

WTF is a SuperColumn?

An Intro to the Cassandra Data Model

by: Arin Sarkissian :: <http://arin.me> :: [@phatduckk](#)

For the last month or two the Digg engineering team has spent quite a bit of time looking into, playing with and finally deploying [Cassandra](#) in production. It's been a super fun project to take on - but even before the fun began we had to spend quite a bit of time figuring out Cassandra's data model... the phrase "WTF is a 'super column'" was uttered quite a few times. :)

If you're coming from an RDBMS background (which is almost everyone) you'll probably trip over some of the naming conventions while learning about Cassandra's data model. It took me and my team members at Digg a couple days of talking things out before we "got it". In recent weeks a [bikeshed](#) went down in the dev mailing list proposing a completely new naming scheme to alleviate some of the confusion. Throughout this discussion I kept thinking: "maybe if there were some decent examples out there people wouldn't get so confused by the naming." So, this is my stab at explaining Cassandra's data model; It's intended to help you get your feet wet & doesn't go into every single detail but, hopefully, it helps clarify a few things.

The Pieces

Let's first go thru the building blocks before we see how they can all be stuck together:

Column

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 a `column` represented in JSON-ish notation:

```
{ // this is a column
  name: "emailAddress",
  value: "arin@example.com",
  timestamp: 123456789
}
```

That's all it is. For simplicity sake let's ignore the timestamp. Just think of it as a name/value pair.

Also, it's worth noting is that the name and value are both binary (technically `byte[]`) and can be of any length.

SuperColumn

A `SuperColumn` is a tuple w/ a binary name & a value which is a map containing an unbounded number of `columns` - keyed by the `column`'s name. Keeping with the JSON-ish notation we get:

```
{ // this is a SuperColumn
  name: "homeAddress",
  // with an infinite list of Columns
  value: {
    // note the keys is the name of the Column
    street: {name: "street", value: "1234 x street", timestamp: 123456789},
    city: {name: "city", value: "san francisco", timestamp: 123456789},
    zip: {name: "zip", value: "94107", timestamp: 123456789},
  }
}
```

Column vs SuperColumn

Columns and SuperColumns are both a tuples w/ a name & value. The key difference is that a standard Column's value is a "string" and in a SuperColumn the value is a Map of Columns. That's the main difference... their values contain different types of data. Another minor difference is that SuperColumn's don't have a timestamp component to them.

Before We Get Rolling

Before I move on I wanna simplify our notation a couple ways: 1) ditch the timestamps from Columns & 2) pull the Columns' & SuperColumns' names component out so that it looks like a key/value pair. So we're gonna go from:

```
{ // this is a super column
  name: "homeAddress",
  // with an infinite list of columns
  value: {
    street: {name: "street", value: "1234 x street", timestamp: 123456789},
    city: {name: "city", value: "san francisco", timestamp: 123456789},
    zip: {name: "zip", value: "94107", timestamp: 123456789},
  }
}
```

to

```
homeAddress: {
  street: "1234 x street",
  city: "san francisco",
  zip: "94107",
}
```

Grouping 'Em

There's a single structure used to group both the Columns and SuperColumns...this structure is called a ColumnFamily and comes in 2 varieties Standard & Super.

ColumnFamily

A ColumnFamily is a structure that contains an infinite number of Rows. Huh, did you say Rows? Ya - rows :) To make it sit easier in your head just think of it as a table in an RDBMS.

OK - each Row has a client supplied (that means you) key & contains a map of Columns. Again, the keys in the map are the names of the Columns and the values are the Columns themselves:

```
UserProfile = { // this is a ColumnFamily
  phatduckk: { // this is the key to this Row inside the CF
    // now we have an infinite # of columns in this row
    username: "phatduckk",
    email: "phatduckk@example.com",
    phone: "(900) 976-6666"
  }, // end row
  ieure: { // this is the key to another row in the CF
    // now we have another infinite # of columns in this row
    username: "ieure",
    email: "ieure@example.com",
    phone: "(888) 555-1212"
    age: "66",
    gender: "undecided"
  },
}
```

Remember: for simplicity we're only showing the value of the Column but in reality the values in the map are the entire Column.

You can think of it as a HashMap/dictionary or associative array. If you start thinking that way then

you're are the right track.

One thing I want to point out is that there's no schema enforced at this level. The Rows do not have a predefined list of Columns that they contain. In our example above you see that the row with the key "ieure" has Columns with names "age" and "gender" whereas the row identified by the key "phatduckk" doesn't. It's 100% flexible: one Row may have 1,989 Columns whereas the other has 2. One Row may have a Column called "foo" whereas none of the rest do. This is the schemaless aspect of Cassandra.

