SESSION 5
DATABASE COMMANDS

Learning Objectives

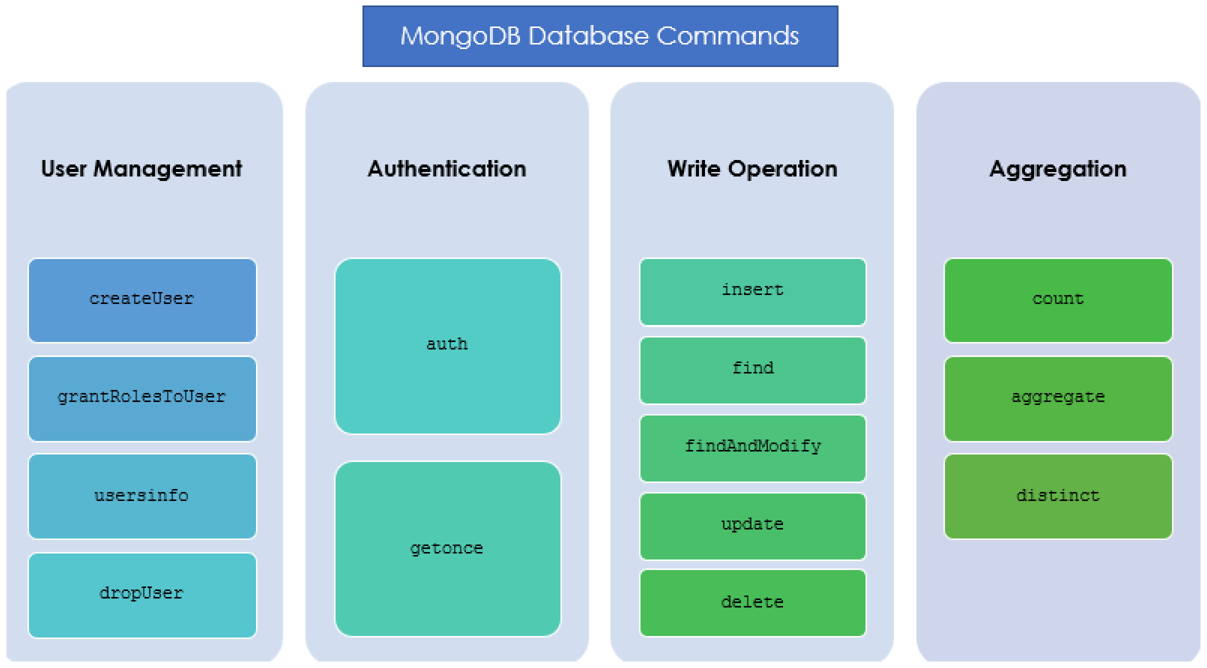
In this session, students will learn to:

- Explain the importance of database commands in MongoDB
- Describe the various types of database commands with examples

Database commands primarily involve operations related to creation and modification of user and collection data, authentication of users, and grouping of multiple documents. Queries involving database commands are often complex. This session explains various categories of database commands. This session also explores the parameters for each of the database commands and how to use them with examples.

5.1 Introduction to Database Commands

Developers use MongoDB database commands to create, modify, and update databases. Some of the categories in which MongoDB offers database commands include:



Before delving into these categories of commands, first let us understand how to run a command. The syntax to use the `runCommand` is:

```
db.runCommand( { <command> } )
```

Here, the user provides the command to be executed as a `document` or a `string` to the command.

Although `db.runCommand` runs the command against the current database, some commands are relevant only for the `admin` database. Therefore, the user must change the database object before running commands related to the `admin` database. Alternatively, the user can use `db.adminCommand` to run an administrative command against the `admin` database irrespective of the current database in use.

To run an administrative command against the `admin` database, the user can use the syntax:

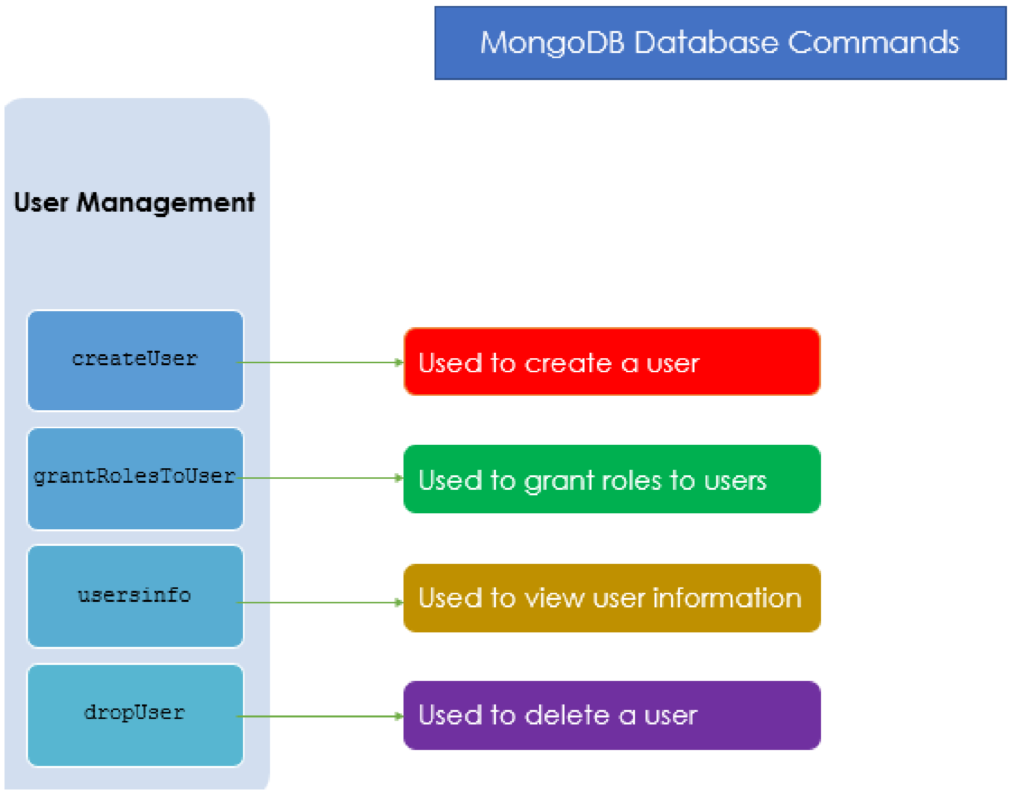
```
db.adminCommand( { <command> } )
```

The `runCommand` command provides a helper to run specified database commands. This method provides a consistent interface between the shell and drivers and hence, is the most preferred method to issue database commands.

The `db.runCommand` and `db.adminCommand` commands take a database command as the parameter which can be a `document` or a `string`. If the database command is specified as a `string`, the `string` is automatically transformed into a `document`.

5.2 User Management Commands

User management commands in MongoDB allow operations related to users such as user creation/deletion, viewing user information, and role assignments.



5.2.1 `createUser` Command

Users can use the `createUser` command to create a new user in the currently active database. If the user already exists, this command returns a **duplicate user** error message. The syntax for the `createUser` command is:

```
db.runCommand(
{
     createUser: "<name>",
     pwd: passwordPrompt(),  // Or "<cleartext password>"
     customData: { <any information> },
     roles: [
            { role: "<role>", db: "<database>" } | "<role>",
            ...
          ],
     writeConcern: { <write concern> },
     authenticationRestrictions: [
             { clientSource: [ "<IP|CIDR range>", ... ],
     serverAddress: [ "<IP|CIDR range>", ... ] },
             ...
          ],
     mechanisms: [ "<scram-mechanism>", ... ],  //Available
     starting in MongoDB 4.0
     digestPassword: <boolean>,
     comment: <any>
}
```

Table 5.1 discusses various parameters in the `createUser` command.

Parameter	Type	Description
`createUser`	`string`	Used to specify the name of the new user
`pwd`	`string`	Used to set the user's password either as a cleartext string or as `passwordPrompt` to prompt for the user's password
`customData`	`document`	Used to specify the extra information that an admin wishes to associate with a particular user
`roles`	`array`	Used to mention the role granted to the user
`digestPassword`	`boolean`	Used to specify whether the server or the client will digest the password

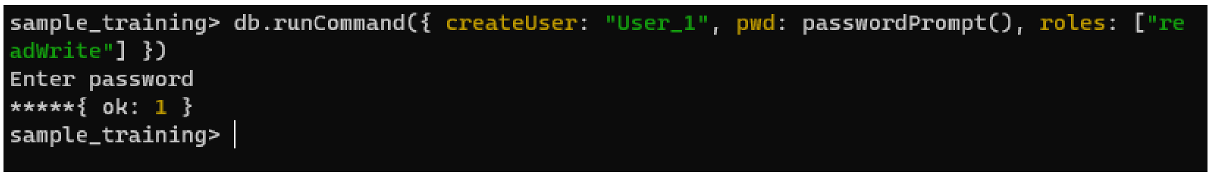
Parameter	Type	Description
writeConcern	document	Used to specify whether the level of write concern for the creation operation is at the transaction level or at the operation level
authentication Restrictions	array	Used to specify the restrictions posed by the server on the user; a list of IP addresses and Classless Inter-Domain Routing (CIDR) ranges from which the user is allowed to connect
mechanism	array	Used to specify the Salted Challenge Response Authentication Mechanism (SCRAM) mechanisms with valid SCRAM values as SCRAM-SHA-1 and SCRAM-SHA-256

Table 5.1: Parameters in the `createUser` Command

For example, to create a user named `User_1` on the `sample_training` database with the `readWrite` role, the user can use the query as:

```
use sample_training
db.runCommand({ createUser: "User_1", pwd:
passwordPrompt(), roles: ["readWrite"] })
```

Figure 5.1 shows the creation of `User_1`. The system will prompt for a password. Type the password as `user1`.



