

# Cassandra in Action with Twitter's Ruby Client

Introduction, Bug fix, Refinement, Resources and Insights!

By Youbang Zhang  
[spanzhang@gmail.com](mailto:spanzhang@gmail.com)

2010-5-12



## Table of Contents

Cassandra in Action with Twitter's Ruby Client .....	0
1. Set up the Environment .....	2
<b>1.1 Install a Cassandra Server</b> .....	2
<b>1.2 Start &amp; Stop the Cassandra Server</b> .....	2
<b>1.3 Install Twitter's Ruby Client</b> .....	3
2. Data Model .....	4
<b>2.1 Storage Configuration File</b> .....	4
<b>2.2 ColumnFamily</b> .....	5
<b>2.3 Example Modeling</b> .....	6
3. Ruby Client .....	8
<b>3.1 Connect &amp; Disconnect</b> .....	8
<b>3.2 Write (Create &amp; Update in CRUD)</b> .....	9
<b>3.3 Delete</b> .....	11
<b>3.4 Read</b> .....	12
<b>3.5 Other Properties and Functions</b> .....	14
<b>3.6 Final Enhanced Class of Cassandra</b> .....	15
4. Resources .....	17
5. Postscript .....	17

Okay, let me get something straight. I'm not going to persuade you to use Cassandra. I just assume you have already chosen Cassandra as your storage solution. Maybe you know the basic things, the underneath mechanisms, about Cassandra or maybe not. Anyway, I assume you don't care what makes Cassandra outstanding and you just want to use it. If that's the case, this tutorial is going to help you, in these 3 aspects:

- 1) to set up the environment correctly,
- 2) to understand how to model you data in a Cassandra way,
- 3) to understand and use Twitter's Cassandra Ruby client.

After reading this tutorial, you should have a working environment and can start your project with a good understanding of doing your job in a very Cassandra way. Here we go!

## 1. Set up the Environment

I use **CentOS 5.3** to write this tutorial. If you have other versions of Linux, it could be similar. But you still need to beware of the difference. Following is a simple table summarizing the software environment.

Operating System	CentOS 5.3
Ruby	ruby 1.8.6 (2007-09-24 patchlevel 111) [i686-linux]
Cassandra	0.6.1 (released on 2010-04-18)
Twitter Cassandra Client Gem	0.8.2 (released on 2010-04-13)

The release dates seem a bit strange because the ruby client was released earlier than Cassandra.

### 1.1 Install a Cassandra Server

Follow these steps to install a Cassandra server:

```
wget http://www.apache.org/dist/cassandra/0.6.1/apache-cassandra-0.6.1-bin.tar.gz
tar -zxvf apache-cassandra-0.6.1-bin.tar.gz
```

If everything went smoothly, you have installed the great Cassandra server on your machine!

### 1.2 Start & Stop the Cassandra Server

To start the server, simply do this:

```
cd apache-cassandra-0.6.1
./bin/cassandra
```

You should see something like this on your screen:

```
INFO 22:30:00,846 Auto DiskAccessMode determined to be standard
INFO 22:30:03,254 Saved Token not found. Using 72943366872202653737800780015756283814
INFO 22:30:03,254 Saved ClusterName not found. Using Test Cluster
INFO      22:30:03,773      Creating      new      commitlog      segment
/var/lib/cassandra/commitlog/CommitLog-1273674603773.log
INFO 22:30:04,249 Starting up server gossip
INFO 22:30:04,773 Binding thrift service to localhost/127.0.0.1:9160
```

If that's what you have on your screen, your Cassandra is ready for use now. To learn more about the parameters provided by this startup script, please go [this place](#). After the server started, a folder located at `/var/lib/cassandra` is created. It has two subfolders. One is **commitlog**, which contains log files for system inner usage. Another one is **data**, which contains the actual data.

To stop your Cassandra server, you have two choices. One is to start the server with `-p` argument and then kill the server like this:

```
kill `cat <pidfile>`
```

Another way is a little brute but very simple:

```
pgrep -f cassandra | xargs kill -9
```

This will kill all processes with cassandra in their name or arguments.

