Introduction to MongoDB

Overview

MongoDB is a Database Management System that provides support of storage, management and retrieval of documents.

Often, MongoDB is classified as a "Document Store".

It also belongs to the category of NoSQL DBMS, as its querying facilities are not declarative.

We use MongoDB in its reduced capability as a glorified Key-Value store.

Documents

The objects MongoDB stores are called documents.

Syntactically, MongoDB documents are same as JSON documents.

```
<document> ::= '{' <pair>* '}'
<pair> ::=  <key> ':' <value>
<key>::= <String>
<value> = <document> |
          <array> |
          true | false | null |
          <SimpleValue>
<SimpleValue> ::= <Number>|<String> | ...
```

Unique Identification. Each document stored by MongoDB contains a unique ID stored in a special key-value pair, with the key `"_id"` and called objectid. The objectid can be generated in one of two ways:

1. **Explicitly:** The document to be stored in MongoDB includes the `"_id": <value>` key-value pair. In this case, the document will use the `value` as the objectid.

2. **Implicitly:** If the document does not include the `"_id"` key, then MongoDB generates a unique `objectId` value, and appends

   `"_id": objectid`

   pair to the document.

Representing "Pure" Key-Value Stores in MongoDB

We can represent "pure" Key-Value stores in MongoDB in a number of ways.

Key-Value Store as a Single Document. A single JSON document is essentially a map/dictionary from the attribute names to their values. Therefore, a single MongoDB document is essentially capable of representing a whole Key-Value store.

Example. The following JSON document

```
{ "1": "a",
  "2": "b",
  "3": "alpha"
}
```

represents a $1 \to$ "a", $2 \to$ "b", $3 \to$ "alpha" mapping.

This way of representing a mapping is compact, but as you will see, the ADT `put` operation is going to translate to MongoDB `update` command, rather than the `insert` command.

Key-Value Store as a Collection. MongoDB supports multiple databases and collections (see below). A MongoDB Collection is a group of MongoDB documents stored under the same namespace. The following trasformation allows us to store arbitrary key-value pairs as individual objects in a single collection.

> Given a key-value pair `key: value`, create a MongoDB document
>
> ```
> { "_id": key,
>   "value": value
> }
> ```
>
> and store this object in the collection.

Example. The dictionary from the example above can be represented as the following collection of `MongoDB documents`:

```
{"_id": 1,
 "value": "a"
},
{"_id": 2,
 "value": "b"
},
{"_id": 3,
 "value": "alpha"
}
```

MongoDB Data Organization

MongoDB organizes data into databases, collections. Each represents a specific namespace.

Databases. Databases are the top level of MongoDB namespace hierarchy. At each moment of time, MongoDB can support multiple databases.

Databases are uniquely identified by their names.

Databases are created dynamically. There is no "CREATE DATABASE" command, but if a user requests to work with a non-existing database, a new empty database under that name is created.

Collections. Each database organizes documents into collections. Each collection within a database is identified by a unique name. Each collection is a set of MongoDB documents grouped logically by the user. Collections form namespaces within databases - all access to data in the collection goes through its name:

```
db.<CollectionName>.<operation>
```

For example, the syntax for an operation of getting a list of all documents in the collection (the `find()` operation) `"storage"` will look as follows:

```
db.storage.find()
```

Collection names can include subnamespaces. For example `storage.today` is a valid collection name. This collection is different from `storage` (i.e., it is not part of it).

MongoDB Functionality

MondoDB functionality can be accessed in three different ways:

1. Interactive session with a MongoDB shell.
2. Batch processing of JavaScript scripts by the MongoDB shell.
3. Connection to the MongoDB server via a MongoDB connectivity driver for a specific programming language.

MongoDB Shell

MongoDB runs on a system as a server process. The default MongoDB port is 27017, although some instances of the MongoDB server can override that.

Note: *On the CSL systems MongoDB runs on the default port 27017. You will be contacting MongoDB from the Linux machine on which it is installed, so you will be using* `"localhost"` *as the server address.*

The command to start the MongoDB shell is

```
$ mongo
```

Note: *On the CSL systems where MongoDB is installed, the* `mongo` *executable is in* `/usr/bin`.

Authentification

Our MongoDB server is set up to authentificate users. MongoDB's authentification procedures are unusual.

Authentification Databases. User accounts in MongoDB are pairs ⟨`database`,`username` ⟩. The `database` part of the account is known as the *authentification database*. In order to perform the authentification operation for `username`, one must switch to the authentification database for this user name.

