as given below. It has four documents. We would like to filter out those documents where the "AccType" has a value other than "S". After the filter, we should be left with three documents where the "Acctype": "S". It is then required to group the documents on the basis of CustID and sum up the "AccBal" for each unique "CustID". This is similar to the output received with group by clause in RDBMS. Once the groups have been formed [as per the example below, there will be only two groups: (a) "CustID" : "C123" and (b) “CustID" : "C111”], filter and display that group where the "TotAccBal" column has a value greater than 1200.


{ CustID: "C123", AccBal: 500, AccType: "S" }
{ CustID: "C123", AccBal: 900, AccType: "S" }
{ CustID: "C111", AccBal: 1200, AccType: "S" }
{ CustID: "C123", AccBal: 1500, AccType: "C" }
Customers
$match
{ CustID: "C123", AccBal: 500, AccType: "S" }
{ CustID: "C123", AccBal: 900, AccType: "S" }
{ CustID: "C111", AccBal: 1200, AccType: "S" }
$group
{ _id: "C123", TotAccBal: 1400 }
{ _id: "C111", TotAccBal: 1200 }

Let us start off by creating the collection "Customers" with the above displayed four documents:

db.Customers.insert([{CustID:"C123",AccBal:500,AccType:"S"},
{CustID:"C123", AccBal: 900, AccType:"S"},
{CustID:"C111", AccBal: 1200, AccType:"S"},
{CustID:"C123", AccBal: 1500, AccType:"C"}]);

To confirm the presence of four documents in the "Customers" collection, use the below syntax:

db.Customers.find().pretty();

To group on "CustID" and compute the sum of "AccBal", use the below syntax:

db.Customers.aggregate({$group:{_id:"$CustID",TotAccBal:{$sum:"$AccBal"}}});

In order to first filter on "AccType:S" and then group it on "CustID" and then compute the sum of "AccBal", use the below syntax:

**db.Customers.aggregate({ $match: {AccType: "S" } },
{$group: { _id: "$CustID",TotAccBal: { $sum : "$AccBal" } } });**

In order to first filter on "AccType:S" and then group it on "CustID" and then to compute the sum of "AccBal" and then filter those documents wherein the "TotAccBal" is greater than 1200, use the below syntax:

**db.Customers.aggregate({ $match : {AccType : "S" } },
{$group: { _id: "$CustID",TotAccBal: { $sum: "$AccBal" } } }, { $match: {TotAccBal : { $gt: 1200 } }});**

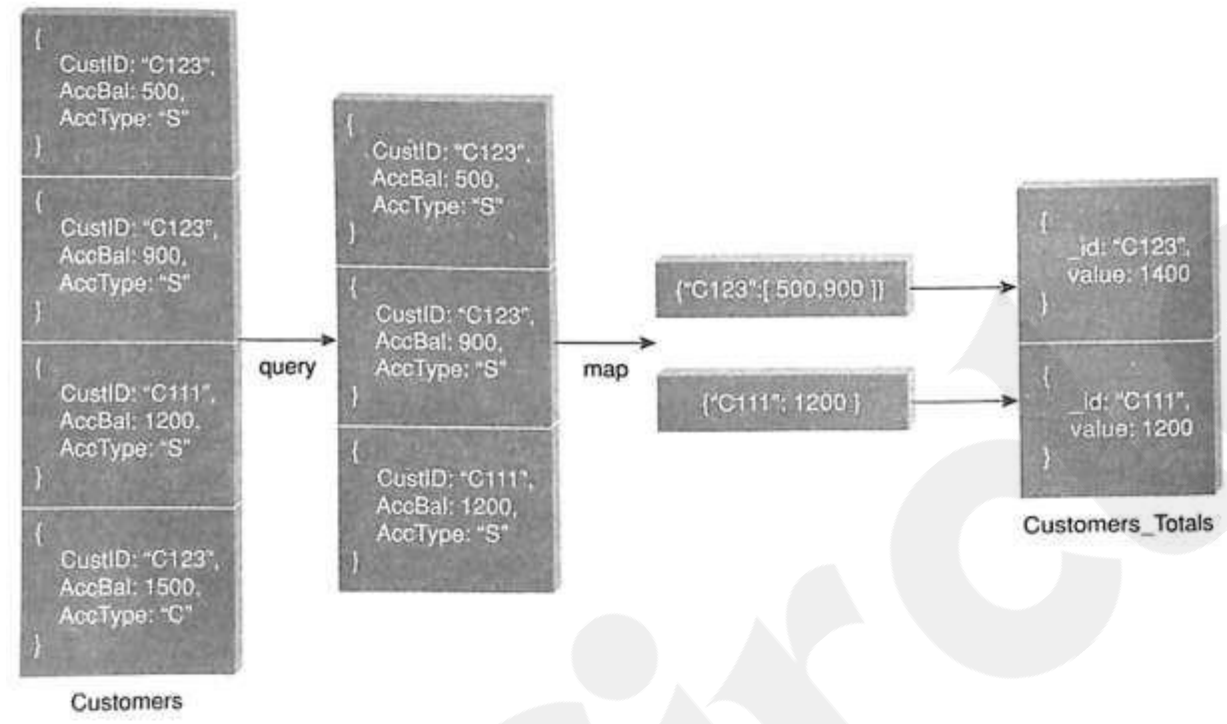
To group on "CustID" and compute the average of the "AccBal" for each group:
db.Customers.aggregate({ $group: { _id: "$CustID", TotAccBal : { $avg: "$AccBal" } } });