## 1.3 Install Twitter's Ruby Client

I assume you already have [Ruby](#) 1.8 or 1.9 installed on your machine. To install the Cassandra Ruby client is very simple, just gem it like this:

```
gem install cassandra
```

The current newest version is 0.8.2. If you don't have an Internet connection on the target machine, you can download the gem package from <http://rubygems.org/downloads/cassandra-0.8.2.gem> and then install it on your target machine manually.

```
gem install cassandra-0.8.2.gem
```

After installation, you can find it under your ruby gem directory. On my machine, it is

```
/usr/local/ruby18/lib/ruby/gems/1.8/gems/cassandra-0.8.2
```

## 2. Data Model

Now you have everything ready. It's time to have a look at how to design your database. This is the most different part from traditional RDBMS. But it's not difficult. Believe me!

Cassandra uses key spaces to organize data. A key space is like a database in conventional RDBMS. We can store infinite **key/value** pairs within one key space. Here, key means index. We always can easily find things via indexes. And one index has only one corresponding value, which can be everything you like, even a set of some other key/value pairs. So, if we use this concept to model data, for example, a person's properties, we can do it like this.

```
Key: a person's identity number  
Value: {name, age, sex, ...}
```

Please note that in the Value set, we use name, age and sex as keys and their corresponding values are the real properties of the person. This is basically how Cassandra models data. It uses a dictionary like method to store and retrieve data. Very fast and neat!

### 2.1 Storage Configuration File

Cassandra server reads a configuration file at startup. This file contains everything Cassandra needs to deal with the storage things, including your database schema. It is by default located under **conf** subfolder named **storage-conf.xml**, where you can design your database structure.

In the storage configuration file, find the **Keyspaces** node and you can see a sample key space

there.

```
<Keyspace Name="Keyspace1">
  <ColumnFamily Name="Standard1" CompareWith="BytesType"/>
  <ColumnFamily Name="Standard2"
    CompareWith="UTF8Type"
    KeysCached="100%"/>
  <ColumnFamily Name="StandardByUUID1" CompareWith="TimeUUIDType" />
  <ColumnFamily Name="Super1"
    ColumnType="Super"
    CompareWith="BytesType"
    CompareSubcolumnsWith="BytesType" />
  <ColumnFamily Name="Super2"
    ColumnType="Super"
    CompareWith="UTF8Type"
    CompareSubcolumnsWith="UTF8Type"
    RowsCached="10000"
    KeysCached="50%"
    Comment="A column family with supercolumns ..."/>
  ...
</Keyspace>
```

The name of the key space is equivalent to a database name in conventional RDBMS. Here we have a key space named **Keyspace1**. In this key space, we have a few **ColumnFamily** nodes. These strange things are equivalent to tables in RDBMS. The name implies a family of columns. But I suggest you to not think of any columns. If you want to think in Cassandra, then treat them as properties. I feel property is a more general concept than column. Column is RDBMS age stuff, somewhat ambiguous in this context.

## 2.2 ColumnFamily

Basically we have two types of ColumnFamily which can be differentiated from the **ColumnType** attribute. The first one is standard ColumnFamily. If you don't specify a value to the ColumnType attribute, it is by default a standard ColumnFamily. Standard ColumnFamily is very similar to our example of the person data model, in which a key maps to a set of key/value pairs. For example,

```
'person_1' => {'name' => 'Youbang Zhang', 'age' => '30'}
```

Here, **'person\_1'** is the key, and **{'name' => 'Youbang Zhang', 'age' => '30'}** is the value. In the

value set, you can put as many key/value pairs as you want. This example is similar to a record in RDBMS. Because a key must be unique in its closure, we cannot have two records using the same key. In RDBMS, we have primary keys and foreign keys. Now you know what I mean! We use keys to index records and attributes.

The other type of ColumnFamily is the so called super ColumnFamily, which means the value set can be something like this:

```
'person_1' =>{'name' =>{'first' =>'Youbang', 'last' =>'Zhang'}, 'age' =>{'lunar' =>'30'}}
```

So, if a ColumnFamily is a super ColumnFamily, a key maps to a set of standard ColumnFamily nodes. But there's no super super ColumnFamily and so fourth. So, the hash maps can only have two layers.

Another important attribute of a ColumnFamily is its CompareWith property, which determines how the records are sorted when store in the database. Basically, we can compare the keys as strings (UTF8Type) or numbers (LongType) or something else. Following is a detailed table.