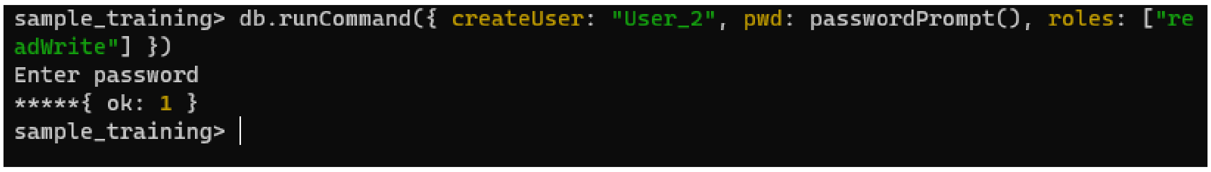
```
sample_training> db.runCommand({ createUser: "User_1", pwd: passwordPrompt(), roles: ["re
adWrite"] })
Enter password
*****{ ok: 1 }
sample_training> 
```

Figure 5.1: Creation of `User_1`

Let us create another user `User_2` in the same database with the `readWrite` role.

```
db.runCommand({ createUser: "User_2", pwd:
passwordPrompt(), roles: ["readWrite"] })
```

Figure 5.2 shows the creation of `User_2`. The system will prompt for a password. Type the password as `user2`.



```
sample_training> db.runCommand({ createUser: "User_2", pwd: passwordPrompt(), roles: ["re
adWrite"] })
Enter password
*****{ ok: 1 }
sample_training> |
```

Figure 5.2: Creation of User_2

5.2.2 `grantRolesToUser` Command

The `grantRolesToUser` command grants additional roles to a user. The syntax for the `grantRolesToUser` command is:

```
db.runCommand(
{
      grantRolesToUser: "<user>",
      roles: [ <roles> ],
      writeConcern: { <write
      concern> },
      comment: <any>
}
)
```

Assigning a role for a user on the currently active database involves:

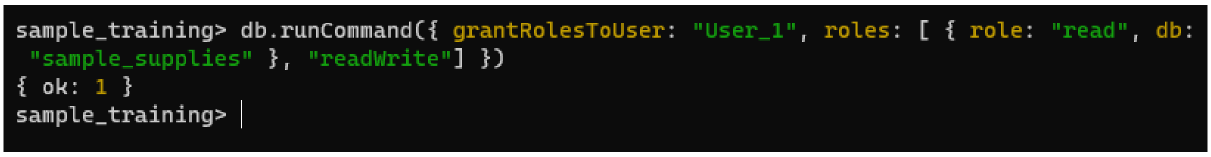
- Specifying the role with the name of the role, for example, `readWrite`, or
- Specifying the role with a document, for example, `{ role: "<role>", db: "<database>" }`

However, to specify a role for a user that exists in a different database, the role with a document must be specified.

Consider the `sample_supplies` database. To grant the **`read`** role on the `sample-supplies` database to `User_1` of the `sample_training` database, the user must execute the query as:

```
db.runCommand({ grantRolesToUser: "User_1", roles: [
{ role: "read", db: "sample_supplies" },
"readWrite"] })
```

Figure 5.3 shows the result of the query execution.



```
sample_training> db.runCommand({ grantRolesToUser: "User_1", roles: [ { role: "read", db:
 "sample_supplies" }, "readWrite"] })
{ ok: 1 }
sample_training>
```

Figure 5.3: Specifying Additional `read` Role for `User_1`

An additional **`read`** role in a different database has been assigned to `User_1`.

5.2.3 `usersInfo` Command

The `usersInfo` command returns information about one or more users from the database. The syntax for the `usersInfo` command is:

```
db.runCommand(
{
      usersInfo: <various>,
      showCredentials: <Boolean>,
      showCustomData: <Boolean>,
      showPrivileges: <Boolean>,
      showAuthenticationRestrictions:
      <Boolean>,
      filter: <document>,
      comment: <any>
}
)
```

Table 5.2 lists the various forms of the `usersInfo` command.

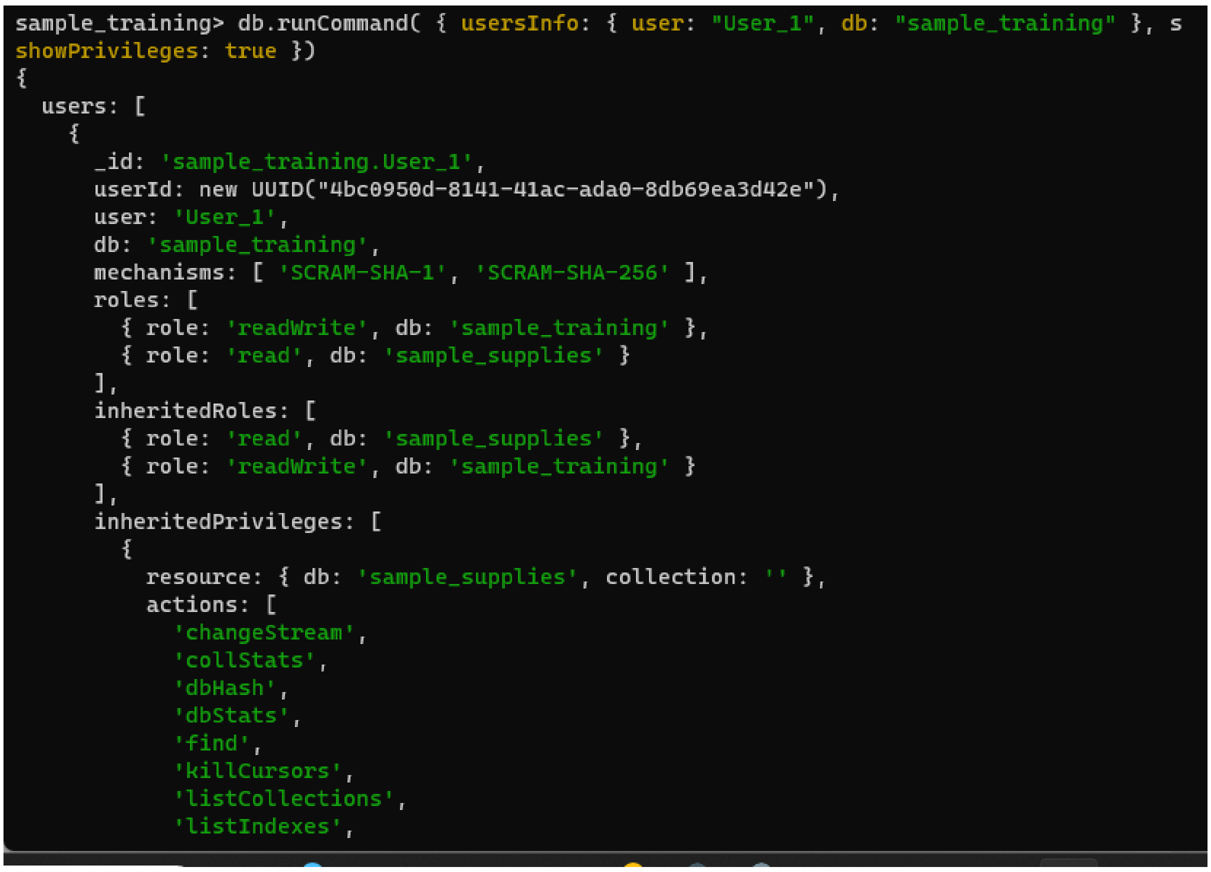
Form	Description
`{ usersInfo: 1 }`	Used to return information about the users in the database where the command is run
`{ usersInfo: <username> }`	Used to return information about a specific user in the database where the command is run
`{ usersInfo: { user: <name>, db: <db> } }`	Used to return information about the specific user in the mentioned database
`{ usersInfo: [ { user: <name>, db: <db> }, … ] }` `{ usersInfo: [ <username1>, … ] }`	Used to return information about the specific users listed
`{ forAllDBs: true }`	Used to return information about users in all databases

Table 5.2: Forms of the `usersInfo` Command

Consider that the user wants to view the information of `User_1` in the `sample_training` database. To do this, the user can execute the query as:

```
db.runCommand( { usersInfo: { user: "User_1", db:
"sample_training" }, showPrivileges: true })
```

Figure 5.4 shows the output of the query.



```
sample_training> db.runCommand( { usersInfo: { user: "User_1", db: "sample_training" }, s
showPrivileges: true })
{
  users: [
    {
      _id: 'sample_training.User_1',
      userId: new UUID("4bc0950d-8141-41ac-ada0-8db69ea3d42e"),
      user: 'User_1',
      db: 'sample_training',
      mechanisms: [ 'SCRAM-SHA-1', 'SCRAM-SHA-256' ],
      roles: [
        { role: 'readWrite', db: 'sample_training' },
        { role: 'read', db: 'sample_supplies' }
      ],
      inheritedRoles: [
        { role: 'read', db: 'sample_supplies' },
        { role: 'readWrite', db: 'sample_training' }
      ],
      inheritedPrivileges: [
        {
          resource: { db: 'sample_supplies', collection: '' },
          actions: [
            'changeStream',
            'collStats',
            'dbHash',
            'dbStats',
            'find',
            'killCursors',
            'listCollections',
            'listIndexes',
```

Figure 5.4: Information on `User_1` from the `sample_training` Database

5.2.4 `dropUser` Command

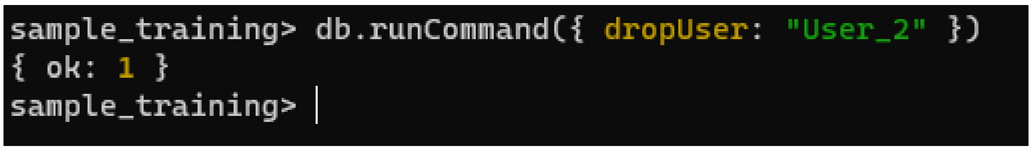
The `dropUser` command removes the user from the currently active database. The syntax for the `dropUser` command is:

```
db.runCommand(
{
       dropUser: "<user>",
       writeConcern: { <write concern> },
       comment: <any>
}
)
```

To drop the user named `User_2` from the `sample_training` database, the user can execute the query as:

```
db.runCommand({ dropUser: "User_2" })
```

Figure 5.5 shows the output of the query.

