API05

Overview

The Cassandra Thrift API changed between 0.3, 0.4, and 0.5; this document explains the 0.5 version.

NOTE: This documents the low-level wire protocol used to communicate with Cassandra. This is not intended to be used directly in applications; rather it is highly recommended that application developers use one of the higher-level clients that are linked to from ClientOptions. That said, this page may still be useful for application developers wanting to better understand the data model or the underlying operations that are available.

The Cassandra Thrift API changed substantially after 0.3, with minor, backwards-compatible changes for 0.4, 0.5 and 0.6; this document explains the 0.5 version with annotations for the changes in 0.6 and 0.7.

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.

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 > 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 the most consistent behavior (* see below). 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.

- Because the repair replication process only requires a write to reach a single node to propagate, a write which 'fails' to meet consistency requirements will still appear eventually so long at it was written to at least one node. With W and R both using QUORUM, the best consistency we can achieve is the guarantee that we will receive the same value regardless of which nodes we read from. However, we can still peform a W=QUORUM that "fails" but reaches one server, perform a R=QUORUM that reads the old value, and then sometime later perform a R=QUORUM that reads the new value.

Terminology: "N" is the ReplicationFactor; "replicas" are the `N` nodes that are directly responsible for the data; "nodes" are any/all nodes in the cluster, including HintedHandoff participants.

Write

Level	Behavior
`ZERO`	Ensure nothing. A write happens asynchronously in background. Until CASSANDRA-685 is fixed: If too many of these queue up, buffers will explode and bad things will happen.
`ANY`	(Requires 0.6) Ensure that the write has been written to at least 1 node, including HintedHandoff recipients.
`ONE`	Ensure that the write has been written to at least 1 replica's commit log and memory table before responding to the client.
`QUORUM`	Ensure that the write has been written to `N / 2 + 1` replicas before responding to the client.
`DCQUORUM`	(No longer in 0.7) Ensure that the write has been written to <ReplicationFactor> / 2 + 1 nodes, within the local datacenter (requires NetworkTopologyStrategy)
`LOCAL_QUORUM`	(Requires 0.7) Ensure that the write has been written to <ReplicationFactor> / 2 + 1 nodes, within the local datacenter (requires NetworkTopologyStrategy)
`EACH_QUORUM`	(Requires 0.7) Ensure that the write has been written to <ReplicationFactor> / 2 + 1 nodes in each datacenter (requires NetworkTopologyStrategy)
`ALL`	Ensure that the write is written to all `N` replicas before responding to the client. Any unresponsive replicas 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 replica to respond. A consistency check is always done in a background thread to fix any consistency issues when `ConsistencyLevel.ONE` is used. This means subsequent calls will have correct data even if the initial read gets an older value. (This is called ReadRepair)
`QUORUM`	Will query all replicas and return the record with the most recent timestamp once it has at least a majority of replicas (`N / 2 + 1`) reported. Again, the remaining replicas will be checked in the background.
`DCQUORUM`	(No longer in 0.7) When using rack aware placement strategy reads are keept within a data center. See https://issues.apache.org/jira/browse/CASSANDRA-492
`LOCAL_QUORUM`	(Requires 0.7) Returns the record with the most recent timestamp once a majority of replicas within the local datacenter have replied.
`EACH_QUORUM`	(Requires 0.7) Returns the record with the most recent timestamp once a majority of replicas within each datacenter have replied.
`ALL`	Will query all replicas and return the record with the most recent timestamp once all replicas have replied. Any unresponsive replicas will fail the operation.

Note: Thrift prior to version 0.6 defaults to a Write Consistency Level of ZERO. 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 milliseconds 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 `SliceRange` 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 _, 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 Comparator defined for the given `ColumnFamily`.
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 `count` results are seen. Otherwise, it must also be a valid value to the `ColumnFamily` Comparator.
reversed	bool	false	Y	Whether the results should be ordered in reversed order. Similar to `ORDER BY blah DESC` in SQL.
count	integer	100	Y	How many columns to return. Similar to `LIMIT 100` 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 `start` of the next instead of increasing `count` arbitrarily large.

KeyRange

Requires Cassandra 0.6

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 inclusive `KeyRange`.
count	i32	100	Y	The total number of keys to permit in the `KeyRange`.

