



# Introduction

The Cassandra data model is designed for distributed data on a very large scale. It trades ACID-compliant data practices for important advantages in performance, availability, and operational manageability. Here are some resources for learning about the data model:

- DataStax (formerly Riptano)  reference documentation on the Cassandra data model
-  An Introduction to the data model by Max Grinev.

The basic concepts are:

- Cluster: the machines (nodes) in a logical Cassandra instance. Clusters can contain multiple keyspaces.
- Keyspace: a namespace for ColumnFamilies, typically one per application.
- ColumnFamilies contain multiple columns, each of which has a name, value, and a timestamp, and which are referenced by row keys.
- SuperColumns can be thought of as columns that themselves have subcolumns.

We'll start from the bottom up, moving from the leaves of Cassandra's data structure (columns) up to the root of the tree (the cluster).

## Columns

The column is the lowest/smallest increment of data. It's a tuple (triplet) that contains a name, a value and a timestamp.

Here's the thrift interface definition of a Column

```
struct Column {  
    1: binary          name,  
    2: binary          value,  
    3: i64             timestamp,  
}
```

And here's a column represented in JSON-ish notation:

```
{  
  "name": "emailAddress",  
  "value": "foo@bar.com",  
  "timestamp": 123456789  
}
```

All values are supplied by the client, including the 'timestamp'. This means that clocks on the clients should be synchronized (in the Cassandra server environment is useful also), as

these timestamps are used for conflict resolution. In many cases the 'timestamp' is not used in client applications, and it becomes convenient to think of a column as a name/value pair. For the remainder of this document, 'timestamps' will be elided for readability. It is also worth noting the name and value are binary values, although in many applications they are UTF8 serialized strings.

Timestamps can be anything you like, but microseconds since 1970 is a convention. Whatever you use, it must be consistent across the application, otherwise earlier changes may overwrite newer ones.

## Column Families

A column family is a container for rows, analogous to the table in a relational system. Each row in a column family can be referenced by its key.

Column families have a configurable ordering applied to the columns within each row, which affects the behavior of the `get_slice` call in the thrift API. Out of the box ordering implementations include ASCII, UTF-8, Long, UUID (lexical or time), Date, combinations of these using `CompositeType`, and others.

## Rows

In Cassandra, each column family is stored in a separate file, and the file is sorted in row (i.e. key) major order. Related columns, those that you'll access together, should be kept within the same column family.

The row key is what determines what machine data is stored on. Thus, for each key you can have data from multiple column families associated with it. However, these are logically distinct, which is why the Thrift interface is oriented around accessing one `ColumnFamily` per key at a time. (TODO given this, is the following JSON more confusing than helpful?)

A JSON representation of the key -> column families -> column structure is

```
{
  "mccv": {
    "Users": {
      "emailAddress": { "name": "emailAddress", "value": "foo@bar.com" },
      "webSite": { "name": "webSite", "value": "http://bar.com" }
    },
    "Stats": {
      "visits": { "name": "visits", "value": "243" }
    }
  },
  "user2": {
    "Users": {
      "emailAddress": { "name": "emailAddress", "value": "user2@bar.com" },
      "twitter": { "name": "twitter", "value": "user2" }
    }
  }
}
```

Note that the key "mccv" identifies data in two different column families, "Users" and "Stats". This does not imply that data from these column families is related. The semantics of having data for the same key in two different column families is entirely up to the application. Also note that within the "Users" column family, "mccv" and "user2" have different column names defined. This is perfectly valid in Cassandra. In fact there may be a virtually unlimited set of column names defined, which leads to fairly common use of the column name as a piece of runtime populated data. This is unusual in storage systems, particularly if you're coming from the RDBMS world.

## Keyspaces

A keyspace is the first dimension of the Cassandra hash, and is the container for column families. Keyspaces are of roughly the same granularity as a schema or database (i.e. a logical collection of tables) in the RDBMS world. They are the configuration and management point for column families, and is also the structure on which batch inserts are applied.

## Super Columns

So far we've covered "normal" columns and rows. Cassandra also supports super columns: columns whose values are columns; that is, a super column is a (sorted) associative array of columns.

One can thus think of columns and super columns in terms of maps: A row in a regular column family is basically a sorted map of column names to column values; a row in a super column family is a sorted map of super column names to maps of column names to column values.

A JSON description of this layout:

```
{
  "mccv": {
    "Tags": {
      "cassandra": {
        "incubator": {"incubator":
"http://incubator.apache.org/cassandra/"},
        "jira": {"jira":
"http://issues.apache.org/jira/browse/CASSANDRA"}
      },
      "thrift": {
        "jira": {"jira": "http://issues.apache.org/jira/browse/THRIFT"}
      }
    }
  }
}
```

Here my column family is "Tags". I have two super columns defined here, "cassandra" and "thrift". Within these I have specific named bookmarks, each of which is a column.

Just like normal columns, super columns are sparse: each row may contain as many or as few as it likes; Cassandra imposes no restrictions.

## Range queries

Cassandra supports pluggable partitioning schemes with a relatively small amount of code. Out of the box, Cassandra provides the hash-based `RandomPartitioner` and a `ByteOrderedPartitioner`. `RandomPartitioner` gives you pretty good load balancing with no further work required. `ByteOrderedPartitioner` on the other hand lets you perform range queries on the keys you have stored, but requires choosing node tokens carefully or active load balancing. Systems that only support hash-based partitioning cannot perform range queries efficiently.

## Modeling your application

Unlike with relational systems, where you model entities and relationships and then just add indexes to support whatever queries become necessary, with Cassandra you need to think about what queries you want to support efficiently ahead of time, and model appropriately. Since there are no automatically-provided indexes, you will be much closer to one `ColumnFamily` per query than you would have been with tables:queries relationally. Don't be afraid to denormalize accordingly; Cassandra is much, much faster at writes than relational systems, without giving up speed on reads.

See the `CassandraLimitations` page for other things to keep in mind when designing a model.

## Attribution

Thanks to phatduckk and asenchi for coming up with examples, text, and reviewing concepts.

DataModel (last edited 2012-03-22 21:01:20 by TylerHobbs)