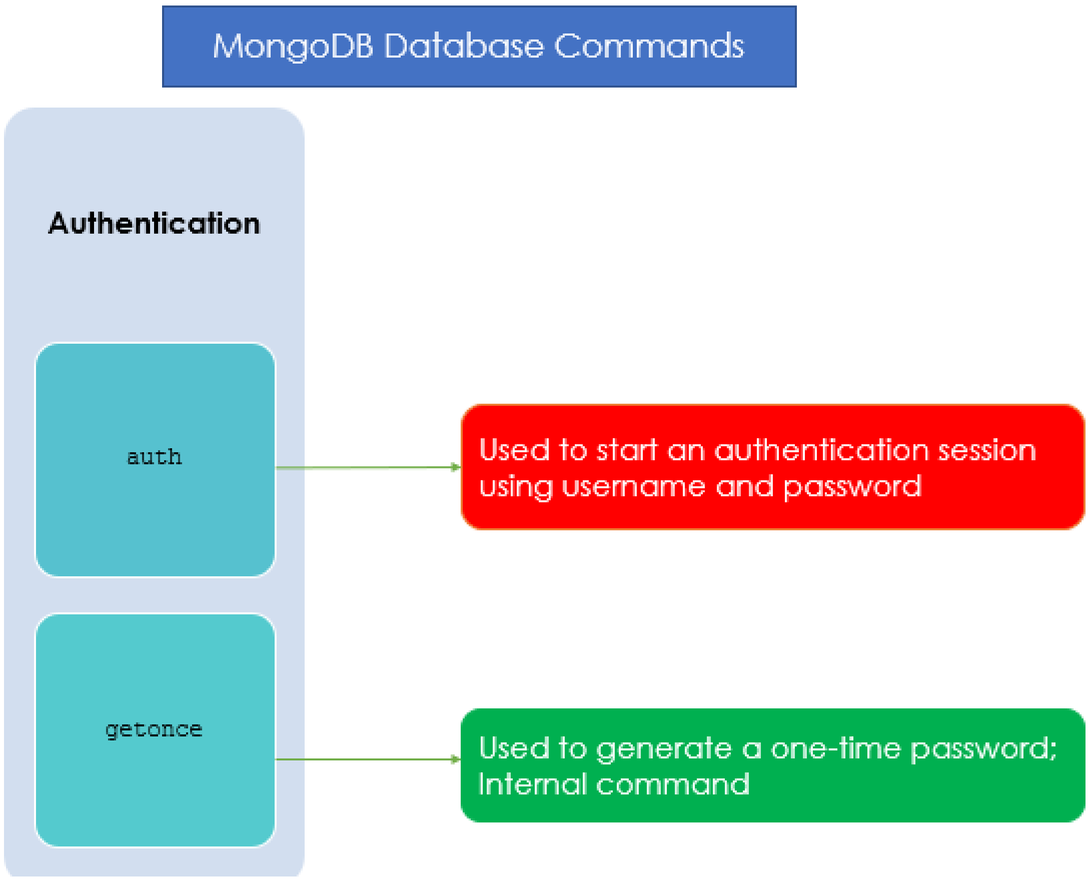


```
sample_training> db.runCommand({ dropUser: "User_2" })
{ ok: 1 }
sample_training>
```

Figure 5.5: Dropping `User_2` from the `sample_training` Database

5.3 Authentication Commands

Authentication is the process in which the user credentials are validated against a stored set of values. MongoDB offers security through its authentication commands, which are:


MongoDB Database Commands
Authentication
auth
getonce
Used to start an authentication session using username and password
Used to generate a one-time password; Internal command

This session discusses only the `auth` command in detail as the `getOnce` command is an internal command used by MongoDB.

5.3.1 Authenticate Command

The `db.auth` command authenticates a user using the x.509 authentication mechanism. This command returns `0` when authentication is not successful and `1` when the authentication is successful. The syntax for the `db.auth` command is:

```
db.auth( {
     user: <username>,
     pwd: passwordPrompt(), // Or "<cleartext
     password>"
     mechanism: <authentication mechanism>,
     digestPassword: <boolean>
} )
```

Table 5.3 lists the parameters of the `db.auth` command.

Parameter	Type	Optional/ Mandatory	Description
`user`	`string`	Mandatory	Used to provide the name of the user with access privileges for this database
`pwd`	`string`	Mandatory	Used to specify the user's password either as a cleartext string or as `passwordPrompt()` to prompt for the user's password
`mechanism`	`string`	Optional	Used to specify an authentication mechanism
`digestPassword`	`boolean`	Optional	Used to determine whether to pre-hash the given password before using it with the specified authentication mechanism

Table 5.3: Parameters of the `db.auth` Command

There are two ways to prompt the user to enter a password. The user can omit the password field or use the `passwordPrompt` method to authenticate a username. The syntax for using the `passwordPrompt` method is:

```
db.auth( <username> )

                    or

db.auth( <username>, passwordPrompt() )
```

The queries will prompt the user to enter a password. To specify a `cleartext` password, the user can use the syntax as:

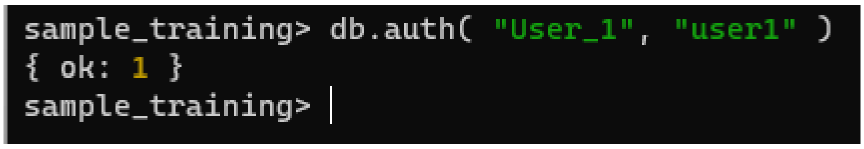
```
db.auth( <username>, <password> )
```

Authentication of user credentials is possible in MongoDB only after a connection to the shell is established.

To authenticate the user `User_1` with the password `user1` in the `sample_training` database, the user can execute the query as:

```
db.auth( "User_1", "user1" )
```

Figure 5.6 shows the output of the query as 1 indicating a successful authentication.



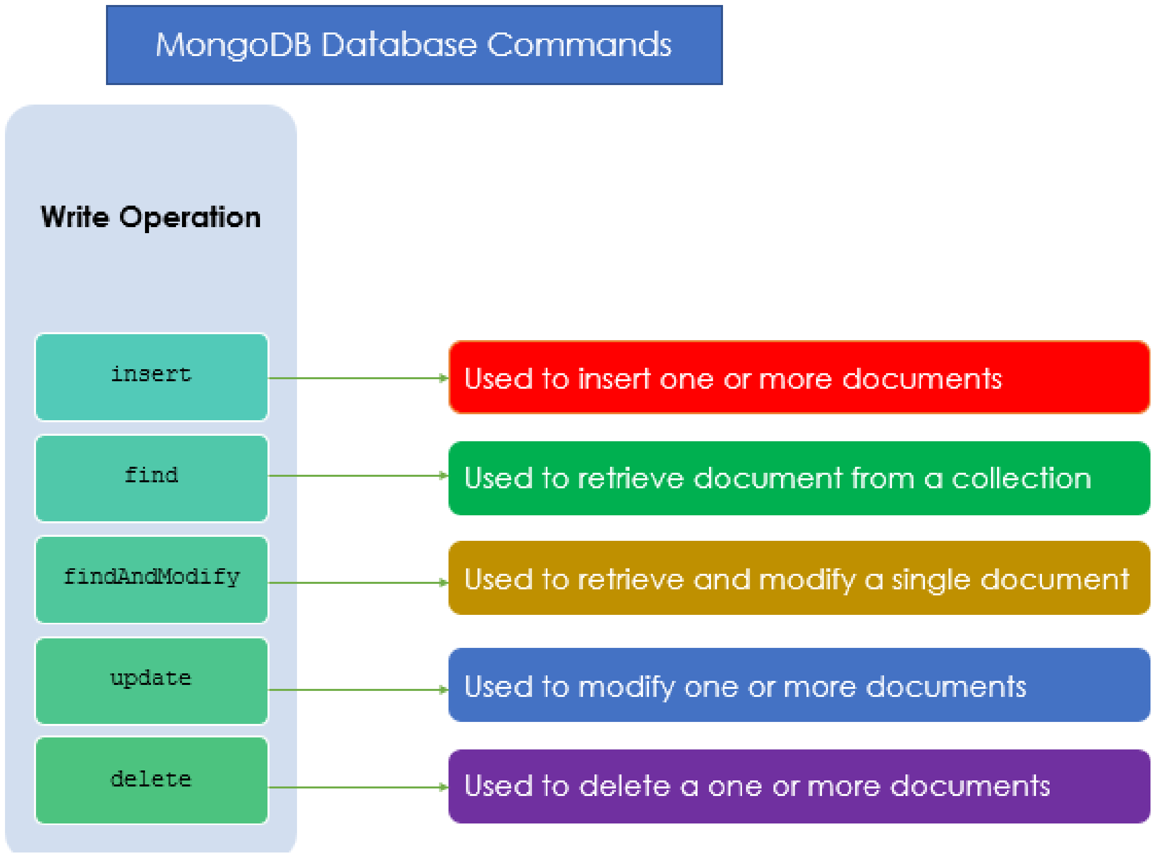
```
sample_training> db.auth( "User_1", "user1" )
{ ok: 1 }
sample_training> 
```

Figure 5.6: Authenticating `User_1` with the Password `user1`

5.4 Write Operation Commands

A write operation is an action used to create or modify data in a MongoDB instance. In MongoDB, write operations target a single collection and affects one or more documents in that collection.

The commands offered by MongoDB to enable write operations are:


MongoDB Database Commands
Write Operation
insert
find
findAndModify
update
delete
Used to insert one or more documents
Used to retrieve document from a collection
Used to retrieve and modify a single document
Used to modify one or more documents
Used to delete a one or more documents

5.4.1 `insert` Command

The `insert` command inserts one or more documents into a collection. This command returns a document containing the status of all insert operations.

