Introduction to the Waterline Criteria Cursor

June 14, 2014

Background

- The criteria cursor was originally developed as one of the final steps of the stable v0.10 release of Waterline.

- The criteria cursor, together with the query heap and the integrator, are what allows Waterline to transparently find and join records across disparate datastores/adapters (or **xD/A**s)

Implementing xD/As

- xdas \`.populate()\`s are *possible* without the cursor (using between 1-3 adapter calls.) But there were challenges:

1. The result sets can only be populated one level deep.

2. \`.populate()\` cannot be safely combined with \`sort\`, \`limit\`, or \`skip\`.

#1
handling deeply nested populates

- The original approach towards xD/A queries in Waterline was tenable because it baked in three different hard-coded algorithms for joining data:

  1. parent records have a foreign key which corresponds to exactly 1 child record (i.e. “belongsTo”)

  2. child records have a foreign key which corresponds to exactly 1 parent record (i.e. “hasMany”)

  3. neither parent nor child records have a foreign key- instead a junction relation maps 1 or more parent records to 1 or more child records (i.e. “hasAndBelongsToMany”)

#1
handling deeply nested populates

- Unfortunately, deeply nested xD/A populates cannot be accomplished using just one of those strategies. We needed something recursive.

- Consider the implementation of a query that looks for doctors over age 40, and populates all of the prescriptions they've ever written. Now imagine that, for each prescription, you need to populate the medication(s) that were prescribed. Oh, and *by the way*, the Doctor collection lives in MongoDB, the Prescription collection lives in MySQL, and the Medication collection lives in MSSQL.

  (continued)

#1
handling deeply nested populates

- Using the new criteria syntax, the query in this example looks something like:

```
Doctor.find()
.where({age:{‘>’:40}})
.select({
  ‘*’: true,
  prescriptionsWritten: {
    medications: { ‘*’: true }
  }
});
```

#1
handling deeply nested populates

- First, you have to do a find query to Mongo to get all the docs over 40.
- Then, depending on the association between doctors and prescriptions, you either use a remote or local foreign key to build a find query to MySQL.
- Then, depending on the association between prescriptions and medications, you use either a remote or local foreign key to build a find query to MSSQL.
- Then you run the integrator and it wires all the records back together.

#1
handling deeply nested populates

- OK fine- we did it! And it only took 3 network calls. And it would all still work if we only wanted a particular type of medication. Or only expired prescriptions. And hypothetically, this could be done recursively, on and on forever!

- *But as soon as you start trying to sort and paginate nested results*, *it all falls apart.*

#2
paginating/sorting populated records

- So what's the big deal with pagination and sorting? To stick with our example, let's say you have 1,000,000 doctors in Mongo, 2,000,000 prescriptions in MySQL, and 3,000,000 medications in MSSQL.

- Let's say that we only wanted to populate the 30 most recent, expired prescriptions written by each doctor, sorted by date. And for each of those prescriptions, we want to populate the 15 least expensive medications with the word "thyroid" in the description.

#2
paginating/sorting populated records

- Using the new criteria syntax, our revised example query looks something like:

```
Doctor.find()
.where({age:{‘>’:40}})
.select({
  ‘*’: true,
  prescriptionsWritten: {
    sort: {dateWritten:-1},
    where:{ expired: true },
    limit: 30,
    select: {
     ‘*’: true,
      medications: {
        select: {
          ‘*’: true
        },
        limit: 15,
        where: {
          description: {contains:{’thyroid’}}
        },
        sort: {price: 1}
      }
    }
  }
});
```

#2
paginating/sorting populated records

- What about paginating the populated records?

```
Doctor.find()
.where({age:{‘>’:40}})
.select({
  ‘*’: true,
  prescriptionsWritten: {
    sort: {dateWritten:-1},
    where:{ expired: true },
    limit: 30,
    skip: 90,
    select: {
     ‘*’: true,
      medications: {
        select: {
          ‘*’: true
        },
        limit: 15,
        skip: 45,
        where: {
          description: {contains:{’thyroid’}}
        },
        sort: {price: 1}
      }
    }
  }
});
```

#2
paginating/sorting populated records

- After first avoiding, then wrestling with the implementation of populate + skip/limit/sort for xD/A queries for almost 6 months, we started to get concerned. The related code had reached a destabilizing level of complexity.

- Cody, Scott, and I finally had to admit that our strategy (and in fact the standard approach taken by every other ORM we'd looked at) would not be sufficient for xD/As.

- And we didn't know what *would* be sufficient. We were beginning to doubt whether it was even possible.

- (spoiler alert: ok *obviously* it was possible)

The Rewrite

- Motivated by something in between desperation and hubris, I started working on a rewrite of the Waterline query engine.

The Rewrite

- The first pass took a little less than a month—there was weekend scribbling, confusing, repetitive conversations with my colleagues and customers, and sudden, suspicious 3AM questions to my girlfriend about how she might find 40 year old men with enough 8 year old dogs (or any other nouns I could think of.)

- And aside from a couple of sobering conversations about my sanity, I thought it was going really well!

The Rewrite

- But not long after the first month, it dawned on me that the algorithms I was struggling with looked a lot like the sort of algorithms you might struggle with if you were some kind of fancy database architect Yale-person.

- And then the disturbing realization that- if you think of the query heap as virtual memory, and physical datastores as B+ trees on disk… well, Waterline *itself* was going to have to become a database.

The Rewrite

- So on one hand, one might think “hey, that’s pretty cool. You get to build a database. Databases are cool.. right?”
- But at 11pm, hunched over a laptop in the driver’s seat of my car parked in the H.E.B. parking lot, where I had slunk off to work on Waterline while my girlfriend visited her parents, it felt a lot more like I had just been sentenced to some kind of unexpected, frightening death. I was Odysseus, the Starbucks-cup-littered vehicle was the Kyklopes’ cave, and most of all, I was NOT A DATABASE ARCHITECT AND I NEVER WENT TO CLASS I’M SORRY I’M SO SORRY DON’T MAKE ME DO IT

The Rewrite

- Preliminary research led me towards a handful of interesting books and articles on the inner workings of a few popular databases- Oracle, DB2, SQLite, and Mongo.

- What I found was that, under the hood, these databases were implementing almost *exactly the same* algorithms and data structures we'd been struggling with in Waterline for almost a year.