Hector Client
for
Apache Cassandra

Riptano

Table of Contents

Document Revision History

Revision Date	Description
10/05/2010	First public release of document for v2
11/23/2010	Updated to reflect auto host discovery and retry, changes to pooling logic. Coincides with Hector 0.7.0-20

Introduction

If you are reading this document, then you are probably interested in connecting to Cassandra using Java. Though a Java-based API to Cassandra is provided with the distribution, it lacks some of the more common enterprise features such as connection pooling, monitoring, etc typically found in common data storage APIs such as those implemented for JDBC.

To a new user, this may seem like an oversight, but it is in fact a design decision made to easily support numerous programming languages with a common wire protocol. Specifically, Cassandra's client API is currently based on Thrift - a service framework with code generation capabilities for numerous languages.

The Java API that ships with Cassandra mentioned above is code auto-generated from a Thrift API definition file and therefore lacks features typically found in other data store client APIs. Hector was created to add these missing features in a manner that would be familiar to Java developers and ideally build on the same foundations where possible.

This document is designed to give the reader an overview of using the Hector API (version 0.7.0-20 at the time of this writing) for managing Cassandra connections. The versions of Cassandra and Thrift used to verify the examples are 0.7.0 and 0.5 respectively.

Though Hector provides a level of abstraction from the Thrift API and it's objects, understanding the structure of Cassandra's Thrift API is essential to correctly using Hector in your projects.
If you have not already, you may want to have a quick read of the Thrift API page on the wiki before continuing this document:
http://wiki.apache.org/cassandra/API07
Suggestions, typos, feedback, additions etc. are welcome and can be sent to:
nate@riptano.com

Features

Pools of pools per server

The basic design of connection pooling logic in Hector is to create a master pool of individual pools of connections based on the number of Cassandra nodes explicitly identified to Hector. For example, if we had three Cassandra nodes configured, HConnectionManager would contain three individual pools of connections – one for each node.

Failover Detection

Detecting and reacting to failed nodes is crucial in any distributed system. The Hector API provides three built-in modes of fail-over specified in the table below by their constant values:

FAIL_FAST	Return the Error to the client without retry
ON_FAIL_TRY_ONE_NEXT_AVAILABLE	Try one more randomly selected server before giving up
ON_FAIL_TRY_ALL_AVAILABLE	Try all known servers before giving up

The fail-over semantics are user configurable per operation. See the section titled Controlling Behavior of Fail-Over for more information.

Basic Load Balancing

Hector provides for plug-able load balancing through the LoadBalancingPolicy interface. Out of the box, two basic implementations are provided: LeastActiveBalancingPolicy (the default) and RoundRobinBalancingPolicy. LeastActiveBalancingPolicy routes requests to the pools with the lowest number of active connections. This ensures a good spread of utilization across the cluster by sending requests to the machine that has the least number of connections. RoundRobinBalancingPolicy implements a simple round-robin distribution algorithm.

If you are interested in load balancing because you are having problems with individual Cassandra instances becoming unresponsive intermittently, this is almost surely a sign of improper Cassandra configuration or, even worse, an inadequate deployment.

Though a full discussion of performance tuning Cassandra is far outside the scope of this document, a few of the most common problems (and some additional references on the subject) are mentioned below in the section on analyzing statistics.

Availability of Metrics

To facilitate smoother operations and better awareness of performance characteristics, Hector exposes both availability counters and, optionally, performance statistics through JMX. These attributes and their interpretations are described in detail in the chapters on JMX and Analyzing Statistics below. Understanding the concepts in these two sections are essential to getting the most out of using Hector to access Cassandra.

Execution time of results are now available directly through the API as well, so developers are free to use these statistics as they would like.

Configuration of Pooling

The behavior of the underlying pools of client connections can be controlled by the ExhaustedPolicy. The following three options are available:

WHEN_EXHAUSTED_FAIL	Fail acquisition when no more clients are available
WHEN_EXHAUSTED_GROW	Grow the pool automatically to react to load
WHEN_EXHAUSTED_BLOCK	Block on acquisition until a client becomes available (the default)

Extensive Logging

Hector uses the slf4j logging API with the Log4J bridge to provide runtime logging. Specifying a level of DEBUG in your log4j configuration for Hector classes will print information on:

- Borrowing and releasing of clients
- Success/fail results of individual operations
- Connection creation and fail-over

Running in DEBUG mode is advised only for development environments.

