PHP Riak Client Documentation

Release 1.0

Fabio B. Silva <fabio.bat.silva@gmail.com>

January 06, 2017

Contents

CHAPTER 1

Taste of Riak: PHP

If you haven't set up a Riak Node and started it, please visit the Prerequisites first.

CHAPTER 2

Client Setup

Here is a Composer example:

```
$ composer require "php-riak/riak-client"
```

If you are using a single local Riak node, use the following to create a new client instance, assuming that the node is running on `localhost` port `8087` Protocol Buffers or port `8098` for http:

The easiest way to get started with the client is using a *RiakClientBuilder* :

```php
<?php
use Riak\Client\RiakClientBuilder;

$builder = new RiakClientBuilder();
$client  = $builder
    ->withNodeUri('http://192.168.1.1:8098')
    ->withNodeUri('proto://192.168.1.2:8087')
    ->build();
```

Once you have a `$client` we are now ready to start interacting with Riak.

CHAPTER 3

Creating Objects in Riak

The first object that we create is a very basic object with a content type of `text/plain`. Once that object is created, we create a `StoreValue` operation that will store the object later on down the line:

```
<?php
use Riak\Client\Command\Kv\StoreValue;
use Riak\Client\Core\Query\RiakObject;
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;

$object    = new RiakObject();
$namespace = new RiakNamespace('default', 'quotes');
$location  = new RiakLocation($namespace, 'icemand');

$object->setValue("You're dangerous, Maverick");
$object->setContentType('text/plain');

// store object
$store  = StoreValue::builder($location, $object)
    ->withPw(1)
    ->withW(2)
    ->build();

// Use our client object to execute the store operation
$client->execute($store);
```

CHAPTER 4

Reading Objects from Riak

After that, we check to make sure that the stored object has the same value as the object that we created. This requires us to fetch the object by way of a `FetchValue` operation:

```
<?php
use Riak\Client\Command\Kv\FetchValue;
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('default', 'quotes');
$location  = new RiakLocation($namespace, 'icemand');

// fetch object
$fetch  = FetchValue::builder($location)
    ->withNotFoundOk(true)
    ->withR(1)
    ->build();

/** @var $result \Riak\Client\Command\Kv\Response\FetchValueResponse */
/** @var $object \Riak\Client\Core\Query\RiakObject */
$result = $client->execute($fetch);
$object = $result->getValue();

echo $object->getValue();
// You're dangerous, Maverick
```

CHAPTER 5

Deleting Objects from Riak

Now that we've stored and then fetched the object, we can delete it by creating and executing a `DeleteValue` operation:

```
<?php
use Riak\Client\Command\Kv\DeleteValue;
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('default', 'quotes');
$location  = new RiakLocation($namespace, 'icemand');

// delete object
$delete  = DeleteValue::builder($location)
    ->withPw(1)
    ->withW(2)
    ->build();

$this->client->execute($delete);
```

CHAPTER 6

Working With Complex Objects

Since the world is a little more complicated than simple integers and bits of strings, let's see how we can work with more complex objects. Take for example, this plain PHP object that encapsulates some knowledge about a book.

```
<?php
class Book implements \JsonSerializable
{
    private $title;
    private $author;
    private $body;
    private $isbn;
    private $copiesOwned;

    // getter and setters.

    public function jsonSerialize()
    {
        return [
            'body'        => $this->body,
            'title'       => $this->title,
            'author'      => $this->author,
            'copiesOwned' => $this->copiesOwned,
        ];
    }
}
```

By default, the PHP Riak client serializes PHP Objets as JSON. Let's create a new Book object to store:

```
<?php
$mobyDick = new Book();

$modyDick->setTitle("Moby Dick");
$mobyDick->setAuthor("Herman Melville");
$mobyDick->setBody("Call me Ishmael. Some years ago...");
$mobyDick->setIsbn("11119799723");
$mobyDick->setCopiesOwned(3);
```

Now we can store that Object object just like we stored the riak object earlier:

```
<?php
use Riak\Client\Command\Kv\StoreValue;
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;
```

```
$namespace = new RiakNamespace('default', 'books');
$location  = new RiakLocation($namespace, 'moby_dick');

/** @var $mobyDick \Book */
$store  = StoreValue::builder($location, $mobyDick)
    ->withPw(1)
    ->withW(2)
    ->build();

$client->execute($store);
```

If we fetch the object using the same method we showed up above, we should get the following:

```
<?php
use Riak\Client\Command\Kv\FetchValue;
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('default', 'books');
$location  = new RiakLocation($namespace, 'moby_dick');

// fetch object
$fetch  = FetchValue::builder($location)
    ->withNotFoundOk(true)
    ->withR(1)
    ->build();

/** @var $result \Riak\Client\Command\Kv\Response\FetchValueResponse */
/** @var $object \Riak\Client\Core\Query\RiakObject */
$result = $client->execute($fetch);
$book   = $result->getValue('Book');
$object = $result->getValue();

echo $book->getTitle();
//  "Moby Dick"

echo $object->getValue();
/*
{
  "title": "Moby Dick",
  "author": "Herman Melville",
  "body": "Call me Ishmael. Some years ago...",
  "isbn": "1111979723",
  "copiesOwned": 3
}
*/
```

CHAPTER 7

Development

All development is done on Github. Use Issues to report problems or submit contributions.

This tutorial documentation its based on the Basho Taste of Riak Docs.

7.1 Contents

7.1.1 Riak Client

The easiest way to get started with the client is using a *RiakClientBuilder* :

```
<?php
use Riak\Client\RiakClientBuilder;

$builder = new RiakClientBuilder();
$client  = $builder
    ->withNodeUri('http://192.168.1.1:8098')
    ->withNodeUri('proto://192.168.1.2:8087')
    ->build();
```

Once you have a $client, commands from the *RiakClientCommand** namespace are built then executed by the client.

```
<?php
use Riak\Client\Command\Kv\FetchValue;
use Riak\Client\Core\Query\RiakObject;
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('bucket_type', 'bucket_name');
$location  = new RiakLocation($namespace, 'object_key');

// fetch object
$fetch  = FetchValue::builder($location)
    ->withNotFoundOk(true)
    ->withR(1)
    ->build();

$result = $client->execute($fetch);
$object = $result->getValue();
```