Authentification procedure. The authentification itself is performed using the

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

command. Here, `<username>` is the username that gets authenticated against the database and `<password>` is the password (*unfortnately, provided in plain text*) for the user name.

The full authentification session for a user `test` in the database `payroll` would look as follows:

```
> use payroll
switched to db payroll
> db.auth("test","aaabbb")
```

If authentication works, MongoDB outputs a message stating

```
Successfully authenticated as principal test on payroll
```

Roles. MongoDB uses role-based access control to data. Each authenticated account may have one of the following permissions on a given database:

Permission
`read`
`readWrite`
`dbAdmin`
`dbOwner`
`userAdmin`

`dbAdmin` role grants permission to work with the database meta data.

`userAdmin` role grants permission to create users in the given database and assign them roles.

`dbOwner` permission grants `readWrite`, `dbAdmin` and `userAdmin` permissions on the database.

Getting Started.

MongoDB accepts the following types of commands:

1. Javascript processing instructions. MongoDB shell is a complete JavaScript interpreter.
2. Shell commands. There is a set of shell commands that help navigate and administrate the system.
3. Database and collection manipulations. These access the data stored in MongoDB. Syntactically, these commands start with `db.` and have the form of a method/function call.

Basic shell commands. Five commands are useful from the very beginning:

Command	Meaning
`help`	Display help message. Shows what types of help can be requested.
`show dbs`	Produces a list of available databases.
`use <dbName>`	Changes the active database.
`show collections`	Lists all collections in the active database.
`exit`	Quit the MongoDB shell.

CRUD. All actual data access operations are conducted through the

```
db.<collection>.<operation>
```

syntax. To see the full list of Collection-level commands, type

```
> db.mycoll.help()
```

Of the various commands, the following are of immediate interest:

Command	Meaning
`db.<collection>.insert(<document>)`	Puts `<document>` into a given `<collection>`.
`db.<collection>.find()`	Shows the list of documents in the collection.
`db.<collection>.remove(<patternDoc>)`	removes documents from the collection that match the pattern specified in `<patternDoc>`
`db.<collection>.update(<patternDoc>, <document>)`	find the document that satisfies the pattern specified in `<paternDoc>` and replace it with `<document>`

Inserting new documents

Inserting one document object at a time. The basic format of the `db.<collection>.insert()` command is

```
db.<collection>.insert(<object>}
```

This command inserts one object `<object>` into the specified `<collection>`.

Note: If `<object>` does not contain an `"_id"` key, then the key-value pair

```
"_id": new ObjectId()
```

will be added to `<object>`.

Bulk Insert. The `insert` command can be used to bulk-insert objects. For this, the argument of the `insert` command needs to change to a JSON (document) array. The format of the command is

```
db.collection.insert([<object>,...,<object>])
```

Example. Starting with an empty collection `ex1`, the following are the effects of a single object `insert` and a bulk `insert`.

```
> db.ex1.find()

> db.ex1.insert({
... "name":"Bob",
... "years": 3})
WriteResult({ "nInserted" : 1 })

> db.ex1.find()
```

```
{ "_id" : ObjectId("569c3f7fd37cd1c58b142632"), "name" : "Bob", "years" : 3 }

> db.ex1.insert({"_id":"Alice", "name":"Alice", "years": 21})
WriteResult({ "nInserted" : 1 })

> db.ex1.find()
{ "_id" : ObjectId("569c3f7fd37cd1c58b142632"), "name" : "Bob", "years" : 3 }
{ "_id" : "Alice", "name" : "Alice", "years" : 21 }

> db.ex1.insert([{"name": "Sue", "years": 4},
  ...            {"name":"Will", "years": 12},
  ...            {"name":"Hank", "lastseen":"yesterday"} ])
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 3,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})

> db.ex1.find()
{ "_id" : ObjectId("569c3f7fd37cd1c58b142632"), "name" : "Bob", "years" : 3 }
{ "_id" : "Alice", "name" : "Alice", "years" : 21 }
{ "_id" : ObjectId("569c4019d37cd1c58b142633"), "name" : "Sue", "years" : 4 }
{ "_id" : ObjectId("569c4019d37cd1c58b142634"), "name" : "Will", "years" : 12 }
{ "_id" : ObjectId("569c4019d37cd1c58b142635"), "name" : "Hank", "lastseen" : "yesterday" }
>
```

Removing Documents and Collections

`db.<collection>.remove` command can be used to remove a full collection, a single document, and a group of documents that match a specific query.

