CS595—BIG DATA TECHNOLOGIES

Module 14

NoSQL Document Database

MongoDB

MongoDB

- MongoDB is a powerful, flexible, and scalable general-purpose database
- It combines the ability to scale out with features such as secondary indexes, range queries, sorting, aggregations, and geospatial indexes

MongoDB | Ease of Use

- MongoDB is a *document-oriented* database, not a relational one
- The primary reason for moving away from the relational model is to make scaling out easier, but there are some other advantages as well
- A document-oriented database replaces the concept of a "row" with a more flexible model, the "document"
- By allowing embedded documents and arrays, the document oriented approach makes it possible to represent complex hierarchical relationships with a single record
- This fits naturally into the way developers in modern object oriented languages think about their data

MongoDB | Ease of Use

- There are also no predefined schemas: a document's keys and values are not of fixed types or sizes
- Without a fixed schema, adding or removing fields as needed becomes easier
- This makes development faster as developers can quickly iterate
- It is also easier to experiment: developers can try dozens of models for the data and then choose the best one

MongoDB | Easy Scaling

- MongoDB was designed to scale out
- Its document-oriented data model makes it easier for it to split up data across multiple servers
- MongoDB automatically takes care of
  - Balancing data and load across a cluster
  - Redistributing documents automatically
  - Routing user requests to the correct machines
- This allows developers to focus on programming the application, not scaling it
- When a cluster need more capacity, new machines can be added and MongoDB will figure out how the existing data should be spread to them

MongoDB | Concepts

- MongoDB is made up of databases
- A database can have zero or more collections
- A collection can be though of as a table (but not quite)
- Collections are made up of zero or more documents
- A document thought of as a row (but not quite)
- A document is made up of one or more fields, which are like columns (but not quite)
- Every document has a special field, "_id", that is unique within a collection
- Indexes in MongoDB function mostly like their RDBMS counterparts
- MongoDB comes with a JavaScript *shell*, which is useful for the administration of MongoDB instances and data manipulation

MongoDB | Concepts

- Cursors are different than the other concepts but are important enough, for their own discussion
- The important thing to understand about cursors is that when you ask MongoDB for data, it returns a pointer to the result set called a cursor…
- Which we can do things to, such as counting or skipping ahead, before actually pulling down data

MongoDB | Concepts

- Why use new terminology (collection vs. table, document vs. row and field vs. column)?
- While these concepts are similar to their relational database counterparts, they are not identical
- The core difference comes from the fact that relational databases define columns at the table level…
- While a document-oriented database defines fields at the document level
- Each document within a collection can have its own unique set of fields
- As such, a collection is a dumbed down container in comparison to a table
- While a document has a lot more information than a row

Documents

- A *document* is the basic unit of data for MongoDB and is roughly equivalent to a row in a relational database

- A MongoDB document is a group of key and value pairs

{someKey1 : someValue1, someKey2 : someValue2 }

- Each key and its value can be thought of as an attribute (field, column) of a document

- The key is always a string and the value can be one of a number of types

- Here is an example of a simple document

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

Documents

- 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}

Documents

- An 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!"}

Documents

- The human readable format of documents is very similar to that of JavaScript Object Notation (JSON)
- One major advantage of this document format is that it is self describing
- You can deduce the name of each entry and its type from the content of the document alone
- You don’t need any external schema to or metadata to interpret JSON encoded data
- Rather the “schema” in MongoDB is, in practice, part of the document itself

Collections

- A *collection* is a group of documents
- If a document is the MongoDB analog of some row in a relational database…
- A collection can be thought of as the analog to a table
- But, as each document is self describing, MongoDB collections do not have a predefined schema
- This means that documents in collections do not all need to have the same key and value pairs in the same order

Collections

- For example, both of the following documents could be stored in a single collection

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

{"foo" : 5}

- Note that the previous documents not only have different types for their values (string versus integer) but also have entirely different keys
- Because any document can be put into any collection, the question arises: “Why do we need separate collections at all?”

Collections

- It’s a good question…

- With no need for separate schemas for different kinds of documents, why *should* we use more than one collection?