RiakCommand classes

- Fetching, storing and deleting objects
  - `Riak\Client\Command\Kv\FetchValue`
  - `Riak\Client\Command\Kv\StoreValue`
  - `Riak\Client\Command\Kv\UpdateValue`
  - `Riak\Client\Command\Kv\DeleteValue`
- Fetching and storing datatypes (CRDTs)
  - `Riak\Client\Command\DataType\FetchCounter`
  - `Riak\Client\Command\DataType\FetchSet`
  - `Riak\Client\Command\DataType\FetchMap`
  - `Riak\Client\Command\DataType\StoreCounter`
  - `Riak\Client\Command\DataType\StoreSet`
  - `Riak\Client\Command\DataType\StoreMap`
- Querying and modifying buckets
  - `Riak\Client\Command\Bucket\FetchBucketProperties`
  - `Riak\Client\Command\Bucket\StoreBucketProperties`
- Secondary index (2i)
  - `Riak\Client\Command\Index\BinIndexQuery`
  - `Riak\Client\Command\Index\IntIndexQuery`
- Yokozuna Search
  - `Riak\Client\Command\Search\DeleteIndex`
  - `Riak\Client\Command\Search\FetchIndex`
  - `Riak\Client\Command\Search\StoreIndex`
  - `Riak\Client\Command\Search\StoreSchema`
  - `Riak\Client\Command\Search\FetchSchema`
  - `Riak\Client\Command\Search\Search`

7.1.2 Buckets & Bucket Types

For more information on how bucket types work and how to managing bucket types see : Basho Bucket Types Docs.

In versions of Riak prior to 2.0, all queries are made to a bucket/key pair, as in the following example read request:

```
<?php
use Riak\Client\Command\Kv\FetchValue;
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('<bucket_type>', '<bucket_name>');
$location  = new RiakLocation($namespace, '<key>');
```

```
// fetch object
$fetch  = FetchValue::builder($location)
    ->withNotFoundOk(true)
    ->withR(1)
    ->build();

/** @var $result \Riak\Client\Command\Kv\Response\FetchValueResponse */
$result = $client->execute($fetch);
```

To modify the properties of a bucket in Riak:

```
<?php
use Riak\Client\Command\Bucket\StoreBucketProperties;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('<bucket_type>', '<bucket_name>');
$store     =StoreBucketProperties::builder($namespace)
    ->withLinkwalkFunction($linkwalkFunction)
    ->withChashkeyFunction($chashkeyFunction)
    ->withPostcommitHook($postcommitFunction)
    ->withPrecommitHook($precommitFunction)
    ->withSearchIndex($searchIndex)
    ->withBasicQuorum($basicQuorum)
    ->withLastWriteWins($wins)
    ->withBackend($backend)
    ->withAllowMulti($allow)
    ->withNotFoundOk($ok)
    ->withSmallVClock($smallVClock)
    ->withYoungVClock($youngVClock)
    ->withOldVClock($ldVClock)
    ->withBigVClock($bigVClock)
    ->withNVal($nVal)
    ->withRw($rw)
    ->withDw($dw)
    ->withPr($pr)
    ->withPw($pw)
    ->withW($w)
    ->withR($w);

$this->client->execute($store);
```

We can simply retrieve the bucket properties :

```
<?php
use Riak\Client\Command\Bucket\FetchBucketProperties;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('<bucket_type>', '<bucket_name>');
$fetch     = FetchBucketProperties::builder()
        ->withNamespace($namespace)
        ->build();

/** @var $response \Riak\Client\Command\Bucket\Response\FetchBucketPropertiesResponse */
/** @var $props \Riak\Client\Core\Query\BucketProperties */
$response = $this->client->execute($fetch);
$props    = $response->getProperties();

echo $props->getNVal();
// 3
```

This tutorial documentation its based on the Basho Bucket Types Docs.

7.1.3 Key/Value commands

StoreValue

```
<?php
use Riak\Client\Command\Kv\StoreValue;
use Riak\Client\Core\Query\RiakObject;
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('bucket_type', 'bucket_name');
$location  = new RiakLocation($namespace, 'object_key');
$object    = new RiakObject();

$object->setContentType('application/json');
$object->setValue('{"name": "FabioBatSilva"}');

// store object
$store  = StoreValue::builder($location)
    ->withReturnBody(true)
    ->withPw(1)
    ->withW(2)
    ->build();

$result = $client->execute($store);
$object = $result->getValue();
```

FetchValue

```
<?php
use Riak\Client\Command\Kv\FetchValue;
use Riak\Client\Core\Query\RiakObject;
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('bucket_type', 'bucket_name');
$location  = new RiakLocation($namespace, 'object_key');

// fetch object
$fetch  = FetchValue::builder($location)
    ->withNotFoundOk(true)
    ->withR(1)
    ->build();

$result = $client->execute($fetch);
$object = $result->getValue();
```

DeleteValue

```
<?php
use Riak\Client\Command\Kv\DeleteValue;
use Riak\Client\Core\Query\RiakObject;
```

```
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('bucket_type', 'bucket_name');
$location  = new RiakLocation($namespace, 'object_key');

// delete object
$delete  = DeleteValue::builder($location)
    ->withPw(1)
    ->withW(2)
    ->build();

$client->execute($delete);
```

ListKeys

```
<?php
use Riak\Client\Command\Kv\ListKeys;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('bucket_type', 'bucket_name');
$location  = new RiakLocation($namespace, 'object_key');

/** @var $result \Riak\Client\Command\Kv\Response\ListKeysResponse */
/** @var $locations \Riak\Client\Core\Query\RiakLocation[] */
$result    = $client->execute($command);
$locations = $result->getLocations();

echo $locations[0]->getKey();
// object_key
```

7.1.4 Data Types

This tutorial documentation its based on the Basho CRDT Docs. If you are not familiar with crdt in Riak Before you start take a look at Basho CRDT Docs for more details.

In versions 2.0 and greater, Riak users can make use of a variety of Riak-specific data types inspired by research on convergent replicated data types, more commonly known as CRDTs.

Location

Here is the general syntax for setting up a bucket type/bucket/key combination to handle a data type:

```
<?php
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('<bucket_type>', '<bucket_name>');
$location  = new RiakLocation($namespace, '<key>');
```

Counters

Counters are a bucket-level Riak Data Type that can be used either by themselves, i.e. associated with a bucket/key pair, or within a map. The examples in this section will show you how to use counters on their own.