A ColumnFamily Can Be Super Too

Now, a ColumnFamily can be of type Standard or Super.

What we just went over was an example of the standard type. What makes it standard is the fact that all the Rows contains a map of *normal* (aka not-Super) Columns... there's no SuperColumns scattered about.

When a ColumnFamily is of type Super we have the opposite: each Row contains a map of SuperColumns. The map is keyed with the name of each SuperColumn and the value is the SuperColumn itself. And, just to be clear, since this ColumnFamily is of type *Super*, there are no *Standard* ColumnFamily's in there. Here's an example:

```
AddressBook = { // this is a ColumnFamily of type Super
  phatduckk: { // this is the key to this row inside the Super CF
    // the key here is the name of the owner of the address book

    // now we have an infinite # of super columns in this row
    // the keys inside the row are the names for the SuperColumns
    // each of these SuperColumns is an address book entry
    friend1: {street: "8th street", zip: "90210", city: "Beverley Hills", state: "CA"},

    // this is the address book entry for John in phatduckk's address book
    John: {street: "Howard street", zip: "94404", city: "FC", state: "CA"},
    Kim: {street: "X street", zip: "87876", city: "Balls", state: "VA"},
    Tod: {street: "Jerry street", zip: "54556", city: "Cartoon", state: "CO"},
    Bob: {street: "Q Blvd", zip: "24252", city: "Nowhere", state: "MN"},
    ...
    // we can have an infinite # of ScuperColumns (aka address book entries)
  }, // end row
  ieure: { // this is the key to another row in the Super CF
    // all the address book entries for ieure
    joey: {street: "A ave", zip: "55485", city: "Hell", state: "NV"},
    William: {street: "Armpit Dr", zip: "93301", city: "Bakersfield", state: "CA"},
  },
}
```

Keyspace

A keyspace is the outer most grouping of your data. All your ColumnFamily's go inside a keyspace. Your keyspace will probably named after your application.

Now, a keyspace can have multiple ColumnFamily's but that doesn't mean there's an imposed relationship between them. For example: they're not like tables in MySQL... you can't join them. Also, just because *ColumnFamily_1* has a Row with key "phatduckk" that doesn't mean *ColumnFamily_2* has one too.

Sorting

OK - we've gone through what the various data containers are about but another key component of the data model is how the data is sorted. Cassandra is not queryable like SQL - you do not specify how you want the data sorted when you're fetching it (among other differences). The data is sorted as soon as you put it into the cluster and it always remains sorted! This is a tremendous performance boost for reads but in exchange for that benefit you're going to have to make sure to plan your data model in a such a way that you're able satisfy your access patterns.

Columns are always sorted within their Row by the Column's name. This is important so i'll say it again: Columns are always sorted by their name! How the names are compared depends on the ColumnFamily's *CompareWith* option. Out of the box you have the following options: *BytesType*, *UTF8Type*, *LexicalUUIDType*, *TimeUUIDType*, *AsciiType*, and *LongType*. Each of these options treats the Columns' name as a different data type giving you quite a bit of flexibility. For example: Using *LongType* will treat your Columns' names as a 64bit Longs. Let's try and clear this up by taking a look at some data before and after it's sorted:

```
// Here's a view of all the Columns from a particular Row in random order
// Cassandra would "never" store data in random order. This is just an example
// Also, ignore the values - they don't matter for sorting at all
{name: 123, value: "hello there"},
{name: 832416, value: "kjjkbcjkcbbd"},
{name: 3, value: "101010101010"},
{name: 976, value: "kjjkbcjkcbbd"}
```

So, given the fact that we're using the *LongType* option, these Columns will look like this when they're sorted:

```
<!--
ColumnFamily definition from storage-conf.xml
-->
<ColumnFamily CompareWith="LongType" Name="CF_NAME_HERE"/>

// See, each Column's name is treated as a 64bit long
// in effect, numerically ordering our Columns' by name
{name: 3, value: "101010101010"},
{name: 123, value: "hello there"},
{name: 976, value: "kjjkbcjkcbbd"},
{name: 832416, value: "kjjkbcjkcbbd"}
```

As you can see the Columns' names were compared as if they were 64bit Longs (aka: numbers that can get pretty big). Now, if we'd used another *CompareWith* option we'd end up with a different result. If we'd set *CompareWith* to *UTF8Type* our sorted Columns' names would be treated as a UTF8 encoded strings yielding a sort order like this:

```
<!--
ColumnFamily definition from storage-conf.xml
-->
<ColumnFamily CompareWith="UTF8Type" Name="CF_NAME_HERE"/>

// Each Column name is treated as a UTF8 string
{name: 123, value: "hello there"},
{name: 3, value: "101010101010"},
{name: 832416, value: "kjjkbcjkcbbd"},
{name: 976, value: "kjjkbcjkcbbd"}
```

The result is completely different!