Encapsulation of Thrift API

As previously mentioned, Thrift is a very useful library, particularly when supporting heterogeneous clients in a distributed system is a design goal. But being generated code, it can be difficult and non-intuitive to work with. Hector now completely encapsulates the Thrift API so developers have to deal only with the Hector client using familiar design patterns. The original API is still available for existing users to transition their current projects as well as for those who are comfortable working with Thrift.

Fully Mavenized

Since the beta release of Cassandra 0.7.0, Riptano has been offering maven repository access for dependecies required for Cassandra usage via Hector. To make use of this repository, add the following repository declaration to your POM file:

```
<repository>
        <id>riptano</id>
        <name>riptano</name>
        <url>http://mvn.riptano.com/content/repositories/public/</url>
</repository>
```

And add the following dependency (modifying version as necessary):

```
<dependency>
        <groupId>me.prettyprint</groupId>
        <artifactId>hector</artifactId>
        <version>0.7.0-19</version>
</dependency>
```

Getting Started

Setting Up Your Local Environment

To use Hector in your code, the following runtime dependencies are required:

Library	**Description**	**Version**
apache-cassandra	Cassandra jar	0.7.0
libthrift	Thrift API	0.5
log4j	Logging Implementation	1.2.14
slf4j-api	Logging facade API	1.5.8
slf4j-log4j	Bridge for using slf4j with log4j	1.5.8
perf4j	Performance metric gathering API	0.9.2
commons-codec	General encoding/decoding algorithms	1.4
google-collections	Additional collections APIs	1.0

If you wish to do integration testing (further described below), the following dependencies will probably be necessary as well:

Library	**Description**	**Version**
high-scale-lib	Utility classes with excellent concurrency	1.0.0
clhm-production	Concurrent linked hashmap implementation	1.0.0
cassandra-javautil	Utility for cleanup after testing	0.7.0
commons-lang	Utility classes from Apache-commons	2.4
spring-beans	Component for SpringFramework	3.0.0.RELEASE
spring-test	Testing integration for SpringFramework	3.0.0.RELEASE
spring-context	Context wireing classes for SpringFramework	3.0.0.RELEASE

Testing

Integration

When you want to test actual functionality of Hector, the EmbeddedServerHelper class can be used to spin up an in-memory Cassandra instance against which you can test your queries. This class will launch a Cassandra instance in a new thread loading the necessary configurations (log4j.xml and storage-conf.xml) from the class path. An additional abstract BaseEmbeddedServerTest class uses JUnit annotations to control startup and tear-down and can be easily extended for testing.

Once you have EmbeddedServerHelper available to your integration tests, experimenting with configuration and API functionality becomes much simpler since the data is not persisted after the testing process exists.

Basic Configuration and Setup

The main entry point for accessing a Cassandra instance through Hector is the Keyspace class. All of the implementations in the api package rely on a Keyspace for interfacing with Cassandra. An instance of Keyspace can be retrieved using the default configurations with very little code:

```
Cluster cluster = HFactory.getOrCreateCluster("TestCluster",
   new CassandraHostConfigurator("localhost:9160"));
Keyspace keyspace = HFactory.createKeyspace("Keyspace1", cluster)
```

The underlying HConnectionManager as well as the host-specific pool for *localhost:9160* will both be created automatically if they do not already exist.
Once a handle to the Keyspace has been obtained, you will probably spend most of your time using HFactory to construct the query appropriate to your use case. Unlike previous versions of the Hector client, the classes returned from HFactory all support method chaining which keeps code compact and more legible.

The Cluster implementation returned from HFactory is designed as a thread-safe class and can be a long-lived object. However, instances of Cluster are held be HFactory internally once created, so using the getOrCreateCluster method repeatedly as outlined above is harmless.

A point to note for users of earlier versions of Hector, is that the meaning of Keyspace has changed significantly. The previous interface and class (Keyspace and KeyspaceImpl respectively) are now embodied in KeyspaceService for a more consistent naming scheme. KeyspaceService is not thread-safe and should be treated as a short-lived, lightweight object.

