

API06

Overview

The Cassandra Thrift API changed between [0.3](#), [0.4](#), [0.5](#) and 0.6; this document explains the 0.6 version.

Cassandra's client API is built entirely on top of Thrift. It should be noted that these documents mention default values, but these are not generated in all of the languages that Thrift supports. Full examples of using Cassandra from Thrift, including setup boilerplate, are found on [ThriftExamples](#). Higher-level clients are linked from [ClientOptions](#).

WARNING: Some SQL/RDBMS terms are used in this documentation for analogy purposes. They should be thought of as just that; analogies. There are few similarities between how data is managed in a traditional RDBMS and Cassandra. Please see [DataModel](#) for more information.

Terminology / Abbreviations

Keyspace

`${renderedContent}`

CF

`${renderedContent}`

SCF

`${renderedContent}`

Key

`${renderedContent}`

Column

`${renderedContent}`

Exceptions

NotFoundException

`${renderedContent}`

InvalidRequestException

`${renderedContent}`

UnavailableException

`${renderedContent}`

TimedOutException

`${renderedContent}`

TApplicationException

`${renderedContent}`

AuthenticationException

`${renderedContent}`

AuthorizationException

`${renderedContent}`

Structures

ConsistencyLevel

The `ConsistencyLevel` is an enum that controls both read and write behavior based on `<ReplicationFactor>` in your `storage-conf.xml`. The different consistency levels have different meanings, depending on if you're doing a write or read operation. Note that if $W + R > \text{ReplicationFactor}$, where W is the number of nodes to block for on write, and R the number to block for on reads, you will have strongly consistent behavior; that is, readers will always see the most recent write. Of these, the most interesting is to do `QUORUM` reads and writes, which gives you consistency while still allowing availability in the face of node failures up to half of `ReplicationFactor`. Of course if latency is more important than consistency then you can use lower values for either or both.

All discussion of "nodes" here refers to nodes responsible for holding data for the given key; "surrogate" nodes involved in [HintedHandoff](#) do not count towards achieving the requested `ConsistencyLevel`.

Write

Level	Behavior
ZERO	Ensure nothing. A write happens asynchronously in background
ANY	Ensure that the write has been written to at least 1 node, including hinted recipients.
ONE	Ensure that the write has been written to at least 1 node's commit log and memory table before responding to the client.
QUORUM	Ensure that the write has been written to $\text{<ReplicationFactor>} / 2 + 1$ nodes before responding to the client.
ALL	Ensure that the write is written to all <code><ReplicationFactor></code> nodes before responding to the client. Any unresponsive nodes will fail the operation.

Read

Level	Behavior
ZERO	Not supported, because it doesn't make sense.
ANY	Not supported. You probably want ONE instead.
ONE	Will return the record returned by the first node to respond. A consistency check is always done in a background thread to fix any consistency issues when <code>ConsistencyLevel.ONE</code> is used. This means subsequent calls will have correct data even if the initial read gets an older value. (This is called <code>read repair</code> .)
QUORUM	Will query all nodes and return the record with the most recent timestamp once it has at least a majority of replicas reported. Again, the remaining replicas will be checked in the background.
ALL	Will query all nodes and return the record with the most recent timestamp once all nodes have replied. Any unresponsive nodes will fail the operation.

Note: Different language toolkits may have their own Consistency Level defaults as well. To ensure the desired Consistency Level, you should always explicitly set the Consistency Level.

ColumnOrSuperColumn

Due to the lack of inheritance in Thrift, `Column` and `SuperColumn` structures are aggregated by the `ColumnOrSuperColumn` structure. This is used wherever either a `Column` or `SuperColumn` would normally be expected.

If the underlying column is a `Column`, it will be contained within the `column` attribute. If the underlying column is a `SuperColumn`, it will be contained within the `super_column` attribute. The two are mutually exclusive - i.e. only one may be populated.

Attribute	Type	Default	Required	Description
column	Column	n/a	N	The Column if this ColumnOrSuperColumn is aggregating a Column.
super_column	SuperColumn	n/a	N	The SuperColumn if this ColumnOrSuperColumn is aggregating a SuperColumn.

Column

The `Column` is a triplet of a name, value and timestamp. As described above, `Column` names are unique within a row. Timestamps are arbitrary - they can be any integer you specify, however they must be consistent across your application. It is recommended to use a timestamp value with a fine granularity, such as microseconds since the UNIX epoch. See [DataModel](#) for more information.

Attribute	Type	Default	Required	Description
name	binary	n/a	Y	The name of the Column.
value	binary	n/a	Y	The value of the Column.
timestamp	i64	n/a	Y	The timestamp of the Column.

SuperColumn

A `SuperColumn` contains no data itself, but instead stores another level of `Columns` below the key. See [DataModel](#) for more details on what `SuperColumns` are and how they should be used.

Attribute	Type	Default	Required	Description
name	binary	n/a	Y	The name of the SuperColumn.
columns	list<Column>	n/a	Y	The Columns within the SuperColumn.