This sorting principle applies to SuperColumns as well but we get an extra dimension to deal with: not only do we determine how the SuperColumns are sorted in a Row but we also determine how the Columns within each SuperColumn are sorted. The sort of the Columns within each SuperColumn is determined by the value of *CompareSubcolumnsWith*. Here's an example:

```
// Here's a view of a Row that has 2 SuperColumns in it.
// currently they're in some random order

{ // first SuperColumn from a Row
  name: "workAddress",
  // and the columns within it
  value: {
    street: {name: "street", value: "1234 x street"},
```

```

        city: {name: "city", value: "san francisco"},
        zip: {name: "zip", value: "94107"}
    }
},
{ // another SuperColumn from same Row
  name: "homeAddress",
  // and the columns within it
  value: {
    street: {name: "street", value: "1234 x street"},
    city: {name: "city", value: "san francisco"},
    zip: {name: "zip", value: "94107"}
  }
}
}

```

Now if we decided to set both *CompareSubcolumnsWith* & *CompareWith* to *UTF8Type* we'd have the following end result:

```

// Now they're sorted

{
  // this one's first b/c when treated as UTF8 strings
  { // another SuperColumn from same Row

    // This Row comes first b/c "homeAddress" is before "workAddress"
    name: "homeAddress",

    // the columns within this SC are also sorted by their names too
    value: {
      // see, these are sorted by Column name too
      city: {name: "city", value: "san francisco"},
      street: {name: "street", value: "1234 x street"},
      zip: {name: "zip", value: "94107"}
    }
  },
  name: "workAddress",
  value: {
    // the columns within this SC are also sorted by their names too
    city: {name: "city", value: "san francisco"},
    street: {name: "street", value: "1234 x street"},
    zip: {name: "zip", value: "94107"}
  }
}
}

```

I want to note that in the last example *CompareSubcolumnsWith* & *CompareWith* were set to *UTF8Type* but this doesn't have to be the case. You can mix and match the values of *CompareSubcolumnsWith* & *CompareWith* as necessary.

The last bit about sorting I want to mention is that you can write a custom class to perform the sorting. The sorting mechanism is pluggable... you can set *CompareSubcolumnsWith* and/or *CompareWith* to any fully-qualified class name as long as that class implements *org.apache.cassandra.db.marshall.IType* (aka you can write custom comparators).

Example Schema

Alrighty - Now we've got all the pieces of the puzzle so let's finally put 'em all together and model a simple blog application. We're going to model a simple app with the following specs:

- support a single blog
- we can have multiple authors
- entries contain title, body, slug & publish date
- entries can be associated with any # of tags
- people can leave comments but cant register: they enter profile info each time (just keeping it simple)
- comments have text, time submitted, commenter's name & commenter's name
- must be able to show all posts in reverse chronological order (newest first)

- must be able to show all posts within a given tag in reverse chronological order

Each of the following sections will describe a `ColumnFamily` that we're going to define in our app's *Keyspace*, show the xml definition, talk about why we picked the particular sort option(s) as well as display the data in the `ColumnFamily` w/ our JSON-ish notation.

Authors ColumnFamily

Modeling the *Authors* `ColumnFamily` is going to be pretty basic; we're not going to do anything fancy here. We're going to give each *Author* their own `Row` & key it by the *Author's* full name. Inside the `Rows` each `Column` is going to represent a single "profile" attribute for the *Author*.

This is an example of using each `Row` to represent an object... in this case an *Author* object. With this approach each `Column` will serve as an attribute. Super simple. I want to point out that since there's no "definition" of what `Columns` must be present within a `Row` we kinda sorta have a schemaless design.