Collections

- One reason is to allow different types of real world concepts to be modeled and managed separately
- Another is to reduce the query overhead that would be needed to filter irrelevant types of documents
  - If you were searching for books you would need to spend compute time removing records about CD's
- Also grouping documents of the same kind together in the same collection allows for data locality
  - Getting blog posts from a collection containing only posts will likely require fewer disk seeks than getting them from a collection having posts and author data
- By putting only documents of a single type into the same collection, we can index our collections more efficiently

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

MongoDB Shell

- MongoDB comes with a JavaScript shell that allows interaction from the command line
- To start the shell, run the mongo executable:

$ mongo

- The shell is a full-featured JavaScript interpreter, capable of running arbitrary JavaScript programs
- But the real power of the shell lies in the fact that it is also a standalone MongoDB client

MongoDB Shell

- On startup, the shell connects to the *test* database on a MongoDB server…
- And assigns this database connection to the global variable db
- This variable is the primary access point to your MongoDB server through the shell

MongoDB Shell

- To see the database db is currently assigned to, type in db and hit Enter:

```
> db
Test
```

- One of the most important operations is selecting which database to use:

```
> use foobar
switched to db foobar
```

- Now if you look at the db variable, you can see that it refers to the *foobar* database:

```
> db
foobar
```

MongoDB Shell

- Collections can be accessed from the db variable
- For example, db.baz returns the *baz* collection in the current database
- Now that we can access a collection in the shell, we can perform almost any database operation

MongoDB Shell

- In interactive mode, mongo prints the results of operations including the content of all cursors
- In scripts, either use the JavaScript print() function or the mongo specific printjson() function which returns formatted JSON

- To print all items in a result cursor in mongo shell scripts, use the following idiom

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

Data Types

- Documents in MongoDB can be thought of as “JSON-like” in that they are conceptually similar to objects in JSON
- The JSON specification can be described in about one paragraph and lists only six data types
- This is a good thing in many ways in that it is easy to understand, parse, and remember
- On the other hand, JSON’s expressive capabilities are limited because the only types are:

null, boolean, numeric, string, array, and object

- So MongoDB adds support for a number of additional data types while keeping JSON’s essential key / value nature

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**}

Data Types

- *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

  {"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}

Data Types

- *array*
  - Sets or lists of values can be represented as arrays:

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

- *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.

  {"x" : ObjectId()}

Data Types | Arrays

- Arrays are values that can be interchangeably used for both ordered operations (lists, stacks, or queues) and unordered operations (as though theywere sets).
- In the following document, the key "things" has an array value

  {"things" : ["pie", 3.14]}
- As we can see from the example, arrays can contain different data types as values
- In fact, array values can be any of the supported values for normal key/value pairs, even nested arrays

Data Types | Arrays

- One of the great things about arrays in documents is that MongoDB “understands” their structure…
- And knows how to reach inside arrays to perform operations on their contents
- This allows us to query on arrays and build indexes using their contents

Data Types | Documents

- Documents can be used as the *value* for a key. This is called an *embedded document*.
- Embedded documents can be used to organize data in a more natural way than just a flat structure of key/value pairs.
- If we have a document representing a person and want to store her address…
- We can nest this information in an embedded "address" document:

```
{
   "name" : "Jane Doe",
   "address" : {
      "street" : "123 Park Street",
      "city" : "Anytown",
      "state" : "NY"
   }
}
```

Data Types | Documents

- As with arrays, MongoDB “understands” the structure of embedded documents and is able to reach inside them to build indexes, perform queries, or make updates
- We’ll discuss schema design in depth later, but even from this example we can see how embedded documents can change the way we work with data
- In a relational database, the previous document would probably be modeled as two separate rows in two different tables (one for “people” and one for “addresses”)
- With MongoDB we can embed the address document directly within the person document.
- When used properly, embedded documents can provide a more natural representation of information.

Data Types | Documents

- The flip side of this is that there can be more data repetition with MongoDB
- Suppose “addresses” were a separate table in a relational database and we needed to fix a typo in an address When we did a join with “people” and “addresses,” we’d get the updated address for everyone who shares it
- With MongoDB, we’d need to fix the typo in each person’s document

Data Types | _id and ObjectId

- Every document stored in MongoDB must have an "_id" key
- The "_id" key's value can be any type, but it defaults to an ObjectId
- In a collection, every document must have a unique value for "_id"…
- Which ensures that every document can be uniquely identified
- That is, if you had two collections, each one could have a document where the value for "_id" was 123
- However, neither collection could contain more than one document with an "_id" of 123