Let's say that we want to create a counter called `traffic_tickets` in our counters bucket to keep track of our legal misbehavior. We can create this counter and ensure that the counters bucket will use our counters bucket type like this:

```
<?php
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('counters', 'counters');
$location  = new RiakLocation($namespace, 'traffic_tickets');
```

Now that our client knows which bucket/key pairing to use for our counter, `traffic_tickets` will start out at 0 by default. If we happen to get a ticket that afternoon, we would need to increment the counter

```
use Riak\Client\Command\DataType\StoreCounter;

<?php
// Increment the counter by 1 and fetch the current value
$store = StoreCounter::builder()
    ->withReturnBody(true)
    ->withLocation($location)
    ->withDelta(1)
    ->build();

$result  = $client->execute($store);
$counter = $result->getDatatype();
$value   = $counter->getValue();

echo $value;
// 1
```

If we're curious about how many tickets we have accumulated, we can simply retrieve the value of the counter at any time:

```
use Riak\Client\Command\DataType\FetchCounter;

<?php
// fetch counter
$fetch = FetchCounter::builder()
    ->withLocation($location)
    ->withR(1)
    ->build();

$result  = $client->execute($store);
$counter = $result->getDatatype();
$value   = $counter->getValue();
```

For a counter to be useful, you need to be able to decrement it in addition to incrementing it. Riak counters enable you to do precisely that. Let's say that we hire an expert lawyer who manages to get one of our traffic tickets stricken from our record:

```
use Riak\Client\Command\DataType\StoreCounter;

<?php
```

```
$store = StoreCounter::builder()
    ->withLocation($location)
    ->withDelta(-1)
    ->build();

$client->execute($store);
```

Sets

As with counters (and maps, as shown below), using sets involves setting up a bucket/key pair to house a set and running set-specific operations on that pair.

Here is the general syntax for setting up a bucket type/bucket/key combination to handle a set:

```
<?php
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('sets', 'travel');
$location  = new RiakLocation($namespace, 'cities');
```

Let's say that we read a travel brochure saying that Toronto and Montreal are nice places to go. Let's add them to our cities set:

```
<?php

use Riak\Client\RiakOption;
use Riak\Client\Command\DataType\StoreSet;

// Store new cities and return the current value
$store = StoreCounter::builder()
    ->withLocation($location)
    ->withReturnBody(true)
    ->build();

$store->add("Toronto")
$store->add("Montreal")

$result = $client->execute($store);
$set    = $set->getDatatype();
$value  = $counter->getValue();

var_dump($value);
// ["Toronto", "Montreal"]
```

Later on, we hear that Hamilton and Ottawa are nice cities to visit in Canada, but if we visit them, we won't have time to visit Montreal, so we need to remove it from the list. It needs to be noted here that removing an element from a set is a bit tricker than adding elements.

```
<?php

use Riak\Client\RiakOption;
use Riak\Client\Command\DataType\StoreSet;
use Riak\Client\Command\DataType\FecthSet;

$fetch = FecthSet::builder()
    ->withLocation($location)
    ->build();
```

```
$fetchResult  = $client->execute($store);
$fetchContext = $result->getContext();

$store = StoreCounter::builder()
    ->withContext($fetchContext)
    ->withLocation($location)
    ->withReturnBody(true)
    ->build();

$store->add("Ottawa");
$store->add("Vancouver");
$store->remove("Montreal");

$result  = $client->execute($store);
$set     = $result->getDatatype();
$value   = $set->getValue();

var_dump($value);
// ["Ottawa","Vancouver","Toronto"]
```

Maps

The map is in many ways the richest of the Riak Data Types because all of the other Data Types can be embedded within them, including maps themselves, to create arbitrarily complex custom Data Types out of a few basic building blocks

Let's say that we want to use Riak to store information about our company's customers. We'll use the bucket `customers` to do so. Each customer's data will be contained in its own key in the `customers` bucket. Let's create a map for the user Ahmed (`ahmed_info`) in our bucket and simply call it `map` for simplicity's sake:

```
<?php
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;

$namespace = new RiakNamespace('maps', 'customers');
$location  = new RiakLocation($namespace, 'ahmed_info');
```

Register & Flags Within Maps

The first piece of info we want to store in our map is Ahmed's name and phone number, both of which are best stored as registers We'll also create an *enterprise_customer* flag to track whether Ahmed has signed up for the new plan:

```
<?php

use Riak\Client\RiakOption;
use Riak\Client\Command\DataType\StoreMap;

$store = StoreMap::builder($location)
    ->updateRegister('first_name', 'Ahmed')
    ->updateRegister('phone_number', '5551234567')
    ->updateFlag('enterprise_customer', false)
    ->withReturnBody(true)
    ->build();

$result = $client->execute($store);
```

```
$map    = $result->getDatatype();

echo $map->get('first_name');
// Ahmed
echo $map->get('phone_number');
// 5551234567
echo $map->get('enterprise_customer');
// false
```

We can retrieve the value of that flag at any time:

```
<?php

use Riak\Client\RiakOption;
use Riak\Client\Command\DataType\FetchMap;

$fetch = FetchMap::builder()
    ->withLocation($location)
    ->build();

$result = $client->execute($fetch);
$map    = $result->getDatatype();
$value  = $map->getValue();

echo $map->get('first_name');
echo $map->get('phone_number');
echo $map->get('enterprise_customer');
```

Counters Within Maps

We also want to know how many times Ahmed has visited our website. We'll use a `page_visits` counter for that and run the following operation when Ahmed visits our page for the first time:

```
<?php

use Riak\Client\RiakOption;
use Riak\Client\Command\DataType\StoreMap;

$store = StoreMap::builder()
    ->withLocation($location)
    ->updateCounter('page_visits', 1)
    ->build();

$client->execute($store);
```

Sets Within Maps

We'd also like to know what Ahmed's interests are so that we can better design a user experience for him. Through his purchasing decisions, we find out that Ahmed likes robots, opera, and motorcyles. We'll store that information in a set inside of our map:

```
<?php

use Riak\Client\Command\DataType\StoreMap;
```

```
$store = StoreMap::builder()
    ->withLocation($location)
    ->updateSet('interests', ['robots', 'opera' , 'motorcycles'])
    ->build();

$client->execute($store);
```

We learn from a recent purchasing decision that Ahmed actually doesn't seem to like opera. He's much more keen on indie pop. Let's change the interests set to reflect that:

```
<?php

use Riak\Client\Command\DataType\FetchMap;
use Riak\Client\Command\DataType\StoreMap;
use Riak\Client\Command\DataType\SetUpdate;

$fetch = FetchMap::builder()
    ->withLocation($location)
    ->build();

$fetchResult  = $client->execute($fetch);
$fetchContext = $fetchResult->getContext();
$setUpdate    = new SetUpdate();

$setUpdate->remove('opera');

$store = StoreMap::builder()
    ->withLocation($location)
    ->withContext($fetchContext)
    ->updateSet('interests', $setUpdate)
    ->withReturnBody(true)
    ->build();

$result = $client->execute($store);
$map    = $result->getDatatype();

var_dump($map->get('interests'));
// ['robots', 'motorcycles']
```

Maps Within Maps

We've stored a wide of variety of information—of a wide variety of types—within the `ahmed_info` map thus far, but we have yet to explore recursively storing maps within maps (which can be nested as deeply as you wish).