We'll be accessing the `Rows` in this `ColumnFamily` via key lookup & will grab every `Column` with each get (ex: we won't ever be fetching the first 3 columns from the `Row` with key 'foo'). This means that we don't care how the `Columns` are sorted so we'll use *BytesType* sort options because it doesn't require any validation of the `Columns's` names.

```
<!--
ColumnFamily: Authors
We'll store all the author data here.

Row Key => Author's name (implies names must be unique)
Column Name: an attribute for the entry (title, body, etc)
Column Value: value of the associated attribute

Access: get author by name (aka grab all columns from a specific Row)

Authors : { // CF
  Arin Sarkissian : { // row key
    // and the columns as "profile" attributes
    numPosts: 11,
    twitter: phatduckk,
    email: arin@example.com,
    bio: "bla bla bla"
  },
  // and the other authors
  Author 2 {
    ...
  }
}
-->
<ColumnFamily CompareWith="BytesType" Name="Authors"/>
```

BlogEntries ColumnFamily

Again, this `ColumnFamily` is going to act as a simple key/value lookup. We'll be storing 1 entry per `Row`. Within that `Row` the `Columns` will just serve as attributes of the entry: title, body, etc (just like the previous example). As a small optimization we'll denormalize the tags into a `Column` as a comma separated string. Upon display we'll just split that `Column's` value to get a list of tags.

The key to each `Row` will be the entries slug. So whenever we want to grab a single entry we can simply look it up by its key (slug).

```
<!--
ColumnFamily: BlogEntries
This is where all the blog entries will go:

Row Key +> post's slug (the seo friendly portion of the uri)
```

Column Name: an attribute for the entry (title, body, etc)
 Column Value: value of the associated attribute

Access: grab an entry by slug (always fetch all Columns for Row)

fyi: tags is a denormalization... its a comma separated list of tags.
 im not using json in order to not interfere with our
 notation but obviously you could use anything as long as your app
 knows how to deal w/ it

```
BlogEntries : { // CF
  i-got-a-new-guitar : { // row key - the unique "slug" of the entry.
    title: This is a blog entry about my new, awesome guitar,
    body: this is a cool entry. etc etc yada yada
    author: Arin Sarkissian // a row key into the Authors CF
    tags: life,guitar,music // comma sep list of tags (basic denormalization)
    pubDate: 1250558004 // unixtime for publish date
    slug: i-got-a-new-guitar
  },
  // all other entries
  another-cool-guitar : {
    ...
    tags: guitar,
    slug: another-cool-guitar
  },
  scream-is-the-best-movie-ever : {
    ...
    tags: movie,horror,
    slug: scream-is-the-best-movie-ever
  }
}
-->
<ColumnFamily CompareWith="BytesType" Name="BlogEntries"/>
```

TaggedPosts ColumnFamily

Alright - here's where things get a bit interesting. This ColumnFamily is going to do some heavy lifting for us. It's going to be responsible for keeping our tag/entry associations. Not only is it going to store the associations but it's going to allow us to fetch all *BlogEntries* for a certain tag in pre-sorted order (remember all that sorting jazz we went thru?).

A design point I want to point out is that we're going have our app logic tag every *BlogEntry* with the tag "__notag__" (a tag I just made up). Tagging every *BlogEntry* with "__notag__" will allow us to use this ColumnFamily to also store a list of all *BlogEntries* in pre-sorted order. We're kinda cheating but it allows us to use a single ColumnFamily to serve "show me all recent posts" and "show me all recent posts tagged 'foo'".

Given this data model if an entry has 3 tags it will have a corresponding column in 4 Rows... 1 for each tag and one for the "__notag__" tag.

Since we're going to want to display lists of entries in chronological order we'll make sure each columns name is a [time UUID](#) and set the ColumnFamily's *CompareWith* to *TimeUUIDType*. This will sort the columns by time satisfying our "chronological order" requirement :) So doing stuff like "get the latest 10 entries tagged 'foo'" is going to be a super efficient operation.

Now when we want display the 10 most recent entries (on the front page, for example) we would:

1. grab the last 10 Columns in the Row w/ key "__notag__" (our "all posts" tag)
2. loop thru that set of Columns
3. while looping, we know the value of each Column is the key to a Row in the *BlogEntries* ColumnFamily
4. so we go ahead and use that to grab the Row for this entry from the *BlogEntries* ColumnFamily. this gives us all the data for this entry
5. one of the Columns from the *BlogEntries* Row we just grabbed is named "author" and the value is the

- key into the *Authors* ColumnFamily we need to use to grab that author's profile data.
6. at this point we've got the entry data and the author data on hand
 7. next we'll split the "tags" Columns value to get a list tags
 8. now we have everything we need to display this post (no comments yet - this aint the permalink page)

