Taste of Riak: Querying with Ruby

A Quick Note on Querying and Schemas

Schemas? Yes we said that correctly, S-C-H-E-M-A-S. It's not a dirty word.
Even with a Key/Value store, you will still have a logical database schema of how all the data relates to one another. This can be as simple as using the same key across multiple buckets for different types of data, to having fields in your data that are related by name. These querying methods will introduce you to some ways of laying out your data in Riak, along with how to query it back.

Denormalization

If you're coming from a relational database, the easiest way to get your application's feet wet with NoSQL is to denormalize your data into related chunks. For example with a customer database, you might have separate tables for Customers, Addresses, Preferences, etc. In Riak, you can denormalize all that associated data into a single object and store it into a `Customer` bucket. You can keep pulling in associated data until you hit one of the big denormalization walls:

- Size Limits (objects greater than 1MB)
- Shared/Referential Data (data that the object doesn't “own”)
- Differences in Access Patterns (objects that get read/written once vs. often)

At one of these points we will have to split the model.

Same Keys - Different Buckets

The simplest way to split up data would be to use the same identity key across different buckets. A good example of this would be a `Customer` object, an `Order` object, and an `OrderSummaries` object that keeps rolled up info about orders such as Total, etc. Let's put some data into Riak so we can play with it.

Ruby

```ruby
# Encoding: utf-8

require 'riak'
require 'pp'

# Starting Client
client = Riak::Client.new protocol: 'pbc', pb_port: 10017

# Creating Data
customer = {
    customer_id: 1,
    name: 'John Smith',
    address: '123 Main Street',
    city: 'Columbus',
    state: 'Ohio',
    zip: '43210',
    phone: '+1-614-555-5555',
    created_date: Time.parse('2013-10-1 14:30:26')
}

orders = [
  {
      order_id: 1,
      customer_id: 1,
      salesperson_id: 9000,
      items: [
          {
            item_id: 'TCV37GIT4NJ',
            title: 'USB 3.0 Coffee Warmer',
            price: 15.99
          },
          {
            item_id: 'PEG10BBF2PP',
            title: 'eTablet Pro, 24GB, Grey',
            price: 399.99
          }
      ],
      total: 415.98,
      order_date: Time.parse('2013-10-1 14:42:26')
  },
  {
      order_id: 2,
      customer_id: 1,
      salesperson_id: 9001,
      items: [
          {
            item_id: 'OAX19XWN0QP',
            title: 'GoSlo Digital Camera',
```

```
            price: 359.99
          }
      ],
      total: 359.99,
      order_date: Time.parse('2013-10-15 16:43:16')
  },
  {
      order_id: 3,
      customer_id: 1,
      salesperson_id: 9000,
      items: [
          {
            item_id: 'WYK12EPU5EZ',
            title: 'Call of Battle: Goats - Gamesphere 4',
            price: 69.99
          },
          {
            item_id: 'TJB84HAA8OA',
            title: 'Bricko Building Blocks',
            price: 4.99
          }
      ],
      total: 74.98,
      order_date: Time.parse('2013-11-3 17:45:28')
  }]

order_summary = {
    customer_id: 1,
    summaries: [
        {
            order_id: 1,
            total: 415.98,
            order_date: Time.parse('2013-10-1 14:42:26')
        },
        {
            order_id: 2,
            total: 359.99,
            order_date: Time.parse('2013-10-15 16:43:16')
        },
        {
            order_id: 3,
            total: 74.98,
            order_date: Time.parse('2013-11-3 17:45:28')
        }
    ]
}

# Creating Buckets and Storing Data
customer_bucket = client.bucket('Customers')
cr = customer_bucket.new(customer[:customer_id].to_s)
cr.data = customer
```

```
cr.store

order_bucket = client.bucket('Orders')
orders.each do |order|
  order_riak = order_bucket.new(order[:order_id].to_s)
  order_riak.data = order
  order_riak.store
end

order_summary_bucket = client.bucket('OrderSummaries')
os = order_summary_bucket.new(order_summary[:customer_id].to_s)
os.data = order_summary
os.store
```

While individual `Customer` and `Order` objects don't change much (or shouldn't change), the `Order Summaries` object will likely change often. It will do double duty by acting as an index for all a customer's orders, and also holding some relevant data such as the order total, etc. If we showed this information in our application often, it's only one extra request to get all the info.

