Package ‘RCassandra’

February 19, 2015

Version 0.1-3

Title R/Cassandra interface

Author Simon Urbanek <`simon.urbanek@r-project.org`>

Maintainer Simon Urbanek <`simon.urbanek@r-project.org`>

Description This packages provides a direct interface (without the use of Java) to the most basic functionality of Apache Cassanda such as login, updates and queries.

License GPL-2

URL `http://www.rforge.net/RCassandra`

NeedsCompilation yes

Repository CRAN

Date/Publication 2013-12-03 22:24:59

R topics documented:

`RCassandra-package` *R/Cassandra interface*

Description

This packages provides a direct interface (without the use of Java) to the most basic functionality of Apache Cassanda ditributed NoSQL database such as login, updates and queries. The focus is on effciency and speed.

Details

RC.connect is used to connect to a Cassandra instance. The obtained handle is then used for all operations until RC.close is used to close the connection.

A set of RC.get functions can be used to query the database. Specialized high-level interface for fixed-column tables (not the most common in Cassandra, thgouh) is also available with RC.read.table.

Updates and inserts can be performed either individually using the RC.insert function or batch-mutations using RC.mutate.

Auxiliary functions retrieving meta-information from the database are described on the RC.version help page.

Currently, communication to Cassanra is performed directly on a blocking TCP/IP socket. This implies that transactions currently cannot be interrupted on the R side and there is no timeout. This may change in future versions. The code does *not* use R connections to avoid extra overhead.

Package:	RCassandra
License:	GPL-2
URL:	http://www.rforge.net/RCassandra

Index:

```
RC.cluster.name         Functions retrieving meta-information from a
                        Cassandra connection
RC.connect              Connect, login, close connection to Cassandra
RC.get                  Functions for querying Cassandra database
RC.insert               Update function to insert data into Cassandra
RC.read.table           Read and write tables into column families in
                        Cassandra
```

Author(s)

Simon Urbanek <simon.urbanek@r-project.org>

RC.cluster.name *Functions retrieving meta-information from a Cassandra connection*

Description

RC.cluster.name returns the name of the cluster.

RC.version returns the protocol version.

RC.describe.keyspace returns a keyspace definition.

RC.describe.keyspaces returns a list of definitions for all keyspaces.

Usage

```
RC.cluster.name(conn)
RC.version(conn)
RC.describe.keyspaces(conn)
RC.describe.keyspace(conn, keyspace)
```

Arguments

`conn`	connection handle are returned by `RC.connect`
`keyspace`	string, name of the keyspace to describe

Value

For `RC.cluster.name` and `RC.version` a string.

For `RC.describe.keyspace` a structure describing the keyspace - see `KsDef` structure in Cassandra.

For `RC.describe.keyspaces` a list of the `KsDef` strctures.

Author(s)

Simon Urbanek

See Also

`RC.connect`, `RC.get`

`RC.connect` *Connect, login, close connection to Cassandra*

Description

`RC.connect` connects a to a host running Cassandra. All subsequent operations are performed on the handle returned by this function.

`RC.close` closes a Cassandra connection.

`RC.login` perform an authentication request.

Usage

```
RC.connect(host = NULL, port = 9160L)
RC.close(conn)
RC.login(conn, username = "default", password = "")
```

Arguments

`host`	host name or IP address to connect to using TCP/IP
`port`	port to connect to on the above host
`conn`	connection handle as returned by `RC.connect`
`username`	username for the authentication dictionary
`password`	password for the authentication dictionary

Details

`RC.connect` return an opaque connection handle that has to be used for all subsequent calls on the same connection. `RCassandra` uses low-level system calls to communicate with Cassandra, this handle is not an R connection.

`RC.close` closes an existing Cassandra connection.

`RC.login` sends an authenticataion request with the given credentials. How this is processed depend on the authentication module use in the Cassandra instance this connection connects to.