Our company is doing well and we have lots of useful information about Ahmed, but now we want to store information about Ahmed's contacts as well. We'll start with storing some information about Ahmed's colleague Annika inside of a map called `annika_info`.

First, we'll store Annika's first name, last name, and phone number in registers:

```
<?php

use Riak\Client\Command\DataType\FetchMap;
use Riak\Client\Command\DataType\StoreMap;

$fetch = FetchMap::builder()
    ->withLocation($location)
```

```
    ->build();

$fetchResult  = $client->execute($fetch);
$fetchContext = $fetchResult->getContext();

$store = StoreMap::builder()
    ->withReturnBody(true)
    ->withContext($fetchContext)
    ->withLocation($location)
    ->updateMap('annika_info', [
        'first_name'   => 'Annika',
        'last_name'    => 'Weiss',
        'phone_number' => '5559876543'
    ])
    ->build();

$result     = $client->execute($store);
$map        = $result->getDatatype();
$annikaInfo = $map->get('annika_info');

echo $annikaInfo['first_name'];
// Annika
```

Map values can also be removed:

```
<?php

use Riak\Client\Command\DataType\FetchMap;
use Riak\Client\Command\DataType\StoreMap;
use Riak\Client\Command\DataType\MapUpdate;

$fetch = FetchMap::builder()
    ->withLocation($location)
    ->build();

$fetchResult  = $client->execute($fetch);
$fetchContext = $fetchResult->getContext();
$mapUpdate    = new MapUpdate();

$mapUpdate->removeRegister('first_name');

$store = StoreMap::builder()
    ->updateMap('annika_info', $mapUpdate)
    ->withContext($fetchContext)
    ->withLocation($location)
    ->build();

$client->execute($store);
```

Now, we'll store whether Annika is subscribed to a variety of plans within the company as well:

```
<?php

use Riak\Client\Command\DataType\FetchMap;
use Riak\Client\Command\DataType\StoreMap;
use Riak\Client\Command\DataType\MapUpdate;

$fetch = FetchMap::builder()
    ->withLocation($location)
```

```
    ->build();

$fetchResult  = $client->execute($fetch);
$fetchContext = $fetchResult->getContext();
$mapUpdate    = new MapUpdate();

$mapUpdate
    ->updateFlag('enterprise_plan', false)
    ->updateFlag('family_plan', false)
    ->updateFlag('free_plan', true);

$store = StoreMap::builder()
    ->updateMap('annika_info', $mapUpdate)
    ->withContext($fetchContext)
    ->withLocation($location)
    ->build();

$client->execute($store);
```

The value of a flag can be retrieved at any time:

```
<?php

use Riak\Client\RiakOption;
use Riak\Client\Command\DataType\FetchMap;

$fetch = FetchMap::builder()
    ->withLocation($location)
    ->build();

$result     = $client->execute($fetch);
$map        = $result->getDatatype();
$annikaInfo = $map->get('annika_info');

echo $annikaInfo['enterprise_plan'];
// false
```

It's also important to track the number of purchases that Annika has made with our company. Annika just made her first widget purchase, w'll also store Annika's interests in a set:

```
<?php

use Riak\Client\Command\DataType\FetchMap;
use Riak\Client\Command\DataType\StoreMap;
use Riak\Client\Command\DataType\MapUpdate;
use Riak\Client\Command\DataType\SetUpdate;

$fetch = FetchMap::builder()
    ->withLocation($location)
    ->withIncludeContext(true)
    ->build();

$fetchResult  = $client->execute($fetch);
$fetchContext = $fetchResult->getContext();
$mapUpdate    = new MapUpdate();
$setUpdate    = new SetUpdate();

$setUpdate
    ->add("tango dancing");
```

```
$mapUpdate
    ->updateCounter('widget_purchases', 1)
    ->updateCounter('interests', $setUpdate);

$store = StoreMap::builder()
    ->updateMap('annika_info', $mapUpdate)
    ->withContext($fetchContext)
    ->withLocation($location)
    ->build();

$client->execute($store);
```

If we wanted to add store information about one of Annika's specific purchases, we could do so within a map:

```
<?php

use Riak\Client\Command\DataType\FetchMap;
use Riak\Client\Command\DataType\StoreMap;
use Riak\Client\Command\DataType\MapUpdate;

$fetch = FetchMap::builder()
    ->withLocation($location)
    ->build();

$fetchResult  = $client->execute($fetch);
$fetchContext = $fetchResult->getContext();
$mapUpdate    = new MapUpdate();

$mapUpdate
    ->updateMap('purchase', [
        'first_purchase' => true,              // flag
        'amount'         => "1271",            // register
        'items'          => ["large widget"], // set
    ]);

$store = StoreMap::builder()
    ->updateMap('annika_info', $mapUpdate)
    ->withContext($fetchContext)
    ->withLocation($location)
    ->build();

$client->execute($store);
```

7.1.5 Secondary Indexes

This tutorial documentation its based on the Basho Secondary Indexes Docs. If you are not familiar with search in Riak Before you start take a look at Basho Secondary Indexes Docs for more details.

Secondary indexing (2i) in Riak gives developers the ability to tag an object stored in Riak, at write time, with one or more queryable values. Those values can be either a binary or string, such as sensor_1_data or admin_user or click_event, or an integer, such as 99 or 141121.

Since key/value data is completely opaque to 2i, applications must attach metadata to objects that tell Riak 2i exactly which attribute(s) to index and what the values of those indexes should be.

