MongoDB-I

UNIT-3

Introduction

- MongoDB is powerful but easy to get started with:

Basic Concepts of MongoDB

- A *document* is the basic unit of data for MongoDB and is roughly equivalent to a row in a relational database management system
- A *collection* can be thought of as a table with a dynamic schema.
- A single instance of MongoDB can host multiple independent *databases*, each of which can have its own collections
- Every document has a special key, "_id", that is unique within a collection
- MongoDB comes with a simple but powerful JavaScript *shell*, which is useful for the administration of MongoDB instances and data manipulation

Documents

- At the heart of MongoDB is the *document*: an ordered set of keys with associated values
- The representation of a document varies by programming language, but most languages have a data structure that is a natural fit, such as a map, hash, or dictionary. In JavaScript, for example, documents are represented as objects:

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

- This simple document contains a single key, "greeting", with a value of "Hello, world!". Most documents will be more complex than this simple one and often will contain multiple key/value pairs:

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

- The keys in a document are strings. Any UTF-8 character is allowed in a key, with a few notable exceptions:
- Keys must not contain the character \0 (the null character). This character is used to signify the end of a key.
- The . and $ characters have some special properties and should be used only in certain circumstances

- MongoDB is type-sensitive and case-sensitive. For example, these documents are distinct:

{"foo" : 3}

{"foo" : "3"}

as are as these:

{"foo" : 3}

{"Foo" : 3}

- A final important thing to note is that documents in MongoDB cannot contain duplicate keys. For example, the following is not a legal document:

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

- Key/value pairs in documents are ordered: {"x" : 1, "y" : 2} is not the same as {"y" : 2, "x" : 1}. Field order does not usually matter and you should not design your schema to depend on a certain ordering of fields.This text will note the special cases where field order is important.

Collections

- *A collection* is a group of documents. If a document is the MongoDB analog of a row in a relational database, then a collection can be thought of as the analog to a table.

Dynamic Schemas:

- Collections have *dynamic schemas*. This means that the documents within a single collection can have any number of different “shapes.” For example, both of the following documents could be stored in a single collection:

```
{"greeting" : "Hello, world!"}
{"foo" : 5}
```

- Keeping different kinds of documents in the same collection can be a nightmare for developers and admins. Developers need to make sure that each query is only returning documents of a certain type or that the application code performing a query can handle documents of different shapes.
- It is much faster to get a list of collections than to extract a list of the types in a collection.
- Grouping documents of the same kind together in the same collection allows for data locality.

Naming

- A collection is identified by its name. Collection names can be any UTF-8 string, with a few restrictions:
- The empty string ("") is not a valid collection name.
- Collection names may not contain the character \0 (the null character) because this delineates the end of a collection name.
- Should not create any collections that start with *system.*, a prefix reserved for internal collections.
  - Eg: system. Users-collection contains the database's users ,system. Namespace collection contains information about all database collections
- User-created collections should not contain the reserved character $ in the name.

Subcollections:

- One convention for organizing collections is to use namespaced subcollections separated by the . Character
  - For example, an application containing a blog might have a collection named *blog. Posts* and a separate collection named *blog. Authors*
- Subcollections are a great way to organize data in MongoDB, and their use is highly recommended

Databases

- In addition to grouping documents by collection, MongoDB groups collections into *databases*.
- A single instance of MongoDB can host several databases, each grouping together zero or more collections.
- A database has its own permissions, and each database is stored in separate files on disk.
- A good rule of thumb is to store all data for a single application in the same database. Separate databases are useful when storing data for several application or users on the same MongoDB server
- Databases are identifyed by name. Database names can be any UTF-8 string, with the following restrictions:
  - The empty string ("") is not a valid database name.
  - A database name cannot contain any of these characters: /, \, ., ", *, <, >, :, |, ?, $, (a single space), or \0 (the null character).
  - Database names are case-sensitive, even on non-case-sensitive filesystems.
  - Database names are limited to a maximum of 64 bytes

- There are also several reserved database names, which you can access but which have special semantics. These are as follows:

Admin :

- This is the “root” database, in terms of authentication. If a user is added to the *admin* database, the user automatically inherits permissions for all databases. There are also certain server-wide commands that can be run only from the *admin* database, such as listing all of the databases or shutting down the server.

Local:

- This database will never be replicated and can be used to store any collections that should be local to a single server

Config:

- When MongoDB is being used in a sharded setup it uses the *config* database to store information about the shards.

By concatenating a database name with a collection in that database you can get a fully qualified collection name called a *namespace*

Basic Operations

Create:

- The insert function adds a document to a collection

```
{
"title" : "My Blog Post",
"content" : "Here's my blog post.",
"date" : ISODate("2012-08-24T21:12:09.982Z")
}
```

- This object is a valid MongoDB document, so we can save it to the *blog* collection using the insert method
  - db.blog.insert(post)
- The blog post has been saved to the database. We can see it by calling find on the collection

```
> db.blog.find()
{
"_id" : ObjectId("5037ee4a1084eb3ffeef7228"),
"title" : "My Blog Post",
"content" : "Here's my blog post.",
"date" : ISODate("2012-08-24T21:12:09.982Z")
}
```