Value

`RC.connect` returns a Cassandra connection handle.

`RC.close` return `NULL`

`RC.login` returns `conn`.

Author(s)

Simon Urbanek

Examples

```
## Not run:
c <- RC.connect("cassandra-host")
RC.login(c, "foo", "bar")
RC.cluster.name(c)
RC.describe.keyspaces(c)
RC.close(c)

## End(Not run)
```

RC.get *Functions for querying Cassandra database*

Description

RC.use selects the keyspace (aka database) to use for all subsequent operations. All functions described below require keyspace to be set using this function.

RC.get queries one key and a fixed list of columns

RC.get.range queries one key and multiple columns

RC.mget.range queries multiple keys and multiple columns

RC.get.range.slices queries a range of keys (or tokens) and a range of columns

RC.consistency sets the desired consistency level for all query operations

Usage

```
RC.use(conn, keyspace, cache.def = TRUE)
RC.get(conn, c.family, key, c.names,
       comparator = NULL, validator = NULL)
RC.get.range(conn, c.family, key, first = "", last = "",
             reverse = FALSE, limit = 1e+07,
             comparator = NULL, validator = NULL)
RC.mget.range(conn, c.family, keys, first = "", last = "",
             reverse = FALSE, limit = 1e+07,
             comparator = NULL, validator = NULL)
RC.get.range.slices(conn, c.family, k.start = "", k.end = "",
                    first = "", last = "", reverse = FALSE,
                    limit = 1e+07, k.limit = 1e+07,
                    tokens = FALSE, fixed = FALSE,
                    comparator = NULL, validator = NULL)
RC.consistency(conn, level = c("one", "quorum", "local.quorum",
               "each.quorum", "all", "any", "two", "three"))
```

Arguments

conn	connection handle as returned by RC.connect
keyspace	name of the keyspace to use
cache.def	if TRUE then in addition to setting the keyspace a query on the keyspace definition is sent and the result cached. This allows automatic detection of comparators and validators, see details section for more information.
c.family	column family (aka table) name
key	row key
c.names	vector of column names
comparator	string, type of the column keys (comparator in Cassandra speak) or NULL to rely on cached schema definitions

`validator`	string, type of the values (validator in Cassandra speak) or NULL to rely on cached schema definitions
`first`	starting column name
`last`	ending column name
`reverse`	if `TRUE` the resutl is returned in reverse order
`limit`	return at most as many columns per key
`keys`	row keys (character vector)
`k.start`	start key (or token)
`k.end`	end key (or token)
`k.limit`	return at most as many keys (rows)
`tokens`	if `TRUE` then keys are interpreted as tokens (i.e. values after hashing)
`fixed`	if `TRUE` then the result if be a single data frame consisting of rows and keys and all columns ever encountered - essentially assuming fixed column structure
`level`	the desired consistency level for query operations on this connection. `"one"` is the default if not explicitly set.

Details

The nomenclature can be a bit confusing and it comes from the literature and the Cassandra API. Put in simple terms, *keyspace* is comparable to a database, and *column family* is somewhat comparable to a table. However, a table may have different number of columns for each row, so it can be used to create a flexible two-dimensional query structure. A row is defined by a (row) *key*. A query is performed by first finding out which row(s) will be fetched according to the key (`RC.get`, `RC.get.range`), keys (`RC.mget.range`) or key range (`RC.get.range.slices`), then selecting the columns of interest. Empty string (`""`) can be used to denote an unspecified range (so the default is to fetch all columns).