Data Types | _id and ObjectId

- MongoDB's distributed nature is the main reason why it uses ObjectIds…
- As opposed to something more traditional, like an auto incrementing primary key
- It is difficult and time-consuming to synchronize auto incrementing primary keys across multiple servers
- if there is no "_id" key present when a document is inserted…
- One will be automatically added to the inserted document

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 one does not already exist) and store it in MongoDB
- 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

- Now that there's data in our database, let's delete it:

> 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 we want to remove everyone from the *mailing.list* collection where the value for "optout" is true:
  - db.mailing.list.remove({"opt-out" : **true**})

- Once data has been removed, it is gone forever

CRUD Methods

- Typically take the following form:

db.<collection>.<method>(<filter>, <options>)

- **db** refers to the current database
- **<collection>** is the name of the target collection for your method
- For **<method>**, substitute the desired method
- Each method has its own **<options>** for what it will do with the matching document(s)

CRUD Methods | Create

- Create or insert operations add new documents to a collection
- If the collection does not currently exist, insert operations will create the collection
- MongoDB provides the following methods to insert documents into a collection:

db.collection.insertOne()

db.collection.insertMany()

CRUD Methods | Create

- The following example inserts a new document into the inventory collection
- If the document does not specify an _id field, MongoDB adds the _id field with an ObjectId value

```
db.inventory.insertOne(
   { item: "canvas", qty: 100, tags: ["cotton"], size: { h: 28, w: 35.5, uom: "cm" } }
)
```

CRUD Methods | Create

- The following example inserts three new documents into the inventory collection
- If the documents do not specify an _id field, MongoDB adds the _id field with an ObjectId value

```
db.inventory.insertMany([
   { item: "journal", qty: 25, tags: ["blank", "red"], size: { h: 14, w: 21, uom: "cm" } },
   { item: "mat", qty: 85, tags: ["gray"], size: { h: 27.9, w: 35.5, uom: "cm" } },
   { item: "mousepad", qty: 25, tags: ["gel", "blue"], size: { h: 19, w: 22.85, uom: "cm" } }
])
```

CRUD Methods | Read

- Read operations retrieves documents from a collection; i.e. queries a collection for documents
- MongoDB provides the following methods to read documents from a collection:

db.collection.find()

You can specify query filters or criteria that identify the documents to return

CRUD Methods | Read

- To select all documents in the collection, pass an empty document as the query filter parameter to the find method

```
db.inventory.find( {} )
```

- These operation corresponds to the following SQL statement

```
SELECT * FROM inventory
```

CRUD Methods | Read

- To specify equality conditions, use <field>:<value> expressions in the query filter document:

{ <field1>: <value1>, ... }

- The following example selects from the inventory collection all documents where the status equals "D":

db.inventory.find({ status: "D" })

CRUD Methods | Read

- A compound query can specify conditions for more than one field in the collection's documents
- Implicitly, a logical AND conjunction connects the clauses of a compound query…
- So that the query selects the documents in the collection that match all the conditions
- The following retrieves all documents in the inventory collection where status equals "A" **and** qty is less than ($lt) 30

```
db.inventory.find( { status: "A", qty: { $lt: 30 } } )
```

- The operation corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status = "A" AND qty < 30
```

CRUD Methods | Read

Name	Description
$eq	Matches values that are equal to a specified value.
$gt	Matches values that are greater than a specified value.
$gte	Matches values that are greater than or equal to a specified value.
$in	Matches any of the values specified in an array.
$lt	Matches values that are less than a specified value.
$lte	Matches values that are less than or equal to a specified value.
$ne	Matches all values that are not equal to a specified value.
$nin	Matches none of the values specified in an array.

CRUD Methods | Read

- Using the $or operator you can specify a compound query that joins each clause with a logical OR conjunction…
- So that the query selects the documents in the collection that match at least one condition
- The following retrieves all documents in the collection where the status equals "A" **or** qty is less than ($lt) 30:

```
db.inventory.find( { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )
```

- The operation corresponds to the following SQL statement:

```
SELECT * FROM inventory WHERE status = "A" OR qty < 30
```

CRUD Methods | Read

- In the following example, the compound query document selects all documents in the collection…
- Where the status equals "A" **and** *either* qty is less than ($lt) 30 *or* item starts with the character p:

```
db.inventory.find( {
    status: "A",
    $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]
} )
```

- MongoDB supports regular expressions $regex queries to perform string pattern matches

CRUD Methods | Read

- Here are some examples of query operations on nested (embedded) documents
- The examples use the inventory collection populated as follows

```
db.inventory.insertMany( [
   { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
   { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
   { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
   { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
   { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]);
```

CRUD Methods | Read

- To specify an equality condition on a field that is an embedded document, use the query filter document { <field>: <value> } where <value> is the document to match
- For example, the following query selects all documents where the field size equals the document { h: 14, w: 21, uom: "cm" }:

```
db.inventory.find( { size: { h: 14, w: 21, uom: "cm" } } )
```

CRUD Methods | Read

- The following query selects all documents where the nested field h is less than 15, the nested field uom equals "in", and the status field equals "D":

```
db.inventory.find(
    { "size.h": { $lt: 15 }, "size.uom": "in", status: "D" } )
```

CRUD Methods | Read

- To specify a query condition on fields in an embedded document, use the dot notation ("field.nestedField").
- The following example selects all documents where the field uom nested in the size field equals "in":

```
db.inventory.find( { "size.uom": "in" } )
```

The following query uses the less than operator ($lt) on the field h embedded in the size field:

```
db.inventory.find( { "size.h": { $lt: 15 } } )
```

CRUD Methods | Read

- The following selects all documents where the nested field h is less than 15, the nested field uom equals "in", and the status field equals "D":

```
db.inventory.find(
    { "size.h": { $lt: 15 }, "size.uom": "in", status: "D" } )
```

CRUD Methods | Read | Cursors

- The database returns results from find using a *cursor*
- The client-side implementations of cursors generally allow you to control a great deal about the eventual output of a query
  - You can limit the number of results
  - Skip over some number of results
  - Sort results by any combination of keys in any direction
  - Perform a number of other operations

CRUD Methods | Read | Cursors

- To create a cursor with the shell, put some documents into a collection, do a query on them, and assign the results to a local variable (variables defined with "var" are local)
- Here, we create a very simple collection and query it, storing the results in the cursor variable:

```
> for(i=0; i<100; i++) {
... db.collection.insert({x : i});
... }
> var cursor = db.collection.find();
```

CRUD Methods | Read | Cursors

- To iterate through the results, you can use the ‘next’ method
- You can use the ‘hasNext’ method to check whether there is another result
- A typical loop through results looks like the following:

```
> while (cursor.hasNext()) {
... obj = cursor.next();
... // do stuff
... }
```

- cursor.hasNext() checks that the next result exists, and cursor.next() fetches it

CRUD Methods | Read | Cursors

- The cursor class also implements JavaScript's iterator interface, so you can use it in a forEach loop:

```
> var cursor = db.people.find();
> cursor.forEach(function(x) {
... print(x.name);
... });
```

CRUD Methods | Read | Cursors

- When you call find, the shell does not query the database immediately
- It waits until you start requesting results, which allows you to chain additional options onto a query before it is performed
- Almost every method on a cursor object returns the cursor itself so that you can chain options in any order
- For instance, all of the following are equivalent:

```
> var cursor = db.foo.find().sort({"x" : 1}).limit(1).skip(10);
> var cursor = db.foo.find().limit(1).sort({"x" : 1}).skip(10);
> var cursor = db.foo.find().skip(10).limit(1).sort({"x" : 1});
```

CRUD Methods | Read | Cursors

- At this point, the query has not been executed yet
- All of these functions merely build the query
- Now, suppose we call the following:
  - cursor.hasNext()

- At this point, the query will be sent to the server
- The shell fetches the first 100 results or first 4 MB of results (whichever is smaller) at once…
- So that the next calls to next or hasNext will not have to make trips to the server
- After the client has run through the first set of results, the shell will again contact the database and ask for more results

CRUD Methods | Read | Cursors

- The most common query options are limiting the number of results returned, skipping a number of results, and sorting
- All these options must be added before a query is sent to the database

- To set a limit, chain the limit function onto your call to find
- For example, to only return three results, use this:

➢ db.c.find().limit(3)

- If there are fewer than three documents matching, only the number of matching documents will be returned

CRUD Methods | Read | Cursors

- skip works similarly to limit:

> db.c.find().skip(3)

- This will skip the first three matching documents and return the rest of the matches
- If there are fewer than three documents in your collection, it will not return any documents

CRUD Methods | Read | Cursors

- sort takes an object: a set of key/value pairs where the keys are key names and the values are sort directions
- Sort direction can be 1 (ascending) or −1 (descending)
- If multiple keys are given, the results will be sorted in that order
- For instance, to sort the results by "username" ascending and "age" descending, we do the following:

```
> db.c.find().sort({username : 1, age : -1})
```

CRUD Methods | Update

- Update operations modify existing documents in a collection
- MongoDB provides the following update methods

db.collection.updateOne(<filter>, <update>, <options>)

db.collection.updateMany(<filter>, <update>, <options>)

db.collection.replaceOne(<filter>, <replacement>, <options>)

- In MongoDB, update operations target a single collection
- Write operations in MongoDB are atomic on the level of a single document

CRUD Methods | Update

- You can specify criteria, or filters, that identify the documents to update
- These filters use the same syntax as read operations

CRUD Methods | Update

- To update a document, MongoDB provides update operators, such as $set, to modify field values
- To use the update operators, pass to the update methods an update document of the form:

```
{
   <update operator>: { <field1>: <value1>, ... },
   <update operator>: { <field2>: <value2>, ... },
... }
```

- Some update operators, such as $set, will create the field if the field does not exist

CRUD Methods | Update

- The following uses the db.collection.updateOne() method on the inventory collection to update the *first* document where item equals "paper":

```
db.inventory.updateOne(
   { item: "paper" },              ← update filter
   {
     $set: { "size.uom": "cm", status: "P" },     ← update action
     $currentDate: { lastModified: true }
   }
)
```

- Uses $set operator to update value of the size.uom field to "cm" and value of the status field to "P",
- Uses $currentDate operator to update value of the lastModified field to the current date
- If lastModified field does not exist, $currentDate will create the field

CRUD Methods | Update

- The following uses the db.collection.updateMany() method on the inventory collection to update all documents where qty is less than 50:

```
db.inventory.updateMany(
   { "qty": { $lt: 50 } },
   {
     $set: { "size.uom": "in", status: "P" },
     $currentDate: { lastModified: true }
   }
)
```

CRUD Methods | Update

- To replace the entire content of a document except for the _id field, pass an entirely new document as the second argument to db.collection.replaceOne()
- The replacement document can have different fields from the original document

CRUD Methods | Update

- The following example replaces the *first* document from the inventory collection that matches the filter item equals "paper":

```
db.inventory.replaceOne(
   { item: "paper" },
   { item: "paper",
     instock: [ { warehouse: "A", qty: 60 }, { warehouse: "B", qty: 40 } ] }
)
```

CRUD Methods | Delete

- To remove all documents from a collection pass an empty filter document {} to the db.collection.deleteMany() method
- The following example deletes *all* documents from the inventory collection:

```
db.inventory.deleteMany({})
```

CRUD Methods | Delete

- You can specify criteria, or filters, that identify the documents to delete
- The filters use the same syntax as read operations

- The following example removes all documents from the inventory collection where the status field equals "A":

```
db.inventory.deleteMany({ status : "A" })
```

CRUD Methods | Delete

- To delete at most a single document that matches a specified filter (even though multiple documents may match the specified filter) use the db.collection.deleteOne() method

- The following example deletes the *first* document where the status is "D“