Finer Grained Configuration

Once you have a good idea of the operating characteristics of your Cassandra cluster, or you just want more control over the connection and pooling settings, you should explore the additional attributes available to CassandraHostConfigurator for configuring your client settings.

The table below shows the settings available for configuration and their default values.

Setting	**Description**	**Default**
cassandraThriftSocketTimeout	The amount of time to wait on the Thrift socket for a response from the server	5000ms
exhaustedPolicy	defined above	WHEN_EXHAUSTED_GROW
maxActive	The maximum number of active clients to allow. The behaviour of hitting this threshold is controlled by exhaustedPolicy	50
maxIdle	The number of unused clients to keep	-1 (keep all clients)
maxWaitTimeWhenExhausted	The maximum amount of time to wait if there are no clients available	-1 (wait indifinetly)
useThriftFramedTransport	Turn on the Thrift's framed transport (if true, the corresponding setting MUST be set in Cassandra's storage.conf)	FALSE
lifo	Use “Last in, first out” pool retrieval policy	TRUE
cassandraThriftSocketTimeout	The socket timeout to set on the underlying Thrift transport	0 (don’t timeout)
retryDownedHosts	Automatically retry hosts that have been marked down in a background thread	TRUE
retryDownedHostsDelayInSeconds	The number of seconds to wait between retry attempts	30
autoDiscoverHosts	True to automatically discover all the hosts on the ring at startup and at intervals thereafter	false
autoDiscoverDelayInSeconds	Number of seconds to wait between checks for new hosts	30 (disabled by default)

The following example shows a CassandraHostConfigurator being built that will create a single pool to *localhost* with several of the connection properties changed from their default values:

```
CassandraHostConfigurator cassandraHostConfigurator =
     new CassandraHostConfigurator("localhost:9160");
cassandraHostConfigurator.setMaxActive(20);
cassandraHostConfigurator.setMaxIdle(5);
cassandraHostConfigurator.setCassandraThriftSocketTimeout(3000);
cassandraHostConfigurator.setMaxWaitTimeWhenExhausted(4000);
```

Using Node Auto Discovery

The NodeAutoDiscoveryService can be a convenient way to automatically apply the same configuration to all the hosts on the ring. The underlying API calls used to determine this information do have some limitations that will cause this feature to not work correctly, or in some cases, at all for certain hardware and network configurations. If you use NodeAutoDiscoveryService, the Thrift and Gossip traffic must be run on the same interfaces. If the Gossip traffic is on a dedicated inteface, this feature will not work, as the call underlying the API methods in Cassandra utilize the ReplicatioStrategy implementation for discovery. Therefore the list of IP Adresses returned will not accept Thrift traffic and will appear as dead hosts.

With this limitation in mind, NodeAutoDiscoveryService can be enabled easily via CassandraHostConfigurator (it is disabled by default due to the limitations mentioned above). To enable this service, the autoDiscoverHosts and autoDiscoverDelayInSeconds should be set to true and the number of seconds (30 by default) to wait between checks respectively.

Automatic Failed-Host Retry

CassandraHostRetryService is another new high-availability feature that will hold the hosts which have been detected as down and retry them at a configured interval. It is on by default with a 30 seconds interval. Use the retryDownedHostDelayInSeconds on CassandraHostConfigurator to customize this interval.

Configuration and Setup with Dependency Injection

For users of Inversion of Control (IoC) containers such as SpringFramework (which we will use for the purposes of these examples), Hector's functionality can be easily configured and provided for injection. The following bean definitions create all the necessary collaborators for connection to a Cassandra server.

```
<bean id="cassandraHostConfigurator"
   class="me.prettyprint.cassandra.service.CassandraHostConfigurator">
   <constructor-arg value="localhost:9170"/>
</bean>

<bean id="cluster" class="me.prettyprint.cassandra.service.ThriftCluster">
   <constructor-arg value="TestCluster"/>
   <constructor-arg ref="cassandraHostConfigurator"/>
</bean>

<bean id="keyspace" class="me.prettyprint.hector.api.factory.HFactory"
   factory-method="createKeyspace">
   <constructor-arg value="Keyspace1"/>
   <constructor-arg ref="cluster"/>
</bean>
```

To use these connections as collaborators in your own Spring-managed beans, Keyspace can be injected into HFactory and used to create Query and Mutator instances by using the via the provided static methods.