`comparator` and `validator` specify the types of column keys and values respectively. Every key or value in Cassandra is simply a byte string, so it can deal with arbitrary values, but sometimes it is convenient to impose some structure on that content by declaring what is represented by that byte string. Unfortunately Cassandra does not include that information in the results, so the user has to define how column names and values are to be interpreted. The default interpretation is simply as a UTF-8 encoded string, but RCassandra also supports following conversions: "UTF8Type", "AsciiType" (stored as character vectors), "BytesType" (opaque stream of bytes, stored as raw vector), "LongType" (8-bytes integer, stored as real vector in R), "DateType" (8-bytes integer, stored as `POSIXct` in R), "BooleanType" (one byte, logical vector in R), "FloatType" (4-bytes float, real vector in R), "DoubleType" (8-bytes float, real vector in R) and "UUIDType" (16-bytes, stored as UUID-formatted string). No other conversions are supported at this point. If the value is `NULL` then `RCassandra` attempts to guess the proper value by taking into account the schema definition obtained by `RC.use(..., cache.def=TRUE)`, otherwise it falls back to "UTF8Type". You can always get the raw form using "BytesType" and decode the values in R.

The `comparator` also determines how the values of `first` and `last` will be interpreted. Regardless of the comparator, it is always possible to pass either `NULL`, `""` (both denoting 0-length value) or a raw vector. Other supported types must match the comparator.

Most users will be happy with the default settings, but if you want to save every nanosecond you can, call `RC.use(..., cache.def = FALSE)` (which saves one extra `RC.describe.keyspace` request to the Cassandra instance) and always specify both `comparator` and `validator` (even if it is just "UTF8String").

Cassandra collects results in memory so key (`k.limit`) and column (`limit`) limits are mandatory. Future versions of RCassandra may abstract this limitation out (by using a limit and repeating queries with new start key/column based on the last result row), but not at this point.

Note that in Cassandra keys are typically hashed, so key range may be counter-intuitive as it is based on the hash and not on the actual value. Columns are always sorted by their name (=key).

The result of queries may be also counter-intuitive, especially when querying fixed column tables as it is not returned in the form that would be expected from a relational database. See `RC.read.table` and `RC.write.table` for retrieving and storing relational structures in rectangular tables (column families with fixed columns). But you have to keep in mind that Cassandra is essentailly key/key/value storage (row key, column key, value) with partitioning on row keys and sorting of column keys, so designing the correct schema for a task needs some thought. Dynamic columns are what makes it so powerful.

Value

`RC.use` and `RC.consistency` returns `conn`

`RC.get` and `RC.get.range` return a data frame with columns `key` (column name), `value` (value in that column) and `ts` (timestamp).

`RC.mget.range` and `RC.get.range.slices` return a named list of data frames as described in `RC.get.range` with names being the row keys, except if `fixed=TRUE` in which case the result is a data frame with row names as keys and values as elements (timestamps are not retrieved in that case).

Author(s)

Simon Urbanek

See Also

`RC.connect`, `RC.read.table`, `RC.write.table`

Examples

```
## Not run:
c <- RC.connect("cassandra-host")
RC.use(c, "testdb")
## you will have to use cassandra-cli to create the schema for the "iris" CF
RC.write.table(c, "iris", iris)
RC.get(c, "iris", "1", c("Sepal.Length", "Species"))
RC.get.range(c, "iris", "1")
## list of 150 data frames
r <- RC.get.range.slices(c, "iris")
## use limit=0 to obtain all row keys without pulling any data
rk <- RC.get.range.slices(c, "iris", limit=0)
y <- RC.read.table(c, "iris")
y <- y[order(as.integer(row.names(y))),]
RC.close(c)
```

```
## End(Not run)
```

RC.insert — *Update functions to insert data into Cassandra*

Description

`RC.insert` updates or inserts new column/value pairs

`RC.mutate` batchwise updates or inserts a list of keys, column families and columns/values.