KeySlice

Requires Cassandra 0.6

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

Requires Cassandra 0.6

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

Requires Cassandra 0.6

A `Mutation` encapsulates either a column to insert, update, 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 or update based on the given key.
deletion	Deletion	n/a	N	The deletion to execute based on the given key.

Deletion

Requires Cassandra 0.6

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 representing the point in time at which the delete occurs
`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

Requires Cassandra 0.6

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.

CFDef, KSDef

Requires Cassandra 0.7

These structures contain fields necessary to describe keyspace and column family definitions.

CFDef

Attribute	Type	Default	Required	Description
`table`	`string`	None	Yes	Keyspace this CFDef belongs to
`name`	`string`	None	Yes	Name of column family. Must be unique to the keyspace
`column_type`	`string`	"Standard"	No	One of "Standard" or "Super"
`comparator_type`	`string`	"BytesType"	No	Name of comparator used for column sorting
`subcomparator_type`	`string`	None	No	Name of comparator used for subcolumns (when column_type="Super" only)
`comment`	`string`	None	No	Human-readable description of column family
`row_cache_size`	`double`	0	No	number of rows to cache
`preload_row_cache`	`boolean`	0 (False)	No	Set to true to automatically load the row cache
`key_cache_size`	`double`	200000	No	Number of keys to cache

KSDef

Attribute	Type	Default	Required	Description
`name`	`string`	None	Yes	Name of keyspace
`strategy_class`	`string`	None	Yes	Fully qualified class name of replication strategy
`replication_factor`	`integer`	None	Yes	Number of data replicas
`cf_defs`	`list<CfDef>`	None	Yes	list of column family definitions. Can be empty, but not null

_

Requires Cassandra 0.7 beta 2_

IndexExpression

Attribute	Type	Default	Required	Description
`column_name`	`binary`	None	Yes	The name of the column to perform the operand on
`op`	`IndexOperator`	None	Yes	The IndexOperator to apply
`value`	`binary`	None	Yes	The value to use in the comparison

IndexClause

Attribute	Type	Default	Required	Description
`expressions`	`list<IndexExpression>`	None	Yes	The list of IndexExpressions to AND together. Semantics from the client work similar to boolean logical operand && or SQL 'AND'
`start_key`	`binary`	None	Yes	Start key range to begin searching on

count	i32	100	No	The maximum rows to return

Method calls

login_

Requires Cassandra 0.6_

- void login(string keyspace, AuthenticationRequest 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(string keyspace, string key, ColumnPath column_path, ConsistencyLevel 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(string keyspace, string key, ColumnParent column_parent, SlicePredicate predicate, ConsistencyLevel 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_

Deprecated in 0.6 - use multiget_slice instead_

- map<string,ColumnOrSuperColumn> multiget(string keyspace, list<string> keys, ColumnPath column_path, ConsistencyLevel consistency_level)

Perform a get for column_path in parallel on the given list<string> keys. The return value maps keys to the ColumnOrSuperColumn found. If no value corresponding to a key is present, the key will still be in the map, but both the column and super_column references of the ColumnOrSuperColumn object it maps to will be null.

multiget_slice

- map<string,list<ColumnOrSuperColumn>> multiget_slice(string keyspace, list<string> keys, ColumnParent column_parent, SlicePredicate predicate, ConsistencyLevel 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_slices (Cassandra 0.6) or 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(string keyspace, string key, ColumnParent column_parent, ConsistencyLevel 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_indexed_slices_

Requires Cassandra 0.7 beta 2_

- {{ list<KeySlice> get_indexed_slices(ColumnParent column_parent, IndexClause index_clause, SlicePredicate column_predicate, ConsistencyLevel consistency_level)}}

Returns a list of key slices that meed the IndexClause critera. Note that index clause must contain at least a single EQ operation. The columns specified in the IndexExpressions will also need to be specified as indexed when the CF is created.

get_range_slice_

Deprecated in 0.6 - use `get_range_slices` instead_

- `list<KeySlice> get_range_slice(string keyspace, ColumnParent column_parent, SlicePredicate predicate, string start_key, string finish_key, i32 row_count=100, ConsistencyLevel consistency_level)`

Replaces `get_key_range`. Returns a list of slices, sorted by row key, starting with start, ending with finish (both inclusive) and at most count long. The empty string ("") can be used as a sentinel value to get the first/last existing key (or first/last column in the column predicate parameter). 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 in 0.5.

