

## Overview

The Cassandra Thrift API changed between 0.3, 0.4, 0.5, 0.6, and 0.7; this document explains the 1.0 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

Contains multiple Column Families.

### CF

ColumnFamily.

### SCF

ColumnFamily of type "Super".

### Key

A unique string that identifies a row in a CF. For clarity, rows are always identified by keys; columns are identified by names. Note that Thrift's Java code [i.e., Cassandra server] assumes that Strings are always encoded as UTF-8, but if you are using a non-Java client, you may need to manually encode non-ascii strings as utf8 first. (This is the major place Thrift does not support interoperability between different platforms well.)

### Column

A tuple of name, value, and timestamp; names are unique within rows.

## Exceptions

### NotFoundException

A specific column was requested that does not exist.

### InvalidRequestException

Invalid request could mean keyspace or column family does not exist, required parameters are missing, or a parameter is malformed. `why` contains an associated error message.

### UnavailableException

Not all the replicas required could be created and/or read.

### TimeoutException

The node responsible for the write or read did not respond during the rpc interval specified in your configuration (default 10s). This can happen if the request is too large, the node is oversaturated with requests, or the node is down but the failure detector has not yet realized it (usually this takes < 30s).

### TApplicationException

Internal server error or invalid Thrift method (possible if you are using an older version of a Thrift client with a newer build of the Cassandra server).

### AuthenticationException

Invalid authentication request (user does not exist or credentials invalid)

### AuthorizationException

Invalid authorization request (user does not have access to keyspace)

### SchemaDisagreementException

Schemas are not in agreement across all nodes

## Structures

### ConsistencyLevel

The `ConsistencyLevel` is an `enum` that controls both read and write behavior based on `<ReplicationFactor>` in your schema definition. 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
ANY	Ensure that the write has been written to at least 1 node, including <code>HintedHandoff</code> 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.
TWO	Ensure that the write has been written to at least 2 replica's before responding to the client.

THREE	Ensure that the write has been written to at least 3 replica's before responding to the client.
QUORUM	Ensure that the write has been written to $N / 2 + 1$ replicas before responding to the client.
LOCAL_QUORUM	Ensure that the write has been written to $\langle \text{ReplicationFactor} \rangle / 2 + 1$ nodes, within the local datacenter (requires <code>NetworkTopologyStrategy</code> )
EACH_QUORUM	Ensure that the write has been written to $\langle \text{ReplicationFactor} \rangle / 2 + 1$ nodes in each datacenter (requires <code>NetworkTopologyStrategy</code> )
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
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 <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 ReadRepair)
TWO	Will query 2 replicas and return the record with the most recent timestamp. Again, the remaining replicas will be checked in the background.
THREE	Will query 3 replicas and return the record with the most recent timestamp.
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.
LOCAL_QUORUM	Returns the record with the most recent timestamp once a majority of replicas within the local datacenter have replied.
EACH_QUORUM	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:** 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 <code>Column</code> if this <code>ColumnOrSuperColumn</code> is aggregating a <code>Column</code> .
super_column	SuperColumn	n/a	N	The <code>SuperColumn</code> if this <code>ColumnOrSuperColumn</code> is aggregating a <code>SuperColumn</code>
counter_column	CounterColumn	n/a	N	The <code>CounterColumn</code> if this <code>ColumnOrSuperColumn</code> is aggregating a <code>CounterColumn</code> .
counter_super_column	CounterSuperColumn	n/a	N	The <code>CounterSuperColumn</code> if this <code>ColumnOrSuperColumn</code> is aggregating a <code>CounterSuperColumn</code>

## 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 (the CLI uses microseconds). See `DataModel` for more information.

Attribute	Type	Default	Required	Description
name	binary	n/a	Y	The name of the <code>Column</code> .
value	binary	n/a	Y	The value of the <code>Column</code> .
timestamp	i64	n/a	Y	The timestamp of the <code>Column</code> .
ttl	i32	n/a	N	An optional, positive delay (in seconds) after which the <code>Column</code> will be automatically deleted.

## 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 <code>SuperColumn</code> .
columns	list<Column>	n/a	Y	The <code>Columns</code> within the <code>SuperColumn</code> .

## CounterColumn

A `CounterColumn` only allows for addition and subtraction. See `Counters` 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.

## CounterSuperColumn

A `CounterSuperColumn` contains no data itself, but instead stores another level of `CounterColumn` below the key.

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

## 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 ' ', 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 count results are seen. Otherwise, it must also be a valid value to the <code>ColumnFamily</code> <code>Comparator</code> .
reversed	bool	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 count 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	binary	n/a	N	The first key in the inclusive <b>KeyRange</b> .
end_key	binary	n/a	N	The last key in the inclusive <b>KeyRange</b> .
start_token	string	n/a	N	The first token in the exclusive <b>KeyRange</b> .
end_token	string	n/a	N	The last token in the exclusive <b>KeyRange</b> .
count	i32	100	Y	The total number of keys to permit in the <b>KeyRange</b> .

## 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	binary	n/a	Y	The key for the slice.
columns	list<ColumnOrSuperColumn>	n/a	Y	The columns in the slice.

