MongoDB (cont.)

Excerpts from

"*The Little MongoDB Book*"

Karl Seguin

Accessing array elements

- $slice: takes the form of array: [skip , limit], where the first value indicates the number of items in the array to skip and the second value indicates the number of items to return:
- db.unicorns.find({}, {loves: {$slice: [0, 1]}})
- db.unicorns.find({}, {loves: {$slice: [1, 1]}})
- db.unicorns.find({}, {loves: {$slice: [0, 2]}})
- db.unicorns.find({},{loves : {$slice: 2}}) → Acá skip es 0
- db.unicorns.find({},{loves: {$slice: -1}})

MongoDB is rich in operators for dealing with arrays…you are encouraged to try them…

- What do these queries do?

```
db.unicorns.find({"loves.0": 'grape'})
db.unicorns.find({"loves.1": 'grape'})
db.unicorns.find({loves: {$size: 3}})
```

Data modeling

- “Having a conversation about modeling with a new paradigm is not as easy.” → Karl Seguin
- “The truth is that most of us are still finding out what works and what doesn’t when it comes to modeling with these new technologies.” → Karl Seguin
- Out of all NoSQL databases, document-oriented databases are probably the most similar to relational databases – at least when it comes to modeling. However, the differences that exist are important.

No Joins:

- Some NoSQL systems do not have joins.*
- To live in a join-less world, we have to do joins ourselves within our application's code.
- Essentially we need to issue a second query to find the relevant data in a second collection. Setting our data up is not any different than declaring a foreign key in a relational database.

* MongoDB sí lo tiene con el operador $lookup, más adelante se muestra un ejemplo. Ver https://docs.mongodb.com/manual/reference/operator/aggregation/lookup

- The first thing we will do is create an employee (here it is an explicit _id so that we can build coherent examples)

```
db.employees.insert({_id: ObjectId("4d85c7039ab0fd70a117d730"),
name: 'Leto'})
```

- Now let us add a couple employees and set their manager as Leto:

```
db.employees.insert({_id: ObjectId(
"4d85c7039ab0fd70a117d731"),
name: 'Duncan',
manager: ObjectId(
"4d85c7039ab0fd70a117d730")});

db.employees.insert({_id: ObjectId(
"4d85c7039ab0fd70a117d732"),
name: 'Moneo',
manager: ObjectId(
"4d85c7039ab0fd70a117d730")});
```

- So to find all of Leto’s employees, one simply executes:

```
db.employees.find({manager: ObjectId("4d85c7039ab0fd70a117d730")})
```

- In this example, the lack of join will merely require an extra query

A simple example with $lookup

```
db.user.remove({})
db.user.insert({code:1, name:'George', gender: 'male'})
db.user.insert({code:2, name:'Saffron', gender: 'female'})
db.user.insert({code:3, name:'Tini', gender: 'female'})

db.account.remove({})
db.account.insert({userCode:1, account:'Ventas'})
db.account.insert({userCode:2, account:'Compras'})
db.account.insert({userCode:1, account:'Cocina'})
```

```
db.user.aggregate([
    {
      $lookup:
        {
          from: "account",
          localField: "code",
          foreignField: "userCode",
          as: "accounts"
        }
    }
]).pretty();
```

```
db.user.aggregate([
    {$lookup:
        {
          from: "account",
          localField: "code",
          foreignField: "userCode",
          as: "accounts"
        }
    },
    {$match: {accounts: {$ne: []}}},
    {$project: {_id: 0}}
]).pretty();
```

- **Arrays and Embedded Documents:**
- Remember that MongoDB supports arrays as first class objects of a document
- It turns out that this is incredibly handy when dealing with many-to-one or many-to-many relationships.
- As a simple example, if an employee could have two managers, we could simply store these in an array:

```
db.employees.insert(
{_id: ObjectId("4d85c7039ab0fd70a117d733"),
name: 'Siona',
manager: [
ObjectId("4d85c7039ab0fd70a117d730"),
ObjectId("4d85c7039ab0fd70a117d732")]
})
```

- Of particular interest is that, for some documents, manager can be a scalar value, while for others it can be an array!
- Our previous find query will work for both:

```
db.employees.find({manager: ObjectId(
"4d85c7039ab0fd70a117d730")})
```

- Besides arrays, MongoDB also supports embedded documents. Go ahead and try inserting a document with a nested document, such as:

```
db.employees.insert(
{_id: ObjectId("4d85c7039ab0fd70a117d734"),
name: 'Ghanima',
family: {mother: 'Chani',
         father: 'Paul',
         brother: ObjectId("4d85c7039ab0fd70a117d730")}
})
```

- Embedded documents can be queried using a dot-notation:

```
db.employees.find({'family.mother': 'Chani'})
```

- Combining the two concepts, we can even embed **arrays of documents**:

```
db.employees.insert(
{_id: ObjectId("4d85c7039ab0fd70a117d735"),
name: 'Chani',
family: [{relation: 'mother', name: 'Ann'},
         {relation: 'father', name: 'Paul'},
         {relation: 'brother', name: 'Duncan'}]
})
```

- **Denormalization**
- “Denormalization refers to the process of optimizing the read performance of a database by adding redundant data or by grouping data.”*
- This process may be accomplished by duplicating data in multiple tables, grouping data for queries.
- With the evergrowing popularity of NoSQL, many of which do not have joins, denormalization as part of normal modeling is becoming common.

