API04

Overview

The Cassandra Thrift API changed between 0.3 and 0.4; this document explains the 0.4 version. The 0.3 API is described in API03.

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.

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}

Exceptions

NotFoundException

${renderedContent}

InvalidRequestException

${renderedContent}

UnavailableException

${renderedContent}

TApplicationException

${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 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.

Write

Level	Behavior
`ZERO`	Ensure nothing. A write happens asynchronously in background
`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 `<ReplicationFactor> / 2 + 1` nodes before responding to the client.
`ALL`	Ensure that the write is written to `<ReplicationFactor>` nodes before responding to the client.

Read

Level	Behavior
ZERO	Not supported, because it doesn't make sense.
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 `ConsistencyLevel.ONE` is used. This means subsequent calls will have correct data even if the initial read gets an older value. (This is called `read repair`.)
QUORUM	Will query all storage 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	Not yet supported, but we plan to eventually.

ColumnPath and ColumnParent

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.

`ColumnPath` is used to looking up a single column. `ColumnParent` is used when selecting groups of columns from the same ColumnFamily. In directory structure terms, imagine `ColumnParent` as `ColumnPath + '/../'`.

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`	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 slice range 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 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 value value to the `ColumnFamily` Comparator.
`reversed`	`bool`	`false`	N	Whether the results should be ordered in reversed order. Similar to `ORDER BY blah DESC` in SQL.
`count`	`integer`	`100`	N	How many keys 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.

ColumnOrSuperColumn

Methods for fetching rows/records from Cassandra will return either a single instance of `ColumnOrSuperColumn` (`get()`) or a list of `ColumnOrSuperColumn`'s (`get_slice()`). If you're looking up a `SuperColumn` (or list of `SuperColumn`'s) then the resulting instances of `ColumnOrSuperColumn` will have the requested `SuperColumn` in the attribute `super_column`. For queries resulting in `Column`'s those values will be in the attribute `column`. This change was made between 0.3 and 0.4 to standardize on single query methods that may return either a `SuperColumn` or `Column`.

Attribute	Type	Default	Required	Description
`column`	`Column`	n/a	N	The `Column` returned by `get()` or `get_slice()`.

super_column	SuperColumn	n/a	N	The SuperColumn returned by get() or get_slice().

Method calls

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

```
map<string,ColumnOrSuperColumn> multiget(keyspace, keys, column_path, consistency_level)
list<string>
```

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(keyspace, keys, column_parent, predicate, consistency_level)
```

Performs a get_slice for column_parent and predicate for the given keys in parallel.

get_count

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

Counts the columns present in column_parent.

get_key_range

```
list<string> get_key_range(keyspace, column_family, start, finish, count=100, 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.) This method is only allowed when using an order-preserving partitioner._

Note'': get_key_range's design is kind of fundamentally broken, so we're deprecating it in favor of get_range_slice starting in 0.5. In trunk (0.5beta) get_range_slice should be used instead.

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_insert

```
batch_insert(keyspace, key, batch_mutation, consistency_level)
```

Insert 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.

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.

Examples

There are a few examples on this page over here.

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