Riak Search serves analogous purposes but is quite different because it parses key/value data itself and builds indexes on the basis of Solr schemas.

Store Object with Secondary Indexes

In this example, the key `john_smith` is used to store user data in the bucket `users`, which bears the `default` bucket type. Let's say that an application would like add a Twitter handle and an email address to this object as secondary indexes.

```
<?php
use Riak\Client\Command\Kv\StoreValue;
use Riak\Client\Core\Query\RiakObject;
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;
use Riak\Client\Core\Query\Index\RiakIndexBin;

$namespace = new RiakNamespace('default', 'users');
$location  = new RiakLocation($namespace, 'john_smith');
$object    = new RiakObject();

$object->setContentType('application/json');
$object->setValue('{"name": "FabioBatSilva"}');
$object->addIndex(new RiakIndexBin('twitter', ['jsmith123']));
$object->addIndex(new RiakIndexBin('email', ['jsmith@basho.com']));

$command = StoreValue::builder($location)
    ->withValue($object)
    ->withW(3)
    ->build();

// store object
$client->execute($command);
```

This has accomplished the following :

- The object has been stored with a primary bucket/key of `users/john_smith`
- The object now has a secondary index called twitter_bin with a value of `jsmith123`
- The object now has a secondary index called email_bin with a value of `jsmith@basho.com`

Querying Objects with Secondary Indexes

Let's query the `users` bucket on the basis of Twitter handle to make sure that we can find our stored object:

```
<?php

use Riak\Client\Core\Query\RiakNamespace;
use Riak\Client\Command\Index\BinIndexQuery;

$namespace = new RiakNamespace('default', 'users');
$command   = BinIndexQuery::builder()
    ->withNamespace($namespace)
    ->withIndexName('twitter')
    ->withMatch('jsmith123')
    ->build();

/** @var $result \Riak\Client\Command\Index\Response\IndexQueryResponse */
/** @var $entries array */
$result  = $this->client->execute($command);
$entries = $result->getEntries();
```

```
echo $entries[0]->getLocation()->getKey();
// john_smith
```

Querying

Exact Match

The following examples perform an exact match index query.

Query a binary index:

```
<?php

use Riak\Client\Core\Query\RiakNamespace;
use Riak\Client\Command\Index\BinIndexQuery;

$namespace = new RiakNamespace('bucket-type', 'bucket-name');
$command   = BinIndexQuery::builder()
    ->withNamespace($namespace)
    ->withIndexName('index-name')
    ->withMatch('index-val')
    ->build();

/** @var $result \Riak\Client\Command\Index\Response\IndexQueryResponse */
$result  = $this->client->execute($command);
```

Query an integer index:

```
<?php

use Riak\Client\Core\Query\RiakNamespace;
use Riak\Client\Command\Index\IntIndexQuery;

$namespace = new RiakNamespace('bucket-type', 'bucket-name');
$command   = IntIndexQuery::builder()
    ->withNamespace($namespace)
    ->withIndexName('index-name')
    ->withMatch(101)
    ->build();

/** @var $result \Riak\Client\Command\Index\Response\IndexQueryResponse */
$result  = $this->client->execute($command);
```

Range

The following examples perform a range query:

Query a binary index :

```
<?php

use Riak\Client\Core\Query\RiakNamespace;
use Riak\Client\Command\Index\BinIndexQuery;

$namespace = new RiakNamespace('bucket-type', 'bucket-name');
$command   = BinIndexQuery::builder()
```

```
    ->withNamespace($namespace)
    ->withIndexName('index-name')
    ->withStart('val1')
    ->withEnd('val9')
    ->build();

/** @var $result \Riak\Client\Command\Index\Response\IndexQueryResponse */
$result  = $this->client->execute($command);
```

Query a integer index :

```
<?php

use Riak\Client\Core\Query\RiakNamespace;
use Riak\Client\Command\Index\BinIndexQuery;

$namespace = new RiakNamespace('bucket-type', 'bucket-name');
$command   = BinIndexQuery::builder()
    ->withNamespace($namespace)
    ->withIndexName('index-name')
    ->withStart(1)
    ->withEnd(100)
    ->build();

/** @var $result \Riak\Client\Command\Index\Response\IndexQueryResponse */
$result  = $this->client->execute($command);
```

Range with terms

When performing a range query, it is possible to retrieve the matched index values alongside the Riak keys using `return_terms=true`. An example from a small sampling of Twitter data with indexed hash tags:

Query a binary index :

```
<?php

use Riak\Client\Core\Query\RiakNamespace;
use Riak\Client\Command\Index\BinIndexQuery;

$namespace = new RiakNamespace('bucket-type', 'bucket-name');
$command   = BinIndexQuery::builder()
    ->withNamespace($namespace)
    ->withIndexName('index-name')
    ->withReturnTerms(true)
    ->withStart('val1')
    ->withEnd('val9')
    ->build();

/** @var $result \Riak\Client\Command\Index\Response\IndexQueryResponse */
$result  = $this->client->execute($command);
```

Pagination

When asking for large result sets, it is often desirable to ask the servers to return chunks of results instead of a firehose. You can do so using max_results=<n>, where n is the number of results you'd like to receive.

Assuming more keys are available, a continuation value will be included in the results to allow the client to request the next page. Here is an example of a range query with both return_terms and pagination against the same Twitter data set:

Note: Index queries are always made using streaming, `IndexQueryResponse#getIterator()` will return a stream iterator that can be used to iterate over the response entries.