Removing full collection. The

```
db.<collection>.remove({})
```

command an empty document as an argument removes all documents from the collection `<collection>`.

Removing documents. The `remove` command takes one input parameter: a query document.

```
db.<collection>.remove(<queryDocument>)
```

This command searches the collection `<collection>` for all documents that match `<queryDocument>`. All these documents are removed from `<collection>`. See the description of the `find()` command for the syntax of `<queryDocument>`.

Example. Here are some examples of the work of the `remove` command.

```
 > db.ex1.find()
{ "_id" : ObjectId("569c3f7fd37cd1c58b142632"), "name" : "Bob", "years" : 3 }
{ "_id" : "Alice", "name" : "Alice", "years" : 21 }
{ "_id" : ObjectId("569c4019d37cd1c58b142633"), "name" : "Sue", "years" : 4 }
{ "_id" : ObjectId("569c4019d37cd1c58b142634"), "name" : "Will", "years" : 12 }
{ "_id" : ObjectId("569c4019d37cd1c58b142635"), "name" : "Hank", "lastseen" : "yesterday" }

>  // deleting a single document
> db.ex1.remove({"_id":"Alice"})
WriteResult({ "nRemoved" : 1 })

> db.ex1.find()
{ "_id" : ObjectId("569c3f7fd37cd1c58b142632"), "name" : "Bob", "years" : 3 }
{ "_id" : ObjectId("569c4019d37cd1c58b142633"), "name" : "Sue", "years" : 4 }
{ "_id" : ObjectId("569c4019d37cd1c58b142634"), "name" : "Will", "years" : 12 }
{ "_id" : ObjectId("569c4019d37cd1c58b142635"), "name" : "Hank", "lastseen" : "yesterday" }

> // deleting multiple documents
> db.ex1.remove({"years": {"$lt" : 5}})
WriteResult({ "nRemoved" : 2 })
> db.ex1.find()
{ "_id" : ObjectId("569c45fdd37cd1c58b142638"), "name" : "Will", "years" : 12 }
{ "_id" : ObjectId("569c45fdd37cd1c58b142639"), "name" : "Hank", "lastseen" : "yesterday" }

> // deleting all documents in a collection
> db.ex1.remove({})
WriteResult({ "nRemoved" : 2 })
> db.ex1.find()
>
```

Querying Documents in a Collection

The MongoDB query command is `db.<collection>.find(<queryDocument>, <projectionDocument>)`.

A `<queryDocument>` is a JSON document describing the specific search parameters that `find` must use when conducting the search.

A `<projectionDocument>` is a JSON document specifying what part of a document matching the query needs to be retrieved. This parameter is *optional*. If missing, full documents matching the query are returned.

List all documents in a collection. The `find()` command without the input parameter lists all objects in a given collection:

```
db.<collection>.find()
```

Query Documents

MongoDB allows for a lot of different types of queries through the use of the query documents.

Query docuements for exact match. To retrieve documents where there is a desired **top-level** key-value pair `key: value`, use the following Query document:

```
{ key: value}
```

Example. Consider the following collection `ex2`:

```
> db.ex2.find()
{ "_id" : 1, "course" : "CSC 365", "roster" : 34 }
{ "_id" : 2, "course" : "CSC 369", "roster" : 61 }
{ "_id" : 3, "course" : "CSC 466", "name" : "Data Mining", "roster" : 34 }
{ "_id" : 4, "course" : "CSC 357", "name" : "Systems Programming", "roster" : 122 }
{ "_id" : 5, "course" : "CSC 101", "roster" : 100 }
{ "_id" : 6, "course" : "CSC 430", "roster" : 60 }
{ "_id" : 7, "course" : "CSC 102", "roster" : 94, "sections" : 3 }
{ "_id" : 8, "course" : "CSC 480", "roster" : 32, "sections" : 1 }
```

To extract information about CSC 365, we construct the following query:

```
> db.ex2.find({"course":"CSC 365"})
{ "_id" : 1, "course" : "CSC 365", "roster" : 34 }
```

To find the courses that have exactly 34 students, we can ask:

```
> db.ex2.find({"roster":34})
{ "_id" : 1, "course" : "CSC 365", "roster" : 34 }
{ "_id" : 3, "course" : "CSC 466", "name" : "Data Mining", "roster" : 34 }
```