BytesType	Simple sort by byte value. No validation is performed.
AsciiType	Like BytesType, but validates that the input can be parsed as US-ASCII.
UTF8Type	A string encoded as UTF8
LongType	A 64bit long
LexicalUUIDType	A 128bit UUID, compared lexically (by byte value)
TimeUUIDType	a 128bit version 1 UUID, compared by timestamp

The interesting thing is you don't need to write your key names in your database schema description. Most of the time, you just specify a name for the ColumnFamily and how the records are comparing with each other. That's enough for Cassandra. The actual key names are defined in your code, the client program will decide this at run-time.

## 2.3 Example Modeling

Social Networking Systems are very popular these days. So we take this as an example to illustrate how to model it in a Cassandra way. We just define several basic data structures for demonstrative purpose, such as Users and AddressBooks.

First we make a key space named SNS. Then put some basic nodes in it, such as ReplicaPlacementStrategy, ReplicationFactor and EndPointSnitch. Then we write ColumnFamily nodes to define the structure of our database.

For the Users 'table', we create such a ColumnFamily.

```
<ColumnFamily Name="Users" CompareWith="BytesType" />
```

That's all! A 'table' is defined, with no data fields defined so far. Please remember, the actual data fields will be defined in your client program. And you can imagine, if you have two different client programs accessing this 'table', they can have two different sets of data fields defined in the code. This is allowed in Cassandra. ColumnFamily only knows the name of the the family and how to sort indexes. It doesn't know what you actually stored. And in deed, it doesn't care about this. But in your client program, you may store something like this in this ColumnFamily.

```
'person_1' => {'password' => 'pass1', 'age' => '16'}  
'person_2' => {'password' => 'pass2', 'age' => '16', 'sex' => 'female'}
```

So, we have two users. The identity or key for the first user is 'person\_1'. If you want to find the information about 'person\_1', you just provide this 'person\_1' key to look up from the dictionary. Cassandra maintains the dictionary, you don't worry.

Now, let's move to the AddressBooks. One user has only one address book. One address book contains many contacts. One contact has many properties. That's what we are going to model. In the previous model, we have modeled users. One user has several properties. It's quite simpler than the AddressBook relationship. So, we used a standard ColumnFamily to model it. But for the Addressbook case, we cannot do the old trick with only one ColumnFamily. In fact, two layered ColumnFamilies are needed.

```
<ColumnFamily Name="AddressBooks" CompareWith="BytesType" />  
<ColumnFamily Name="Addresses" CompareWith="BytesType" />
```

In AddressBooks ColumnFamily, we may have this kind of 'record':

```
'person_1' => {'contact_1' => 'address_1'}
```

And in Addresses ColumnFamily, the 'address\_1' maps to a real address 'record'

```
'address_1' => {'contact' => 'Youbang', 'street' => '50 Nanyang Ave', 'city' => 'Singapore'}
```

Obviously, 'address\_1' in AddressBooks is just a dummy key mapping to the real data. We have a better way to model this kind of situation. That is to use super ColumnFamily.

```
<ColumnFamily Name="AddressBooks" CompareWith="BytesType" ColumnType="Super"
CompareSubcolumnWith="UTF8Type" />
```

Then we can have records in this super ColumnFamily in this way:

```
'person_1' => {
  'John' => {'street' => 'Hurong Road', 'city' => 'Changsha'}
  'Kim' => {'street' => '50 Nanyang Ave', 'city' => 'Singapore'}
}
'person_2' => {
  'Kim' => {'street' => 'Naodong West Road', 'city' => 'Shanghai'}
  'Tod' => {'street' => 'Tianhe Street', 'city' => 'Guangzhou'}
}
```

Super ColumnFamily is useful when every entity has a number of private entities.

## 3. Ruby Client

It's time to write the client program to communicate with the Cassandra server to do the real data CRUD thing. If you want to test the codes provided here. You can open a ruby **irb** session and then:

```
irb(main):001:0> require 'rubygems'
=> true
irb(main):002:0> require 'cassandra'
=> true
```

### 3.1 Connect & Disconnect

To connect to a Cassandra server, simply to create a new Cassandra object will do the trick.

```
irb(main):003:0> db = Cassandra.new('Keyspace1', "127.0.0.1:9160", :retries => 2)
=> #<Cassandra:-6038, @keyspace="Keyspace1", @schema={}, @servers=["127.0.0.1:9160"]>
```

Only the first argument is mandatory. The original initialize function is defined as

```
def initialize(keyspace, servers = "127.0.0.1:9160", thrift_client_options = {})
```

To disconnect:

```
db.disconnect!
```

If you do this just after the db object is created, you'll get an error. This is actually a bug. To fix this, simply modify the source code to add a new line to check if @client variable is nil or not.

