**A Quick Introduction to the Cassandra Data Model**

In a nutshell, Cassandra data model can be described as follows:

**1) Cassandra is based on a key-value model**

A database consists of *column families*. A column family is a set of key-value pairs. I know the terminology is confusing but so far it is just basic key-value model. Drawing an analogy with relational databases, you can think about column family as table and a key-value pair as a record in a table.

**2) Cassandra extends basic key-value model with two levels of nesting**

At the first level the value of a record is in turn a sequence of key-value pairs. These nested key-value pairs are called *columns* where key is the *name* of the column. In other words you can say that a record in a column family has a key and consists of columns. This level of nesting is mandatory – a record must contain at least one column (so in the first point above value of a record was an intermediate notion as value is actually a sequence of columns).

At the second level, which is arbitrary, the value of a nested key-value pair can be a sequence of key-value pairs as well. When the second level of nesting is presented, outer key-value pairs are called *super columns* with key being the name of the super column and inner key-value pairs are called *columns*.

**3) The names of both columns and super columns can be used in two ways: as names or as values (usually reference value).**

First, names can play the role of attribute names. For example, the name of a column in a record about User can be Email. That is how we used to think about columns in relational databases.

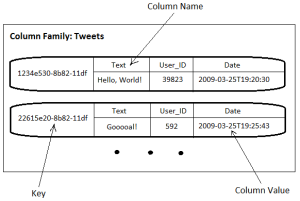
Second, names can also be used to store values! For example, column names in a record which represent Blog can be identifiers of the posts of this blog and the corresponding column values are posts themselves. You can really use column (or super column) names to store some values because (a) theoretically there is no limitation on the number of columns (or super columns) for any given record and (b) names are byte arrays so that you can encode any value in it.

**4) Columns and super columns are stored ordered by names.**

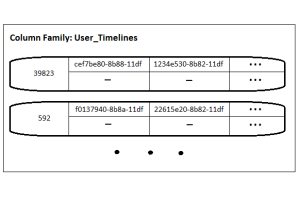
You can specify sorting behavior by defining how Cassandra treats the names of (super) columns (recall that a name is just an byte array). Name can be treated as Bytes Type, Long Type, Ascii Type, UTF8 Type, Lexical UUID Type, Time UUID Type.

So now you know everything you need to know. Let’s consider an classical :)example of Twitter database to demonstrate the points.

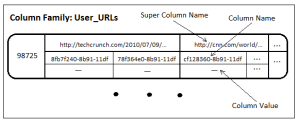
Column family Tweetscontains records representing tweets. The key of a record is of Time UUID type and generated when the tweet is received (we will use this feature in User\_Timelines column family below). The records consist of columns (no super columns here). Columns simply represent attributes of tweets. So it is very similar to how one would store it in a relational database.



The next example is User\_Timelines (i.e. tweets posted by a user). Records are keyed by user IDs (referenced by User\_ID columns in Tweets column family). User\_Timelines demonstrates how column names can be used to store values – tweet IDs in this case. The type of column names is defined as Time UUID. It means that tweets IDs are kept ordered by the time of posting. That is very useful as we usually want to show the last N tweets for a user. Values of all columns are set to an empty byte array (denoted “-”) as they are not used.



To demonstrate super columns let us assume that we want to collect statistics about URLs posted by each user. For that we need to group all the tweets posted by a user by URLs contained in the tweets. It can be stored using super columns as follows.



In User\_URLs the names of the super columns are used to store URLs and the names of the nested columns are the corresponding tweet IDs.

Important note: currently Cassandra automatically supports indexes for column names but does not support indexes for the names of super columns. In our example it means that you cannot efficiently retrieve/update tweet ids by URL.

