Taste of Riak: Querying with Java

Java Version Setup

For the java version, please download the source from GitHub by either cloning the source code repository or downloading thecurrent zip of the master branch. The code for this chapter is in `/java/Ch02-Schemas-and-Indexes`. You may import this code into your favorite editor, or just run it from the command line using the commands in `BuildAndRun.sh` if you are running on a *nix OS.

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. You can find the source for these POJO's in `Customer.java`, `Order.java` and `OrderSummaries.java`. Let's put some data into Riak so we can play with it.

Java

```
// From SipOfRiak.java

private static Customer createCustomer() {
    Customer customer = new Customer();
    customer.CustomerId = 1;
    customer.Name = "John Smith";
    customer.Address = "123 Main Street";
    customer.City = "Columbus";
    customer.State = "Ohio";
    customer.Zip = "43210";
    customer.Phone = "+1-614-555-5555";
    customer.CreatedDate = "2013-10-01 14:30:26";
    return customer;
}

private static ArrayList<Order> createOrders() {
    ArrayList<Order> orders = new ArrayList<Order>();

    Order order1 = new Order();
    order1.OrderId = 1;
    order1.CustomerId = 1;
    order1.SalespersonId = 9000;
    order1.Items.add(
            new Item("TCV37GIT4NJ",
```

```
                "USB 3.0 Coffee Warmer",
                15.99));
        order1.Items.add(
                new Item("PEG10BBF2PP",
                "eTablet Pro; 24GB; Grey",
                399.99));
        order1.Total = 415.98;
        order1.OrderDate = "2013-10-01 14:42:26";
        orders.add(order1);

        Order order2 = new Order();
        order2.OrderId = 2;
        order2.CustomerId = 1;
        order2.SalespersonId = 9001;
        order2.Items.add(
                new Item("OAX19XWN0QP",
                "GoSlo Digital Camera",
                359.99));
        order2.Total = 359.99;
        order2.OrderDate = "2013-10-15 16:43:16";
        orders.add(order2);

        Order order3 = new Order();
        order3.OrderId = 3;
        order3.CustomerId = 1;
        order3.SalespersonId = 9000;
        order3.Items.add(
                new Item("WYK12EPU5EZ",
                "Call of Battle = Goats - Gamesphere 4",
                69.99));
        order3.Items.add(
                new Item("TJB84HAA8OA",
                "Bricko Building Blocks",
                4.99));
        order3.Total = 74.98;
        order3.OrderDate = "2013-11-03 17:45:28";
        orders.add(order3);
        return orders;
```

```
}

private static OrderSummary createOrderSummary(ArrayList<Ord
    OrderSummary orderSummary = new OrderSummary();
    orderSummary.CustomerId = 1;
    for(Order order: orders)
    {
        orderSummary.Summaries.add(new OrderSummaryItem(orde
    }
    return orderSummary;
}

public static void main(String[] args) throws RiakException

    System.out.println("Creating Data");
    Customer customer = createCustomer();
    ArrayList<Order> orders = createOrders();
    OrderSummary orderSummary = createOrderSummary(orders);

    System.out.println("Starting Client");
    IRiakClient client = RiakFactory.pbcClient("127.0.0.1",


    System.out.println("Creating Buckets");
    Bucket customersBucket = client.fetchBucket("Customers")
    Bucket ordersBucket = client.fetchBucket("Orders").lazyL
    Bucket orderSummariesBucket = client.fetchBucket("OrderS

    System.out.println("Storing Data");
    customersBucket.store(String.valueOf(customer.CustomerId
    for (Order order : orders) {
        ordersBucket.store(String.valueOf(order.OrderId), or
    }
    orderSummariesBucket.store(String.valueOf(orderSummary.C
```

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.

Java

```
System.out.println("Fetching related data by shared key"
String key = "1";
String fetchedCust = customersBucket.fetch(key).execute(
String fetchedOrdSum = orderSummariesBucket.fetch(key).e
System.out.format("Customer     1: %s\n", fetchedCust);
System.out.format("OrderSummary 1: %s\n", fetchedOrdSum)
```

Which returns our amalgamated objects:

Shell

```
Fetching related data by shared key
Customer     1: {"CustomerId":1,"Name":"John Smith","Address
OrderSummary 1: {"CustomerId":1,"Summaries":[{"OrderId":1,"T
```

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.

Java

```
System.out.println("Adding Index Data");
IRiakObject riakObj = ordersBucket.fetch("1").execute();
riakObj.addIndex("SalespersonId", 9000);
riakObj.addIndex("OrderDate", "2013-10-01");
ordersBucket.store(riakObj).execute();
```

```
IRiakObject riakObj2 = ordersBucket.fetch("2").execute()
riakObj2.addIndex("SalespersonId", 9001);
riakObj2.addIndex("OrderDate", "2013-10-15");
ordersBucket.store(riakObj2).execute();

IRiakObject riakObj3 = ordersBucket.fetch("3").execute()
riakObj3.addIndex("SalespersonId", 9000);
riakObj3.addIndex("OrderDate", "2013-11-03");
ordersBucket.store(riakObj3).execute();
```

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 `SalespersonId` integer index for Jane's id of `9000`.

Java

```
// Query for orders where the SalespersonId index is set
List<String> janesOrders = ordersBucket.fetchIndex(IntIn
                                        .withValue(9000).

System.out.format("Jane's Orders: %s\n", StringUtil.Join
```

Which returns:

Text

```
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 `OrderDate` binary index for entries between `2013-10-01` and `2013-10-31`.

Java

```
// Query for orders where the OrderDate index is between
List<String> octoberOrders = ordersBucket.fetchIndex(Bin
                                        .from("2013-10-
```

```
    System.out.format("October's Orders: %s\n", StringUtil.J
```

Which returns:

```
Text
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