Notice that is not possible to rewind a stream iterator, If you need to re-use the result use `IndexQueryResponse#getEntries()` instead.

```
<?php

use Riak\Client\Core\Query\RiakNamespace;
use Riak\Client\Command\Index\IntIndexQuery;

$namespace = new RiakNamespace('bucket-type', 'bucket-name');
$command   = IntIndexQuery::builder()
    ->withNamespace($namespace)
    ->withIndexName('index-name')
    ->withReturnTerms(true)
    ->withMaxResults(100)
    ->withStart(1)
    ->withEnd(99999)
    ->build();

/** @var $result \Riak\Client\Command\Index\Response\IndexQueryResponse */
/** @var $iterator \Iterator */
$result   = $this->client->execute($command);
$iterator = $result->getIterator();

/** @var $entry \Riak\Client\Command\Index\Response\IndexEntry */
foreach ($iterator as $entry) {
    /// ...
}
```

After after iterating over the response entries Take the continuation value from the previous result set and feed it back into the query

```
<?php

use Riak\Client\Core\Query\RiakNamespace;
use Riak\Client\Command\Index\IntIndexQuery;

/** @var $result \Riak\Client\Command\Index\Response\IndexQueryResponse */
$continuation = $result->getContinuation();
$nextCommand  = IntIndexQuery::builder()
    ->withNamespace($namespace)
    ->withContinuation($continuation)
    ->withIndexName('index-name')
    ->withReturnTerms(true)
    ->withMaxResults(100)
    ->withStart(1)
    ->withEnd(99999)
    ->build();

/** @var $result \Riak\Client\Command\Index\Response\IndexQueryResponse */
$nextResult = $this->client->execute($nextCommand);
```

7.1.6 Yokozuna Search

This tutorial documentation its based on the Basho Search Docs. If you are not familiar with search in Riak Before you start take a look at Basho Search Docs for more details.

Riak Search 2.0 is a new open-source project integrated with Riak. It allows for distributed, scalable, fault-tolerant, transparent indexing and querying of Riak values. It's easy to use. After connecting a bucket (or bucket type) to a Solr index, you simply write values (such as JSON, XML, plain text, Riak Data Types, etc.) into Riak as normal, and then query those indexed values using the Solr API.

Riak Search 2.0 is an integration of Solr (for indexing and querying) and Riak (for storage and distribution). There are a few points of interest that a user of Riak Search will have to keep in mind in order to properly store and later query for values.

Schemas explain to Solr how to index fields Indexes are named Solr indexes against which you will query Bucket-index association signals to Riak when to index values (this also includes bucket type-index association)

Create Index

Let's start by creating an index called famous that uses the default schema.

```
<?php
use Riak\Client\Command\Search\StoreIndex;
use Riak\Client\Core\Query\Search\YokozunaIndex;

$indexName = 'famous';
$index     = new YokozunaIndex($indexName, '_yz_default');
$command   = StoreIndex::builder()
    ->withIndex($index)
    ->build();

$client->execute($command);
```

Deleting Indexes

Indexes may be deleted if they have no buckets associated with them:

```
<?php
use Riak\Client\Command\Search\DeleteIndex;

$indexName = 'famous';
$command   = DeleteIndex::builder()
    ->withIndexName($indexName)
    ->build();

$client->execute($command);
```

Bucket search index

The last setup item that you need to perform is to associate either a bucket or a bucket type with a Solr index. You only need do this once per bucket type, and all buckets within that type will use the same Solr index.

Although we recommend that you use all new buckets under a bucket type, if you have existing data with a type-free bucket (i.e. under the default bucket type) you can set the search_index property for a specific bucket.

```php
<?php

use Riak\Client\Core\Query\RiakNamespace;
use Riak\Client\Command\Search\StoreIndex;
use Riak\Client\Core\Query\BucketProperties;
use Riak\Client\Command\Bucket\StoreBucketProperties;

$namespace = new RiakNamespace("cats")
$command   = StoreBucketProperties::builder()
    ->withSearchIndex('famous')
    ->withNamespace($namespace)
    ->build();

$client->execute($command);
```

Indexing Values

With a Solr schema, index, and association in place (and possibly a security setup as well), we're ready to start using Riak Search. First, populate the cat bucket with values, in this case information about four cats: Liono, Cheetara, Snarf, and Panthro.

```php
<?php

use Riak\Client\Core\Query\RiakNamespace;
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakObject;
use Riak\Client\Command\Kv\StoreValue;

$lionoObject    = new RiakObject();
$cheetaraObject = new RiakObject();
$snarfObject    = new RiakObject();
$panthroObject  = new RiakObject();

$lionoObject->setContentType('application/json');
$lionoObject->setValue(json_encode([
    'name_s'   => 'Lion-o',
    'leader_b' => true,
    'age_i'    => 30,
]));

$cheetaraObject->setContentType('application/json');
$cheetaraObject->setValue(json_encode([
    'name_s'   => 'Cheetara',
    'leader_b' => false,
    'age_i'    => 30,
]));

$snarfObject->setContentType('application/json');
$snarfObject->setValue(json_encode([
    'name_s'   => 'Snarf',
    'leader_b' => false,
    'age_i'    => 43,
]));

$panthroObject->setContentType('application/json');
$panthroObject->setValue(json_encode([
    'name_s'   => 'Panthro',
```

```
    'leader_b' => false,
    'age_i'    => 36,
]));

// All the store commands can be built the same way
$namespace  = new RiakNamespace('default', 'cats');
$location   = new RiakLocation($namespace, $key);
$lionoStore = StoreValue::builder($location, $lionoObject)
    ->withPw(1)
    ->withW(2)
    ->build();

// The other storage operations can be performed the same way
$client->execute($lionoStore);
```

Querying

All distributed Solr queries are supported, which actually includes most of the single-node Solr queries. This example searches for all documents in which the name_s value begins with Lion by means of a glob (wildcard) match.

```
<?php

use Riak\Client\Command\Search\Search;

$search = Search::builder()
    ->withQuery('name_s:Lion*')
    ->withIndex("famous")
    ->build();

$searchResult  = $this->client->execute($search);
$numResults    = $searchResult->getNumResults();
$allResults    = $searchResult->getAllResults();
$singleResults = $searchResult->getSingleResults();

echo $numResults;
// 1

echo $singleResults[0]['name_s'];
// Lion-o

echo json_encode($allResults[0]['name_s']);
// ["Lion-o"]
```

The response to a query will be an object containing details about the response, such as a query's max score and a list of documents which match the given query.

Note: While `SearchResponse#getSingleResults()` returns only the first entry of each element from the search query result. `SearchResponse#getAllResults()` will return a list containing all the result sets, so if you have a multi-valued field you should probably use `getAllResults`

Range Queries

Range queries are searches within a range of numerical or date values.

To find the ages of all famous cats who are 30 or younger: `age_i:[0 TO 30]`. If you wanted to find all cats 30 or older, you could include a glob as a top end of the range: `age_i:[30 TO *]`.

