Getting Initial Hands-on Experience

- Tasting NoSQL technology
- Exploring MongoDB and Apache Cassandra basics
- Accessing MongoDB and Apache Cassandra from some of the popular high-level programming languages

- This lecture is a variation of the essential programming tutorial first step: Hello World!
- It introduces the initial examples.
- Although elementary, these examples go beyond simply printing a hello message on the console and give you a first hands-on flavor of the topic.
- The topic in this case is NoSQL, which is an abstraction for a class of data stores.
- NoSQL is a concept, a classification, and a new-generation data storage viewpoint.
- It includes a class of products and a set of alternative non-relational data store choices.
- The examples in this chapter use MongoDB and Cassandra so you may want to install and set up those products to follow along.

Why Only Mongodb And Apache Cassandra?

- The choice of MongoDB and Cassandra to illustrate NoSQL examples is quite arbitrary.
- This lecture intends to provide a first flavor of the deep and wide NoSQL domain.
- There are numerous NoSQL products and many offer compelling features and advantages.
- Choosing a couple of products to start with NoSQL was not easy.
- For example, Couchbase server could have been chosen over MongoDB and Hbase could have been used instead of Cassandra.
- The examples could have been based on products like Redis, Membase, Hypertable, or Riak.
- Many NoSQL databases are covered in this class.
- We will discuss a lot about the various alternative options in the NoSQL space.

Examples

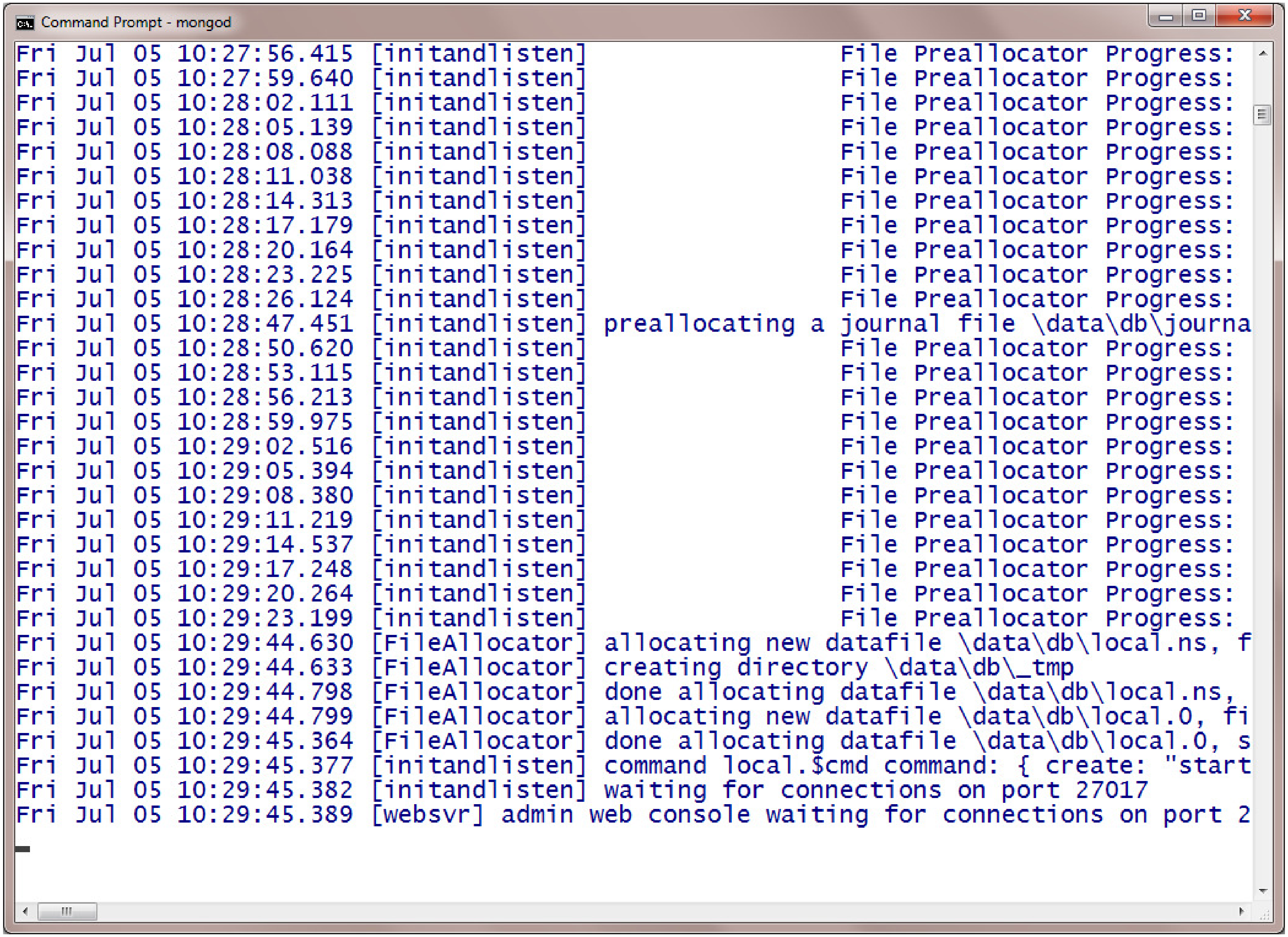
- Without further delay or long prologues, it’s time to dive right into your first two simple examples.
- The first example creates a trivial location preferences store and the second one manages a car make and model database.
- Both the examples focus on the data management aspects that are pertinent in the context of NoSQL.