Comparisons. MongoDB uses the `"$<keyword>"` syntax for keywords expressing a variety of different requests. The following keywords are reserved for comparison operators:

Operator	MongoDB keyword
$=$	`"$eq"`
$\neq$	`"$ne"`
$<$	`"$lt"`
$\leq$	`"$lte"`
$>$	`"$ge"`
$\geq$	`"$gte"`

To use comparisons in a query, prepare the query document according to the following template:

```
{<key> : {"$comparison" : <value>}}
```

For example, to find all classes with fewer than 40 students in them, we can issue the following query:

```
> db.ex2.find({"roster": {"$lt": 40}})
{ "_id" : 1, "course" : "CSC 365", "roster" : 34 }
{ "_id" : 3, "course" : "CSC 466", "name" : "Data Mining", "roster" : 34 }
{ "_id" : 8, "course" : "CSC 480", "roster" : 32, "sections" : 1 }
```

Comparisons on a single key can be combined:

```
> db.ex2.find({"roster": {"$lt": 40, "$gt": 33}})
{ "_id" : 1, "course" : "CSC 365", "roster" : 34 }
{ "_id" : 3, "course" : "CSC 466", "name" : "Data Mining", "roster" : 34 }
```

Boolean Combinations. *Conjunction*, *disjunction* and *negation* are handled via conditional keywords. Some special cases may also be handled in a separate way. The relevant keywords are:

Operator	Operation	MongoDB keyword
$\&$	conjunction	`"$and"`
$\vee$	disjunction	`"$or"`
$\in$	belongs to	`"$in"`
$\notin$	does not belong to	`"$nin"`
$\neg$	negation	`"$not"`

1. **Conjunction.** Use multiple `key:value` pairs in a single query for a conjunctive query:

   ```
   > db.ex2.find({"roster":34, "name":"Data Mining"})
   { "_id" : 3, "course" : "CSC 466", "name" : "Data Mining", "roster" : 34 }
   ```

   Alternatively, use the built-in keyword `"$and"` as follows:

   ```
   {"$and": [ <conjunct1>,...,<conjunctN>]}
   ```

   Here, `<conjunct1>,..., <conjunctN>` are query documents specifying each conjunct.

   For example, the same query as above can be restated as follows:

   ```
   > db.ex2.find({"$and": [{"roster":34},{"course":"CSC 466"}]})
   { "_id" : 3, "course" : "CSC 466", "name" : "Data Mining", "roster" : 34 }
   ```

2. **Disjunction.** Arbitrary disjunction can be represented using the following query document template:

   ```
   {"$or" : [<option1>, <option2>,...,<optionN>]}
   ```

   Here, `"$or"` is a disjunction keyword, and `<option1>,...,<optionN>` are query documents specifying the disjuncts.

   The following example retrieves CSC 365 and CSC 369 records:

   ```
   > db.ex2.find({"$or": [{"course":"CSC 365"}, {"course": "CSC 369"}]})
   { "_id" : 1, "course" : "CSC 365", "roster" : 34 }
   { "_id" : 2, "course" : "CSC 369", "roster" : 61 }
   ```

   In this query, we ask for either a CSC 365 record or records of all courses with more than 50 students in them

```
> db.ex2.find({"$or": [{"course": "CSC 365"}, {"roster": {"$gt": 50}}]})
{ "_id" : 1, "course" : "CSC 365", "roster" : 34 }
{ "_id" : 2, "course" : "CSC 369", "roster" : 61 }
{ "_id" : 4, "course" : "CSC 357", "name" : "Systems Programming", "roster" : 122 }
{ "_id" : 5, "course" : "CSC 101", "roster" : 100 }
{ "_id" : 6, "course" : "CSC 430", "roster" : 60 }
{ "_id" : 7, "course" : "CSC 102", "roster" : 94, "sections" : 3 }
```

A simplified version of disjunction, ranging over multiple values of a single key (e.g., "find all courses with either 34 or 100 students") can be encoded using an `"$in"` keyword. The query document shall look like this:

```
{<key> : {"$in": [<value1>,..., <valueN>]}}
```

The query mentioned above can be encoded as follows:

```
> db.ex2.find({"roster": {"$in": [34, 100]}})
{ "_id" : 1, "course" : "CSC 365", "roster" : 34 }
{ "_id" : 3, "course" : "CSC 466", "name" : "Data Mining", "roster" : 34 }
{ "_id" : 5, "course" : "CSC 101", "roster" : 100 }
```