The syntax for the `insert` command is:

```
db.runCommand(
{
      insert: <collection>,
      documents: [ <document>, <document>,
      <document>, ... ],
      ordered: <boolean>,
      writeConcern: { <write concern> },
      bypassDocumentValidation: <boolean>,
      comment: <any>
}
)
```

Table 5.4 lists the parameters of the `insert` command.

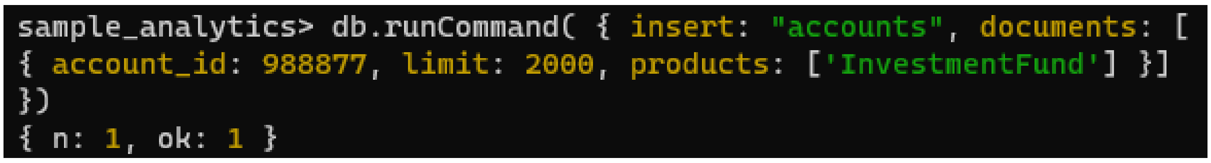
Parameter	Type	Description
`insert`	`string`	Used to specify name of the target collection
`documents`	`array`	Used to provide the array of documents to be inserted into the named collection
`ordered`	`boolean`	Used to specify if the remaining commands must be performed in case this `insert` command fails; if set to `true`, all remaining commands will not be performed; if set to `false`, all remaining commands will be executed Optional; Default value is `true`.
`writeConcern`	`document`	Used to specify whether the level of write concern for the insert operation is at the transaction level or at the operation level Optional
`bypassDocumentValidation`	`boolean`	Used to insert documents that do not meet the validation criteria Optional
`comment`	`any`	Used to provide an arbitrary string that helps in tracing the operation through the database profiler, `currentOp`, and logs Optional

Table 5.4: Parameters of the `insert` Command

Consider the `accounts` collection in the `sample_analytics` database. If the user wants to insert a single document with `account_id` as `988877`, `limit` as `2000`, and `products` as `InvestmentFund`, then the user can execute the query as:

```
db.runCommand( { insert: "accounts", documents: [{
account_id: 988877, limit: 2000, products:
['InvestmentFund'] }] })
```

Figure 5.7 shows the output of the query.



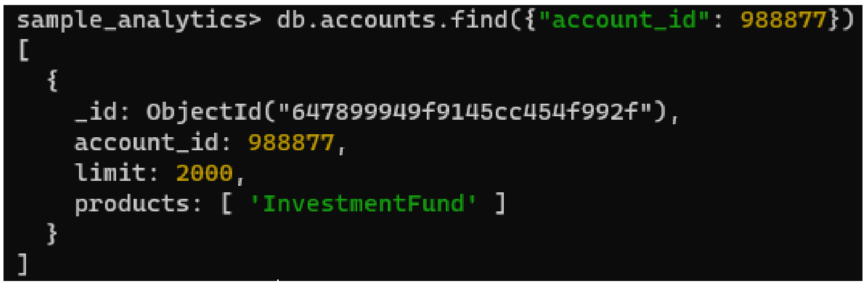
```
sample_analytics> db.runCommand( { insert: "accounts", documents: [
{ account_id: 988877, limit: 2000, products: ['InvestmentFund'] }]
})
{ n: 1, ok: 1 }
```

Figure 5.7: Inserting a Single Document

To view the inserted document, the user can execute the query as:

```
db.accounts.find({"account_id": 988877})
```

Figure 5.8 shows the output of the query.



```
sample_analytics> db.accounts.find({"account_id": 988877})
[
  {
    _id: ObjectId("647899949f9145cc454f992f"),
    account_id: 988877,
    limit: 2000,
    products: [ 'InvestmentFund' ]
  }
]
```

Figure 5.8: Insertion of the Document with `account_id` as `988877`

5.4.2 `find` Command

The `find` command selects the documents that match the given query criteria from a collection and returns a cursor to the selected documents.

A cursor is a pointer to the resultant set of a query. Users can iterate through a cursor to perform operations on the results of the query. By default, cursors that are not opened within a session automatically time out after 10 minutes of inactivity. Cursors opened under a session will close when the session ends or times out.

The syntax for the `find` command is:

```
db.runCommand(
{
      find: <string>,
      filter: <document>,
      sort: <document>,
      projection: <document>,
      hint: <document or string>,
      skip: <int>,
      limit: <int>,
      batchSize: <int>,
      singleBatch: <bool>,
      comment: <any>,
      maxTimeMS: <int>,
      readConcern: <document>,
      max: <document>,
      min: <document>,
      returnKey: <bool>,
      showRecordId: <bool>,
      tailable: <bool>,
      oplogReplay: <bool>,
      noCursorTimeout: <bool>,
      awaitData: <bool>,
      allowPartialResults: <bool>,
      collation: <document>,
      allowDiskUse : <bool>,
      let: <document> // Added in MongoDB 5.0
   }
)
```

Table 5.5 lists some of the parameters in the `find` command.

Parameter	Type	Description
`Find`	`string`	Used to specify the name of the collection
`filter`	`document`	Used to provide the match criteria
`Sort`	`document`	Used to specify the order for sorting the result of the query

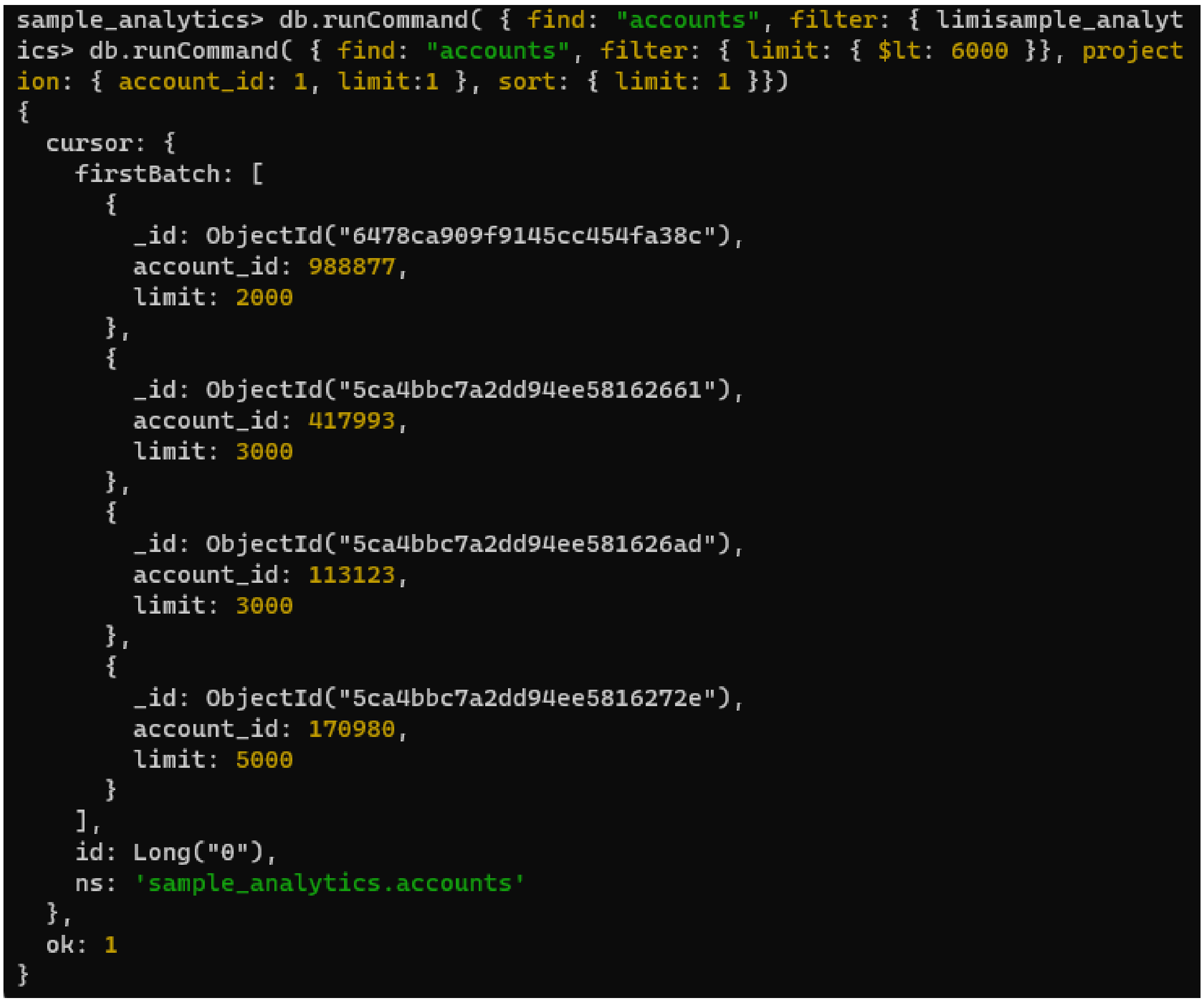
Parameter	Type	Description
`projection`	`document`	Used to specify the projection specification which lists the fields to be included in the result
`Hint`	`string`	Used to specify the index name as a string or as an index key pattern
`Skip`	`positive integer`	Used to list the number of documents to be skipped
`Limit`	`Non-negative integer`	Used to set the maximum number of documents to be returned
`batchSize`	`Non-negative integer`	Used to specify the number of documents to be returned in a batch of the result
`singleBatch`	`boolean`	Used to specify whether to close the cursor after returning the first batch of documents
`maxTimeMS`	`positive integer`	Used to provide the time limit for the processing operation on the cursor
`readConcern`	`document`	Used to set the read concern level as: `readConcern: { level: <value> }`
`Max`	`document`	Used to provide the upper bound for the given index
`Min`	`boolean`	Used to provide the lower bound for the given index
`returnKey`	`boolean`	Used to restrict the result to index keys, if set to true
`showRecordID`	`boolean`	Used to return the record identifier for each document, if set to true
`noCursorTimeout`	`boolean`	Used to prevent the server from timing out idle cursors
`collation`	`document`	Used to specify the collation rules for the operation that includes language-specific rules for string comparison, such as casing of the alphabets and accent marks