A Simple Set of Persistent Preferences Data

- Location-based services are gaining importance as local businesses are trying to connect with users who are in the neighborhood and large companies are trying to customize their online experience and offerings based on where people are stationed.
- A few common occurrences of location-based preferences are visible in popular applications like Google Maps, which allows local search, and online retailers like Walmart.com that provide product availability and promotion information based on your closest Walmart store location.
- Sometimes a user is asked to input location data and other times user location is inferred.
- Inference may be based on a user’s IP address, network access point (especially if a user accesses data from a mobile device), or any combination of these techniques.
- Irrespective of how the data is gathered, you will need to store it effectively and that is where the example starts.
- To make things simple, the location preferences are maintained for users only in the United States so only a user identifier and a zip code are required to find the location for a user.
- Let’s start with usernames as their identifiers.
- Data points like “John Doe, 10001,” “Lee Chang, 94129,” “Jenny Gonzalez 33101,” and “Srinivas Shastri, 02101” will need to be maintained.
- To store such data in a flexible and extendible way, this example uses a non-relational database product named MongoDB.
- In the next few slides, we will create a MongoDB database and store a few sample location data points.

Starting MongoDB and Storing Data

- Assuming you have installed MongoDB successfully, start the server and connect to it.
- You can start a MongoDB server by running the mongod program within the bin folder of the distribution.
- Distributions vary according to the underlying environment, which can be Windows, Mac OS X, or a Linux variant, but in each case the server program has the same name and it resides in a folder named bin in the distribution.
- The simplest way to connect to the MongoDB server is to use the JavaScript shell available with the distribution.
- Simply run mongo from your command-line interface.
- The mongo JavaScript shell command is also found in the bin folder.


Command Prompt - mongod

```
Fri Jul 05 10:27:56.415 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:27:59.640 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:28:02.111 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:28:05.139 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:28:08.088 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:28:11.038 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:28:14.313 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:28:17.179 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:28:20.164 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:28:23.225 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:28:26.124 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:28:47.451 [initandlisten] preallocating a journal file \data\db\journa
Fri Jul 05 10:28:50.620 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:28:53.115 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:28:56.213 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:28:59.975 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:29:02.516 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:29:05.394 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:29:08.380 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:29:11.219 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:29:14.537 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:29:17.248 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:29:20.264 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:29:23.199 [initandlisten]                 File Preallocator Progress:
Fri Jul 05 10:29:44.630 [FileAllocator] allocating new datafile \data\db\local.ns, f
Fri Jul 05 10:29:44.633 [FileAllocator] creating directory \data\db\_tmp
Fri Jul 05 10:29:44.798 [FileAllocator] done allocating datafile \data\db\local.ns,
Fri Jul 05 10:29:44.799 [FileAllocator] allocating new datafile \data\db\local.0, fi
Fri Jul 05 10:29:45.364 [FileAllocator] done allocating datafile \data\db\local.0, s
Fri Jul 05 10:29:45.377 [initandlisten] command local.$cmd command: { create: "start
Fri Jul 05 10:29:45.382 [initandlisten] waiting for connections on port 27017
Fri Jul 05 10:29:45.389 [websvr] admin web console waiting for connections on port 2
```