It is also possible to query for values to **not** be included in the list, by using the `"$nin"` keyword instead of `"$in"`.

```
> db.ex2.find({"roster": {"$nin": [34, 100]}})
{ "_id" : 2, "course" : "CSC 369", "roster" : 61 }
{ "_id" : 4, "course" : "CSC 357", "name" : "Systems Programming", "roster" : 122 }
{ "_id" : 6, "course" : "CSC 430", "roster" : 60 }
{ "_id" : 7, "course" : "CSC 102", "roster" : 94, "sections" : 3 }
{ "_id" : 8, "course" : "CSC 480", "roster" : 32, "sections" : 1 }
```

3. **Negation.** Negation must be applied to an *operation* within a query document. The negation keyword is `"$not"`. The syntax for negation is as follows:

```
{<key> : {"$not" : {"$operation" : <document>}}}
```

For example, finding all courses that do not have 3 sections can be done as follows:

```
> db.ex2.find({"sections": {"$not" : {"$eq": 3}}})
{ "_id" : 1, "course" : "CSC 365", "roster" : 34 }
{ "_id" : 2, "course" : "CSC 369", "roster" : 61 }
{ "_id" : 3, "course" : "CSC 466", "name" : "Data Mining", "roster" : 34 }
{ "_id" : 4, "course" : "CSC 357", "name" : "Systems Programming", "roster" : 122 }
{ "_id" : 5, "course" : "CSC 101", "roster" : 100 }
{ "_id" : 6, "course" : "CSC 430", "roster" : 60 }
{ "_id" : 8, "course" : "CSC 480", "roster" : 32, "sections" : 1 }
```

Note: Notice the **semantics** of this query. The query returns both:

- documents that do not have a key-value pair for the specified key;
- documents that have a key-value pair for the specified key, with the value not equal to the value specified in the query document.

The `"db.ex2.find("sections": "$ne": 3)"` returns the same result, so, broadly speaking, this is how MongoDB handles *inequality*.

Matching `null` The behavior of the negation is explained by how matching of `null` values is handled.

The query document

```
{<key>: null}
```

matches:

- All documents in the collection that contain an explicit key-value pair `<key>: null`;
- All documents in the collection that do not contain a key-value pair for `<key>`.

Example. Consider the collection `ex2` used in the examples above, enhanced with two more objects:

```
> db.ex2.insert({"_id":9, "course": "CSC 400", "roster": null, "sections": 8})
WriteResult({ "nInserted" : 1 })
> db.ex2.insert({"_id":10, "course": "CSC 103",  "sections": 8})
WriteResult({ "nInserted" : 1 })
> db.ex2.find()
{ "_id" : 1, "course" : "CSC 365", "roster" : 34 }
{ "_id" : 2, "course" : "CSC 369", "roster" : 61 }
{ "_id" : 3, "course" : "CSC 466", "name" : "Data Mining", "roster" : 34 }
{ "_id" : 4, "course" : "CSC 357", "name" : "Systems Programming", "roster" : 122 }
{ "_id" : 5, "course" : "CSC 101", "roster" : 100 }
{ "_id" : 6, "course" : "CSC 430", "roster" : 60 }
{ "_id" : 7, "course" : "CSC 102", "roster" : 94, "sections" : 3 }
{ "_id" : 8, "course" : "CSC 480", "roster" : 32, "sections" : 1 }
{ "_id" : 9, "course" : "CSC 400", "roster" : null, "sections" : 8 }
{ "_id" : 10, "course" : "CSC 103", "sections" : 8 }
```

Asking for all courses with `"roster"` set to `null` yields the following response:

```
> db.ex2.find({"roster":null})
{ "_id" : 9, "course" : "CSC 400", "roster" : null, "sections" : 8 }
{ "_id" : 10, "course" : "CSC 103", "sections" : 8 }
```

Querying Arrays

Querying arrays has interesting semantics in MongoDB.

Let `"key": [ value1, value2,..., valueN]` be a JSON document.

A query document `"key": value` will match the JSON object above if `value` is equal to **one of** the values `value1,...,valueN`.

At the same time:

A query document `"key": [v1, ..., vM]` matches the JSON object above, if `M = N` and, `value1 = v1`, `value2 = v2`,..., `valueN = vM`.

