Creating a table

Data in Cassandra is organized into storage structures known as tables. Some older documentation may refer to tables as column families. If someone refers to column families, it is safe to assume that they are referring to structures in older versions of Apache Cassandra:

```
CREATE TABLE [IF NOT EXISTS] [keyspace_name.]<table_name>
 <column_name> <column_type>,
 [additional <column_name> <column_type>,]
 PRIMARY KEY ((<partition_key>[,additional <partition_key>])[,<clustering_keys]);
[WITH <options>];
```

Simple table example

For a small table with simple query requirements, something like this might be sufficient:

```
CREATE TABLE users (
 username TEXT,
 email TEXT,
 department TEXT,
 title TEXT,
 ad_groups TEXT,
 PRIMARY KEY (username));
```

Clustering key example

Let's say that we wanted to be able to offer up the same data, but to support a query for users by department. The table definition would change to something like this:

```
CREATE TABLE users_by_dept (
  username TEXT,
  email TEXT,
  department TEXT,
  title TEXT,
  AD_groups TEXT,
  PRIMARY KEY ((department),username))
WITH CLUSTERING ORDER BY (username ASC)
  AND COMPACTION = {'class':'LeveledCompactionStrategy',
  'sstable_size_in_mb':'200'};
```

As the preceding table will be read more than it will be written to (users queried more often than they are added), we also designate use of `LeveledCompactionStrategy`.

Note

The default compaction strategy is `SizeTieredCompactionStrategy`, which tends to favor read and write patterns that are either even (50% read, 50% write) or more write-heavy.

For a later example, let's write some data to that table:

```
 INSERT INTO users_by_dept(department,username,title,email) VALUES ('Engineering','Dinesh','Dev Lead','dinesh@piedpiper.com');
INSERT INTO users_by_dept(department,username,title,email) VALUES ('Engineering','Gilfoyle','Sys Admin/DBA','thedarkone@piedpiper.co
INSERT INTO users_by_dept(department,username,title,email) VALUES ('Engineering','Richard','CEO','richard@piedpiper.com');
INSERT INTO users_by_dept(department,username,title,email) VALUES ('Marketing','Erlich','CMO','erlichb@aviato.com');
INSERT INTO users_by_dept(department,username,title,email) VALUES ('Finance/HR','Jared','COO','donald@piedpiper.com');
```

Composite partition key example

Solving data model problems attributed to unbound row growth can sometimes be done by adding another partition key. Let's assume that we want to query security entrance logs for employees. If we were to use a clustering key on a new column named time to one of the preceding examples, we would be continually adding cells to each partition. So we'll build this `PRIMARY KEY` to partition our data by `entrance` and `day`, as well as cluster it on `checkpoint_time` and `username`:

```
CREATE TABLE security_log (
 entrance TEXT,
 day BIGINT,
 checkpoint_time TIMESTAMP,
 username TEXT,
 email TEXT,
 department TEXT,
 title TEXT,
 PRIMARY KEY ((entrance,day),checkpoint_time,username))
WITH CLUSTERING ORDER BY (checkpoint_time DESC, username ASC)
AND default_time_to_live=2592000;
```

The preceding table will store data sorted within each partition by both `checkpoint_time` and `username`. Note that, since we care more about the most recent data, we have designated the clustering on `checkpoint_time` to be in descending order. Additionally, we'll assume we have a requirement to only keep security log data for the last 30 days, so we'll set `default_time_to_live` to 30 days (`2592000` seconds).

Table options

There are several options that can be adjusted at table-creation time. Most of these options are present on all tables. If they are omitted from table creation, their default values are assumed. The following are a few of the table options:

- Clustering order specifies the column(s) and direction by which the table data (within a partition) should be stored on disk. As Cassandra stores data written sequentially, it's also faster when it can read sequentially, so learning how to utilize clustering keys in your models can help with performance:

```
CLUSTERING ORDER BY (<clustering_key_name <ASC|DESC>)
```

- This setting represents the false positive probability for the table's bloom filter. The value can range between `0.0` and `1.0`, with a recommended setting of `0.1` (10%):

```
bloom_filter_fp_chance = 0.1
```

- The `caching` property dictates which caching features are used by this table. There are two types of caching that can be used by a table: **key caching** and **row caching**. Key caching is enabled by default. Row caching can take up larger amounts of resources, so it defaults to disabled or `NONE`:

```
caching = {'keys': 'ALL', 'rows_per_partition': 'NONE'}
```

- This property allows for the addition of `comment` or description for the table, assuming that it is not clear from its name:

```
comment = ''
```

- This enables the table for **Change Data Capture** (**CDC**) logging and `cdc` defaults to `FALSE`:

```
cdc = FALSE
```

- A map structure that allows for the configuration of the compaction strategy, and the properties that govern it:

```
compaction = {'class': 'org.apache.cassandra.db.compaction.LeveledCompactionStrategy'}
```

Apache Cassandra 3.0 ships with three compaction strategies:

- `SizeTieredCompactionStrategy`: This is the default strategy, and works well in most scenarios, specifically write-heavy throughput patterns. It works by finding similarly-sized SSTable files and combining them once they reach a certain size. If you create a table without configuring compaction, it will be set to the following options:

```
compaction = {'class': 'org.apache.cassandra.db.compaction.SizeTieredCompactionStrategy',
 'max_threshold': '32', 'min_threshold': '4'}
```

Under these settings, a minor compaction will trigger for the table when at least `4` (and no more than `32`) SSTables are found to be within a calculated threshold of the average SSTable file size.