Similarly, here is an example of the SimpleCassandraDao class included with the Hector API being injected with the Keyspace object, keyspace name and column family name:

```
<bean id="simpleCassandraDao"
   class="me.prettyprint.cassandra.dao.SimpleCassandraDao">
   <property name="keyspace" ref="keyspace"/>
   <property name="columnFamilyName" value="Standard1"/>
</bean>
```

API Overview

The Hector client underwent a number of major changes recently reflecting feedback from the community. The results of these changes are a cleaner API that is more intuitive and completely encapsulates Thrift. The original API remains largely unchanged (including direct access to Thrift should you want it). There has been some refactoring to enforce the separation of operations against a Keyspace versus a cluster as well as some cleanup of internals with regards to fail-over, but these changes are minimal and were done against an extensive set of pre-existing unit tests.

Common API Usage Examples

Insert a Single Row with a Single Column

The following example inserts a Column with the column name "first" and the column value of "John" under the key "jsmith". The retrieving and instance of the Mutator class from HFactory makes this action straight-forward.

```
Mutator<String> mutator = HFactory.createMutator(keyspace, stringSerializer);
```

```
mutator.insert("jsmith", "Standard1", HFactory.createStringColumn("first", "John"));
```

Retrieve a Single Column

Retrieving a column from Cassandra is a straight forward operation. Using the ColumnQuery object to retrieve the information we inserted in the example above, we would have:

```
ColumnQuery<String, String, String> columnQuery =
   HFactory.createStringColumnQuery(keyspace);

columnQuery.setColumnFamily("Standard1").setKey("jsmith").setName("first");
Result<HColumn<String, String>> result = columnQuery.execute();
```

Note the strong typing provided by the ColumnQuery and Result objects.

Removing a Single Column

To remove the column used in the previous two examples, an instance of Mutator can be obtained through HFactor or reused.

```
mutator.delete(“jsmith”, “Standard1”, “first”, stringSerializer);
```

Inserting Multiple Columns with Mutator

The Mutator class also allows the insertion of multiple columns at one time (even for multiple keys). The following example inserts three columns: first, last and middle under the same key:

```
mutator.addInsertion(”jsmith,” "Standard1",
   HFactory.createStringColumn(“first”, “John”))
   .addInsertion(“jsmith”, “Standard1”, HFactory.createStringColumn(“last”, “Smith”))
   .addInsertion(“jsmith”, “Standard1”, HFactory.createStringColumn(“middle”, “Q”));
mutator.execute();
```

Retrieving Multiple Columns with SliceRange

To retrieve the three columns inserted in the previous example in a single query, use HFactory to construct a SliceQuery for those columns:

```
// given Keyspace ko and StringSerializer se
SliceQuery<String, String> q = HFactory.createSliceQuery(ko, se, se, se);
q.setColumnFamily(cf)
.setKey("jsmith")
.setColumnNames("first", "last", "middle");
Result<ColumnSlice<String, String>> r = q.execute();
```

Deleting Multiple Columns with Mutator

To delete the three columns from the two examples above, we can again use (reuse even) the Mutator class. The following example deletes all columns for the key “jsmith”:

```
mutator.delete(“jsmith”, "Standard1", null, stringSerializer);
```

The null passed as the column name above triggers the delete for all the columns. If only the “middle” column was to be deleted, that would have been passed in place of the null. If I wanted to delete only the middle and last columns of this example, I could call the example above twice using for each of the two columns in place of null, or I could construct the Mutator in a more detailed manner to efficiently batch the operations into the underlying batch_mutate API call:

```
mutator.addDeletion(“jsmith”, "Standard1", “middle”, stringSerializer)
.addDeletion(“jsmith”, "Standard1", “last”, stringSerializer)
.execute();
```

Getting Multiple Rows with MultigetSliceQuery

To retrieve multiple rows of data for a given set of keys, *MultigetSliceQuery* provides the most efficient query. The following example inserts 10 columns, keeping track of their keys in a list and then explicitly selects 10 rows of information from Cassandra using that list:

```
MultigetSliceQuery<String, String, String> multigetSliceQuery =
   HFactory.createMultigetSliceQuery(keyspace, stringSerializer, stringSerializer, stringSerializer);
multigetSliceQuery.setColumnFamily("Standard1");
multigetSliceQuery.setKeys("fake_key_0", "fake_key_1",
   "fake_key_2", "fake_key_3", "fake_key_4");
multigetSliceQuery.setRange("", "", false, 3);
Result<Rows<String, String, String>> result = multigetSliceQuery.execute();
```