### 3.2 Write (Create & Update in CRUD)

Because Cassandra is a key/value storage database, we only need to set a value for a specified key to create or update the record. So, there's no difference between Create and Update. If you really want to differentiate them, check the data existence before you do write operations.

```
def insert(column_family, key, hash, options = {})
```

This is the write operation support function. It takes three mandatory arguments and several optional arguments. Valid optional parameters are:

**:timestamp**

The transaction timestamp. Defaults to the current time in milliseconds. This is used for conflict resolution by the server; you normally never need to change it.

**:consistency**

The consistency level of the request. Defaults to `Cassandra::Consistency::ONE` (one node must respond). Other valid options are

`Cassandra::Consistency::ZERO`

`Cassandra::Consistency::QUORUM`

`Cassandra::Consistency::ALL`.

I haven't mentioned these two concepts before because I don't want to confuse you at this moment. If you want to know more about this, see this [architecture overview](#).

Here are some examples of writing things to Cassandra.

```
# create a new record
db.insert(:Standard1, 'person_1', {'password' => 'pass', 'sex' => 'male'})

# update password
db.insert(:Standard1, 'person_1', {'password' => 'pass2'})

# add new property
db.insert(:Standard1, 'person_1', {'age' => '30'})
```

NOTE: For the second statement, it just updates the password property. It does not affect other properties of the 'record'. So, I feel this **insert** is actually a **set** function. If you want to make an alias of the **insert** function, the following code will be of help.

```
require 'rubygems'
require 'cassandra'

class Cassandra
  alias_method :set, :insert
end
```

Furthermore, if you seldom use the last argument of options, you can do it this way. I think it's a good tradeoff.

```
def correct_hash_keys(hash)
  if hash.class == Hash
    ret = {}
    hash.each {|k, v| ret[k.to_s] = correct_hash_keys(v)}
    return ret
  end
  return hash.to_s
end

def set(column_family, key, hash = {})
  insert(column_family.to_sym, key.to_s, correct_hash_keys(hash))
end
```

Here, **correct\_hash\_keys** makes sure the keys and values in a hash must be of class String. If you pass a hash with symbol keys to the insert function, it will fail to convert symbols to strings. So I fix it before pass it to the insert function. With this, you can write data like this:

```
db.set(:Standard1, :person_1, :password => 'pass', :sex => :female, :age => 34)
```

This is a bit condensed, because you don't need to write strings every time. And it eliminates ambiguity when you modify properties.

```
db.set(:Standard1, :person_1, :password => 'pass', :sex => :female)
db.set(:Standard1, :person_1, :password => 'pass2') # set password to a new value
```

Now you see, the second statement is to set a new password for :person\_1. It's really straightforward and no ambiguity anymore. And in this way, we type less! If you need to write a lot of such sort of statements, this will save you much time. All programmers are lazy!

To complete this section, let's have a look at some super ColumnFamily writing examples.

```
db.set(:Standard1, :person_1, :john => {:email => 'j@c.com', :sex => :female})
db.set(:Standard1, :person_1, :tim => {:age => 45}, :tod => {:age => 30})
```

### 3.3 Delete

There are basically three functions to delete data at three different levels. The first one is to remove a specific key/value pair.

```
def remove(column_family, key, *columns_and_options)
```

This function remove things in a certain ColumnFamily. It can remove a sub-key from the given key or remove the entire key. For example:

```
db.remove(:Standard1, 'person_1', 'sex') # removes sub-key 'sex' from 'person_1'
db.remove(:Standard1, 'person_1', 'sex', 'age') # only removes 'sex' if not super CF
db.remove(:Super1, 'person_1', ['john', 'age']) # applies for super CF
```

```
db.remove(:Standard1, 'person_1') # removes the entire key
```

Another two functions are a bit high level. They deal with the entire ColumnFamily and key space.

```
def clear_column_family!(column_family, options = {})
def clear_keyspace!(options = {})
```

I don't think any further explanation is needed here. ☺

### 3.4 Read

There are a bunch of methods to support reading things from Cassandra. First of all, let's look at some valid option parameters for these methods.