Usage

```
RC.insert(conn, c.family, key, column, value = NULL,
          comparator = NULL, validator = NULL)
RC.mutate(conn, mutation)
```

Arguments

`conn`	connection handle obtained from `RC.connect`
`c.family`	name of the column family (string)
`key`	row key name (string) or a vector of (preferably contiguous) keys to use with the column names vector
`column`	column name - any vector supported by the comparator
`value`	optinally values to add into the columns - if specified, must be the same length as `column`. If `NULL` only the column is created
`comparator`	comparator (column name type) to be used - see `RC.get` for details
`validator`	validator (value type) to be used - see `RC.get` for details
`mutation`	a structure describing the desired mutation (see Cassandra documentation). In its simplest form it is a nested list: `list(row.key1=list(c.family1=list(col1=val1, ...), ...),` so to add column "foo" with value "bar" to column family "table" and row "key" the mutation would be `list(key=list(table=list(foo="bar")))`. The innermost list can optionally be a character vector (if unnamed it specifies the column names, otherwise names are column names and elements are values). Only string keys and column names are supproted.

Value

`conn`

Note

RC.insert supports multi-column insertions where column and value are vectors. For the scalar case insert message is used, for vector case batch_mutate. If key is a scalar, all column/value pairs are added to that row key. Alternatively, key can be a vector of the same length as column in which case the mutation will consist of key/column/value triplets. Note that key should be contiguous as the mutation will only group contiguous sequences (see coalesce from the fastmatch package for a fast way of obtaining contiguous sequences).

RC.insert honors both the validator and comparator (the latter is taken from the cache if not specified).

RC.mutate currently only uses "UTF8Type" validator and comparator as there is no way to specify either in the mutation object.

Cassandra requires timestamps on all objects that specify columns/values for conflict resolution. All functions above generate such timestamps from the system time as POSIX time in milliseconds.

Author(s)

Simon Urbanek

See Also

RC.connect, RC.use, RC.get

RC.read.table *Read and write tables into column families in Cassandra*

Description

RC.read.table reads the contents of a column family into a data frame

RC.write.table writes the contents of a data frame into a column familly

Usage

```
RC.read.table(conn, c.family, convert = TRUE, na.strings = "NA",
              as.is = FALSE, dec = ".")
RC.write.table(conn, c.family, df)
```

Arguments

conn	connection handle as obtained form RC.connect
c.family	column family name (string)
convert	logical, if TRUE the resulting data frame is processed using type.convert, otherwise all columns will be character vectors
na.strings	passed to type.convert
as.is	passed to type.convert
dec	passed to type.convert
df	data frame - it must have both row and column names

Details

Cassandra is a key/value store with dynamic columns, so tables are not the native format. Row names are used as keys and columns are treated as fixed. `RC.read.table` is really jsut a wrapper for `RC.get.range.slices(conn, c.family, fixed=TRUE)`. `RC.write.table` uses the same facility as `RC.mutate` but without actually creating the mutation object on the R side.

Note that all updates in Cassandra are "upserts", i.e., `RC.write.table` updates any existing row key/coumn name combinations or creates new ones where not present (insert). Additonal columns (or even keys) may still exist in the column family and they will not be touched.

`RC.read.table` creates a data frame from all columns that are ever encountered in at least one key. All other values are filled with `NA`s.

Value

`RC.read.table` returns the resulting data frame

`RC.write.table` returns `conn`

Note

IMPORTANT: Cassandra does *NOT* preserve order of keys and columns. Internally, keys are ordered by their hash value and columns are ordered lexicographically (treated as bytes). However, due to the fact that columns are dynamic the order of columns will vary if keys have different columns, because columns are added to the data frame in the sequence they are encountered as the keys are loaded. You may want to use `df <- df[order(as.integer(row.names(df))),]` on the result of `RC.read.table` for tables with automatic row names to obtain the original order of rows.

`RC.read.table` is more effcient than `RC.get.range.slices` because it can store columns into vectors and can pre-allocate the whole structure in advance.

Note that the current implementation of tables (`RC.read.table` and `RC.write.table`) supports only string-based representation of columns and values ("UTF8Type", "AsciiType" or similar).

Author(s)

Simon Urbanek

See Also

`RC.connect`, `RC.use`, `RC.get`

Index