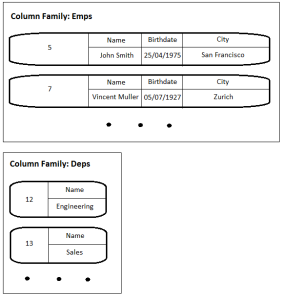
**[Update:** The above note is incorrect! It is subcolumn names that are not indexed inside super columns. Supercolumn names are always indexed. It is a great news as it enables the use-case of data denormalization to speed up queries. For more on this, find the first comment by Jonathan Ellis below. I cover denormalization use-cases in [my next post](http://maxgrinev.com/2010/07/12/do-you-really-need-sql-to-do-it-all-in-cassandra/).**]**

## Do You Really Need SQL to Do It All in Cassandra?

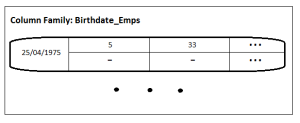
NoSQL database systems are designed for scalability. The down side of that is a primitive key-value data model and, as the name suggest, no support for SQL. It might sound like a serious limitation – how can I “select”, “join”, “group” and “sort” the data? This post explains how all these operations can be quite naturally and efficiently implemented in one of the most famous NoSQL system – Cassandra.

To understand this post you need to know the Cassandra data model. You can find [a quick introduction](http://maxgrinev.com/2010/07/09/a-quick-introduction-to-the-cassandra-data-model/) in my previous post. The power of the Cassandra data model is that it extends a basic key-value store with efficient data nesting (via columns and super columns). It means that you can read/update a column (or a super column) without retrieving the whole record. Below I describe how we can exploit data nesting to support various query operations.

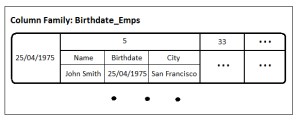
Let’s consider a basic example: departments and employees with one-to-many relationships respectively.  So we have two column families: Emps and Deps. In Emps employee IDs are used as keys and there are Name, Birthdate, and City columns. In Deps keys are department IDs and the single column is Name.

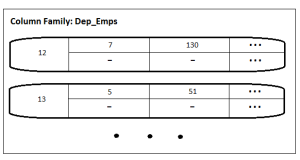


1) Select  
For example: select \* from Emps where Birthdate = '25/04/1975'  
To support this query we need to add one more column family named Birthdate\_Emps in which key is a date and column names are IDs of those employees that were born on the date. The values are not used here and can be an empty byte array (denoted “-”). Every time when a new employee is inserted/deleted into/from Emps we need to update Birthdate\_Emps. To execute the query we just need to retrieve all the columns for the key '25/04/1975' from Birthdate\_Emps.

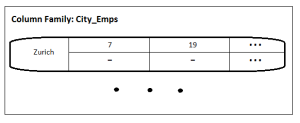


Notice that Birthdate\_Emps is essentially an index that allows us to execute the query very efficiently. And this index is scalable as it is distributed across Cassandra nodes. You can go even further to speed up the query by redundantly storing information about employees (i.e. employee’s columns from Emps) in Birthdate\_Emps. In this case employee IDs becomes names of super columns that contain corresponding employee columns.



2) Join  
For example: select \* from Emps e, Deps d where e.dep\_id = d.dep\_id  
What does join essentially mean? It constructs records that represent relationship between entities. Such relationships can be easily (and even more naturally) represented via nesting. To do that add column family Dep\_Emps in which key is a department ID and column names are IDs of the corresponding employees.  


3) Group By  
For example: select count(\*) from Emps group by City  
From implementation viewpoint Group By is very similar to select/indexing described above. You just need to add a column family City\_Emps with cities as keys and employee IDs as column names. In this case you will count the number of employees on retrieval. Or you can have a single column named count which value is the pre-calculated number of employees in the city.



4) Order By  
To keep data sorted in Cassandra you can use two mechanisms: (a) records can be sorted by keys using OrderPreservingPartitioner with range queries (more on this in [Cassandra: RandomPartitioner vs OrderPreservingPartitioner](http://ria101.wordpress.com/2010/02/22/cassandra-randompartitioner-vs-orderpreservingpartitioner/)). To keep nested data sorted you can use automatically supported ordering for column names.

To support all these operations we store redundant data optimized for each particular query. It has two implications:  
1) You must know queries in advance (i.e. no support for ad-hoc queries). However, typically in Web applications and enterprise OLTP applications queries are well known in advance, few in number, and do not change often. Read [Mike Stonebraker convincingly talking about that](http://cs-www.cs.yale.edu/homes/dna/papers/vldb07hstore.pdf).  BTW, Constraint Tree Schema, described in the latter paper, also exploits nesting to organize data for predefined queries.