This does not mean you should duplicate every piece of data in every document.

* https://quizlet.com/145056951/cassandra-flash-cards

- Consider modeling your data based on what information belongs to what document.
- For example, say you are writing a forum application. The traditional way to associate a specific user with a post is via a userid column within posts.
- With such a model, you can not display posts without retrieving (joining to) users.

- A possible alternative is simply to store the name as well as the userid with each post.
- Of course, if you let users change their name, you may have to update each document (which is one multi-update) → But it is not very common that users change their name…
- Adjusting to this kind of approach will not come easy to some.
- Do not be afraid to experiment with this approach though, it can be suitable in some circumstances

- **Some alternatives**
- Arrays of ids can be a useful strategy when dealing with one-to-many or many-to-many scenarios. But more commonly, developers are left deciding between using embedded documents versus doing “manual” referencing.
- Embedded documents are frequently took advantage of*, but mostly for smaller pieces of data which we want to always pull with the parent document.
- A real world example may be to store an addresses documents with each user, something like:

* In the original the author uses “leveraged”; however, see http://this.isfluent.com/2010/1/are-you-stupid-enough-to-use-leverage-as-a-verb

```
db.employees.insert(
{name: 'leto',
email: 'leto@dune.gov',
addresses: [
{street: "229 W. 43rd St", city: "New York", state:"NY",zip:"10036"},
{street: "555 University", city: "Palo Alto", state:"CA",zip:"94107"}]
})
```

- This does not mean you should underestimate the power of embedded documents or write them off as something of minor utility.
- Having your data model map directly to your objects makes things a lot simpler and often removes the need to join.
- This is especially true when you consider that MongoDB lets you query and index fields of an embedded documents and arrays.

- **Few or Many Collections**
- Given that collections do not enforce any schema, it is entirely possible to build a system using a single collection with a mishmash of documents!!! **But it would be a very bad idea**.
- The conversation gets even more interesting when you consider embedded documents.
- The example that frequently comes up is a blog. Should you have a posts collection and a comments collection, or should each post have an array of comments embedded within it?

- Setting aside the document size limit for the time being*, most developers should prefer to separate things out. It is simply cleaner, gives you better performance and more explicit.
- MongoDB's flexible schema allows you to combine the two approaches by keeping comments in their own collection but embedding a few comments (maybe the first few) in the blog post to be able to display them with the post.

This follows the principle of keeping together data

that you want to get back in one query.

*16MB in MongoDB

- There is no hard rule.
- Play with different approaches and you will get a sense of what does and does not feel right.

When To Use MongoDB?

- There are enough new and competing storage technologies that it is easy to get overwhelmed by all of the choices.
- Only you know whether the benefits of introducing a new solution outweigh the costs.
- MongoDB (and in general, NoSQL-databases) should be seen as a direct *alternative* to relational databases.

Notice that we did **not** call MongoDB a ***replacement*** for relational databases, but rather an ***alternative***.

- It is a tool that can do what a lot of other tools can do. Some of it MongoDB does better, some of it MongoDB does worse. Let us dissect things a little further.

Flexible Schema

- An oft-touted* benefit of document-oriented database is that they do not enforce a fixed schema.
- This makes them much more flexible than traditional database tables.

*Muy promocionado

- People talk about schema-less as though you will suddenly start storing a crazy mishmash of data.
- “There are domains and data sets which can really be a pain to model using relational databases, but I see those as edge cases.” → Karl Seguin
- Schema-less is cool, but most of your data is going to be highly structured
- There is nothing a nullable column probably would not solve just as well.

A lot of features…

Writes

- MongoDB has something called a *capped** collection.
- We can create a capped collection by using the db.createCollection command and flagging it as capped:
- //limit our capped collection to 1 megabyte

```
db.createCollection('logs', {capped: true , size: 1048576})
```

* Que tienen un tope

A lot of features…

- When our capped collection reaches its 1MB limit, old documents are automatically purged.
- A limit on the number of documents, rather than the size, can be set using max.
- If you want to “expire” your data based on time rather than overall collection size, you can use TTL indexes where TTL stands for “time-to-live”. See: https://docs.mongodb.com/manual/core/index-ttl

A lot of features…

Full Text Search

- MongoDB includes text search capabilities. See: https://docs.mongodb.com/manual/reference/operator/query/text
- It supports fifteen languages with stemming and stop words, see next slide.
- With MongoDB's support for arrays and text search you will only need to look to other solutions if you need a more powerful and full-featured text search engine.