In this example the query fields are returned because they're stored in Solr. This depends on your schema. If they are not stored, you'll have to perform a separate Riak GET operation to retrieve the value using the _yz_rk value.

```
<?php

use Riak\Client\Command\Search\Search;
use Riak\Client\Command\Kv\FetchValue;
use Riak\Client\Core\Query\RiakLocation;
use Riak\Client\Core\Query\RiakNamespace;

$search = Search::builder()
    ->withQuery('age_i:[30 TO *]')
    ->withIndex("famous")
    ->build();

/** @var $result \Riak\Client\Command\Search\Response\SearchResponse */
/** @var $results array */
$searchResult = $this->client->execute($search);
$results      = $searchResult->getSingleResults();

//  retrieve ``_yz_`` values
$bucketType = $results[0]["_yz_rt"];
$bucketName = $results[0]["yz_rb"];
$key        = $results[0]["_yz_rk"];

// create reference object locations
$namespace = new RiakNamespace($bucketType , $bucketName;
$location  = new RiakLocation($namespace, $key);

// fetch object
$fetch  = FetchValue::builder($location)
    ->withNotFoundOk(true)
    ->withR(1)
    ->build();

/** @var $result \Riak\Client\Command\Kv\Response\FetchValueResponse */
/** @var $object \Riak\Client\Core\Query\RiakObject */
$result = $client->execute($fetch);
$object = $result->getValue();

echo $object->getValue();
// {"name_s": "Lion-o", "age_i": 30, "leader_b": true}
```

Pagination

A common requirement you may face is paginating searches, where an ordered set of matching documents are returned in non-overlapping sequential subsets (in other words, pages). This is easy to do with the start and rows parameters, where start is the number of documents to skip over (the offset) and rows are the number of results to return in one go.

For example, assuming we want two results per page, getting the second page is easy, where start is calculated as (rows per page) * (page number - 1).

```
<?php

use Riak\Client\Command\Search\Search;
```

```
$rowsPerPage = 2;
$page        = 2;
$start       = $rowsPerPage * ($page - 1);

$search = Search::builder()
    ->withNumRows($rowsPerPage)
    ->withIndex("famous")
    ->withStart($start)
    ->withQuery('*:*')
    ->build();

/** @var $result \Riak\Client\Command\Search\Response\SearchResponse */
/** @var $results array */
$searchResult = $this->client->execute($search);
$results      = $searchResult->getAllResults();
```

7.1.7 Map Reduce

This tutorial documentation its based on the Basho MapReduce Docs.

How it Works

The MapReduce framework helps developers divide a query into steps, divide the dataset into chunks, and then run those step/chunk pairs in separate physical hosts.

In Riak, MapReduce is one of the primary methods for non-primary-key-based querying in Riak, alongside secondary indexes. Riak allows you to run MapReduce jobs using Erlang or JavaScript.

There are two steps in a MapReduce query:

- `Map` — The data collection phase, which breaks up large chunks of work into smaller ones and then takes action on each chunk. Map phases consist of a function and a list of objects on which the map operation will operate.
- `Reduce` — The data collation or processing phase, which combines the results from the map step into a single output. The reduce phase is optional.

Riak MapReduce queries have two components:

- A list of inputs
- A list of phases

The elements of the input list are object locations as specified by bucket type, bucket, and key. The elements of the phases list are chunks of information related to a map, a reduce, or a link function.

MapReduce Commands

- *BucketMapReduce* Map-Reduce operation over a bucket in Riak.
- *BucketKeyMapReduce* Map-Reduce operation over a specific set of keys in a bucket.
- *IndexMapReduce* Map-Reduce operation using a secondary index (2i) as input.
- *SearchMapReduce* Map-Reduce operation with a search query as input.

BucketMapReduce

Command used to perform a Map Reduce operation over a bucket in Riak.

Here is the general syntax for setting up a bucket map reduce combination to handle a range of keys:

```
<?php
use Riak\Client\Command\MapReduce\BucketMapReduce;
use Riak\Client\Core\Query\Func\NamedJsFunction;
use Riak\Client\Command\MapReduce\KeyFilters;
use Riak\Client\Core\Query\RiakNamespace;

$map       = new NamedJsFunction('Riak.mapValuesJson');
$namespace = new RiakNamespace('bucket_type', 'bucket_name');
$filter    = KeyFilters::filter()->between('key1', 'key9', false);
$command   = BucketMapReduce::builder()
    ->withMapPhase($map, null, true)
    ->withNamespace($namespace)
    ->withKeyFilter($filter)
    ->build();

/* @var $result \Riak\Client\Command\MapReduce\Response\BucketMapReduceResponse */
$result = $this->client->execute($command);
$values = $result->getResultForPhase(0);

echo $values[0];
// ... first element response
```

See Basho KeyFilters Docs. for more details on filters

BucketKeyMapReduce

Command used to perform a map reduce operation over a specific set of keys in a bucket.

Here is the general syntax for setting up a bucket map over a specific set of keys:

```
<?php
use Riak\Client\Command\MapReduce\BucketKeyMapReduce;
use Riak\Client\Core\Query\Func\AnonymousJsFunction;
use Riak\Client\Core\Query\Func\ErlangFunction;
use Riak\Client\Core\Query\RiakNamespace;
use Riak\Client\Core\Query\RiakLocation;

$reduce = new ErlangFunction('riak_kv_mapreduce', 'reduce_sum');
$map    = new AnonymousJsFunction('function(entry) {
    return [JSON.parse(entry.values[0].data)];
}');

$namespace = new RiakNamespace('bucket_type', 'bucket_name');
$command   = BucketKeyMapReduce::builder([])
    ->withMapPhase($map)
    ->withReducePhase($reduce, null, true)
    ->withLocation(new RiakLocation($namespace, 'key1'))
    ->withLocation(new RiakLocation($namespace, 'key2'))
    ->withLocation(new RiakLocation($namespace, 'key3'))
    ->build();

/* @var $result \Riak\Client\Command\MapReduce\Response\BucketKeyMapReduceResponse */
$result = $this->client->execute($command);
```

```
$values = $result->getResultForPhase(1);

echo $values[0];
// 10
```

IndexMapReduce

Command used to perform a map reduce operation using a secondary index (2i) as input.