Table 5.5: Parameters of the `find` Command

Consider the `accounts` collection in the `sample_analytics` database. The user wants to find the documents where `limit` is less than `6000`, display only the `account_id` and `limit` value, and sort the resulting documents by the `limit` value. To do this, the user can run the query as:

```
db.runCommand( { find: "accounts", filter: { limit:
{ $lt: 6000 }}, projection: { account_id: 1, limit:1
}, sort: { limit: 1 }})
```

Figure 5.9 shows the output of the query.



```
sample_analytics> db.runCommand( { find: "accounts", filter: { limisample_analyt
ics> db.runCommand( { find: "accounts", filter: { limit: { $lt: 6000 }}, project
ion: { account_id: 1, limit:1 }, sort: { limit: 1 }})
{
  cursor: {
    firstBatch: [
      {
        _id: ObjectId("6478ca909f9145cc454fa38c"),
        account_id: 988877,
        limit: 2000
      },
      {
        _id: ObjectId("5ca4bbc7a2dd94ee58162661"),
        account_id: 417993,
        limit: 3000
      },
      {
        _id: ObjectId("5ca4bbc7a2dd94ee581626ad"),
        account_id: 113123,
        limit: 3000
      },
      {
        _id: ObjectId("5ca4bbc7a2dd94ee5816272e"),
        account_id: 170980,
        limit: 5000
      }
    ],
    id: Long("0"),
    ns: 'sample_analytics.accounts'
  },
  ok: 1
}
```

Figure 5.9: Result of the `find` Command

5.4.3 `findAndModify` Command

The `findAndModify` command filters, modifies, and returns a single document as result with or without the modifications.

The syntax for the `findAndModify` command is:

```
db.runCommand(
{
      findAndModify: <collection-name>,
      query: <document>,
      sort: <document>,
      remove: <boolean>,
      update: <document or aggregation
      pipeline>,
      new: <boolean>,
      fields: <document>,
      upsert: <boolean>,
      bypassDocumentValidation: <boolean>,
      writeConcern: <document>,
      collation: <document>,
      arrayFilters: <array>,
      hint: <document|string>,
      comment: <any>,
      let: <document> // Added in MongoDB 5.0
})
```

Table 5.6 lists some of the parameters in the `findAndModify` command.

Parameter	Type	Description
`findAndModify`	`string`	Used to specify the name of the collection in which a document must be searched for to modify or remove
`query`	`document`	Used to specify the selection criteria for the documents Optional; if omitted, defaults to an empty document
`sort`	`document`	Used to specify the document to be modified; the first document in the sorted result collection is modified
`remove`	`boolean`	Used to specify whether the document filtered by the query must be removed; either `remove` or `update` field must be set to true
`update`	`Document or array`	Used to specify the modifications to be made to the document selected by `query`
`new`	`boolean`	Used to specify whether the modified document must be returned

Parameter	Type	Description
fields	document	Used to list the fields to be returned
writeConcern	document	Used to specify whether the level of write concern for the operation is at the transaction level or at the operation level Optional
collation	document	Used to specify the collation rules for the operation that includes language-specific rules for string comparison, such as casing of the alphabets and accent marks

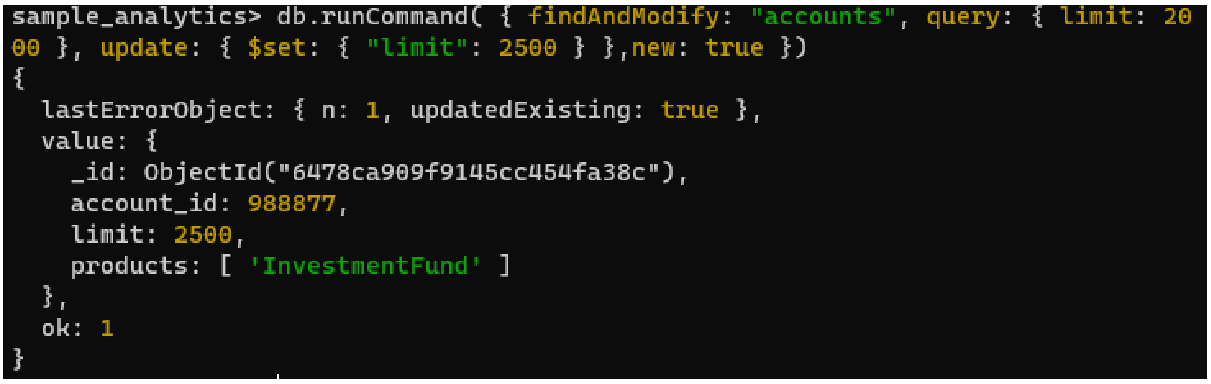
Table 5.6: Parameters of the `findAndModify` Command

By default, this command returns a document without the modifications. To return the document with the modifications, the user must include the `new` field set to `true`.

Consider the `accounts` collection in the `sample_analytics` database. The user wants to find the documents where `limit` is `2000`, modify the `limit` to `2500`, and display the modified document. To do this, the user can run the query as:

```
db.runCommand( { findAndModify: "accounts", query: {
limit: 2000 }, update: { $set: { "limit": 2500 }
},new: true })
```

Figure 5.10 shows the modified document.



```
sample_analytics> db.runCommand( { findAndModify: "accounts", query: { limit: 20
00 }, update: { $set: { "limit": 2500 } },new: true })
{
  lastErrorObject: { n: 1, updatedExisting: true },
  value: {
    _id: ObjectId("6478ca909f9145cc454fa38c"),
    account_id: 988877,
    limit: 2500,
    products: [ 'InvestmentFund' ]
  },
  ok: 1
}
```

Figure 5.10: Result of the `findAndModify` Command

If no document matches the query condition, the `findAndModify` command returns a document with null in the value field.

5.4.4 `update` Command

The `update` command modifies one or more documents in a collection. A single update command can contain a single update statement or multiple update statements. The syntax for the `update` command is:

```
db.runCommand(
{
     update: <collection>,
     updates: [
     {
          q: <query>,
          u: <document or pipeline>,
          c: <document>, // Added in MongoDB 5.0
          upsert: <boolean>,
          multi: <boolean>,
          collation: <document>,
          arrayFilters: <array>,
          hint: <document|string>
     },
     ...
     ],
     ordered: <boolean>,
     writeConcern: { <write concern> },
     bypassDocumentValidation: <boolean>,
     comment: <any>,
     let: <document> // Added in MongoDB 5.0
}
)
```

Table 5.7 describes the parameters of the `update` command.

Parameter	Type	Description
`update`	string	Used to specify the name of the target collection
`updates`	array	Used to provide an array of one or more update statements which will perform modifications on the named collection
`ordered`	`boolean`	Used to specify if the remaining commands must be performed in case this `update` command fails

Parameter	Type	Description
		• if set to `true`, the remaining commands will not be executed • if set to `false`, the remaining commands will be executed • Optional • Default is `true`
`writeConcern`	`document`	Used to specify whether the level of write concern for the operation is at the transaction level or at the operation level Optional
`bypassDocumentValidation`	`boolean`	Used to update documents that do not meet the validation criteria Optional
`comment`	`any`	Used to provide an arbitrary string that helps in tracing the operation through the database profiler, `currentOp`, and logs Optional

Table 5.7: Parameters of the `update` Command

Table 5.8 lists the fields in the `updates` array of the `update` command.

Field	Type	Description
`q`	`document`	Used to specify the selection criteria for documents using the query selectors
`u`	`document` or `array`	Used to specify the modifications to be made in the documents that match the selection

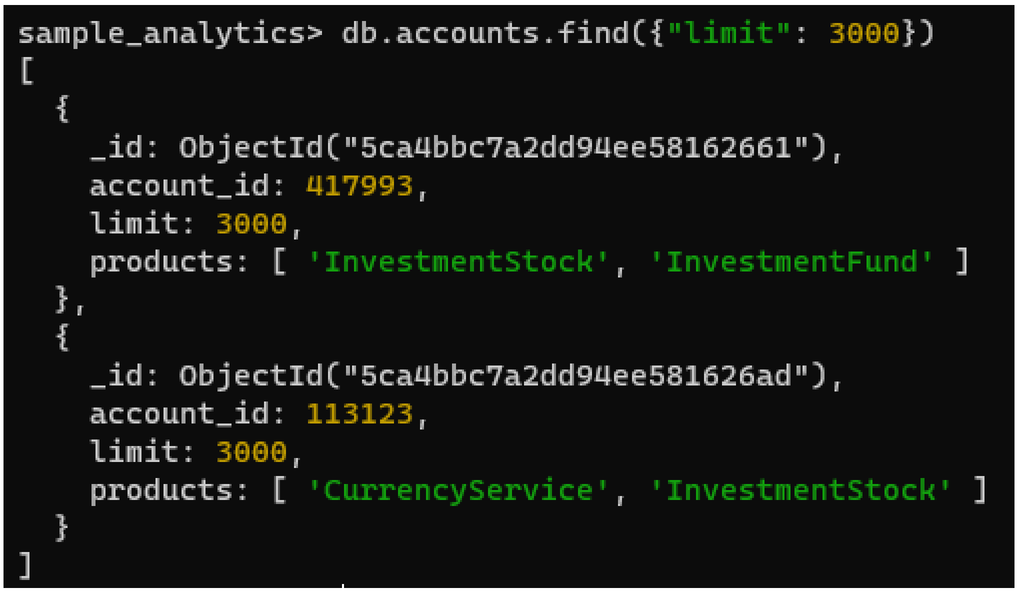
Table 5.8: Fields of the `updates` Array

The user can either update a specific field in a single document or single field in multiple documents. Similarly, the user can also update multiple fields in a single document or multiple documents.