## IndexOperator

An enum that details the type of operator to use in an **IndexExpression**. Currently, on **EQ** is supported for configuring a **ColumnFamily**, but the other operators may be used in conjunction with and **EQ** operator on other non-indexed columns.

Operator	Description
EQ	Equality
GTE	Greater than or equal to
GT	Greater than
LTE	Less than or equal to
LT	Less than

## IndexExpression

A struct that defines the **IndexOperator** to use against a column for a lookup value. Used only by the **IndexClause** in the `get_indexed_slices` method.

Attribute	Type	Default	Required	Description
column_name	binary	n/a	Y	The column name to against which the operator and value will be applied
op	IndexOperator	n/a	Y	The <b>IndexOperator</b> to use. Currently only <b>EQ</b> is supported for direct queries, but other <b>IndexExpression</b> structs may be created and passed to <b>IndexClause</b>
value	binary	n/a	Y	The value to be compared against the column value

## IndexClause

Defines one or more **IndexExpressions** for `get_indexed_slices`. An **IndexExpression** containing an **EQ** **IndexOperator** must be present.

Attribute	Type	Default	Required	Description
expressions	list<IndexExpression>	n/a	Y	The list of <b>IndexExpression</b> objects which must contain one <b>EQ</b> <b>IndexOperator</b> among the expressions
start_key	binary	n/a	Y	Start the index query at the specified key - can be set to ' ', i.e., an empty byte array, to start with the first key
count	integer	100	Y	The number of results to which the index query will be constrained

## 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 <b>TokenRange</b> .
end_token	string	n/a	Y	The last token in the <b>TokenRange</b> .
endpoints	list<string>	n/a	Y	A list of the endpoints (nodes) that replicate data in the <b>TokenRange</b> .

## 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
<code>column_or_supercolumn</code>	<code>ColumnOrSuperColumn</code>	n/a	N	The column to insert in to the key.
<code>deletion</code>	<code>Deletion</code>	n/a	N	The deletion to execute on the key.

## Deletion

A `Deletion` encapsulates an operation that will delete all columns less than the specified `timestamp` and matching the `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
<code>timestamp</code>	<code>i64</code>	n/a	N	The timestamp of the delete operation. Must only be unset in the case of counter deletions.
<code>super_column</code>	<code>binary</code>	n/a	N	The super column to delete the column(s) from.
<code>predicate</code>	<code>SlicePredicate</code>	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
<code>credentials</code>	<code>map&lt;string, string&gt;</code>	n/a	Y	A map of named credentials.

## IndexType

Type	Behavior
KEYS	A <code>ColumnFamily</code> backed index.

## ColumnDef

Describes a column in a column family.

Attribute	Type	Default	Required	Description
<code>name</code>	<code>binary</code>	n/a	Y	The column name
<code>validation_class</code>	<code>string</code>	n/a	Y	The <code>validation_class</code> of the column as a class name
<code>index_type</code>	<code>IndexType</code>	n/a	N	The type of index
<code>index_name</code>	<code>string</code>	n/a	N	Name for the index. Both an index name and type must be specified.

## CfDef

Describes a column family

Attribute	Type	Default
<code>keyspace</code>	<code>string</code>	n/a
<code>name</code>	<code>string</code>	n/a
<code>column_type</code>	<code>string</code>	<code>Standard</code>
<code>comparator_type</code>	<code>string</code>	<code>BytesType</code>
<code>subcomparator_type</code>	<code>string</code>	n/a
<code>comment</code>	<code>string</code>	n/a
<code>row_cache_size</code>	<code>double</code>	0
<code>key_cache_size</code>	<code>double</code>	200000
<code>read_repair_chance</code>	<code>double</code>	1.0
<code>column_metadata</code>	<code>list&lt;ColumnDef&gt;</code>	n/a
<code>gc_grace_seconds</code>	<code>i32</code>	n/a
<code>default_validation_class</code>	<code>string</code>	n/a
<code>id</code>	<code>i32</code>	n/a
<code>min_compaction_threshold</code>	<code>i32</code>	n/a
<code>max_compaction_threshold</code>	<code>i32</code>	n/a

row_cache_save_period_in_seconds	i32	n/a
key_cache_save_period_in_seconds	i32	n/a
memtable_flush_after_mins	i32	n/a
memtable_throughput_in_mb	i32	n/a
memtable_operations_in_millions	double	n/a
replicate_on_write	bool	n/a
merge_shards_chance	double	n/a
key_validation_class	string	n/a
row_cache_provider	string	org.apache.cassandra.cache.ConcurrentLinkedHashCacheProvid
key_alias	binary	n/a

## KsDef

Describes a keyspace.

Attribute	Type	Default	Required
name	string	n/a	Y
strategy_class	string	n/a	Y
strategy_options	map<string,string>	n/a	N
cf_defs	list<CfDef>	n/a	Y
durable_writes	bool	true	N

## Compression

Type
GZIP
NONE

## CqlResultType

Type
ROWS
VOID
INT

## CqlRow

Row returned from a CQL query.

Attribute	Type	Default	Required
key	binary	n/a	Y
columns	list<Column>	n/a	Y

## CqlRow

Result returned from a CQL query.