Example. Consider the following collection `ex3`:

```
> db.ex3.find()
{ "_id" : 1, "name" : "Joe", "courses" : [ 365, 369, 466 ] }
{ "_id" : 2, "name" : "Paula", "courses" : [ 471, 365 ] }
{ "_id" : 3, "name" : "Vanda", "courses" : 365 }
{ "_id" : 4, "name" : "Morris", "courses" : [ 102, 103, 225, 357 ] }
```

Consider the four queries shown below. The first one finds all students who took CSC 365. The second one shows that no student took exactly CSC 365 and CSC 369. The third query matches nothing, because the courses are in the wrong order (arrays are ordered). Finally, the last query discovers the person who took CSC 471 and CSC 365 (in the specified order).

```
> db.ex3.find({"courses": 365})
{ "_id" : 1, "name" : "Joe", "courses" : [ 365, 369, 466 ] }
{ "_id" : 2, "name" : "Paula", "courses" : [ 471, 365 ] }
{ "_id" : 3, "name" : "Vanda", "courses" : 365 }
> db.ex3.find({"courses": [365, 369]})
> db.ex3.find({"courses": [365, 471]})
> db.ex3.find({"courses": [471, 365]})
{ "_id" : 2, "name" : "Paula", "courses" : [ 471, 365 ] }
```

Accessing individual array elements. To access individual array elements one can construct the query document according to the following template:

```
{ "<key>.N" : <value>}
```

For example, the following query retrieves the information about students who list CSC 369 as their second course:

```
> db.ex3.find({"courses.1" : 369} )
{ "_id" : 1, "name" : "Joe", "courses" : [ 365, 369, 466 ] }
```

Accessing Array Size. The keyword `"$size"` in a query document evaluates to the size (number of elements) in the array stored under a given key. The following syntax allows to query for arrays of certain sizes:

```
{<key> : {"$size": Number}}
```

For example, the following query retrieves records of all students taking exactly two courses. Note also, that *no conditionals/comparison operators can be used together with* `"$size"`: the second query, ostensible asking for all records with more the two courses returns a syntax error.

```
 > db.ex3.find({"courses": {"$size": 2}})
{ "_id" : 2, "name" : "Paula", "courses" : [ 471, 365 ] }

> db.ex3.find({"courses": {"$size": {"$gt": 2}}})
Error: error: {
        "waitedMS" : NumberLong(0),
        "ok" : 0,
        "errmsg" : "$size needs a number",
        "code" : 2
}
```

Array elements and conditionals. A query document of the form

```
{<key>: {"$gt": value1, "$lt": value2}}
```

matches the following types of documents:

- a document that has a key-value pair for key `<key>` with a simple value `val` that is greater than `value1` and smaller than `value2`.
- a document that has a key-value pair for key `<key>` with an array value that has values `val1` and `val2` (possibly the same, but not required to be), such that

$$\texttt{val1} > \texttt{value1} \text{ and } \texttt{val2} < \texttt{value2}$$

We can illustrate this behavior on the example of the following query:

```
> db.ex3.find({"courses": {"$gt": 370, "$lt": 400}})
{ "_id" : 1, "name" : "Joe", "courses" : [ 365, 369, 466 ] }
{ "_id" : 2, "name" : "Paula", "courses" : [ 471, 365 ] }
```

Document Queries for Matching Embedded Documents

When a value for a specific key in a document is a JSON document itself, we can query in two different ways.

1. **Full match queries.** A query document can be constructed to match **exactly** the contents of the embedded document. The syntax is similar to the standard query document syntax:

   ```
   { "<key>": { "<ekey1>": value1, ..., "<ekeyN>": valueN}}
   ```

This query will match exactly a document that has a key-value pair with the key `<key>`, whose value is an embedded object

```
{ "<ekey1>": value1,
  ...
  "<ekeyN>": valueN
}
```

2. **Partial match queries.** Often, rather than quering for the full contents of an object, we want to achieve the same result as with top-level key-value pairs: match and object if *it contains a specific key-value pair*. This can be done using the following syntax:

   ```
   { "<key>.<ekey>": value}
   ```

   This query document matches documents that have a key-value pair with the key `<key>` whose value is an embedded document, which has a key-value pair `<ekey>: value`.