```
db.inventory.deleteOne( { status: "D" } )
```

SQL to MongoDB

SQL SELECT Statements	MongoDB find() Statements
SELECT * **FROM** people	db.people.find()
SELECT id, user_id, status **FROM** people	db.people.find({ }, { user_id: 1, status: 1 })
SELECT user_id, status **FROM** people	db.people.find({ }, { user_id: 1, status: 1, _id: 0 })
SELECT * **FROM** people **WHERE** status = "A"	db.people.find({ status: "A" })
SELECT user_id, status **FROM** people **WHERE** status = "A"	db.people.find({ status: "A" }, { user_id: 1, status: 1, _id: 0 })

SQL to MongoDB

SELECT * **FROM** people **WHERE** status != "A"	db.people.find({ status: { $ne: "A" } })
SELECT * **FROM** people **WHERE** status = "A" **AND** age = 50	db.people.find({ status: "A", age: 50 })
SELECT * **FROM** people **WHERE** status = "A" **OR** age = 50	db.people.find({ $or: [{ status: "A" } , { age: 50 }] })
SELECT * **FROM** people **WHERE** age > 25	db.people.find({ age: { $gt: 25 } })
SELECT * **FROM** people **WHERE** age < 25	db.people.find({ age: { $lt: 25 } })

SQL to MongoDB

SELECT * **FROM** people **WHERE** age > 25 **AND** age <= 50	db.people.find({ age: { $gt: 25, $lte: 50 } })
SELECT * **FROM** people **WHERE** status = "A" **ORDER BY** user_id **ASC**	db.people.find({ status: "A" }).sort({ user_id: 1 })
SELECT * **FROM** people **WHERE** status = "A" **ORDER BY** user_id **DESC**	db.people.find({ status: "A" }).sort({ user_id: -1 }

SQL to MongoDB

SQL Update Statements	MongoDB updateMany() Statements
UPDATE people **SET** status = "C" **WHERE** age > 25	db.people.updateMany({ age: { $gt: 25 } }, { $set: { status: "C" } })
UPDATE people **SET** age = age + 3 **WHERE** status = "A"	db.people.updateMany({ status: "A" } , { $inc: { age: 3 } })

CRUD Methods | Update Operators

Fields

Name	Description
$currentDate	Sets the value of a field to current date, either as a Date or a Timestamp.
$inc	Increments the value of the field by the specified amount.
$min	Only updates the field if the specified value is less than the existing field value.
$max	Only updates the field if the specified value is greater than the existing field value.
$mul	Multiplies the value of the field by the specified amount.
$set	Sets the value of a field in a document.
$unset	Removes the specified field from a document.

CRUD Methods | Update Operators

Arrays

Name	Description
$addToSet	Adds elements to an array only if they do not already exist
$pop	Removes the first or last item of an array.
$pull	Removes all array elements that match a specified query.
$pushAll	Deprecated. Adds several items to an array.
$push	Adds an item to an array.
$pullAll	Removes all matching values from an array.

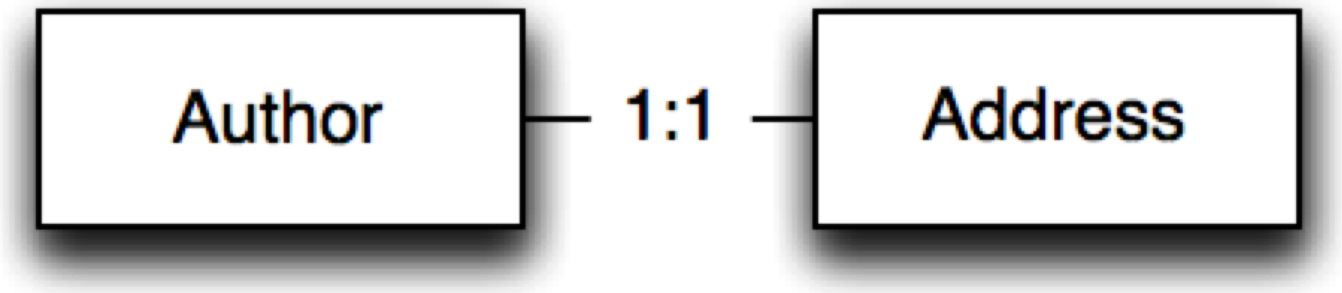
Data Organization

- We will explore basic relationships among data items and how they relate to the MongoDB document model

One-to-One (1:1)

- Describes the relationship between exactly two entities

- The *Author* has a single *Address* relationship where an *Author* lives at a single *Address* and an *Address* only contains a single *Author*



One-to-One | Data Model

Example User

Key	Value
name	“Sam Smith”
age	27

Example Address

Key	Value
street	“2921 N Michigan Ave.”
city	“Chicago”

One-to-One | Embedding

- Include the Address document as an embedded document in the User Document

```
{
        “name” : “Sam Smith,
        “age” : 27,
        “address” : {
                “street” : “2921 N Michigan Ave.”,
                “city” : “Chicago”
        }
}
```

One-to-One | Embedding

- The strength of embedding the *Address* document directly in the *User* document…

- We can retrieve the user and its addresses in a single read operation…
- Versus having to first read the user document and then the address documents for that specific user

- Since addresses have a quite strong affinity to the user document the embedding makes sense here

One-to-One | Linking

- The second approach is to link the address and user document using something like a foreign key