Read:

find and findOne can be used to query a collection. If we just want to see one document from a collection, we can use findOne:

```
> db.blog.findOne()
{
"_id" : ObjectId("5037ee4a1084eb3ffeef7228"),
"title" : "My Blog Post",
"content" : "Here's my blog post.",
"date" : ISODate("2012-08-24T21:12:09.982Z")
}
```

- find and find One can also be passed criteria in the form of a *query document*. This will restrict the documents matched by the query

Update:

- update takes (at least) two parameters: the first is the criteria to find which document to update, and the second is the new document

- The first step is to modify the variable post and add a "comments" key:

```
> post.comments = []
[ ]
```

- Then we perform the update, replacing the post titled “My Blog Post” with our new version of the document:
  - db.blog.update({title : "My Blog Post"}, post)

```
> db.blog.find()
{
"_id" : ObjectId("5037ee4a1084eb3ffeef7228"),
"title" : "My Blog Post",
"content" : "Here's my blog post.",
"date" : ISODate("2012-08-24T21:12:09.982Z"),
"comments" : [ ]
}
```

Delete

- remove permanently deletes documents from the database. Called with no parameters, it removes all documents from a collection.

```
> db.blog.remove({title : "My Blog Post"})
```

Data Types

- ✓ *Null:*

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

{"x" : **null**}

- ✓ *Boolean:*

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

{"x" : **true**}

- ✓ *Number:*

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

{"x" : 3.14}

or

{"x" : 3}

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

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

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

- ✓ *string*

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

{"x" : "foobar"}

- ✓ *date*

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

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

- ✓ *regular expression*

Queries can use regular expressions using JavaScript's regular expression syntax:

{"x" : /foobar/i}

- ✓ *array*

Sets or lists of values can be represented as arrays:

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

- ✓ *embedded document*

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

{"x" : {"foo" : "bar"}}

- ✓ *object id*

An object id is a 12-byte ID for documents. See the section “_id and ObjectIds” on page 20 for details:

```
{"x" : ObjectId()}
```

- ✓ *binary data*

Binary data is a string of arbitrary bytes. It cannot be manipulated from the shell.

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

code

- ✓ Queries and documents can also contain arbitrary JavaScript code:

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

There are a few types that are mostly used internally (or superseded by other types).These will be described in the text as needed.

Dates:

- the Date class is used for MongoDB’s date type. When creating a new Date object, always call new Date().

Arrays:

- Arrays are values that can be interchangeably used for both ordered operations (as though they were lists, stacks, or queues) and unordered operations (as though they were sets).

In the following document, the key "things" has an array value:

```
{"things" : ["pie", 3.14]}
```

Creating, Updating, and Deleting Documents

Inserting and Saving Documents:

Inserts are the basic method for adding data to MongoDB. To insert a document into a collection, use the collection's insert method:

```
> db.foo.insert({"bar" : "baz"})
```

This will add an "_id" key to the document (if o.e does not already exist) and store it in MongoDB.

Batch Insert

- The batch Insert function, which is similar to insert except that it takes an array of documents to insert

```
> db.foo.batchInsert([{"_id" : 0}, {"_id" : 1}, {"_id" : 2}])
> db.foo.find()
{ "_id" : 0 }
{ "_id" : 1 }
{ "_id" : 2}
```

- Sending dozens, hundreds, or even thousands of documents at a time can make inserts significantly faster.

- db.foo.batchInsert([{"_id" : 0}, {"_id" : 1}, {"_id" : 1}, {"_id" : 2}])

Insert Validation:

- MongoDB does minimal checks on data being inserted: it check's the document's basic structure and adds an "_id" field if one does not exist. One of the basic structure checks is size: all documents must be smaller than 16 MB.

Removing Documents:

> db.foo.remove()

- This will remove all of the documents in the *foo* collection. This doesn't actually remove the collection, and any meta information about it will still exist.
- The remove function optionally takes a query document as a parameter. When it's given, only documents that match the criteria will be removed. Suppose, for instance, that we want to remove everyone from the *mailing.list* collection where the value for "optout" is true:
- db.mailing.list.remove({"opt-out" : **true**})

Remove Speed

- Removing documents is usually a fairly quick operation, but if you want to clear an entire collection, it is faster to *drop* it

For example, suppose we insert a million dummy elements with the following:

> **for** (**var** i = 0; i < 1000000; i++) {

... db.tester.insert({"foo": "bar", "baz": i, "z": 10 - i})

... }

Updating Documents

- A document is stored in the database, it can be changed using the update method. Update takes two parameters: a query document, which locates documents to update, and a modifier document, which describes the changes to make to the documents found.
- Updating a document is atomic: if two updates happen at the same time, whichever one reaches the server first will be applied, and then the next one will be applied

Document Replacement

The simplest type of update fully replaces a matching document with a new one. This can be useful to do a dramatic schema migration. For example, suppose we are making major changes to a user document, which looks like the following:

```
{
"_id" : ObjectId("4b2b9f67a1f631733d917a7a"),
"name" : "joe",
"friends" : 32,
"enemies" : 2
}
```

Updating Multiple Documents

- Updates, by default, update only the first document found that matches the criteria. If there are more matching documents, they will remain unchanged. To modify all of the documents matching the criteria, you can pass true as the fourth parameter to update.
- Multiupdates are a great way of performing schema migrations or rolling out new features to certain users.

```
> db.users.update({"birthday" : "10/13/1978"},
... {"$set" : {"gift" : "Happy Birthday!"}}, false, true)
```