Getting Multiple Rows with RangeSlicesQuery

The RangeSlicesQuery method functions similarly to MultigetSliceQuery. The difference is that RangeSlicesQuery uses a contiguous range of keys as opposed to the specific set of keys used by MultigetSliceQuery. If the OrderPreservingPartitioner (OPP) were configured, the same query from the MultigetSliceQuery example above could be done with the RangeSlicesQuery:

```
RangeSlicesQuery<String, String, String> rangeSlicesQuery =
   HFactory.createRangeSlicesQuery(keyspace, stringSerializer, stringSerializer, stringSerializer);
rangeSlicesQuery.setColumnFamily("Standard1");
rangeSlicesQuery.setKeys("fake_key_0", "fake_key_4");
rangeSlicesQuery.setRange("", "", false, 3);
Result<OrderedRows<String, String, String>> result = rangeSlicesQuery.execute();
```

If you did not want to explicitly set the column names, or as above, get the first three columns in order, but instead wanted to return all columns starting with a common prefix, you could change the RangeSlicesQuery from the example above as follows:

```
rangeSlicesQuery.setRange("fake_column_", "", false, 3);
```

If you were using OPP the endKey could be provided as well to further limit the results.

It is important to note that unless you are using OPP in your storage configuration, the results returned from a RangeSlicesQuery will not be in any key order. Therefore using a RangeSlicesQuery with partitioners other than OPP is really only useful when you want to iterate over the entire keyspace. See the Cassandra API documentation for further discussion on partitioner selection and the resulting effects on queries.

Using Secondary Column Indexes

Secondary column indexes are a new feature to Cassandra 0.7. Though similar to a RangeSlicesQuery and it's underlying get_range_slices API call, IndexSlicesQuery requires some configuration in the ColumnFamily. The following example creates a ColumnFamily called *Users* with an index on the *birthdate* column name.

```
- name: Users
  default_validation_class: LongType
  column_metadata:
    - name: birthdate
      validator_class: LongType
      index_type: KEYS
```

The following IndexedSlicesQuery uses this ColumnFamily to return a slice of all the rows containing the *name* and *birthdate* columns where *birthyear* has a value of 1975.

```
IndexedSlicesQuery<String, String, Long> indexedSlicesQuery =
    HFactory.createIndexedSlicesQuery(keyspace, se, se,, LongSerializer.get());
indexedSlicesQuery.addEqualsExpression("birthyear", 1975L);
indexedSlicesQuery.setColumnNames("birthdate","firstname");
indexedSlicesQuery.setColumnFamily("Users");
indexedSlicesQuery.setStartKey("");

QueryResult<OrderedRows<String, String, Long>> result =
    indexedSlicesQuery.execute();
```

Rows returned by the query will be ordered in partitioner (token) order.

IndexedSlicesQuery can also have expressions applied on other columns in conjunction with the *addEqualsExpression*. There are two important points to note in using additional index expressions:

- At least one equals expression against an indexed column must always be present via addEqualsExpression
- The columns for additional clauses do **not** have have to be configured as indexed for the

ColumnFamily

Taking these points into account, extending the above example to limit the query to the months of April to June (4 and 6 respectively) would be:

```
indexedSlicesQuery.addGteExpression(“birthmonth”,4L);
indexedSlicesQuery.addLteExpression(“birthmonth”,6L);
```

IndexedSlicesQuery also has the methods addGtExpression and addLtExpression for exclusive *greater than* and *less than* clauses.