2) We shift the burden from querying to updating because what we essentially do is supporting materialized views (i.e. pre-computed results of queries). But it makes a lot of sense in case of using Cassandra as Cassandra is very much optimized for updates (thanks to eventual consistency and “log-structured” storage borrowed from Google BigTable). So we can use fast updates to speed up query execution. Moreover, use-cases typical for social applications are proven to be only scalable with push-on-change model (i.e. preliminary data propagation via updates with simple queries – the approach taken in this post) in comparison with pull-on-demand model (i.e. data are stored normalized and combined by queries on demand – classical relational approach). On push-on-change versus pull-on-demand read [WHY ARE FACEBOOK, DIGG, AND TWITTER SO HARD TO SCALE?](http://highscalability.com/blog/2009/10/13/why-are-facebook-digg-and-twitter-so-hard-to-scale.html)

## Step 2: Basic Configuration

The Cassandra configuration files can be found in the conf directory of binary and source distributions. If you have installed Cassandra from a deb or rpm package, the configuration files will be located in /etc/cassandra.

### Step 2.1: Directories Used by Cassandra

If you've installed Cassandra with a deb or rpm package, the directories that Cassandra will use should already be created an have the correct permissions. Otherwise, you will want to check the following config settings.

In conf/cassandra.yaml you will find the following configuration options: data\_file\_directories (/var/lib/cassandra/data), commitlog\_directory (/var/lib/cassandra/commitlog), and saved\_caches\_directory (/var/lib/cassandra/saved\_caches). Make sure these directories exist and can be written to.

By default, Cassandra will write its logs in /var/log/cassandra/. Make sure this directory exists and is writeable, or change this line in conf/log4j-server.properies:

log4j.appender.R.File=/var/log/cassandra/system.log

### Step 2.2: Configure Memory Usage (Optional)

By default, Cassandra will allocate memory based on physical memory your system has, using somewhere between 1/4 and 1/2 of the available RAM.

If you want to specify how much memory Cassandra should use explicitly, edit conf/cassandra-env.sh, find the following lines, uncomment them, and change their values:

#MAX\_HEAP\_SIZE="4G"

#HEAP\_NEWSIZE="800M"

For MAX\_HEAP\_SIZE use as little as you can get away with. It's recommended to stay within 8G because much beyond that, the CMS GC pauses interfere with normal operations. For HEAP\_NEWSIZE use the number of cores \* 100 but don't exceed 800M. With too much allocated, [ParNew](http://wiki.apache.org/cassandra/ParNew) GC pauses become detrimental.

## Step 3: Start Cassandra

And now for the moment of truth, start up Cassandra by invoking 'bin/cassandra -f' from the command line[1](http://wiki.apache.org/cassandra/GettingStarted" \l "fnref-cac370d9c0a5bbfcc8ba5b7e5a5e742501e0181d). The service should start in the foreground and log gratuitously to the console. Assuming you don't see messages with scary words like "error", or "fatal", or anything that looks like a Java stack trace, then everything should be working.

Press "Control-C" to stop Cassandra.

If you start up Cassandra without the "-f" option, it will run in the background. You can stop the process by killing it, using 'pkill -f CassandraDaemon', for example.

## Step 4: Using cassandra-cli

bin/cassandra-cli is an interactive command line interface for Cassandra. You can alter the schema and interact with data using the cli. Run the following command to connect to your local Cassandra instance:

bin/cassandra-cli

You should see the following prompt, if successful:

Connected to: "Test Cluster" on 127.0.0.1/9160

Welcome to Cassandra CLI version 1.0.7

Type 'help;' or '?' for help.

Type 'quit;' or 'exit;' to quit.

[default@unknown]

You can access to the online help with 'help;' command. Commands are terminated with a semicolon (';') in the cli.

[default@unknown] help;

First, create a keyspace for your test.