Consider the `accounts` collections in the `sample_analytics` database. This collection contains documents that have the `limit` value set to `3000`. To view these documents, the user can execute the query as:

```
db.accounts.find({"limit": 3000})
```

Figure 5.11 shows two documents with `limit` as `3000`.



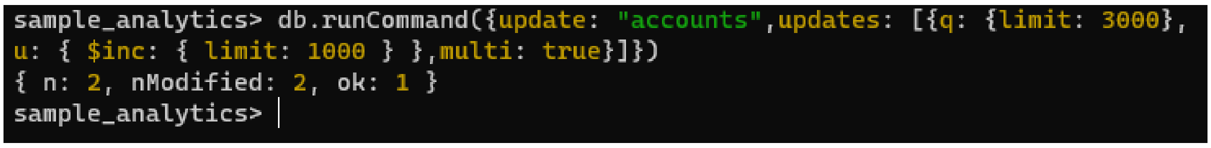
```
sample_analytics> db.accounts.find({"limit": 3000})
[
  {
    _id: ObjectId("5ca4bbc7a2dd94ee58162661"),
    account_id: 417993,
    limit: 3000,
    products: [ 'InvestmentStock', 'InvestmentFund' ]
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee581626ad"),
    account_id: 113123,
    limit: 3000,
    products: [ 'CurrencyService', 'InvestmentStock' ]
  }
]
```

Figure 5.11: Viewing Documents with `limit` as `3000`

To update all the documents with the `limit` value of `3000` to `4000`, the user can execute the query as:

```
db.runCommand({update: "accounts", updates: [{q:
{limit: 3000},u: { $inc: { limit: 1000 } },multi:
true}]})
```

Figure 5.12 shows the output of this query.



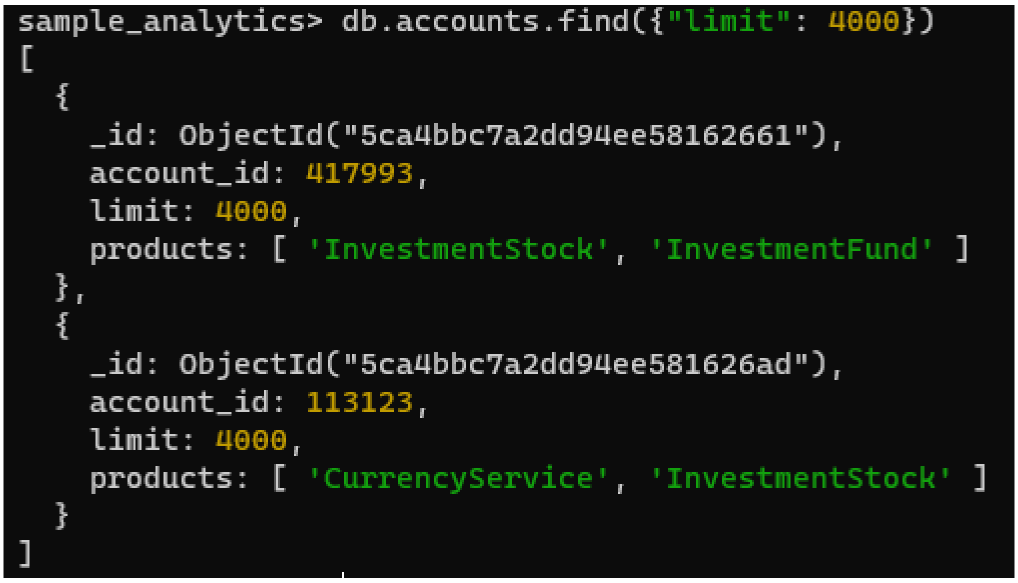
```
sample_analytics> db.runCommand({update: "accounts",updates: [{q: {limit: 3000},
u: { $inc: { limit: 1000 } },multi: true}]})
{ n: 2, nModified: 2, ok: 1 }
sample_analytics> 
```

Figure 5.12: Updating the `limit` as `4000`

To view the update, the user can execute the query as:

```
db.accounts.find({"limit": 4000})
```

Figure 5.13 shows the output of this query.



```
sample_analytics> db.accounts.find({"limit": 4000})
[
  {
    _id: ObjectId("5ca4bbc7a2dd94ee58162661"),
    account_id: 417993,
    limit: 4000,
    products: [ 'InvestmentStock', 'InvestmentFund' ]
  },
  {
    _id: ObjectId("5ca4bbc7a2dd94ee581626ad"),
    account_id: 113123,
    limit: 4000,
    products: [ 'CurrencyService', 'InvestmentStock' ]
  }
]
```

Figure 5.13: Viewing the Updation

5.4.5 `delete` Command

The `delete` command deletes one or more documents from a collection. A single delete command can contain multiple delete specifications.

The syntax for the `delete` command is:

```
db.runCommand(
{
     delete: <collection>,
     deletes: [
     {
          q : <query>,
          limit : <integer>,
          collation: <document>,
          hint: <document|string>
          },
          ...
     ],
     comment: <any>,
     let: <document>, // Added in MongoDB 5.0
     ordered: <boolean>,
     writeConcern: { <write concern> }
}
)
```

Table 5.9 lists the parameters in the `delete` command.

Parameter	Type	Description
`delete`	`string`	Used to specify the name of the target collection from which the document must be deleted
`deletes`	`array`	Used to provide an array of delete statements within the delete command
`ordered`	`boolean`	Used to specify if the remaining commands must be executed in case the `delete` command fails • if set to `true`, the remaining commands will not be executed • if set to `false`, the remaining commands will be executed • Optional • Default is `true`
`writeConcern`	`document`	Used to specify whether the level of write concern for the operation is at the transaction level or at the operation level Optional

Table 5.9: Parameters of the `delete` Command

Table 5.10 lists the fields in the `deletes` array of the `delete` command.

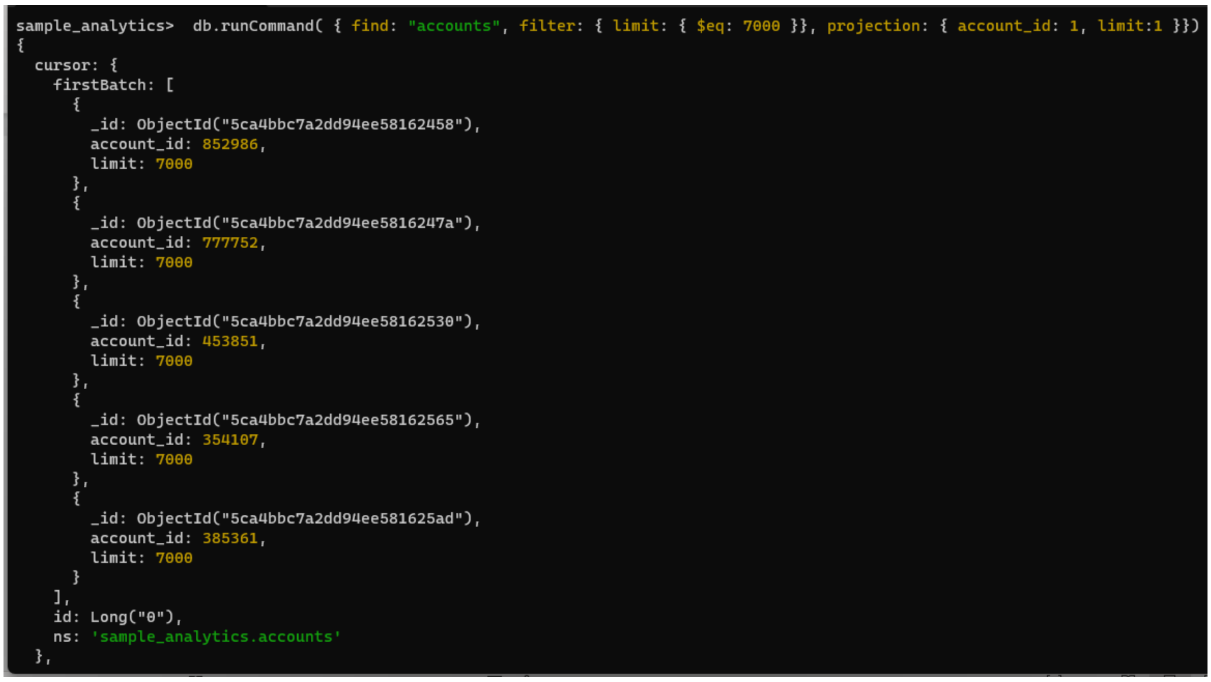
Field	Type	Description
`q`	`document`	Used to specify the selection criteria for the documents using the query selectors
`limit`	`Non-negative integer`	Used to set the number of matching documents to be deleted; must be set to 0 to delete all matching documents and 1 to delete a single matching document
`collation`	`document`	Used to specify the collation rules for the operation that includes language-specific rules for string comparison, such as casing of the alphabets and accent marks

Table 5.10: Fields of the `deletes` Array

Consider the `accounts` collection in the `sample_analytics` database. This collection contains documents that has `limit` set to 7000. To view these documents, the user can run the query as:

```
db.runCommand( { find: "accounts", filter: { limit:
{ $eq: 7000 }}, projection: { account_id: 1, limit:1
}})
```

Figure 5.14 shows five documents that match this query.