ColumnPath

The `ColumnPath` is the path to a single column in Cassandra. It might make sense to think of `ColumnPath` and `ColumnParent` in terms of a directory structure.

Attribute	Type	Default	Required	Description
column_family	string	n/a	Y	The name of the CF of the column being looked up.
super_column	binary	n/a	N	The super column name.
column	binary	n/a	N	The column name.

ColumnParent

The `ColumnParent` is the path to the parent of a particular set of `Columns`. It is used when selecting groups of columns from the same `ColumnFamily`. In directory structure terms, imagine `ColumnParent` as `ColumnPath` + `'/../../'`.

Attribute	Type	Default	Required	Description
column_family	string	n/a	Y	The name of the CF of the column being looked up.
super_column	binary	n/a	N	The super column name.

SlicePredicate

A `SlicePredicate` is similar to a [mathematic predicate](#), which is described as "a property that the elements of a set have in common."

`SlicePredicate`'s in Cassandra are described with either a list of `column_names` or a `SliceRange`.

Attribute	Type	Default	Required	Description
column_names	list<binary>	n/a	N	A list of column names to retrieve. This can be used similar to Memcached's "multi-get" feature to fetch N known column names. For instance, if you know you wish to fetch columns 'Joe', 'Jack', and 'Jim' you can pass those column names as a list to fetch all three at once.
slice_range	SliceRange	n/a	N	A <code>SliceRange</code> describing how to range, order, and/or limit the slice.

If `column_names` is specified, `slice_range` is ignored.

SliceRange

A `SliceRange` is a structure that stores basic range, ordering and limit information for a query that will return multiple columns. It could be thought of as Cassandra's version of `LIMIT` and `ORDER BY`.

Attribute	Type	Default	Required	Description
start	binary	n/a	Y	The column name to start the slice with. This attribute is not required, though there is no default value, and can be safely set to <code>''</code> , i.e., an empty byte array, to start with the first column name. Otherwise, it must be a valid value under the rules of the <code>Comparator</code> defined for the given <code>ColumnFamily</code> .
finish	binary	n/a	Y	The column name to stop the slice at. This attribute is not required, though there is no default value, and can be safely set to an empty byte array to not stop until <code>count</code> results are seen. Otherwise, it must also be a valid value to the <code>ColumnFamily</code> <code>Comparator</code> .
reversed	boolean	false	Y	Whether the results should be ordered in reversed order. Similar to <code>ORDER BY blah DESC</code> in SQL.
count	integer	100	Y	How many columns to return. Similar to <code>LIMIT 100</code> in SQL. May be arbitrarily large, but Thrift will materialize the whole result into memory before returning it to the client, so be aware that you may be better served by iterating through slices by passing the last value of one call in as the <code>start</code> of the next instead of increasing <code>count</code> arbitrarily large.

KeyRange

A `KeyRange` is used by `get_range_slices` to define the range of keys to get the slices for.

The semantics of start keys and tokens are slightly different. Keys are start-inclusive; tokens are start-exclusive. Token ranges may also wrap – that is, the end token may be less than the start one. Thus, a range from `keyX` to `keyX` is a one-element range, but a range from `tokenY` to `tokenY` is the full ring.

Attribute	Type	Default	Required	Description
start_key	string	n/a	N	The first key in the inclusive KeyRange.
end_key	string	n/a	N	The last key in the inclusive KeyRange.
start_token	string	n/a	N	The first token in the exclusive KeyRange.
end_token	string	n/a	N	The last token in the exclusive KeyRange.
count	i32	100	Y	The total number of keys to permit in the KeyRange.

KeySlice

A `KeySlice` encapsulates a mapping of a key to the slice of columns for it as returned by the `get_range_slices` operation. Normally, when slicing a single key, a `list<ColumnOrSuperColumn>` of the slice would be returned. When slicing multiple or a range of keys, a `list<KeySlice>` is instead returned so that each slice can be mapped to their key.

Attribute	Type	Default	Required	Description
key	string	n/a	Y	The key for the slice.
columns	list<ColumnOrSuperColumn>	n/a	Y	The columns in the slice.

TokenRange

A structure representing structural information about the cluster provided by the `describe` utility methods detailed below.

Attribute	Type	Default	Required	Description
start_token	string	n/a	Y	The first token in the TokenRange.
end_token	string	n/a	Y	The last token in the TokenRange.
endpoints	list<string>	n/a	Y	A list of the endpoints (nodes) that replicate data in the TokenRange.

Mutation

A `Mutation` encapsulates either a column to insert, or a deletion to execute for a key. Like `ColumnOrSuperColumn`, the two properties are mutually exclusive - you may only set one on a `Mutation`.

Attribute	Type	Default	Required	Description
column_or_supercolumn	ColumnOrSuperColumn	n/a	N	The column to insert in to the key.
deletion	Deletion	n/a	N	The deletion to execute on the key.