To group on "CustID" and determine the maximum "AccBal" for each group:
db.Customers.aggregate({ $group: { _id: "$CustID", TotAccBal: { $max : "$AccBal" } } });

To group on "CustID" and determine the minimum "AccBal" for each group:
db.Customers.aggregate({$group: { _id: "$CustID", TotAccBal: { $min : "$AccBal" } } });

3.5.10 MapReduce Function

Objective: Consider the collection "Customers" below. There are four documents. Run a query to filter out those documents where the key "AccType" has a value other than “S”. Then for each unique CustID, prepare a list of AccBal values. For example, for CustID: "C123", the AccBals are 500,900. This task will be assigned to the mapper function. The output from the mapper function serves as the input to the reducer function. The reducer function then aggregates the AccBal for each CustID. For example, for CustID: "C123", the value is 1400, etc.


{ CustID: "C123", AccBal: 500, AccType: "S" }
{ CustID: "C123", AccBal: 900, AccType: "S" }
{ CustID: "C111", AccBal: 1200, AccType: "S" }
{ CustID: "C123", AccBal: 1500, AccType: "C" }
Customers
query
{ CustID: "C123", AccBal: 500, AccType: "S" }
{ CustID: "C123", AccBal: 900, AccType: "S" }
{ CustID: "C111", AccBal: 1200, AccType: "S" }
map
{"C123":[500,900]}
{"C111": 1200 }
{ _id: "C123", value: 1400 }
{ _id: "C111", value: 1200 }
Customers_Totals

Given below is the syntax that we will use to accomplish the objective.

```
db.Customers.mapReduce (
        map → function() { emit (this. CustID, this.AccBal ); },
        reduce→ function(key, values) { return Array.sum (values ) },
                {
        query→ query: { AccType: "S"},
        output→  out:  "Customer_Totals"
                }
                )
```

Map Function

```
var map=function(){
emit (this. CustID, this.AccBal );}
```

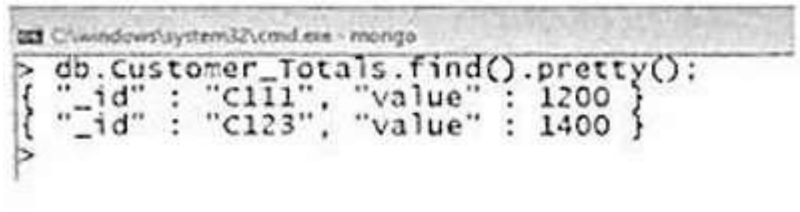
Reduce Function

```
var reduce = function (key, values) { return Array.sum(values); }
```

To execute the query

```
db.Customers.mapReduce(map,reduce,{out:"Customer_Totals",query:{AccType:"
S"}});
```

The output as archived in "Customer_Totals" collection:

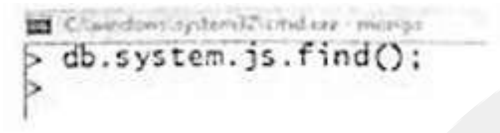


```
C:\windows\system32\cmd.exe - mongo
> db.Customer_Totals.find().pretty();
{ "_id" : "C111", "value" : 1200 }
{ "_id" : "C123", "value" : 1400 }
>
```

3.5.11 Java Script Programming

Objective: To compute the factorial of a given positive number. The user is required to create a func- tion by the name "factorial" and insert it into the "system.js" collection.