Ruby

```
shared_key = '1'
customer = customer_bucket.get(shared_key).data
customer[:order_summary] = order_summary_bucket.get(shared_k
puts "Combined Customer and Order Summary: "
pp customer
```

Which returns our amalgamated objects:

Ruby

```
#Combined Customer and Order Summary:
{"customer_id"=>1,
 "name"=>"John Smith",
 "address"=>"123 Main Street",
 "city"=>"Columbus",
 "state"=>"Ohio",
 "zip"=>"43210",
 "phone"=>"+1-614-555-5555",
 "created_date"=>"2013-10-01 14:30:26 -0400",
 :order_summary=>
  {"customer_id"=>1,
   "summaries"=>
    [{"order_id"=>1,
      "total"=>415.98,
      "order_date"=>"2013-10-01 14:42:26 -0400"},
     {"order_id"=>2,
      "total"=>359.99,
      "order_date"=>"2013-10-15 16:43:16 -0400"},
```

```
      {"order_id"=>3,
       "total"=>74.98,
       "order_date"=>"2013-11-03 17:45:28 -0500"}]}}
```

While this pattern is very easy and extremely fast with respect to queries and complexity, it's up to the application to know about these intrinsic relationships.

Secondary Indexes

If you're coming from a SQL world, Secondary Indexes (2i) are a lot like SQL indexes. They are a way to quickly lookup objects based on a secondary key, without scanning through the whole dataset. This makes it very easy to find groups of related data by values, or even ranges of values. To properly show this off, we will now add some more data to our application, and add some secondary index entries at the same time.

Ruby

```ruby
(1..3).each do |i|
  order = order_bucket.get(i.to_s)
  # Initialize our secondary indices
  order.indexes['salesperson_id_int'] = []
  order.indexes['order_date_bin'] = []

  order.indexes['salesperson_id_int'] <<  order.data['salesp
  order.indexes['order_date_bin'] << Time.parse(order.data['
                                        .strftime('%Y%m%d')

  order.store
end
```

As you may have noticed, ordinary Key/Value data is opaque to 2i, so we have to add entries to the indexes at the application level. Now let's find all of Jane Appleseed's processed orders, we'll lookup the orders by searching the `saleperson_id_int` index for Jane's id of `9000`.

Ruby

```ruby
puts "#Jane's Orders: "
pp order_bucket.get_index('salesperson_id_int', 9000)
```

Which returns:

Ruby

```ruby
#Jane's Orders:
["1", "3"]
```

Jane processed orders 1 and 3. We used an “integer” index to reference Jane's id, next let's use a “binary” index. Now, let's say that the VP of Sales wants to know how many orders came in during October 2013. In this case, we can exploit 2i's range queries. Let's search the `order_date_bin` index for entries between `20131001` and `20131031`.

Ruby

```ruby
puts "#October's Orders: "
pp order_bucket.get_index('order_date_bin', '20131001'..'201
```

Which returns:

Ruby

```ruby
#October's Orders:
["1", "2"]
```

Boom, easy-peasy. We used 2i's range feature to search for a range of values, and demonstrated binary indexes.

So to recap:

- You can use Secondary Indexes to quickly lookup an object based on a secondary id other than the object's key.
- Indexes can have either Integer or Binary(String) keys
- You can search for specific values, or a range of values
- Riak will return a list of keys that match the index query

These May Also Interest You

- Five-Minute Install
- Advanced Secondary Indexes
- Replication Properties
- Advanced Commit Hooks
- Advanced MapReduce
- Advanced Search