Attribute	Type	Default	Required
type	CqlResultType	n/a	Y
rows	list<CqlRow>	n/a	N
num	i32	n/a	N

## Method calls

### login

```
void login(AuthenticationRequest auth_request)
```

Authenticates with the cluster 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.

### set\_keyspace

```
void set_keyspace(string keyspace)
```

Set the keyspace to use for subsequent requests. Throws `InvalidRequestException` for an unknown keyspace.

### get

```
ColumnOrSuperColumn get(binary 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(binary 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\_slice

```
map<string, list<ColumnOrSuperColumn>> multiget_slice(list<binary> 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`, except it operates on a set of non-contiguous keys instead of a range of keys.

### get\_count

```
i32 get_count(binary key, ColumnParent column_parent, SlicePredicate predicate, ConsistencyLevel consistency_level)
```

Counts the columns present in `column_parent` within the `predicate`.

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.

### multiget\_count

```
map<string, i32> multiget_count(list<binary> keys, ColumnParent column_parent, SlicePredicate predicate, ConsistencyLevel consistency_level)
```

A combination of `multiget_slice` and `get_count`.

### get\_range\_slices

```
list<KeySlice> get_range_slices(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. Note that when using `RandomPartitioner`, keys are stored in the order of their MD5 hash, making it impossible to get a meaningful range of keys between two endpoints.

### get\_indexed\_slices

```
list<KeySlice> get_indexed_slices(ColumnParent column_parent, IndexClause index_clause, SlicePredicate predicate, ConsistencyLevel consistency_level)
```

Like `get_range_slices`, returns a list of slices, but uses `IndexClause` instead of `KeyRange`. To use this method, the underlying `ColumnFamily` of the `ColumnParent` must have been configured with a `column_metadata` attribute, specifying at least the `name` and `index_type` attributes. See `CfDef` and `ColumnDef` above for the list of attributes. Note: the `IndexClause` must contain one `IndexExpression` with an `EQ` operator on a configured index column. Other `IndexExpression` structs may be added to the `IndexClause` for non-indexed columns to further refine the results of the `EQ` expression.

### insert

```
insert(binary key, ColumnParent column_parent, Column column, ConsistencyLevel consistency_level)
```

Insert a `Column` consisting of (`name`, `value`, `timestamp`, `ttl`) at the given `ColumnParent`. Note that a `SuperColumn` cannot directly contain binary values -- it can only contain sub-Columns. Only one sub-Column may be inserted at a time, as well.

### batch\_mutate

```
batch_mutate(map<binary, map<string, list<Mutation>>> mutation_map, ConsistencyLevel consistency_level)
```

Executes the specified mutations on the keyspace. `mutation_map` is a `map<string, map<string, vector<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, vector<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.

### add

```
add(binary key, ColumnParent column_parent, CounterColumn column, ConsistencyLevel consistency_level)
```

Increments a `CounterColumn` consisting of (`name`, `value`) at the given `ColumnParent`. Note that a `SuperColumn` cannot directly contain binary values -

- it can only contain sub-Columns.

### remove

```
remove(binary 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.

### remove\_counter

```
remove_counter(binary key, ColumnPath column_path, ConsistencyLevel consistency_level)
```

Remove a counter from the row specified by `key` at the granularity specified by `column_path`. 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 counters have limited support for deletes: if you remove a counter, you must wait to issue any following update until the delete has reached all the nodes and all of them have been fully compacted.

### truncate

```
truncate(string column_family)
```

Removes all the rows from the given column family.

### describe\_cluster\_name

```
string describe_cluster_name()
```

Gets the name of the cluster.

### describe\_schema\_versions

```
map<string, list<string>> describe_schema_versions
```

For each schema version present in the cluster, returns a list of nodes at that version. Hosts that do not respond will be under the key `DatabaseDescriptor.INITIAL_VERSION`. The cluster is all on the same version if the size of the map is 1.

### describe\_keyspace

```
KsDef describe_keyspace(string keyspace)
```

Gets information about the specified keyspace.

### describe\_keyspaces

```
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`.)

### describe\_partitioner

```
string describe_partitioner()
```

Gets the name of the partitioner for the cluster.

### 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\_snitch

```
string describe_snitch()
```

Gets the name of the snitch used for the cluster.

### describe\_version

```
string describe_version()
```

Gets the Thrift API version.

### system\_add\_column\_family

```
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**

```
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\_add\_keyspace**

```
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**

```
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\_update\_keyspace**

```
string system_update_keyspace(KsDef ks_def)
```

Updates properties of a keyspace. returns the new schema id.

### **system\_update\_column\_family**


```
string system_update_column_family(CfDef cf_def)
```

### **execute\_cql\_query**

```
CqlResult execute_cql_query(binary query, Compression compression)
```

Executes a CQL (Cassandra Query Language) statement and returns a CqlResult containing the results. Throws InvalidRequestException, UnavailableException, TimedOutException, SchemaDisagreementException.

## **Examples**

 There are a few examples on this page over here.

API (last edited 2011-11-24 01:41:59 by BlairZajac)