Now that the database server is up and running, use the mongo JavaScript shell to connect to it. The initial output of the shell should be as follows:

```
PS C:\applications\mongodb-win32-x86_64-1.8.1> bin/mongo
MongoDB shell version: 1.8.1
connecting to: test
>
```

By default, the mongo shell connects to the “test” database available on localhost. From mongod (the server daemon program) console output, you can also guess that the MongoDB server waits for connections on port 27017. To explore a possible set of initial commands just type help on the mongo interactive console. On typing help and pressing the Enter (or Return) key, you should see a list of command options like so:

```
> help
        db.help()                    help on db methods
        db.mycoll.help()             help on collection methods
        rs.help()                    help on replica set methods
        help connect                 connecting to a db help
        help admin                   administrative help
        help misc                    misc things to know
        help mr                      mapreduce help

        show dbs                     show database names
        show collections             show collections in current database
        show users                   show users in current database
        show profile                 show most recent system.profile entries
                                         with time >= 1ms
        use <db_name>                set current database
        db.foo.find()                list objects in collection foo
        db.foo.find( { a : 1 } )     list objects in foo where a == 1
        it                           result of the last line evaluated;
                                         use to further iterate
        DBQuery.shellBatchSize = x   set default number of items to display
                                         on shell
        exit                         quit the mongo shell
>
```

Customizing The Mongodb Data Directory And Port

By default, MongoDB stores the data files in the `/data/db` (`C:\data\db` on Windows) directory and listens for requests on port 27017. You can specify an alternative data directory by specifying the directory path using the `dbpath` option, as follows:

```
mongod --dbpath  /path/to/alternative/directory
```

Make sure the data directory is created if it doesn't already exist. Also, ensure that `mongod` has permissions to write to that directory.

In addition, you can also direct MongoDB to listen for connections on an alternative port by explicitly passing the port as follows:

```
mongod --port 94301
```

To avoid conflicts, make sure the port is not in use.

To change both the data directory and the port simultaneously, simply specify both the `--dbpath` and `--port` options with the corresponding alternative values to the `mongod` executable.

Creating the Preferences Database

To start out, create a preferences database called `prefs`. After you create it, store tuples (or pairs) of usernames and zip codes in a collection, named `location`, within this database. Then store the available data sets in this defined structure. In MongoDB terms it would translate to carrying out the following steps:

1. Switch to the `prefs` database.
2. Define the data sets that need to be stored.
3. Save the defined data sets in a collection, named `location`.