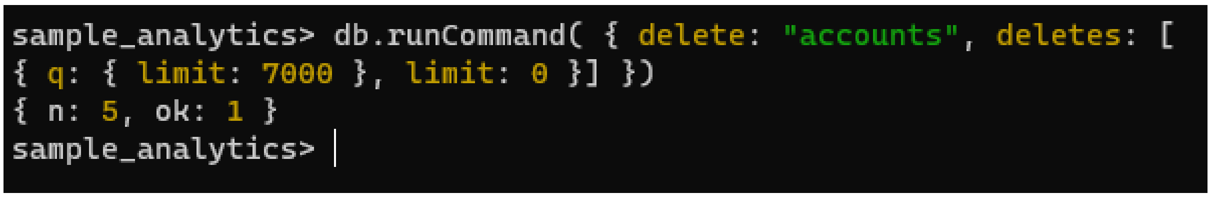
```
sample_analytics>  db.runCommand( { find: "accounts", filter: { limit: { $eq: 7000 }}, projection: { account_id: 1, limit:1 }})
{
  cursor: {
    firstBatch: [
      {
        _id: ObjectId("5ca4bbc7a2dd94ee58162458"),
        account_id: 852986,
        limit: 7000
      },
      {
        _id: ObjectId("5ca4bbc7a2dd94ee5816247a"),
        account_id: 777752,
        limit: 7000
      },
      {
        _id: ObjectId("5ca4bbc7a2dd94ee58162530"),
        account_id: 453851,
        limit: 7000
      },
      {
        _id: ObjectId("5ca4bbc7a2dd94ee58162565"),
        account_id: 354107,
        limit: 7000
      },
      {
        _id: ObjectId("5ca4bbc7a2dd94ee581625ad"),
        account_id: 385361,
        limit: 7000
      }
    ],
    id: Long("0"),
    ns: 'sample_analytics.accounts'
  },
```

Figure 5.14: Viewing all Documents with `limit` as `7000`

To delete all documents with `limit` as `7000`, the user can execute the query as:

```
db.runCommand( { delete: "accounts", deletes: [{ q:
{ limit: 7000 }, limit: 0 }] })
```

In the query, `limit` is set to `0` because all the documents must be deleted. Figure 5.15 shows the output of this query.



```
sample_analytics> db.runCommand( { delete: "accounts", deletes: [
{ q: { limit: 7000 }, limit: 0 }] })
{ n: 5, ok: 1 }
sample_analytics> |
```

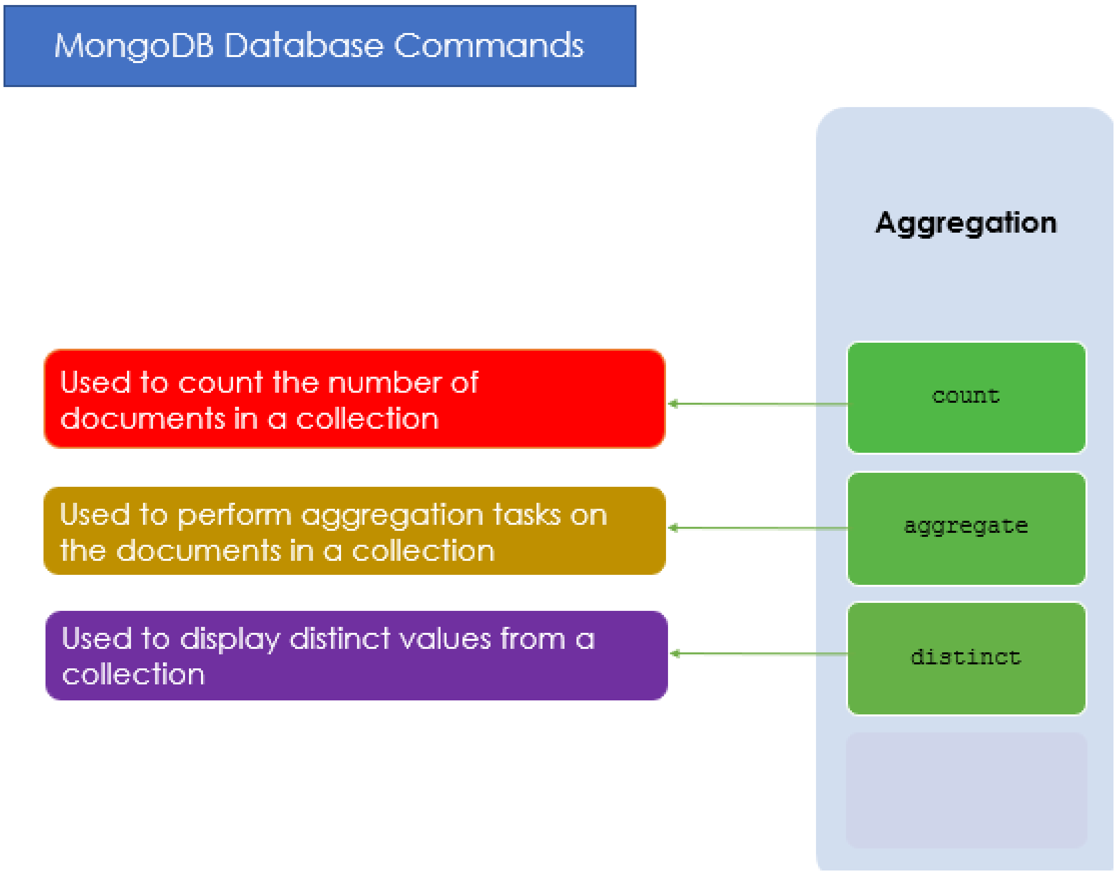
Figure 5.15: Deletion of documents with `limit` as `7000`

5.5 Aggregation Commands

Aggregation commands allow the user to group values from multiple documents, perform operations on the grouped data, and return a single result.

To perform aggregation operations, the user can use aggregation pipelines or single aggregation methods.

Aggregation methods offered by MongoDB are:


MongoDB Database Commands
Aggregation
Used to count the number of documents in a collection
count
Used to perform aggregation tasks on the documents in a collection
aggregate
Used to display distinct values from a collection
distinct

5.5.1 `count` Command

As the name suggests, the `count` command counts the number of documents in a collection. It then returns a document that contains this count and the command status. The syntax of the `count` command is:

```
db.runCommand(
   {
     count: <collection or view>,
     query: <document>,
     limit: <integer>,
     skip: <integer>,
     hint: <hint>,
     readConcern: <document>,
     collation: <document>,
     comment: <any>
   }
)
```

Table 5.11 lists the parameters of the `count` command.

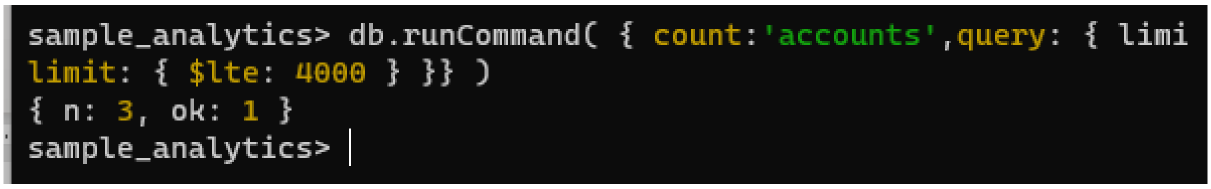
Parameter	Type	Description
`count`	`string`	Used to specify the name of the collection on which the count operation is to be performed
`query`	`document`	Used to specify the selection criteria to filter the documents to count in the collection
`limit`	`integer`	Used to specify the maximum number of matching documents to be returned
`skip`	`integer`	Used to specify the number of matching documents to be skipped before returning results
`hint`	`string`	Used to specify the index to be used
`readConcern`	`document`	Used to specify the read concern level of the operation
`collation`	`document`	Used to specify the collation rules for the operation that includes language-specific rules for string comparison, such as casing of the alphabets and accent marks

Table 5.11: Parameters of the `count` Command

Consider the `accounts` collection in the `sample_analytics` database. To count the number of documents that has `limit` less than or equal to `4000`, the user can execute the query as:

```
db.runCommand({count: "accounts",query: (limit:
($lte: 4000}}})
```

Figure 5.16 shows the output of this query.



```
sample_analytics> db.runCommand( { count:'accounts',query: { limi
limit: { $lte: 4000 } }} )
{ n: 3, ok: 1 }
sample_analytics> 
```

Figure 5.16: Counting Documents that Match a Query

5.5.2 `aggregate` Command

The `aggregate` command performs the aggregation operation using the aggregation pipeline.

The syntax for the `aggregate` command is:

```
db.runCommand(
{
     aggregate: "<collection>" || 1,
     pipeline: [ <stage>, <...> ],
     explain: <boolean>,
     allowDiskUse: <boolean>,
     cursor: <document>,
     maxTimeMS: <int>,
     bypassDocumentValidation: <boolean>,
     readConcern: <document>,
     collation: <document>,
     hint: <string or document>,
     comment: <any>,
     writeConcern: <document>,
     let: <document> // Added in MongoDB 5.0
     }
)
```

Table 5.12 lists the parameters of the `aggregate` command.