Utilities: mongoimport and mongoexport (JSON and CSV files)

$text example

```
db.articles.createIndex( { subject: "text" } )
db.articles.insertMany( [
    { _id: 1, subject: "coffee", author: "xyz", views: 50 },
    { _id: 2, subject: "Coffee Shopping", author: "efg", views: 5 },
    { _id: 3, subject: "Baking a cake", author: "abc", views: 90  },
    { _id: 4, subject: "baking", author: "xyz", views: 100 },
    { _id: 5, subject: "Café Con Leche", author: "abc", views: 200 },
    { _id: 6, subject: "Сырники", author: "jkl", views: 80 },
    { _id: 7, subject: "coffee and cream", author: "efg", views: 10 },
    { _id: 8, subject: "Cafe con Leche", author: "xyz", views: 10 },
    { _id: 9, subject: "John eats pork and blueberries", author: "xyz", views: 10 }
] )
```

- The following query specifies a $search string of coffee:

```
db.articles.find( { $text: { $search: "coffee" } } )
```

- The following query specifies a $search string of three terms delimited by space, "bake coffee cake":

```
db.articles.find( { $text: { $search: "bake coffee cake" }})
```

- The following query searches for the phrase coffee shop:

```
db.articles.find( { $text: { $search: "\"coffee shop\"" } } )
```

- The following example searches for documents that contain the words coffee but do not contain the term shop, or more precisely the stemmed version of the words:

```
db.articles.find( { $text: { $search: "coffee -shop" } } )
```

- The following query specifies en, i.e., English, as the language that determines the tokenization, stemming, and stop words:

```
db.articles.find({ $text: { $search: "eat", $language: "en" } } )
db.articles.find({ $text: { $search: "blueberry", $language: "en" } } )
```

For more examples and options see:

https://www.mongodb.com/docs/manual/reference/operator/query/text

A lot of features…

Data Processing

- Before version 2.2 MongoDB relied on MapReduce for most data processing jobs.
- As of 2.2 it has added a powerful feature called aggregation framework* or pipeline, so you will only need to use MapReduce in rare cases where you need complex functions for aggregations that are not yet supported in the pipeline.
- For parallel processing of very large data, you may need to rely on something else, such as Hadoop.

*Similar to GROUP BY in SQL, you are encouraged to to try it…See a basic example next

- A basic aggregation example: What does this code do?

```
db.unicorns.aggregate([
                { $match: { } },
                { $group: { _id: "$gender", total: { $sum: 1 } } }
               ])
```

$match is similar to where in SQL, here it can be removed…

See also:

https://docs.mongodb.com/manual/reference/method/db.collection.aggregate

A lot of features…

Geospatial

- A particularly powerful feature of MongoDB is its support for geospatial indexes. This allows you to store either geoJSON (*x* and *y* coordinates within documents and many more geospatial data…)

See: https://docs.mongodb.com/manual/reference/geojson

Parallel and distributed execution across sharded nodes

- Replicas

Many, many more features…

A lot of features…

Indexes

- MongoDB creates a unique index on the _id field
- The _id index avoids inserting two documents with the same value for the _id field
- You cannot drop this index on the _id field.

- To create an index use db.collection.createIndex():
- Example:

```
db.unicorns.createIndex({name: 1})
```

1: ascending

-1: descending

For single key indexes it does not matter whether it is 1 or -1, because MongoDB can traverse the index in either direction.

For compound indexes it *does* matter...

- db.unicorns.getIndexes()
- db.unicorns.find({"name": "Horny"})
- db.unicorns.find({"name": "Horny"}).explain()
- db.unicorns.dropIndex("name_1")
- db.unicorns.hideIndex("name_1")
- db.unicorns.unhideIndex("name_1")
- db.unicorns.explain("executionStats").find({"name": "Horny"})

Very briefly: Cursors

- The db.collection.find() method returns a cursor.
- By default, the cursor will be iterated automatically when the result of the query is returned.
- You can also manually iterate a cursor: In the mongo shell, when you assign the cursor returned from the find() method to a variable using the var keyword, the cursor does not automatically iterate.
- Cursors are rich in methods, see

https://docs.mongodb.com/manual/reference/method/js-cursor

Example 1

```
var myCursor = db.unicorns.find({});
while (myCursor.hasNext()) {
  print(tojson(myCursor.next()));
}
```

- As an alternative consider the printjson() method to replace print(tojson()):

```
var myCursor = db.unicorns.find({});
while (myCursor.hasNext()) {
  printjson(myCursor.next());
}
```

Example 2: What does this example do?

```
var micursor = db.unicorns.find().sort({weight: 1})
var i = 0;
while (micursor.hasNext()) {
   if (i%2 == 1) printjson(micursor.next());
   else micursor.next();
   i++;
}
```