Some examples illustrating it are below. We use the following collection `ex4` (indentation added for clarity):

```
> db.ex4.find()
{ "_id" : 1, "person" : { "first" : "Bob", "last" : "Smith" }, "age" : 37 }
{ "_id" : 2, "person" : { "first" : "Anna", "last" : "McCarthy" },
             "occupation" : "medical" }
{ "_id" : 3, "person" : { "first" : "Alice", "last" : "Brown", "middlie" : "Matilda" },
             "email" : "amb456@gmail.com" }
```

Querying for both first and last name of Bob Smith returns his record. Querying in a "standard" way for just "Bob" returns nothing. Using the dot.notation we can retrieve Bob's record just using his first name.

```
> db.ex4.find({"person": {"first": "Bob"}})

> db.ex4.find({"person": {"first": "Bob", "last":"Smith" }})
{ "_id" : 1, "person" : { "first" : "Bob", "last" : "Smith" }, "age" : 37 }

> db.ex4.find({"person.first":  "Bob"})
{ "_id" : 1, "person" : { "first" : "Bob", "last" : "Smith" }, "age" : 37 }
```

Projection operation

In relational databases, projection operation allows for the return of only the relevant parts of the database tables. MongoDB employs the second (optional) argument of the `find()` command to restrict the retrieved content.

Returning the exact objects stored in the collections. When `db.<collection>.find()` is invoked *without the second parameter*, the result of the query will show the full contents of all retrieved objects.

Limiting the retrieved content. To limit the information returned for each retrieved document, use the second parameter of the `find()` command. We refer to this parameter as the Projection document.

Projection Documents. There are two types of projection actions allowed in projection documents: an inclusion action and an exclusion action.

1. **Inclusions.** To specify the list of key-value pairs to be included in the output, construct the following projection document:

   ```
   {"<key1>": 1,
    "<key2>": 1,
     ...
    "<keyN>": 1
   }
   ```

   If this projection document is specified as the second parameter of a `find()` command, all retrieved objects will only contain key-value pairs with keys coming from the list `"<key1>",..., "<keyN>"`. All other key-value pairs will not be placed in the output.

2. **Exclusions.** To specify the list of key-value pairs to be excluded in the output, construct the following projection document:

   ```
   {"<key1>": 0,
    "<key2>": 0,
     ...
    "<keyN>": 0
   }
   ```

   If this projection document is specified as the second parameter of a `find()` command, all retrieved objects will only contain key-value pairs with keys **not found** in the list `"<key1>",..., "<keyN>"`. All other key-value pairs will be placed in the output.

Example. Using our collection `ex2`, we can test projections on a few queries. The next to last query shows that mixing the exclusion and inclusion in a single projections document is not allowed. The only exception from this rule is the `"_id"` key, which can be excluded anytime (see the last query).

```
> db.ex2.find({"roster": 34}, {"course": 1})
{ "_id" : 1, "course" : "CSC 365" }
{ "_id" : 3, "course" : "CSC 466" }

> db.ex2.find({"roster": 34}, {"course": 0})
{ "_id" : 1, "roster" : 34 }
{ "_id" : 3, "name" : "Data Mining", "roster" : 34 }

> db.ex2.find({"course": {"$in": ["CSC 365", "CSC 466" ]}}, {"course": 1, "name": 1})
```

```
{ "_id" : 1, "course" : "CSC 365" }
{ "_id" : 3, "course" : "CSC 466", "name" : "Data Mining" }

> db.ex2.find({"course": {"$in": ["CSC 365", "CSC 466" ]}}, {"course": 0, "name": 0})
{ "_id" : 1, "roster" : 34 }
{ "_id" : 3, "roster" : 34 }

> db.ex2.find({"course": {"$in": ["CSC 365", "CSC 466" ]}}, {"course": 0, "name": 1})
Error: error: {
        "waitedMS" : NumberLong(0),
        "ok" : 0,
        "errmsg" : "Projection cannot have a mix of inclusion and exclusion.",
        "code" : 2
}

> db.ex2.find({"course": {"$in": ["CSC 365", "CSC 466" ]}}, {"course": 1, "_id":0})
{ "course" : "CSC 365" }
{ "course" : "CSC 466" }
```

Finishing Query Results

MongoDB allows the results of any `find()` request to be post-processed using one of the following commands:

Command	Effect
`db.<collection>.find(...).limit(<Number>)`	set an upper limit on the number of returned documents
`db.<collection>.find(...).skip(<Number>)`	skip the first `<Number>` of matching documents
`db.<collection>.find(...).sort(<SortDocument>)`	sort the output according to the spec in `<SortDocument>`
`db.<collection>.find(...).count()`	return the number of retrieved documents

Sort Document Syntax. A sort document provides a specification on the sort order. It has the format:

```
{ "<key1>": SortOrder1, ..., "<keyM>": SortOrderM }
```

Here, `SortOrder1`,... `SortOrderM` are the specifications for the sort order on the specific key. The sort order specifications can take the following values:

Value	Meaning
1	ascending sort order
−1	descending sort order

The three postprocessing commands/methods can be chained.