Before we proceed, a quick check on what is contained in the "system.js" collection:



```
C:\windows\system32\cmd.exe - mongo
> db.system.js.find();
>
```

Act:

```
db.system.js.insert({_id:"factorial",
value:function(n)
{
        if (n==1)
        return 1;
        else
        return n* factorial(n-1);
}
        }
);
```

To execute the function "factorial”, use the eval() method.

db.eval("factorial(3)");

db.eval("factorial(5)");

db.eval("factorial(1)");

3.5.12 Cursors in MongoDB

Objective: To create a collection named **"alphabets"** and insert 26 documents into it. Each document contains:

- _id: Numeric values from 1 to 26.
- alphabet: Alphabet letters from 'a' to 'z'.

Insertion Syntax:

Example insert statement:

```
db.alphabets.insert({_id:1, alphabet:"a"});
```

We must perform 26 such insertions to cover all lowercase alphabets.

Using Cursor to Iterate:

Using find():

```
var cursor = db.alphabets.find();
```

This retrieves all documents in the collection.

- If find() is not assigned to a variable, MongoDB shell automatically prints **only the first 20 documents**, followed by "Type "it" for more".

To print the remaining, type:

```
it
```

Manual Cursor Iteration Methods:

Method 1: hasNext()

Returns a **Boolean** indicating if more documents exist in the cursor.

Usage:

```
while (cursor.hasNext()) {
    printjson(cursor.next());
}
```

Method 2: next()

- Retrieves the **next document** in the cursor.

Using forEach Loop:

Alternative to hasNext():

```
db.alphabets.find().forEach(function (myrec) {
    print("Alphabet is: " + myrec.alphabet);
```

});

Verification:

To confirm the presence of all 26 documents:

db.alphabets.find()

3.5.13 Indexes

Sample Data:

Collection:books

Contains 5 documents with fields like:

- _id, Category, Bookname, Author, Qty, Price, Pages

Example Categories:

- Machine Learning
- Web Mining
- Python
- Visualization

Creating an Index

To create an index on the Category field in the books collection:

```
db.books.ensureIndex({"Category": 1});
```

Checking Index Status

1. **To check index stats:**

```
db.books.stats();
```

Shows count, storage size, index count, index sizes, etc.

Example:

```
"indexes" : 2,
"indexSizes" : {
  "_id_" : 8176,
  "Category_1" : 8176
}
```

2. **To list all indexes:**

```
db.books.getIndexes();
```

- Shows keys and names of all indexes.

Using Index in Queries

To force use of a specific index:

```
db.books.find({"Category":  "Web  Mining"}).hint({"Category":  1});
```

To understand how indexes are used:

```
db.books.find({"Category":  "Web  Mining"}).hint({"Category":1}).explain();
```

- cursor: "BtreeCursor Category_1"
- indexOnly: false (unless it's a covered index)
- nscanned: Number of documents scanned
- nscannedObjects: Number of actual documents examined

Covered Index

Covered index = All fields in query are part of the index.

Example:

```
db.books.find({"Category":"Web Mining"}, {"Category":1, _id:0})
        .hint({"Category":1})
        .explain();
```

- indexOnly: true
- _id field is explicitly excluded to allow the index to fully satisfy the query.

To have a covered index, only indexed fields should be returned (projected). MongoDB skips retrieving the full document if the index contains all the queried fields, improving performance.

3.5.14 mongoimport

Purpose:

The mongoimport command is used to import data into MongoDB from:

- CSV (Comma-Separated Values)
- TSV (Tab-Separated Values)
- JSON (JavaScript Object Notation)

Objective:

Import a CSV file named sample.txt located in the D: drive into the MongoDB collection SampleJSON within the test database.