We can go through the same procedure above using any tag... so it works for "all entries" and "entries tagged 'foo'". Kinda nice.

```
<!--
  ColumnFamily: TaggedPosts
  A secondary index to determine which BlogEntries are associated with a tag

  Row Key => tag
  Column Names: a TimeUUIDType
  Column Value: row key into BlogEntries CF

  Access: get a slice of entries tagged 'foo'

  We're gonna use this CF to determine which blog entries to show for a tag page.
  We'll be a bit ghetto and use the string __notag__ to mean
  "don't restrict by tag". Each entry will get a column in here...
  this means we'll have to have #tags + 1 columns for each post.

  TaggedPosts : { // CF
    // blog entries tagged "guitar"
    guitar : { // Row key is the tag name
      // column names are TimeUUIDType, value is the row key into BlogEntries
      timeuuid_1 : i-got-a-new-guitar,
      timeuuid_2 : another-cool-guitar,
    },
    // here's all blog entries
    __notag__ : {
      timeuuid_1b : i-got-a-new-guitar,

      // notice this is in the guitar Row as well
      timeuuid_2b : another-cool-guitar,

      // and this is in the movie Row as well
      timeuuid_2b : scream-is-the-best-movie-ever,
    },
    // blog entries tagged "movie"
    movie: {
      timeuuid_1c: scream-is-the-best-movie-ever
    }
  }
-->
<ColumnFamily CompareWith="TimeUUIDType" Name="TaggedPosts"/>
```

Comments ColumnFamily

The last thing we need to do is figure out how to model the comments. Here we'll get to bust out some SuperColumns.

We'll have 1 Row per entry. The key to the Row will be the entries slug. Within each Row we'll have a SuperColumn for each comment. The name of the SuperColumns will be a UUID that we'll be applying the *TimeUUIDType* to. This will ensure that all our comments for an entry are sorted in chronological order. The Columns within each SuperColumn will be the various attributes of the comment (commenter's name, comment time etc).

So, this is pretty simple as well... nothing fancy.

```
<!--
  ColumnFamily: Comments
```

We store all comments here

Row key => row key of the BlogEntry
SuperColumn name: TimeUUIDType

Access: get all comments for an entry

```
Comments : {
  // comments for scream-is-the-best-movie-ever
  scream-is-the-best-movie-ever : { // row key = row key of BlogEntry
    // oldest comment first
    timeuuid_1 : { // SC Name
      // all Columns in the SC are attribute of the comment
      commenter: Joe Blow,
      email: joe@example.com,
      comment: you're a dumb douche, the godfather is the best movie ever
      commentTime: 1250438004
    },
    ... more comments for scream-is-the-best-movie-ever

    // newest comment last
    timeuuid_2 : {
      commenter: Some Dude,
      email: sd@example.com,
      comment: be nice Joe Blow this isnt youtube
      commentTime: 1250557004
    },
  },
  // comments for i-got-a-new-guitar
  i-got-a-new-guitar : {
    timeuuid_1 : { // SC Name
      // all Columns in the SC are attribute of the comment
      commenter: Johnny Guitar,
      email: guitardude@example.com,
      comment: nice axe dawg...
      commentTime: 1250438004
    },
  }
  ..
  // then more Super CF's for the other entries
}
-->
<ColumnFamily CompareWith="TimeUUIDType" ColumnType="Super"
  CompareSubcolumnsWith="ByteType" Name="Comments"/>
```

Woot!

That's it. Our little blog app is all modeled and ready to go. It's quite a bit to digest but in the end you end up with a pretty small chunk of XML you've gotta store in the *storage-conf.xml*:

```
<Keyspace Name="BloggyAppy">
  <!-- other keyspace config stuff -->

  <!-- CF definitions -->
  <ColumnFamily CompareWith="ByteType" Name="Authors"/>
  <ColumnFamily CompareWith="ByteType" Name="BlogEntries"/>
  <ColumnFamily CompareWith="TimeUUIDType" Name="TaggedPosts"/>
  <ColumnFamily CompareWith="TimeUUIDType" Name="Comments"
    CompareSubcolumnsWith="ByteType" ColumnType="Super"/>
</Keyspace>
```

Now all you need to do is figure out how to get the data in and out of Cassandra ;). That's all accomplished via the [Thrift Interface](#). The [API wiki page](#) does a decent job at explaining what the various endpoints do so I won't go into all those details. But, in general, you just compile the `cassandra.thrift` file and use the

generated code to access the various [endpoints](#). Alternatively you can take advantage of this [Ruby client](#) or this [Python client](#).

Alrighty... hopefully all that made sense & you finally understand WTF a `superColumn` is and can start building some awesome apps.