get_range_slices_

Requires Cassandra 0.6_

_
In Cassandra 0.7, first parameter "keyspace" is omitted, since the connection should already be authenticated to a keyspace._

- `list<KeySlice> get_range_slices(string keyspace, ColumnParent column_parent, SlicePredicate predicate, KeyRange range, ConsistencyLevel 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.

get_key_range_

Deprecated in 0.5 - use `get_range_slice` instead_

_
Removed in 0.6 - use `get_range_slices` instead_

- `list<string> get_key_range(string keyspace, ColumnFamily column_family, string start, string finish, i32 count=100, ConsistencyLevel consistency_level)`

Returns a list of keys starting with `start`, ending with `finish` (both inclusive), and at most `count` long. The empty string ("") can be used as a sentinel value to get the first/last existing key. (The semantics are similar to the corresponding components of `SliceRange`.)

insert

- `void insert(string keyspace, string key, ColumnPath column_path, binary value, i64 timestamp, ConsistencyLevel consistency_level)`

Insert or update 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_insert_

Deprecated in 0.6 - use `batch_mutate` instead_

- `void batch_insert(string keyspace, string key, map<string,list<ColumnOrSuperColumn>> batch_mutation, ConsistencyLevel consistency_level)`

Insert or update Columns or SuperColumns across different Column Families for the same row key. `batch_mutation` is a `map<string, list<ColumnOrSuperColumn>>` – a map which pairs column family names with the relevant `ColumnOrSuperColumn` objects to insert or update with.

batch_mutate_

Requires Cassandra 0.6_

- `void batch_mutate(string keyspace, map<string,map<string,list<Mutation>>> mutation_map, ConsistencyLevel 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 columns to insert, update or delete. See `Mutation` and `Deletion` above for more details.

remove

- `void remove(string keyspace, string key, ColumnPath column_path, i64 timestamp, ConsistencyLevel 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_cluster_name_

Requires Cassandra 0.6_

- `string describe_cluster_name()`

Gets the name of the cluster.

describe_version_

Requires Cassandra 0.6_

- `string describe_version()`

Gets the Thrift API version.

describe_ring_

Requires Cassandra 0.6_

- `list<TokenRange> describe_ring(string 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_

Requires Cassandra 0.7_

- `KsDef describe_keyspace(string keyspace)`

Gets information about the specified keyspace.

describe_keyspaces_

Requires Cassandra 0.7_

- `list<KsDef> describe_keyspaces()`

Gets a list of all the keyspaces configured for the cluster. (Equivalent to calling describe_keyspace(k) for k in keyspaces.)

truncate_

Requires Cassandra 0.7_

- `truncate(string column_family)`

Removes all the rows from the given column family.

system_add_column_family_

Requires Cassandra 0.7_

- `string system_add_column_family(CFDef cf_def)`

Adds a column family. This method will throw an exception if a column family with the same name is already associated with the keyspace. Returns the new schema version ID.

system_drop_column_family_

Requires Cassandra 0.7_

- `string system_drop_column_family(ColumnFamily column_family)`

Drops a column family. Creates a snapshot and then submits a 'graveyard' compaction during which the abandoned files will be deleted. Returns the new schema version ID.

system_rename_column_family_

Requires Cassandra 0.7_

- `string system_rename_column_family(string old_name, string new_name)`

Renames a column family if the new name doesn't collide with an existing column family associated with the same keyspace. This operation blocks while the operating system renames files on disk. Returns the new schema version ID.

system_add_keyspace_

Requires Cassandra 0.7_

- `string system_add_keyspace(KSDef ks_def)`

Creates a new keyspace and any column families defined with it. Callers **are not required** to first create an empty keyspace and then create column families for it. Returns the new schema version ID.

system_drop_keyspace_

Requires Cassandra 0.7_

- `string system_drop_keyspace(string keyspace)`

Drops a keyspace. Creates a snapshot and then submits a 'graveyard' compaction during which the abandoned files will be deleted. Returns the new schema version ID.

system_rename_keyspace_

Requires Cassandra 0.7"

- `string system_rename_keyspace(string old_name, string new_name)`

Renames a keyspace if the new name doesn't collide with an existing keyspace. This operation blocks while the operating system renames files on disk. Returns the new schema version ID.

Examples

There are a few examples on this page over here.