```
{
    “_id” : 1,
    “name” : “Sam Smith”,
    “age” : 27
}

{
    “_id” : 527
    “user_id” : 1
    “street” : “2921 N Michigan Ave.”,
    “city” : “Chicago”
}
```

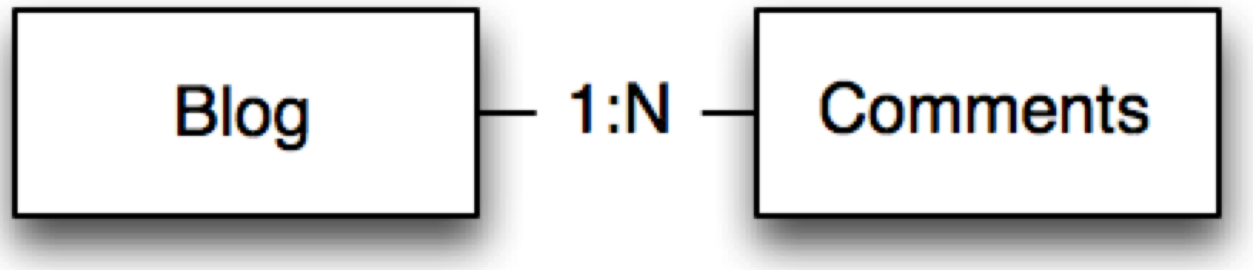
One-to-One | Linking

- This is similar to how traditional relational databases would store the data
- Note that MongoDB does not enforce any foreign key constraints…
- So the relation only exists at the application level
- In the one to one relationship embedding is the preferred way to model the relationship…
- As it is simpler and also more efficient to retrieve the document

One-to-Many (1:N)

- Refers to the relationship between two entities A and B in which an element of A may be linked to many elements of B but a member of B is linked to only one element of A

- A Blog (post) might have many *Comments* but a *Comment* is only related to a single Blog (post)



One-to-Many | Data Model

Example Blog

Key	Value
title	“An awesome blog”
url	“http://awesomeblog.com”
text	“This is so awesome”

One-to-Many | Data Model

Example Comments | For each blog we can have 1 or more comments

Key	Value
name	“Peter Critic”
created_on	ISODate("2014-01-01T10:01:22Z")
comment	“Awesome blog post”

Key	Value
name	“Joh Page”
created_on	ISODate("2014-01-01T10:01:22Z")
comment	“Not so awesome blog post”

One-to-Many | Embedding

The first approach is to embed the Comments in the Blog

```
{
        “_id” : 1
        title: "An awesome blog",
        url: "http://awesomeblog.com",
        text: "This is an awesome blog",
        comments: [{
                name: "Peter Critic",
                created_on: ISODate("2014-01-01T10:01:22Z"),
                comment: "Awesome blog post"
        }, {
                name: "John Page",
                created_on: ISODate("2014-01-01T11:01:22Z"),
                comment: "Not so awesome blog"
        }]
}
```

One-to-Many | Embedding

- Embedding of comments in the *Blog* means we can easily retrieve all the comments belong to a particular *Blog*
- Adding new comments is as simple as appending the new comment document to the end of the comments array
- However, there are potential problems associated with this approach that one should be aware of...

1. The comments array might grow larger than the maximum document size of 16 MB
2. There is no way to limit the comments returned from the Blog using normal finds…
   - So we will have to retrieve the whole blog document and filter the comments in our application

One-to-Many | Linking

- Link comments to the Blog Post using a more traditional foreign key

```
{
   _id: 1,
   title: "An awesome blog",
   url: "http://awesomeblog.com",
   text: "This is an awesome blog"
}
```

- …

One-to-Many | Linking

```
{
    _id : 23,
    blog_entry_id: 1,
    name: "Peter Critic",
    created_on: ISODate("2014-01-01T10:01:22Z"),
    comment: "Awesome blog post"
}