Getting Information from the Result

A new feature in the Hector API is the exposure of connection and query information on the ExecutionResult object. All results have the following two properties exposed:

- execution time in micro seconds
- the CassandraHost object from which the result came

When dealing with Query results, the Query implementation used is also available. The following example shows some of the information exposed from ExecutionResult when querying via a ColumnQuery:

```
ColumnQuery<String, String, String> columnQuery =
   HFactory.createStringColumnQuery(keyspace);
columnQuery.setColumnFamily("Standard1")
   .setKey("jsmith").setName("first");

Result<HColumn<String, String>> colResult = columnQuery.execute();
System.out.println("Execution time: " + colResult.getExecutionTimeMicro());
System.out.println("CassandraHost used: " + colResult.getHostUsed());
System.out.println("Query Execute: " + colResult.getQuery());
```

Execution of the above code produces the following output:

```
Execution time: 24828224
CassandraHost used: localhost(127.0.0.1):9160
Query Execute: AbstractColumnQuery(jsmith,first)
```

Dealing with Tombstones

Tombstones present some unusual issues to developers who are used to working with results retrieved from a database. Although these issues tend to add some burden to the client code, they are quite necessary for coordinating the deletion of data in a distributed system. The Cassandra wiki has some additional information on this subject if you are curious: http://wiki.apache.org/cassandra/DistributedDeletes
http://wiki.apache.org/cassandra/FAQ#range_ghosts

To put this in the context of an example, say we have just created 10 rows of data with three columns each. If half the columns are later deleted, and a compaction has not yet occurred, these columns will show up in *get_range_slices* queries as empty. Using *RangeSlicesQuery* as described in the previous section, we would have 10 results returned, but only five of them will have values. More importantly, calls to *get* (via ColumnQuery) by design assume the Column you are retrieving exists in the store. Therefore if you call *get* on tombstoned data, null is returned (note: this is different than previous versions of Hector where the underlying NotFoundException was propogated up the stack).

Common API Usage Examples Using SuperColumns

One of the more cumbersome portions of dealing directly with Cassandra's Thrift API, is the prevalence of the ColumnOrSuperColumn (CoSC) object. Though it provides for some convenience, it can lead to a lot of boiler-plate code. Previous versions of Hector encapsulated this functionality into separate method calls. The current version follows on that design with different implementations of the Query interface for SuperColumn queries.

Inserting a Column on a SuperColumn

Inserting a Column on a SuperColumn with Mutator is quite similar to what we have already seen. The only difference is we are now creating an HSuperColumn in place of HColumn to provide some additional structure. For example, if we wanted to store our users under the "billing" department, we would use the following call to Mutator:

```
Mutator<String> mutator =
   HFactory.createMutator(keyspace, stringSerializer);
mutator.insert("billing", "Super1", HFactory.createSuperColumn("jsmith",
   Arrays.asList(HFactory.createStringColumn("first", "John")),
   stringSerializer, stringSerializer, stringSerializer));
```

As for retrieval of a SuperColumn, the simple case is almost identical to retrieval of a standard Column. The only difference is the Query implementation used.

```
SuperColumnQuery<String, String, String, String> superColumnQuery =
                HFactory.createSuperColumnQuery(keyspace, stringSerializer,
            stringSerializer, stringSerializer, stringSerializer);
superColumnQuery.setColumnFamily("Super1")
   .setKey("billing").setSuperName("jsmith");
Result<HSuperColumn<String, String, String>> result = superColumnQuery.execute();
```

Advanced Use Cases

Controlling Iteration a.k.a “Paging”

Iterating over results from a RDBMS is a use case with which most Java developers should be familiar. If you have a result set with several thousand rows, it is most common to break up the results in smaller chunks or pages. To do this in Cassandra, some additional work is required on the part of the developer.
As an example say we wanted to page over the first 100 or so rows of the *Standard1* ColumnFamily in groups of 10. A portion of this code is shown here:

```
RangeSlicesQuery<String, String, String> rangeSlicesQuery =
HFactory.createRangeSlicesQuery(keyspace, stringSerializer, stringSerializer,
stringSerializer);
rangeSlicesQuery.setColumnFamily("Standard1");
rangeSlicesQuery.setKeys("", "");
rangeSlicesQuery.setRange("", "", false, 3);

rangeSlicesQuery.setRowCount(11);
Result<OrderedRows<String, String, String>> result = rangeSlicesQuery.execute();
OrderedRows<String, String, String> orderedRows = result.get();
Row<String,String,String> lastRow = orderedRows.peekLast();
```

To get the first 10 results and prepare for retrieving the next 10, we actually have to ask for 11. The reason for this is that because the count attribute of SliceRange (Cassandra's concept of a SQL LIMIT) only applies to number of results being returned, not their position in the result set. We need the 11th item to set the startKey on the RangeSlicesQuery for the next call as the startKey to endKey range is inclusive. Note also that we remove the 11th item from the "results" map as it will be shown first in the next set of results.