Option Name	Description
:count	How many results to return. Defaults to 100.
:start	Column name token at which to start iterating, inclusive. Defaults to nil, which means the first column in the collation order.
:finish	Column name token at which to stop iterating, inclusive. Defaults to nil, which means no boundary.
:reversed	Swap the direction of the collation order.
:consistency	The consistency level of the request. Defaults to Cassandra::Consistency::ONE

Let's start with the very basic function.

```
def get(column_family, key, *columns_and_options)
```

This function returns a hash (actually, a Cassandra::Orderedhash) or a single value representing the element at the column\_family:key:[column]:[sub\_column] path you request. Supports all the options mentioned above. Let's make a wrap to fix the Symbol to String conversion.

```
class Cassandra
  alias_method :old_get, :get
  def get(column_family, key, *columns_and_options)
    cols = []
    columns_and_options.each {|x| cols << (x.class == Hash ? x : x.to_s)}
```

```

    old_get(column_family.to_sym, key.to_s, *cols)
  end
end

```

Then we look at some examples. We write data first and retrieve it afterwards.

```

irb(main):034:0> db.set(:Super1, :person_1, :data => {'1' => 'v1', '2' => 'v2', '3' => 'v3'})
=> nil
irb(main):035:0> db.get(:Super1, :person_1)
=> #<OrderedHash {"data"=>#<OrderedHash {"1"=>"v1", "2"=>"v2", "3"=>"v3"}>>
irb(main):036:0> db.get(:Super1, :person_1, :data)
=> #<OrderedHash {"1"=>"v1", "2"=>"v2", "3"=>"v3"}>
irb(main):037:0> db.get(:Super1, :person_1, :data, '1')
=> "v1"
irb(main):038:0> db.get(:Super1, :person_1, :data, :count => 1)
=> #<OrderedHash {"1"=>"v1"}>
irb(main):039:0> db.get(:Super1, :person_1, :data, :count => 2, :reversed => true)
=> #<OrderedHash {"2"=>"v2", "3"=>"v3"}>

```

Next is `multi_get`, which takes an array of keys.

```

def multi_get(column_family, keys, *columns_and_options)

```

where `keys` is an array, like `['person_1', 'person_2']`. We can still fix the Symbol to String conversion like this:

```

alias_method :old_multi_get, :multi_get
def multi_get(column_family, keys, *columns_and_options)
  options = []
  columns_and_options.each do |x|
    options << (x.class == Array ? Array.new(x.length) {|i| x[i].to_s} : x)
  end
  keys_str = Array.new(keys.length) {|i| keys[i].to_s}
  old_multi_get(column_family.to_sym, keys_str, *options)
end

```

Here's an example of how to use `multi_get`.

```

irb(main):053:0> db.set(:Standard1, :desk1, :prop => 'v1')
=> nil
irb(main):054:0> db.set(:Standard1, :desk2, :prop => 'v2')
=> nil
irb(main):055:0> db.multi_get(:Standard1, [:desk1, :desk2])
=> #<OrderedHash {"desk1"=>#<OrderedHash {"prop"=>"v1"}>, "desk2"=>#<OrderedHash {"prop"=>"v2"}>>

```

### 3.5 Other Properties and Functions

There are some handy properties to check the environment. For your convenience, property names are highlighted in bold.

```

irb(main):030:0> db.keyspace
=> "Keyspace1"
irb(main):031:0> db.servers
=> ["127.0.0.1:9160"]
irb(main):032:0> db.thrift_client_options
=> {:thrift_client_class=>ThriftClient, :transport_wrapper=>Thrift::BufferedTransport}
irb(main):033:0> db.thrift_client_class
=> ThriftClient
irb(main):034:0> db.keyspaces
=> ["Keyspace1", "system", "SNS"]
irb(main):035:0> db.inspect
=> "#<Cassandra:-6988, @keyspace=\"Keyspace1\", @schema={:StandardByUUID1 => nil, :Super1
=> nil, :Super2 => nil, :Standard1 => nil, :Standard2 => nil},
@servers=[\"127.0.0.1:9160\"]>

```

And a few statistical functions are also available for you to check existence of a given key or something like this.

```

# Count the elements at the column_family:key:[super_column] path you
# request. Supports the :consistency option.
def count_columns(column_family, key, *columns_and_options)

# Multi-key version of Cassandra#count_columns. Supports options :count,
# :start, :finish, :reversed, and :consistency.