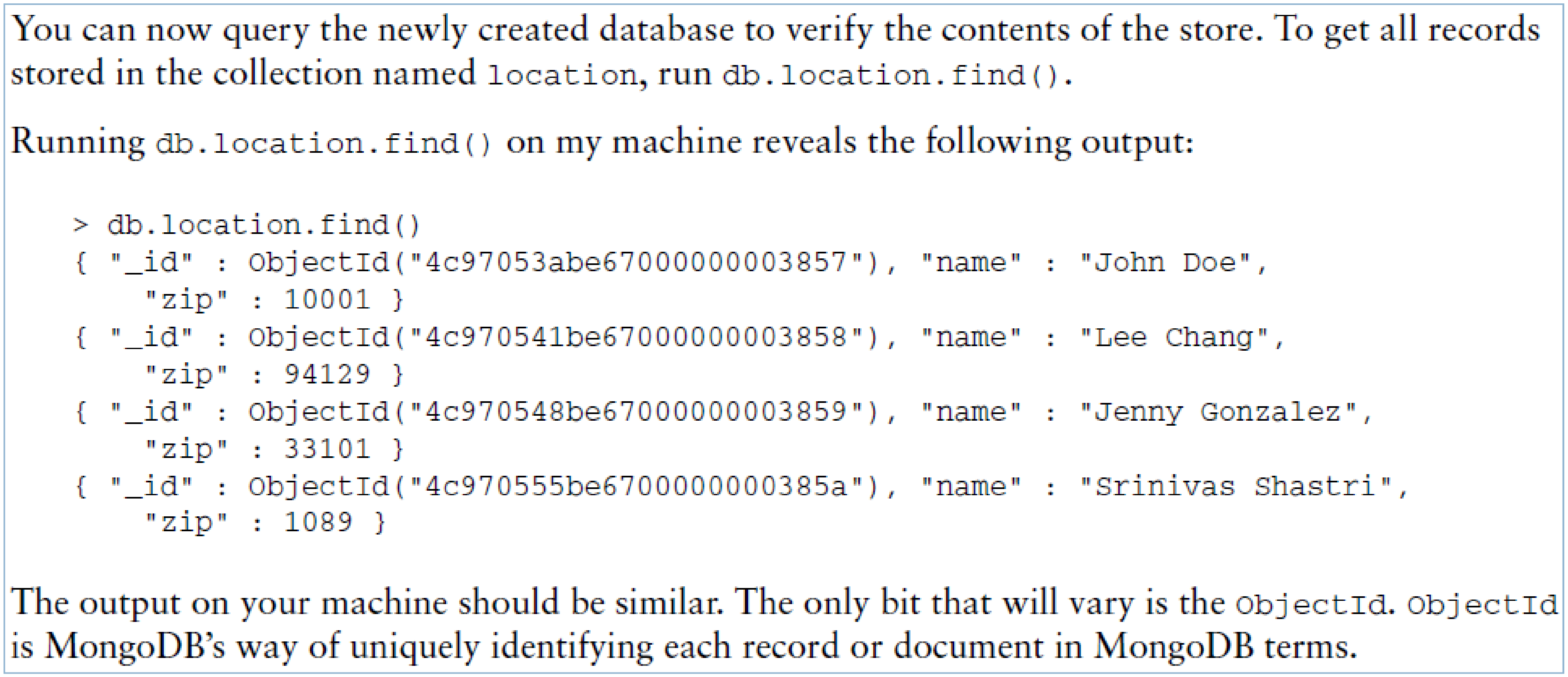
To carry out these steps, type the following on your Mongo JavaScript console:

```
use prefs
w = {name: "John Doe", zip: 10001};
x = {name: "Lee Chang", zip: 94129};
y = {name: "Jenny Gonzalez", zip: 33101};
z = {name: "Srinivas Shastri", zip: 02101};
db.location.save(w);
db.location.save(x);
db.location.save(y);
db.location.save(z);
```

That's it! A few simple steps and the data store is ready. Some quick notes before moving forward though: The `use prefs` command changed the current database to the database called `prefs`. However, the database itself was never explicitly created. Similarly, the data points were stored in the `location` collection by passing a data point to the `db.location.save()` method. The collection wasn't explicitly created either. In MongoDB, both the database and the collection are created only when data is inserted into it. So, in this example, it's created when the first data point, `{name: "John Doe", zip: 10001}`, is inserted.

You can now query the newly created database to verify the contents of the store. To get all records stored in the collection named `location`, run `db.location.find()`.

Querying Database



You can now query the newly created database to verify the contents of the store. To get all records stored in the collection named `location`, run `db.location.find()`.

Running `db.location.find()` on my machine reveals the following output:

```
> db.location.find()
{ "_id" : ObjectId("4c97053abe67000000003857"), "name" : "John Doe",
    "zip" : 10001 }
{ "_id" : ObjectId("4c970541be67000000003858"), "name" : "Lee Chang",
    "zip" : 94129 }
{ "_id" : ObjectId("4c970548be67000000003859"), "name" : "Jenny Gonzalez",
    "zip" : 33101 }
{ "_id" : ObjectId("4c970555be6700000000385a"), "name" : "Srinivas Shastri",
    "zip" : 1089 }
```

The output on your machine should be similar. The only bit that will vary is the `ObjectId`. `ObjectId` is MongoDB's way of uniquely identifying each record or document in MongoDB terms.