Now when rangeSlicesQuery is called with the key value of the last Row provided as the startKey, the results map will start on the 11th result, continuing where we left off above:

```
rangeSlicesQuery.setKeys(lastRow.getKey(), "");
```

Retrieving Keys Without Deserializing Columns

Occasionally, you may find yourself in a situation where you need to retrieve a large number of keys and not really need or care about the columns to which they are associated. The problem here is that most of the underlying API calls actually deserialize the column data whether it is needed or not. Depending on the size of your ColumnFamily, this can be extremely inefficient. Using *RangeSlicesQuery* as described below to retrieve keys, avoids this deserialization overhead:

```
RangeSlicesQuery<String, String, String> rangeSlicesQuery =
   HFactory.createRangeSlicesQuery(keyspace, stringSerializer, stringSerializer,
stringSerializer);
rangeSlicesQuery.setColumnFamily("Standard1");
rangeSlicesQuery.setKeys("fake_key_", "");
rangeSlicesQuery.setReturnKeysOnly();

rangeSlicesQuery.setRowCount(5);
```

```
Result<OrderedRows<String, String, String>> result =  rangeSlicesQuery.execute();
```

Like the previous *RangeSliceQuery* examples, you can limit the request to a subset of keys by using a prefix for the start key if needed.

Controlling Behaviour of Fail-Over

In any distributed system component failure should be anticipated and even expected. To facilitate this, Hector provides for control of fail-over semantics in each operation with the FailoverPolicy class. This class exposes two important properties to the user: the number of times a request will be repeated if an unavailable exception was received and the amount of time to pause between these operations. By default, queries created through HFactory all use the built-in policy ON_FAIL_TRY_ALL_AVAILABLE which continues to retry the operation with no pauses until it succeeds.

To create your own approach to fail-over, for example to try the next three hosts pausing 200 milliseconds between each, you would construct the following FailoverPolicy:

```
FailoverPolicy failoverPolicy = new FailoverPolicy(3, 200);
```

Aquiring a Keyspace with this FailoverPolicy is then a matter of just using the overloaded version of createKeyspace in HFactory which takes the FailoverPolicy as one of the arguments.

Gathering Statistics

JMX Attributes Available to Hector

The fact that you are reading this guide probably means that you new to Cassandra or at least new to connecting via a Java application. Like any new system being considered, particularly one that embodies as many conceptual changes to the day-to-day of most Java programmers, it is really important that you understand the performance characteristics of your application and your cluster as much as possible.

Recognizing this need, Hector has a number of attributes exposed via JMX. These attributes fall into two categories: status and counters on the connections and pools, and aggregate performance statistics. Knowing what these attributes mean and spending some time exploring them as you develop will make base lining, monitoring and tunning your applications significantly easier.

To get the most out of the available output, you should use a remote monitoring tool that supports JMX queries and has the ability to capture and store statistics over time.

Health Check Attributes Available for Hector

RecoverableTimedOutCount	Number of recoverable TimedOut exceptions. Those exceptions may happen when certain nodes are under heavy load that they can't provide the service
RecoverableUnavailableCount	Number of recoverable Unavailable exceptions
RecoverableTransportExceptionCount	Number of recoverable Transport exceptions
SkipHostSuccess	Number of times that a successful skip-host (fail-over) has occurred.
WriteFail	Number of failed write operations
ReadFail	Number of failed read operations
RecoverableLoadBalanceConnectErrors	Number of recoverable load-balance connection errors.
NumConnectionErrors	Number of connection errors (initial connection to the ring for retrieving metadata)
ExhaustedPoolNames	The list of exhausted connection pools.
KnownHosts	the list of known hosts in the ring. This list will be used by the client in case fail-over is required.
NumActive	number of currently active connections (all pools)
NumBlockedThreads	Number of currently blocked threads.
NumExhaustedPools	Number of currently exhausted connection pools.
NumIdleConnections	Number of currently idle connections (all pools)
NumPoolExhaustedEventCount	Number of times threads have encountered the pool-exhausted state (and were blocked)
NumPools	Number of connections pools. This is also the number of unique hosts in the ring that this client has communicated with. The number may be one or more, depending on the load balance policy and fail-over attempts.