Contents of sample.txt:

_id,FName,LName

1,Samuel,Jones

2,Virat,Kumar

3,Raul,"A Simpson"

4,,"Andrew Simon"

Command to Import CSV File:

Run the following command in the command prompt:

mongoimport --db test --collection SampleJSON --type csv --headerline --file d:\sample.txt

- --db test → Target database
- --collection SampleJSON → Target collection
- --type csv → Input file type
- --headerline → Use the first line of the CSV file as field names
- --file → Path to the input file

Successful Output Message:

connected to: 127.0.0.1

imported 4 objects

Verifying the Import in Mongo Shell:

Steps:

1. Start Mongo shell
2. Switch to test database:
3. use test
4. View collections:
5. show collections
6. Query the data:
7. db.SampleJSON.find().pretty();

Sample Output of JSON Documents:

```
{ "_id": 1, "FName": "Samuel", "LName": "Jones" }
{ "_id": 2, "FName": "Virat", "LName": "Kumar" }
{ "_id": 3, "FName": "Raul", "LName": "A Simpson" }
{ "_id": 4, "FName": "", "LName": "Andrew Simon" }
```

- Quoted values (e.g., "A Simpson") are correctly interpreted.
- Empty fields are stored as empty strings (e.g., FName in record 4).
- --headerline ensures correct mapping of CSV headers to MongoDB fields.

3.5.15 mongoexport

Purpose:

The mongoexport command is used at the command prompt to export MongoDB JSON documents into:

- CSV (Comma-Separated Values),
- TSV (Tab-Separated Values), or
- JSON (JavaScript Object Notation) formats.

Objective:

Export data from the Customers collection in the test database into a CSV file named Output.txt in the D: drive.

Sample Data in MongoDB (Customers Collection):

```
{ "_id": ObjectId("..."), "CustID": "C123", "AccBal": 500, "AccType": "S" }
{ "_id": ObjectId("..."), "CustID": "C123", "AccBal": 900, "AccType": "S" }
{ "_id": ObjectId("..."), "CustID": "C111", "AccBal": 1200, "AccType": "S" }
{ "_id": ObjectId("..."), "CustID": "C123", "AccBal": 1500, "AccType": "C" }
```

Steps to Export the Data:

Step 1: Create fields.txt file

This file should contain the field names **exactly as they appear in the MongoDB collection**, one per line:

```
CustID
```

AccBal

AccType

□□ **Important:** Field names are case-sensitive. Only one field name should be placed per line.

Step 2: Run the Export Command

```
mongoexport --db test --collection Customers --csv --fieldFile d:\fields.txt --out d:\output.txt
```

- --db test → Specifies the database.
- --collection Customers → Target collection.
- --csv → Specifies the output format.
- --fieldFile → Points to the list of fields to include.
- --out → Output file location.

Expected Command Line Output:

```
connected to: 127.0.0.1
exported 4 records
```

Final Output File (Output.txt in D: Drive):

```
CustID,AccBal,AccType
"C123",500.0,"S"
"C123",900.0,"S"
"C111",1200.0,"S"
"C123",1500.0,"C"
```

3.5.16 Automatic Generation of Unique Numbers for the _id Field

This technique is useful to **automatically assign a unique, incrementing ID** to each new document inserted into a collection.

Step 1: Initialize the Counter Document

Insert an initial document in a new collection named usercounters:

```
db.usercounters.insert(
  {
    _id: "empid",
    seq: 0
  }
)
```

- _id is a custom name (e.g., "empid") used to identify the sequence.
- seq is initialized to 0.

Step 2: Create a JavaScript Function getnextseq

This function will **find and increment the sequence value** using findAndModify().

```
function getnextseq(name) {
  var ret = db.usercounters.findAndModify({
    query: { _id: name },
    update: { $inc: { seq: 1 } },
    new: true
  });
  return ret.seq;
}
```

- findAndModify() atomically finds the document and increments seq by 1.
- new: true returns the modified document after the update.
- Returns the incremented seq value.

Step 3: Use getnextseq() While Inserting New Documents

Use the getnextseq() function when inserting into a collection (e.g., users) to auto-assign a unique _id:

```
db.users.insert(
  {
    _id: getnextseq("empid"),
    Name: "sarah jane"
  }
)
```

- The _id will now have an auto-incremented value based on the "empid" sequence in usercounters.

Benefits:

- Ensures unique and sequential _id values.
- Useful in applications needing custom ID schemes (e.g., employee numbers, customer IDs).

********END********