API03

Cassandra Data Model and Operations

This page was created by someone trying to understand Cassandra. Until it is reviewed & blessed by someone who really knows you read it at your own risk...

This page is an alternate attempt at capturing the Cassandra data model and its operations. The descriptions below show the original Thrift API (as of 0.3) as well as a simplified notation borrowed from the Bright Yellow Cow blog entry, i.e. using [] to mean 'list of' and (,) for tuple construction.

Simple Column families

A column family has a name and an arbitrary number of columns, each column is a name, value, and timestamp tuple. Columns may be name sorted or time sorted, which affects range operations on them. In pseudo-notation:

```
family -> [(name, value, timestamp)]
```

Since each (top-level) row has an arbitrary set of columns in each column family, we can really think of this as a two dimensional map:

```
family -> [(key1, key2, value, timestamp)]
```

In the Thrift API all this is defined as:

```
struct column_t {
   1: string                       columnName,
   2: binary                       value,
   3: i64                          timestamp,
}

typedef map< string, list<column_t>  > column_family_map
```

insert

Insert a column.

```
insert(family, key1, key2, value, timestamp)
```

I believe the block_for parameter is to wait for N replicas to ACK the write. From the Thrift API:

```
void insert(1:string tablename, 2:string key, 3:string columnFamily_column, 4:binary cellData,
            5:i64 timestamp, 6:i32 block_for=0)
throws (1: InvalidRequestException ire, 2: UnavailableException ue),
```

remove

Remove a column

```
remove(family, key1, key2, timestamp)
```

The timestamp specifies exactly which insertion is removed (the column could have been re-inserted "later"). From the Thrift API:

```
void remove(1:string tablename, 2:string key, 3:string columnFamily_column, 4:i64 timestamp,
            5:i32 block_for=0)
throws (1: InvalidRequestException ire, 2: UnavailableException ue),
```

get_column

Retrieve a specific column for a key.

```
get_column(family, key1, key2) -> (key2, value, timestamp)
```

From the Thrift API:

```
column_t       get_column(1:string tablename, 2:string key, 3:string columnFamily_column)
throws (1: InvalidRequestException ire, 2: NotFoundException nfe),
```

get_slice

Retrieve all columns for a key:

```
get_slice(family, key1) -> [(key2, value, timestamp)]
```

plus `start`/`count` parameters allow pagination of the results. From the Thrift API:

```
list<column_t> get_slice(1:string tablename, 2:string key, 3:string columnFamily_column,
                         4:i32 start=-1, 5:i32 count=-1)
throws (1: InvalidRequestException ire, 2: NotFoundException nfe),
```

get_slice_by_name_range

Retrieve a range of columns for a key:

```
get_slice(family, key1, key2_start, key2_end) -> [(key2, value, timestamp)]
```

plus a `count` parameter allows limiting the result. From the Thrift API:

```
list<column_t> get_slice_by_name_range(1:string tablename, 2:string key, 3:string columnFamily,
                                       4:string start, 5:string end, 6:i32 count=-1)
throws (1: InvalidRequestException ire, 2: NotFoundException nfe),
```

get_slice_by_names

Retrieve a specific set of columns for a key:

```
get_slice_by_names(family, key1, [key2_1, key2_2, ..., key2_N]) -> [(key2, value, timestamp)]
```

From the Thrift API:

```
list<column_t> get_slice_by_names(1:string tablename, 2:string key, 3:string columnFamily, 4:list<string>
columnNames)
throws (1: InvalidRequestException ire, 2: NotFoundException nfe),
```

get_slice_from

Retrieve columns for a key starting from a specific column.

```
get_slice_from(family, key1, key2_start) -> [(key, value, timestamp)]
```

plus an ascending/descending flag and a count determine the direction and limit of the enumeration. From the Thrift API:

```
list<column_t> get_slice_from(1:string tablename, 2:string key, 3:string columnFamily_column,
                              4:bool isAscending, 5:i32 count)
throws (1: InvalidRequestException ire, 2: NotFoundException nfe),
```

get_columns_since

Retrieves columns for a key starting from a specific timestamp.

```
get_columns_since(family, key1, key2, timestamp) -> [(key, value, timestamp)]
```

From the Thrift API:

```
list<column_t> get_columns_since(1:string tablename, 2:string key, 3:string columnFamily_column, 4:i64
timeStamp)
throws (1: InvalidRequestException ire, 2: NotFoundException nfe),
```

get_column_count

Return the number of columns for a key.

```
get_column_count(family, key1, key2) -> count
```