PoolNames	The list of known pools

Optional Performance Counters

The following counter tags are available via Perf4j's JmxAttributeStatisticsAppender class. The tags available via Perf4j are listed below:

READ.success_	Read operations that were successful
WRITE.success_	Write operations that were successful
READ.fail_	Failed read operations
WRITE.fail_	Failed write operations
META_READ.success_	Successful reads against the meta API
META_READ.fail_	Failed reads against the meta API

Each of the above tags expose six attributes that are by default gathered in 10 seconds intervals. These attributes are intended to be used to gather performance-over-time statistics and, due to their potential to impact on performance, must be enabled and disabled explicitly via *exposeTag* and *removeTag* operations respectively.

Mean	The statistical mean of the operation
StdDev	The standard deviation of the operation
Min	The minimum amount of time an operation took in this interval
Max	The maxiumum amount of time an operation took in this interval
Count	The total count of operations performed
TPS	Transactions Per Second for this operations

To translate this all into JMX parlance, invoking the *exposeTag* method with *READ.success_* will enable performance statistics gathering for successful read operations, via the six attributes listed above. Specifically, if we wanted the count of successful read operations for a given 10 second interval, the attribute *READ.success_Count* would need to be invoked.

If you do not want the overhead of having Pef4j running, it is possible to disable the statistics gathering entirely by setting the system property *com.prettyprint.cassandra.load_hector_log4j* to false at system startup and setting the level in your logging configuretion to WARN for the appender named: *me.prettyprint.hector.TimingLogger*

Monitoring and Diagnosis

The most immediately useful attributes in monitoring the health of a Hector-based application are those dealing with pool exhaustion and operation failures. Particularly, should you observe increases in *NumPoolExhaustedEventCount*, *ReadFail*, *WriteFail*, this indicative of one of two things:

- You are not correctly releasing connections back to the pool in your application code
- One or more of your Cassandra instances are under duress (typically this is accompanied with *RecoverableTimedOutCount* increases as well)

In either of the above cases, you should also check and/or there are entries in *ExhaustedPoolNames* as this attribute will provide the listing of Cassandra server instances which currently maximized their connections. If the same host repeatedly shows up on this list, there may be an issue with the hardware or this host may have a disproportionate allotment of the token ring.

As for performance statistics, the *READ.success_* attributes provides very useful information about cluster performance trending as your data set grows. A common monitoring setup that shows this information involves two different graphs: one for monitoring the TPS attribute, another for correlating *Min*, *Max* and *Mean*. Though there is a performance overhead in gathering these statistics, it is a small price to pay if you are not sure about your capacity plan or you want more insight into your cluster's behavior.

Regardless of which of Hector's JMX attributes you choose to monitor and analyze, correlating this data with the JMX output from individual Cassandra instances is essential if you want to get an accurate picture of cluster behavior as a system. Unfortunately, a discussion of Cassandra JMX attributes is outside the scope of this document. See the Operations and JMX attributes pages of the Cassandra wiki for more information on monitoring:
http://wiki.apache.org/cassandra/Operations
http://wiki.apache.org/cassandra/JmxInterface

Appendix

Maven Repository for Dependencies

In previous versions of Hector, dependencies that were not themselves mavenized had to be included explicitly with the maven-install plugin. Riptano has offered to host the maven repository for Cassandra and related non-mavenized dependencies. See “Fully Mavenized” above for how to include Hector and the related dependencies in your own projects.

Finding Help

The Hector github Site

http://github.com/rantav/hector
Hector has a number of test cases that could be helpful examples of how to use the API.

Mailing List

Address: hector-users@googlegroups.com
Archives page: http://groups.google.com/group/hector-users

Obtaining Complete Code Samples

The code used in the example sections of this document can be found on github:
http://github.com/zznate/hector-examples
Each example is independent and can be executed from the command line by using maven.
See the README file on the project page for details.

Stress Testing

The following project can be used to stress test your cluster through the Hector API. This is an excellent way to quickly gather feedback on failure scenarios as they impact your cluster.
http://github.com/zznate/cassandra-stress