Deletion

A `Deletion` encapsulates an operation that will delete all columns matching the specified `timestamp` and `predicate`. If `super_column` is specified, the `Deletion` will operate on columns within the `SuperColumn` - otherwise it will operate on columns in the top-level of the key.

Attribute	Type	Default	Required	Description
timestamp	i64	n/a	Y	The timestamp of the column(s) to be deleted.
super_column	binary	n/a	N	The super column to delete the column(s) from.
predicate	SlicePredicate	n/a	N	A predicate to match the column(s) to be deleted from the key/super column.

AuthenticationRequest

A structure that encapsulates a request for the connection to be authenticated. The authentication credentials are arbitrary - this structure simply provides a mapping of credential name to credential value.

Attribute	Type	Default	Required	Description
credentials	map<string, string>	n/a	Y	A map of named credentials.

Method calls

login

- `void login(keyspace, auth_request)`

Authenticates with the cluster for operations on the specified keyspace using the specified `AuthenticationRequest` credentials. Throws `AuthenticationException` if the credentials are invalid or `AuthorizationException` if the credentials are valid, but not for the specified keyspace.

get

- `ColumnOrSuperColumn get(keyspace, key, column_path, consistency_level)`

Get the `Column` or `SuperColumn` at the given `column_path`. If no value is present, `NotFoundException` is thrown. (This is the only method that can throw an exception under non-failure conditions.)

get_slice

- `list<ColumnOrSuperColumn> get_slice(keyspace, key, column_parent, predicate, consistency_level)`

Get the group of columns contained by `column_parent` (either a `ColumnFamily` name or a `ColumnFamily/SuperColumn` name pair) specified by the given `SlicePredicate` struct.

multiget_slice

- `map<string, list<ColumnOrSuperColumn>> multiget_slice(keyspace, keys, column_parent, predicate, consistency_level)`

Retrieves slices for `column_parent` and `predicate` on each of the given keys in parallel. Keys are a `list<string>` of the keys to get slices for.

This is similar to `get_range_slice` (Cassandra 0.5) except operating on a set of non-contiguous keys instead of a range of keys.

get_count

- `i32 get_count(keyspace, key, column_parent, consistency_level)`

Counts the columns present in `column_parent`.

The method is not $O(1)$. It takes all the columns from disk to calculate the answer. The only benefit of the method is that you do not need to pull all the columns over Thrift interface to count them.

get_range_slices

- `list<KeySlice> get_range_slices(keyspace, column_parent, predicate, range, consistency_level)`

Replaces `get_range_slice`. Returns a list of slices for the keys within the specified `KeyRange`. Unlike `get_key_range`, this applies the given predicate to all keys in the range, not just those with undeleted matching data. This method is only allowed when using an order-preserving partitioner.

insert

- `insert(keyspace, key, column_path, value, timestamp, consistency_level)`

Insert a `Column` consisting of (`column_path.column`, `value`, `timestamp`) at the given `column_path.column_family` and optional `column_path.super_column`. Note that `column_path.column` is here required, since a `SuperColumn` cannot directly contain binary values – it can only contain sub-Columns.

batch_mutate

- `batch_mutate(keyspace, mutation_map, consistency_level)`

Executes the specified mutations on the keyspace. `mutation_map` is a `map<string, map<string, list<Mutation>>>`; the outer map maps the key to the inner map, which maps the column family to the `Mutation`; can be read as: `map<key : string, map<column_family : string, list<Mutation>>>`. To be more specific, the outer map key is a row key, the inner map key is the column family name.

A `Mutation` specifies either columns to insert or columns to delete. See `Mutation` and `Deletion` above for more details.

remove

- `remove(keyspace, key, column_path, timestamp, consistency_level)`

Remove data from the row specified by `key` at the granularity specified by `column_path`, and the given `timestamp`. Note that all the values in `column_path` besides `column_path.column_family` are truly optional: you can remove the entire row by just specifying the `ColumnFamily`, or you can remove a `SuperColumn` or a single `Column` by specifying those levels too. Note that the `timestamp` is needed, so that if the commands are replayed in a different order on different nodes, the same result is produced.

describe_keyspaces

- `set<string> describe_keyspaces()`

Gets a list of all the keyspaces configured for the cluster.

describe_cluster_name

- `string describe_cluster_name()`

Gets the name of the cluster.

describe_version

- `string describe_version()`

Gets the Thrift API version.

describe_ring

- `list<TokenRange> describe_ring(keyspace)`

Gets the token ring; a map of ranges to host addresses. Represented as a set of `TokenRange` instead of a map from range to list of endpoints, because you can't use Thrift structs as map keys: <https://issues.apache.org/jira/browse/THRIFT-162> for the same reason, we can't return a set here, even though order is neither important nor predictable.

describe_keyspace

- `map<string, map<string, string>> describe_keyspace(keyspace)`

Gets information about the specified keyspace.

Examples

[There are a few examples on this page over here.](#)

<https://c.statcounter.com/9397521/0/fe557aad/1/> | stats