Examples. The following examples show how these commands work.

```
> db.ex2.find().limit(3)
{ "_id" : 1, "course" : "CSC 365", "roster" : 34 }
{ "_id" : 2, "course" : "CSC 369", "roster" : 61 }
{ "_id" : 3, "course" : "CSC 466", "name" : "Data Mining", "roster" : 34 }

> db.ex2.find().skip(3)
{ "_id" : 4, "course" : "CSC 357", "name" : "Systems Programming", "roster" : 122 }
{ "_id" : 5, "course" : "CSC 101", "roster" : 100 }
```

```
{ "_id" : 6, "course" : "CSC 430", "roster" : 60 }
{ "_id" : 7, "course" : "CSC 102", "roster" : 94, "sections" : 3 }
{ "_id" : 8, "course" : "CSC 480", "roster" : 32, "sections" : 1 }
{ "_id" : 9, "course" : "CSC 400", "roster" : null, "sections" : 8 }
{ "_id" : 10, "course" : "CSC 103", "sections" : 8 }


> db.ex2.find({"roster": 34}).sort({"_id": -1})
{ "_id" : 3, "course" : "CSC 466", "name" : "Data Mining", "roster" : 34 }
{ "_id" : 1, "course" : "CSC 365", "roster" : 34 }

> db.ex2.find().limit(3).skip(3).sort({"roster": -1})
{ "_id" : 2, "course" : "CSC 369", "roster" : 61 }
{ "_id" : 6, "course" : "CSC 430", "roster" : 60 }
{ "_id" : 1, "course" : "CSC 365", "roster" : 34 }
```

Aggregation. The `count()` method aggregates the results of the query. Instead of reporting the documents, it reports the total number of the documents returns.

```
> db.ex2.find({"roster": {"$in": [34, 100]}})
{ "_id" : 1, "course" : "CSC 365", "roster" : 34 }
{ "_id" : 3, "course" : "CSC 466", "name" : "Data Mining", "roster" : 34 }
{ "_id" : 5, "course" : "CSC 101", "roster" : 100 }
> db.ex2.find({"roster": {"$in": [34, 100]}}).count()
3
```

Javascript Scripting

As mentioned above, MongoDB shell is a full-scale Javascript processor. Javascript code can be run within the shell, Javascript programs can be loaded and run from the shell. Additionally, Javascript programs can be run by invoking `mongo`, the MongoDB shell in batch mode.

To build simple Javascript programs that batch-execute of MongoDB commands, you need to know how to connect to the server instance and how to select a database.

Connection to server. Use `Mongo()` constructor. The following variants are available:

Constructor	Action
`new Mongo()`	creates connection to the MongoDB instance on local host at the default port
`new Mongo("<host>")`	creates connection to the MongoDB instance at the specified host address at the default port
`new Mongo("<host>:<port>")`	creates connection to the MongoDB instance at the specified host and port

To connect to the local server place this code at the top of your script:

```
connection = new Mongo(); // use any variable name you like
```

Database Selection. To select a database to work with, use the `getDB()` method on the connection object. The method is invoked after the connection object is instantiated. The syntax is as follows:

```
<dbVar> = <connVar>.getDB("<dbName>");
```

For example, to work with the database `"examples"` on the local server, put this code at the beginning of your script:

```
connection = new Mongo();
db = connection.getDB("examples");
```

Running collection commands. You can put MongoDB collection manipulation commands directly into the body of your Javascript program. The following script inserts one object into the `ex1` collection.

```
connection = new Mongo();
db = connection.getDB("examples");
db.ex1.insert({"item": "one"});
```

Cursors. To retrieve data from MongoDB use Javascript cursors. The following code runs through the contents on an entire collection and prints each object:

```
connection = new Mongo();
db = connection.getDB("examples");

cursor = db.ex1.find();
while (cursor.hasNext()) {
  x = cursor.next();
  printjson(x);
}
```