Here is the general syntax for setting up a bin secondary index map-reduce:

```
<?php
use Riak\Client\Command\MapReduce\IndexMapReduce;
use Riak\Client\Core\Query\Func\AnonymousJsFunction;
use Riak\Client\Core\Query\Func\ErlangFunction;
use Riak\Client\Core\Query\RiakNamespace;

$reduce = new ErlangFunction('riak_kv_mapreduce', 'reduce_sort');
$map    = new AnonymousJsFunction('function(entry) {
    return [JSON.parse(entry.values[0].data).email];
}');

$namespace = new RiakNamespace('bucket_type', 'bucket_name');
$command   = IndexMapReduce::builder()
    ->withMapPhase($map)
    ->withReducePhase($reduce, null, true)
    ->withNamespace($namespace)
    ->withIndexBin('department_index')
    ->withMatchValue('dev')
    ->build();

/* @var $result \Riak\Client\Command\MapReduce\Response\IndexMapReduceResponse */
$result = $this->client->execute($command);
$values = $result->getResultsFromAllPhases();

echo implode(",", $values);
// fabio.bat.silva@gmail.com,dev@gmail.com,riak@basho.com,...
```

For a int secondary index map-reduce:

```
<?php
use Riak\Client\Command\MapReduce\IndexMapReduce;
use Riak\Client\Core\Query\Func\AnonymousJsFunction;
use Riak\Client\Core\Query\Func\ErlangFunction;
use Riak\Client\Core\Query\RiakNamespace;

$reduce = new ErlangFunction('riak_kv_mapreduce', 'reduce_sort');
$map    = new AnonymousJsFunction('function(entry) {
    return [JSON.parse(entry.values[0].data).email];
}');

$namespace = new RiakNamespace('bucket_type', 'bucket_name');
$command   = IndexMapReduce::builder()
    ->withMapPhase($map)
    ->withReducePhase($reduce, null, true)
    ->withNamespace($namespace)
    ->withIndexInt('year')
    ->withRange(2010, 2015)
```

```
    ->build();

/* @var $result \Riak\Client\Command\MapReduce\Response\IndexMapReduceResponse */
$result = $this->client->execute($command);
```

SearchMapReduce

```
<?php
use Riak\Client\Command\MapReduce\SearchMapReduce;
use Riak\Client\Core\Query\Func\AnonymousJsFunction;
use Riak\Client\Core\Query\Func\ErlangFunction;
use Riak\Client\Core\Query\RiakNamespace;

$reduce = new ErlangFunction('riak_kv_mapreduce', 'reduce_sort');
$map    = new AnonymousJsFunction('function(entry) {
    return [JSON.parse(entry.values[0].data).email];
}');

$namespace = new RiakNamespace('cats_type', 'cats_bucket');
$command   = SearchMapReduce::builder()
    ->withMapPhase($map)
    ->withReducePhase($reduce, null, true)
    ->withQuery('name_s:Snarf')
    ->withIndex('famous')
    ->build();

/* @var $result \Riak\Client\Command\MapReduce\Response\SearchMapReduceResponse */
/* @var $iterator \Iterator*/
$result   = $this->client->execute($command);
$iterator = $result->getIterator();

/** @var $entry \Riak\Client\Command\MapReduce\Response\MapReduceEntry */
foreach ($iterator as $entry) {
    echo $entry->getPhase();
    // 1

    echo $entry->getResponse();
    // ["Snarf"]
}
```

Note: Map-reduce operations are always made using streaming, `Response#getIterator()` will return a stream iterator that can be used to iterate over the response entries.

Notice that is not possible to rewind a stream iterator, If you need to re-use the result use `Response#getResults()` instead.

7.1.8 Merging siblings

Siblings are an important feature in Riak, for this purpose we have the interface `Riak\Client\Resolver\ConflictResolver`, you implement this interface to resolve siblings into a single object.

Below is a simple example that will just merge the content of siblings :

Assuming we have the following domain :

```
<?php
use Riak\Client\Annotation as Riak;

class MyDomainObject
{
    /**
     * @var string
     *
     * @Riak\ContentType
     */
    private $contentType = 'application/json';

    /**
     * @var string
     *
     * @Riak\VClock
     */
    private $vClock;

    /**
     * @var string
     */
    private $value;

    // getters and setters
}
```

By implementing the interface `Riak\Client\Resolver\ConflictResolver` we can merge siblings,

The method `resolve($siblings)` will be called every time we have siblings and invoke `Riak\Client\Command\Kv\Response\FetchValueResponse#getValue('MyDomainObject')` :

```
<?php
use \Riak\Client\Resolver\ConflictResolver;
use \Riak\Client\Core\Query\RiakObject;
use \Riak\Client\Core\Query\RiakList;

class MySimpleResolver implements ConflictResolver
{
    /**
     * {@inheritdoc}
     */
    public function resolve(RiakList $siblings)
    {
        $result  = new MyDomainObject();
        $content = "";

        /** @var $object \MyDomainObject */
        foreach ($siblings as $object) {
            $content .= $object->getValue();
        }

        $result->setValue($content);

        return $result;
    }
}
```

Register your resolver during the application start up :

```
<?php
/** @var $builder \Riak\Client\RiakClientBuilder */
$client = $builder
    ->withConflictResolver('MyDomainObject' new MySimpleResolver())
    ->withNodeUri('http://localhost:8098')
    ->build();
```

Finally whe can fetch the object with siblings and resolve any possible conflict :

```
<?php
$namespace = new RiakNamespace('bucket_type', 'bucket_name');
$location  = new RiakLocation($namespace, 'object_key');
$fetch     = FetchValue::builder($location)
    ->withNotFoundOk(true)
    ->build();

/** @var $domain \MyDomainObject */
$result = $client->execute($fetch);
$domain = $result->getValue('MyDomainObject');

echo $result->getNumberOfValues();
// 2
```

See Siblings for more details on conflict resolution on riak

7.1.9 Performance

This library is faster than most riak clients written in php, It is about 50% percent faster is some cases, mostly because it uses protocol buffer and and iterators every where it is possible.

Fetch/Get Operation

Fetch/Get Operation	Iterations	Average Time	Ops/second
php_riak extension	1,000	0.0005543117523	1,804.03897
riak-client (proto)	1,000	0.0007790112495	1,283.67851
basho/riak	1,000	0.0017048845291	586.54999

Store/Put Operation

Store/Put Operation	Iterations	Average Time	Ops/second
php_riak extension	1,000	0.0010141553879	986.04219
~~riak-client (proto)~~	~~1,000~~	~~0.0013224580288~~	~~756.16767~~
basho/riak	1,000	0.0025912058353	385.92071

For more details and riak clients performance comparison see : https://github.com/FabioBatSilva/riak-clients-performance-comparison