MongoDB uniquely identifies each document in a collection using the `ObjectId`*. The* `ObjectId` *for a document is stored as the* `_id` *attribute of that document. While inserting a record, any unique value can be set as the* `ObjectId`*. The uniqueness of the value needs to be guaranteed by the developer. You could also avoid specifying the value for the* `_id` *property while inserting a record. In such cases, MongoDB creates and inserts an appropriate unique id. Such generated ids in MongoDB are of the BSON, short for binary JSON, format, which can be best summarized as follows:*

- BSON Object Id is a 12-byte value.
- The first 4 bytes represent the creation timestamp. It represents the seconds since epoch. This value must be stored in big endian, which means the most significant value in the sequence must be stored in the lowest storage address.
- The next 3 bytes represent the machine id.
- The following 2 bytes represent the process id.
- The last 3 bytes represent the counter. This value must be stored in big endian.
- The BSON format, apart from assuring uniqueness, includes the creation timestamp. BSON format ids are supported by all standard MongoDB drivers.

The find method, with no parameters, returns all the elements in the collection. In some cases, this may not be desirable and only a subset of the collection may be required. To understand querying possibilities, add the following additional records to the location collection:

- Don Joe, 10001
- John Doe, 94129

You can accomplish this, via the mongo shell, as follows:

```
> a = {name:"Don Joe", zip:10001};
{ "name" : "Don Joe", "zip" : 10001 }
> b = {name:"John Doe", zip:94129};
{ "name" : "John Doe", "zip" : 94129 }
> db.location.save(a);
> db.location.save(b);
>
```

To get a list of only those people who are in the 10001 zip code, you could query as follows:

```
> db.location.find({zip: 10001});
{ "_id" : ObjectId("4c97053abe67000000003857"), "name" : "John Doe",
    "zip" : 10001 }
{ "_id" : ObjectId("4c97a6555c760000000054d8"), "name" : "Don Joe",
    "zip" : 10001 }
```

To get a list of all those who have the name "John Doe," you could query like so:

```
> db.location.find({name: "John Doe"});
{ "_id" : ObjectId("4c97053abe67000000003857"), "name" : "John Doe",
    "zip" : 10001 }
{ "_id" : ObjectId("4c97a7ef5c760000000054da"), "name" : "John Doe",
    "zip" : 94129 }
```

Find method

- In both these queries that filter the collection, a query document is passed as a parameter to the find method.
- The query document specifies the pattern of keys and values that need to be matched.
- MongoDB supports many advanced querying mechanisms beyond simple filters, including pattern representation with the help of regular expressions.
- Because a database includes newer data sets, it is possible the structure of the collection will become a constraint and thus need modification.
- In traditional relational database sense, you may need to alter the table schema.
- In relational databases, altering table schemas also means taking on a complicated data migration task to make sure data in the old and the new schema exist together.
- In MongoDB, modifying a collection structure is trivial.
- More accurately, collections, analogous to tables, are schema-less and so it allows you to store disparate document types within the same collection.

Duplicate Entries

Consider an example where you need to store the location preferences of another user, whose name and zip code are identical to a document already existing in your database, say, another `{name: "Lee Chang", zip: 94129}`. Intentionally and not realistically, of course, the assumption was that a name and zip pair would be unique!

To distinctly identify the second Lee Chang from the one in the database, an additional attribute, the street address, is added like so:

```
> anotherLee = {name:"Lee Chang", zip: 94129, streetAddress:"37000 Graham Street"};
{
        "name" : "Lee Chang",
        "zip" : 94129,
        "streetAddress" : "37000 Graham Street"
}
> db.location.save(anotherLee);
```

Now getting all documents, using `find`, returns the following data sets:

```
> db.location.find();
{ "_id" : ObjectId("4c97053abe67000000003857"), "name" : "John Doe",
    "zip" : 10001 }
{ "_id" : ObjectId("4c970541be67000000003858"), "name" : "Lee Chang",
   "zip" : 94129 }
{ "_id" : ObjectId("4c970548be67000000003859"), "name" : "Jenny Gonzalez",
    "zip" : 33101 }
{ "_id" : ObjectId("4c970555be6700000000385a"), "name" : "Srinivas Shastri",
    "zip" : 1089 }
{ "_id" : ObjectId("4c97a6555c760000000054d8"), "name" : "Don Joe",
    "zip" : 10001 }
{ "_id" : ObjectId("4c97a7ef5c760000000054da"), "name" : "John Doe",
    "zip" : 94129 }
{ "_id" : ObjectId("4c97add25c760000000054db"), "name" : "Lee Chang",
    "zip" : 94129, "streetAddress" : "37000 Graham Street" }
```

You can access this data set from most mainstream programming languages, because drivers for those exist.