{
    _id: 24,
    blog_entry_id: 1,
    name: "John Page",
    created_on: ISODate("2014-01-01T11:01:22Z"),
    comment: "Not so awesome blog"
}
```

One-to-Many | Linking

- An advantage this model has is that additional comments will not grow the original Blog Post document
- It’s also much easier to return paginated comments as the application can slice and dice the comments more easily But, if we have 1000 comments on a blog post, we would need to retrieve all 1000 documents causing lots of reads from the database

One-to-Many | Bucketing

- Balance the rigidity of the embedding strategy with the flexibility of the linking strategy
- For this example, we will split the comments into buckets with a maximum of 50 comments in each
- An advantage this model has is that additional comments will not grow the original Blog Post document

One-to-Many | Bucketing

```
{
   _id: 12
   blog_entry_id: 1,
   page: 1,
   count: 50,
   comments: [{
       name: "Peter Critic",
       created_on: ISODate("2014-01-01T10:01:22Z"),
       comment: "Awesome blog post“
       }, ...]
}
```

One-to-Many | Bucketing

```
{
_id: 41
blog_entry_id: 1,
page: 2,
count: 1,
comments: [{
    name: "John Page",
    created_on: ISODate("2014-01-01T11:01:22Z"),
    comment: "Not so awesome blog"
}]
}
```

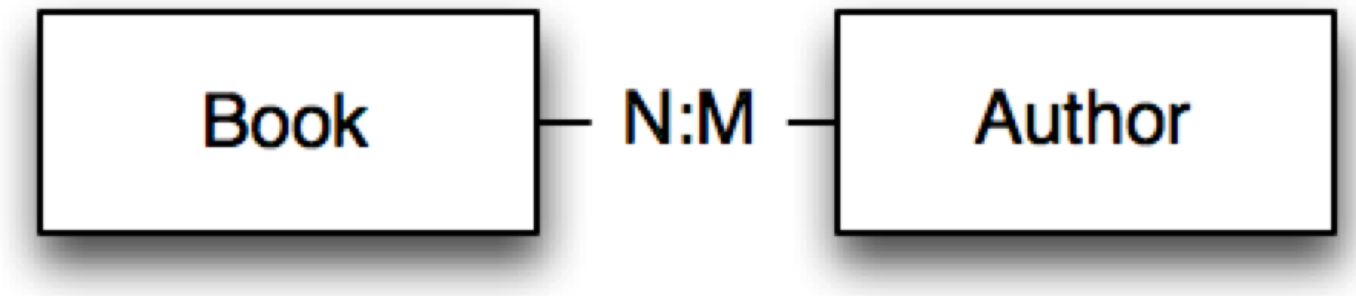
One-to-Many | Bucketing

- The main benefit of using buckets in this case is that we can perform a single read to fetch 50 comments at a time
- When you have the possibility of splitting up your documents into discreet batches, it makes sense to consider bucketing to speed up document retrieval
- Typical cases where bucketing is appropriate are ones such as bucketing data by hours, days or number of entries on a page (like comments pagination)

Many-to-Many (N:M)

- Relationship between two entities where they both might have many relationships between each other

- A *Book* was written by many *Authors* while at the same time an *Author* might have written many *Books*



Many-to-Many (N:M)

- *N:M* relationships are modeled in relational databases by using a join table
- A good example is the relationship between books and authors

Many-to-Many | Two Way Embedding

- We will include the *Book* foreign keys under the books field of each author document

```
{
   _id: 1,
   name: "Peter Standford",
   books: [1, 2]          ← _id's of book documents
}

{
   _id: 2,
   name: "Georg Peterson",
   books: [2]             ← _id's of book documents
}
```

Many-to-Many | Two Way Embedding

- Mirroring the *Author* document, for each *Book document* we include *Author* foreign keys under the *Author* field

```
{
    _id: 1,
    title: "A tale of two people",
    categories: ["drama"],
    authors: [1, 2]
}

{
    _id: 2,
    title: "A tale of two space ships",
    categories: ["scifi"],
     authors: [1]
}
```

Many-to-Many | Queries

- Fetch the books by a specific author

```
author = db.authors.findOne({name: "Peter Standford"});
books = db.books.find({_id: {$in: author.books}}).toArray();
```

- Fetch authors by a specific book

```
book = db.books.findOne({title: "A tale of two space ships"});
authors = db.author.find({_id: {$in: book.authors}}).toArray();
```

- We have to perform two queries in both directions…

1. Find either the author or the book…
2. Perform a $in query to find the books or authors