From the Thrift API:

```
i32 get_column_count(1:string tablename, 2:string key, 3:string columnFamily_column)
throws (1: InvalidRequestException ire),
```

batch_insert

Insert a batch of columns for a key.

```
batch_insert(family, key1, [(key2, value, timestamp)])
```

From the Thrift API:

```
struct batch_mutation_t {
   1: string                        table,
   2: string                        key,
   3: column_family_map             cfmap,
}

void     batch_insert(1: batch_mutation_t batchMutation, 2:i32 block_for=0)
throws (1: InvalidRequestException ire, 2: UnavailableException ue),
```

Super Column

A super column family has a name and an arbitrary number of super columns, each super column has an arbitrary number of columns. "Currently" supercolumns are always name-sorted, and their subcolumns are always time-sorted. In pseudo-notation:

```
super_family -> [(super_column, [(column_name, value, timestamp)])]
```

It is tempting but inaccurate to think of this as a three dimensional map:

```
super_family -> [(key1, key2, key3, value, timestamp)]
```

What's more accurate is to continue thinking of this as a two-dimensional map, just like regular column families, but where the values are really sets of name-value pairs (plus timestamps to be accurate). So it's really like this:

```
Simple column families:
  column_family -> [(key1, key2, value, timestamp)]
Super column families:
  column_family -> [(key1, key2, [(key3, value, timestamp)])]
```

In the Thrift API all this is defined as:

```
struct superColumn_t {
   1: string          name,
   2: list<column_t>   columns,
}

typedef map< string, list<superColumn_t>  > superColumn_family_map
```

get_superColumn

Retrieves a super column from a column family for a key.

```
get_superColumn(super_family, key1, key2) -> (key2, [(key3, value, timestamp)])
```

From the Thrift API:

```
superColumn_t get_superColumn(1:string tablename, 2:string key, 3:string columnFamily)
throws (1: InvalidRequestException ire, 2: NotFoundException nfe),
```

Note that the 3rd argument should really be called `columnFamily_superColumnName`

get_slice_super

Retrieve the super columns in a super column family for a key.

```
get_slice_super(super_family, key1) -> [(key2, [(key3, value, timestamp)])]
```

The `start`/`count` parameters allow pagination of the results. From the Thrift API:

```
list<superColumn_t> get_slice_super(1:string tablename, 2:string key, 3:string columnFamily_superColumnName,
                                    4:i32 start=-1, 5:i32 count=-1)
throws (1: InvalidRequestException ire),
```

Note that the 3rd argument should really be called `columnFamily`

get_slice_super_by_names

Retrieve a set of super columns in a super column family.

```
get_slice_super_by_names(family, key1, [key2_1, key2_2, ..., key2_N]) -> [(key2, [(key3, value, timestamp)])]
```

From the Thrift API:

```
list<superColumn_t> get_slice_super_by_names(1:string tablename, 2:string key, 3:string columnFamily,
                                             4:list<string> superColumnNames)
throws (1: InvalidRequestException ire),
```

batch_insert_superColumn

Insert a super column.

```
batch_insert_superColumn(family, key1, key2, [(key3, value, timestamp)])
```

From the Thrift API:

```
struct batch_mutation_super_t {
   1: string                        table,
   2: string                        key,
   3: superColumn_family_map        cfmap,
}

void batch_insert_superColumn(1:batch_mutation_super_t batchMutationSuper, 2:i32 block_for=0)
throws (1: InvalidRequestException ire, 2: UnavailableException ue),
```

Other operations

get_key_range

Retrieve the list of keys that exist in a range. A key exists if at least on column in one column family exists for the key. A list of column families can be passed into the call to reduce the search to columns in those families.

```
get_key_range(family, key1_start, key1_end, [key2_1, key2_2, ..., key2_N]) -> [key1_1, key1_2, ..., key1_M]
```

From the Thrift API:

```
# range query: returns matching keys
list<string> get_key_range(1:string tablename, 2:list<string> columnFamilies=[], 3:string startWith="", 4:
string stopAt="",
                           5:i32 maxResults=1000)
throws (1: InvalidRequestException ire),
```

touch

Intended to force index information for the key into cache, but is buggy and to be deprecated.

```
touch(key1)
```

From the Thrift API:

```
oneway void touch(1:string key, 2:bool fData),
```