- `LeveledCompactionStrategy`: The leveled compaction strategy builds out SSTable files as levels, where each level is 10 times the size of the previous level. In this way, a row can be guaranteed to be contained within a single SSTable file 90% of the time. A typical leveled `compaction` config looks like this:

```
compaction = {'class': 'org.apache.cassandra.db.compaction.LeveledCompactionStrategy',
 'sstable_size_in_mb': '160'}
```

Leveled `compaction` is a good idea for operational patterns where reads are twice (or more) as frequent as writes.

- `TimeWindowCompactionStrategy` : Time-window compaction builds SSTable files according to time-based buckets. Rows are then stored according to these (configurable) buckets. Configuration for time window `compaction` looks like this:

```
compaction = {'class': 'org.apache.cassandra.db.compaction.TimeWindowCompactionStrategy',
 'compaction_window_unit': 'hours', 'compaction_window_size': '24'}
```

This is especially useful for time-series data, as it allows data within the same time period to be stored together.

Note

Time-window compaction can greatly improve query performance (over other strategies) for time-series data that uses**time to live** (**TTL**). This ensures that data within a bucket is tombstoned together, and therefore the TTL tombstones should not interfere with queries for newer data.

Note

`TimeWindowCompactionStrategy` is new as of Apache Cassandra 3.0, and replaces `DateTieredCompactionStrategy`, which was shipped with Apache Cassandra 2.1 and 2.2.

```
compression = {'chunk_length_in_kb': '64', 'class': 'org.apache.cassandra.io.compress.LZ4Compressor'}
```

This property is another map structure that allows the compression settings for a table to be configured. Apache Cassandra ships with three compressor classes: `LZ4Compressor` (default), `SnappyCompressor`, and `DeflateCompressor`. Once the class has been specified, `chunk_length` can also be specified. By default, a table will use `LZ4Compressor`, and will be configured as shown here:

```
compression = {'class': 'org.apache.cassandra.io.compress.LZ4Compressor',
 'chunk_length_in_kb': '64'}
```

Note

To disable compression, specify `compression = {'sstable_compression': ''}` .

It should be noted that the default `chunk_length_in_kb` of `64` is intended for write-heavy workloads. Access patterns that are more evenly read and write, or read-heavy may see a performance benefit from bringing that value down to as low as four. As always, be sure to test significant changes, like this:

```
crc_check_chance = 1.0
```

With compression enabled, this property defines the probability that the CRC compression checksums will be verified on a read operation. The default value is `1.0`, or 100%:

```
dclocal_read_repair_chance = 0.1
```

This property specifies the probability that a read repair will be invoked on a read. The read repair will only be enforced within the same (local) data center:

```
default_time_to_live = 0
```

This allows all data entered into a table to be automatically deleted after a specified number of seconds. The default value is `0`, which disables TTL by default. TTLs can also be set from the application side, which will override this setting. The maximum value is `630720000`, or 20 years.

Note

Don't forget that TTLs also create tombstones. So, if your table employs a TTL, be sure to account for that in your overall data modeling approach.

```
gc_grace_seconds = 864000
```

This property specifies the amount of time (in seconds) that tombstones in the table will persist before they are eligible for collection via compaction. By default, this is set to `864000` seconds, or 10 days.

Note

Remember that the 10 day default exists to give your cluster a chance to run a repair. This is important, as all tombstones must be successfully propagated to avoid data ghosting its way back into a result set. If you plan to lower this value, make sure that you also increase the frequency by which the repair is run.

```
min_index_interval & max_index_interval
```

These properties work together to determine how many entries end up in the table's index summary (in RAM). The more entries in the partition index, the quicker a partition can be located during an operation. The trade-off, is that more entries equal more RAM consumed. The actual value used is determined by how often the table is accessed. Default values for these properties are set as follows:

```
 max_index_interval = 2048
min_index_interval = 128
memtable_flush_period_in_ms = 0
```

The number of milliseconds before the table's memtables are flushed from RAM to disk. The default is zero, effectively leaving the triggering of memtable flushes up to the commit log and the defined value of `memtable_cleanup_threshold`:

```
read_repair_chance = 0.0
```

Similar to `dclocal_read_repair_chance`, this setting specifies the probability that a cross-data center read repair will be invoked. This defaults to zero (0.0), to ensure that read repairs are limited to a single data center at a time (and therefore less expensive):

```
speculative_retry = '99PERCENTILE';
```

This property configures rapid read protection, in that read requests are sent to other replicas despite their consistency requirements having been met. This property has four possible values:

- `ALWAYS`: Every ready sends additional read requests to other replicas.
- `XPERCENTILE`: Sends additional read requests only for requests that are determined to be in the slowest X percentile. The default value for this property is `99PERCENTILE`, which will trigger additional replica reads for request latencies in the 99^th^ percentile, based on past performance of reads for that table.
- `XMS`: Sends additional replica reads for requests that do not complete within X milliseconds.
- `NONE`: Disables this functionality.