Parameter	Type	Description
`aggregate`	`string`	Used to specify the name of the aggregation pipeline
`pipeline`	`array`	Used to give an array that transforms the list of documents as a part of the aggregation pipeline

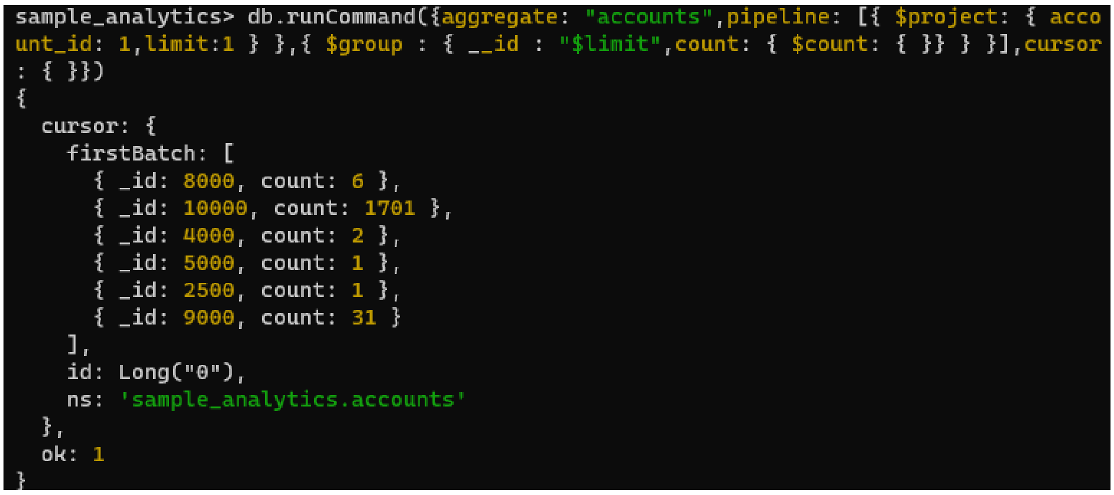
Parameter	Type	Description
`explain`	`boolean`	Used to return information about the processing of the pipeline
`allowDiskUse`	`boolean`	Used to allow the command to write to the temporary files, if set to `true`
`cursor`	`document`	Used to specify the documents that can control the creation of the cursor object
`maxTimeMS`	`non-negative integer`	Used to set a time limit for the processing operations on a cursor
`readConcern`	`document`	Used to specify the read concern level: `local`, `available`, `majority`, or `linearizable`
`collation`	`document`	Used to specify the collation rules for the operation that includes language-specific rules for string comparison, such as casing of the alphabets and accent marks
`comment`	`string`	Used to provide an arbitrary string that helps in tracing the operation through the database profiler, `currentOp`, and logs Optional
`writeConcern`	`document`	Used to set the default write concern with the `$out` or `$merge` pipeline stages

Table 5.12: Parameters of the `aggregate` Command

Consider the `accounts` collection in the `sample_analytics` database. If the user wants a list of available `limit` and the number of documents in each `limit`, then the user can run the query as:

```
db.runCommand({aggregate: "accounts",pipeline: [{
$project: { account_id: 1,limit:1 } },{ $group : {
_id : "$limit",count: { $count: { }} } }],cursor: {
}})
```

Figure 5.17 shows the output of this query.



```
sample_analytics> db.runCommand({aggregate: "accounts",pipeline: [{ $project: { acco
unt_id: 1,limit:1 } },{ $group : { _id : "$limit",count: { $count: { }} } }],cursor
: { }})
{
  cursor: {
    firstBatch: [
      { _id: 8000, count: 6 },
      { _id: 10000, count: 1701 },
      { _id: 4000, count: 2 },
      { _id: 5000, count: 1 },
      { _id: 2500, count: 1 },
      { _id: 9000, count: 31 }
    ],
    id: Long("0"),
    ns: 'sample_analytics.accounts'
  },
  ok: 1
}
```

Figure 5.17: Result of the aggregate Command

5.5.3 `distinct` Command

The `distinct` command locates the distinct values for a specified field from a single collection. This command returns the resultant data as a document containing an array of distinct values. The syntax for this command is:

```
db.runCommand(
{
       distinct: "<collection>",
       key: "<field>",
       query: <query>,
       readConcern: <read concern document>,
       collation: <collation document>,
       comment: <any>
}
)
```

Table 5.13 lists the parameters of the `distinct` command.

Parameter	Type	Description
`distinct`	`string`	Used to specify the name of the collection to use for the query
`key`	`string`	Used to provide the field for which the command must return distinct values
`query`	`document`	Used to specify the documents from where the distinct values must be retrieved
`readConcern`	`document`	Used to specify the read concern level

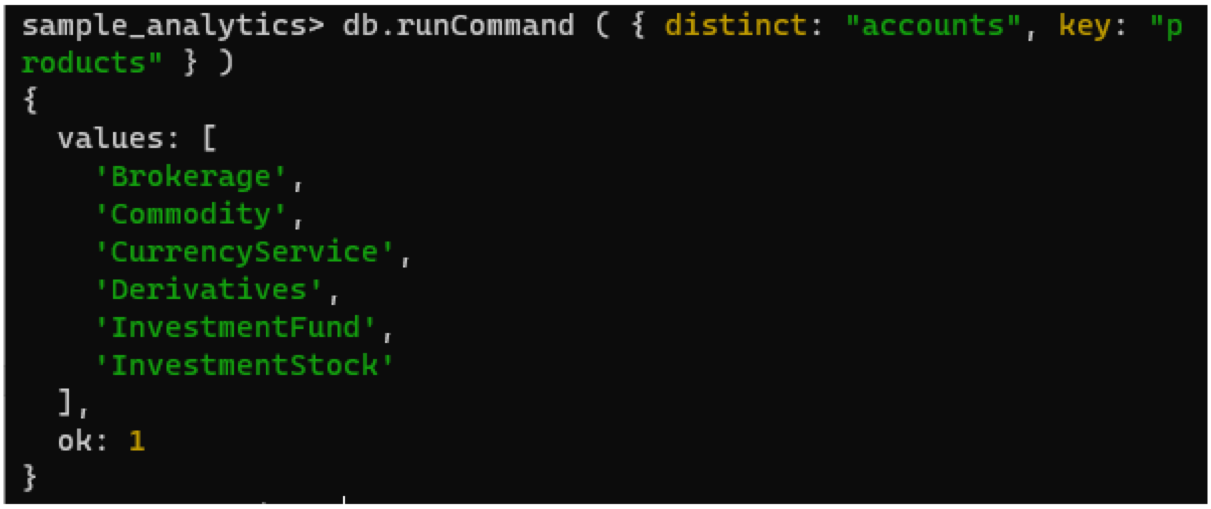
Parameter	Type	Description
`collation`	`document`	Used to specify the collation rules for the operation that includes language-specific rules for string comparison, such as casing of the alphabets and accent marks

Table 5.13: Parameters of the `distinct` Command

Consider the `accounts` collection in the `sample_analytics` database. If the user wants a list of distinct `products` available, then the user can run the query as:

```
db.runCommand ( { distinct: "accounts", key:
"products" } )
```

Figure 5.18 shows the output of the query.



```
sample_analytics> db.runCommand ( { distinct: "accounts", key: "p
roducts" } )
{
  values: [
    'Brokerage',
    'Commodity',
    'CurrencyService',
    'Derivatives',
    'InvestmentFund',
    'InvestmentStock'
  ],
  ok: 1
}
```

Figure 5.18: Distinct Values of the products Array Field

5.6 Summary

- Database commands allow users to create and edit databases.
- MongoDB offers database commands for user management, authentication, write operations, and aggregation.
- User management commands include user creation, user deletion, viewing of user information, and granting user roles.
- Authentication happens using the SCRAM mechanism in MongoDB.
- Write operation commands in MongoDB allows insertion of data to a collection, modifying existing data, and deleting data from a collection.
- Aggregation commands perform operations on grouped values from multiple documents to return a single result.

Test Your Knowledge

1. Which of the following option is used to run a command against the current database?

   a. `db.commandRun`
   b. `db.runAdmin`
   c. `db.runDatabase`
   d. `db.runCommand`

2. Which of the following options are the required fields of createUser command?

   a. `pwd`
   b. `mechanisms`
   c. `createUser`
   d. `roles`

3. Which of the following field of the `deletes` array field of `delete` command represents the query that matches the documents to delete?

   a. `query`
   b. `q`
   c. `qy`
   d. `deleteQuery`

4. Which of the following query will return the total number of documents in the `inventory` collection?

   a. `db.runCommand({count: "inventory"})`
   b. `db.runCommand({aggregate: "inventory"})`
   c. `db.runCommand({aggregate: {count:"inventory"}})`
   d. `db.runCommand({sum: {count:"inventory"}})`

5. Which of the following query returns the distinct values for the field `designation` from all the documents in the `employee` collection?

   a. `db.runCommand ( { distinct: "employee", : "designation" } )`
   b. `db.runCommand ( { unique: "employee", key: "designation" } )`
   c. `db.runCommand ( { distinct: "employee", key: "designation" } )`
   d. `db.runCommand ( { unique: "employee", field: "designation" } )`

Answers to Test Your Knowledge

1	d
2	a, c, d
3	b
4	a
5	c

Try it Yourself

1. Create a database named `product` and a collection named `sales_product`.
2. Insert the following four documents into the `sales_product` collection.

```
[
   { product_id:
1243,product_type:"stationary",branch:"B-
111",units_sold:678,profit_millions:5},
   { product_id: 1144,product_type:"grocery",branch:"B-
112",units_sold:2500,profit_millions:8},
   { product_id: 1345,product_type:"baby
items",branch:"B-113",units_sold:1500,profit_millions:3},
   { product_id: 1567,product_type:"pet care",branch:"B-
114",units_sold:1725,profit_millions:4}
]
```

   Using the `sales_product` collection, perform the following tasks:

3. Using database commands create three users named `db_user1`, `db_user2` and `db_user3` for the database `product` using the password `db1`, `db2`, and `db3`, respectively.
4. Grant `readwrite` role to user `db_user1` and `db_user2` on the `product` database and grant `read` role for both these users on the `inventory` database, which was created in the previous session.
5. Grant `read` role to user `db_user3` on the `product` database and grant `readwrite` role for this user on the `inventory` database.
6. Use a database command to view the information of `db_user1` and `db_user3`.
7. Use a database command to remove the user `db_user2` from the `product` database.
8. Use database command to authenticate the user `db_user1` on the `product` database.
9. Insert the document into the database `product` as:

```
{  product_id: 1899, product_type:"diary", branch:"B-
115", units_sold:3005, profit_millions:6 }
```

10. Update the field `branch` as `B-117` for the documents with `product_type` as "`diary`".
11. Using the `findAndModify` database command, find a product with `product_type` as "`baby items`", modify the `product_type` as "`baby`

foods", and return the document with the modifications made after update.

12. Use aggregate database commands to retrieve the details of the product that has profit_millions greater than 5.