```

```
# FIXME Not real multi; needs server support
def multi_count_columns(column_family, keys, *options)

# Return a list of single values for the elements at the
# column_family:key:column[s]:[sub_columns] path you request. Supports the
# :consistency option.
def get_columns(column_family, key, *columns_and_options)

# Multi-key version of Cassandra#get_columns. Supports the :consistency
# option.
# FIXME Not real multi; needs to use a Column predicate
def multi_get_columns(column_family, keys, *options)

# Return true if the column_family:key:[column]:[sub_column] path you
# request exists. Supports the :consistency option.
def exists?(column_family, key, *columns_and_options)

# Return a list of keys in the column_family you request. Requires the
# table to be partitioned with OrderPreservingHash. Supports the
# :count, :start, :finish, and :consistency
# options.
def get_range(column_family, options = {})

# Count all rows in the column_family you request. Requires the table
# to be partitioned with OrderPreservingHash. Supports the :start,
# :finish, and :consistency options.
def count_range(column_family, options = {})
```

### 3.6 Final Enhanced Class of Cassandra

Put all the fixes and enhances together, we get this final enhanced class of Cassandra.

```
require 'rubygems'
require 'cassandra'

class Cassandra
  def correct_hash_keys(hash)
    if hash.class == Hash
      ret = {}
      hash.each {|k, v| ret[k.to_s] = correct_hash_keys(v)}
      return ret
    end
  end
end
```

```
end
  return hash.to_s
end

def set(column_family, key, hash = {})
  insert(column_family.to_sym, key.to_s, correct_hash_keys(hash))
end

alias_method :old_get, :get
def get(column_family, key, *columns_and_options)
  cols = []
  columns_and_options.each {|x| cols << (x.class == Hash ? x : x.to_s)}
  old_get(column_family.to_sym, key.to_s, *cols)
end

alias_method :old_multi_get, :multi_get
def multi_get(column_family, keys, *columns_and_options)
  options = []
  columns_and_options.each do |x|
    options << (x.class == Array ? Array.new(x.length) {|i| x[i].to_s} : x)
  end

  keys_str = Array.new(keys.length) {|i| keys[i].to_s}

  old_multi_get(column_family.to_sym, keys_str, *options)
end

alias_method :old_remove, :remove
def remove(column_family, key, *columns_and_options)
  cao = []
  columns_and_options.each do |x|
    if x.class == Array
      cao << (Array.new(x.length) {|i| x[i].to_s})
    elsif x.class == Hash
      cao << x
    else
      cao << x.to_s
    end
  end

  old_remove(column_family.to_sym, key.to_s, *cao)
end

def disconnect!
  return if @client.nil?
end
```

```

@client.disconnect!

@client = nil

end

end

```

## 4. Resources

Here's a table of resources that you can access via Internet.

Resource Name	URL and Description
<b>Important Websites</b>	
Cassandra Official Website	<a href="http://cassandra.apache.org">http://cassandra.apache.org</a> for download, bug-tracking, mailing-lists, etc
Cassandra Wiki	<a href="http://wiki.apache.org/cassandra">http://wiki.apache.org/cassandra</a>
Twitter Cassandra Client	gem <a href="http://rubygems.org/gems/cassandra">http://rubygems.org/gems/cassandra</a>  github <a href="http://github.com/fauna/cassandra">http://github.com/fauna/cassandra</a>
Twissandra	<a href="http://github.com/ericflo/twissandra">http://github.com/ericflo/twissandra</a> Twissandra is an example project, created to learn and demonstrate how to use Cassandra. Running the project will present a website that has similar functionality to Twitter: <a href="http://twissandra.com/">http://twissandra.com/</a>
<b>Useful Blogs</b>	
Up and running with cassandra	<a href="http://blog.evanweaver.com/articles/2009/07/06/up-and-running-with-cassandra/">http://blog.evanweaver.com/articles/2009/07/06/up-and-running-with-cassandra/</a>
Cassandra and Ruby: A Love Affair?	<a href="http://www.engineyard.com/blog/2009/cassandra-and-ruby-a-love-affair/">http://www.engineyard.com/blog/2009/cassandra-and-ruby-a-love-affair/</a>

## 5. Postscript

To use Cassandra to model and store data is a lot of fun. It is flexible. Schemaless gives you the utmost flexibility to design your data storage strategy on the fly. If you like Ruby, you'll definitely like Cassandra.

Any comments or suggestions please send to [spanzhang@gmail.com](mailto:spanzhang@gmail.com). Thanks for spending your precious time reading this article!