[default@unknown] create keyspace DEMO

with placement\_strategy = 'org.apache.cassandra.locator.SimpleStrategy'

and strategy\_options = {replication\_factor:1};

f53dff10-5bd8-11e1-0000-915a024292eb

Waiting for schema agreement...

... schemas agree across the cluster

[default@unknown]

Don't forget to add a semicolon (';') at end of the command.

Second, authenticate to the DEMO keyspace:

[default@unknown] use DEMO;

Authenticated to keyspace: DEMO

[default@DEMO]

Third, create a Users column family:

[default@DEMO] create column family Users

... with key\_validation\_class = 'UTF8Type'

... and comparator = 'UTF8Type'

... and default\_validation\_class = 'UTF8Type';

[default@DEMO]

Now you can store data into Users column family:

[default@DEMO] set Users[1234][name] = scott;

Value inserted.

Elapsed time: 10 msec(s).

[default@DEMO] set Users[1234][password] = tiger;

Value inserted.

Elapsed time: 10 msec(s).

[default@DEMO]

You have inserted a row into the Users column family. The row key is '1234', and we set values for two columns in the row: 'name', and 'password'.

Now let's fetch the data you inserted:

[default@DEMO] get Users[1234];

=> (column=name, value=scott, timestamp=1350769161684000)

=> (column=password, value=tiger, timestamp=1350769245191000)

Returned 2 results.

Elapsed time: 67 msec(s).

[default@DEMO]

You can easily specify types other than UTF-8 when creating or updating a column family. See 'help update column family;' and 'help create column family;' for more details.

To be certain though, take some time to try out the examples in [CassandraCli](http://wiki.apache.org/cassandra/CassandraCli) before moving on Also, if you run into problems, Don't Panic, calmly proceed to [If Something Goes Wrong](http://wiki.apache.org/cassandra/GettingStarted" \l "if_something_goes_wrong).

* Users of recent Linux distributions and Mac OS X Snow Leopard should be able to start up Cassandra simply by untarring and invoking bin/cassandra -f with root privileges. Snow Leopard ships with Java 1.6.0 and does not require changing the JAVA\_HOME environment variable or adding any directory to your PATH. On Linux just make sure you have a working Java JDK package installed such as the openjdk-6-jdk on Ubuntu Lucid Lynx.

## Configuring Multinode Cluster

Now you have single working Cassandra node. It is a Cassandra cluster which has only one node. By adding more nodes, you can make it a multi node cluster.

Setting up a Cassandra cluster is almost as simple as repeating the above procedures for each node in your cluster. There are a few minor exceptions though.

Cassandra nodes exchange information about one another using a mechanism called Gossip, but to get the ball rolling a newly started node needs to know of at least one other, this is called a Seed. It's customary to pick a small number of relatively stable nodes to serve as your seeds, but there is no hard-and-fast rule here. Do make sure that each seed also knows of at least one other, remember, the goal is to avoid a chicken-and-egg scenario and provide an avenue for all nodes in the cluster to discover one another.

In addition to seeds, you'll also need to configure the IP interface to listen on for Gossip and Thrift, (listen\_address and rpc\_address respectively). Use a 'listen\_address that will be reachable from the listen\_address used on all other nodes, and a rpc\_address` that will be accessible to clients.

One other thing you need to care at multi node cluster is Token. Each node in the cluster owns a part of token range from 0 to 2^127-1. If the Nth node in the cluster has token value T(N), the node owns range from T(N-1)+1 to T(N). Cassandra decide nodes where a data should be stored based on the consistent mapping of the row key and token range (refer to [RandomPartitioner](http://wiki.apache.org/cassandra/RandomPartitioner), [ByteOrderedPartitioner](http://wiki.apache.org/cassandra/ByteOrderedPartitioner)).

The token can be assigned to node by initial\_token parameter in cassandra.yaml. The parameter is effective only at the first boot of the node. Once you boot a node, use 'nodetool move' command to change the assigned token. You need to specify appropriate initial\_token for each node to balance data load across the nodes. Here is a python script to calculate balanced tokens.

# Number of nodes in the cluster

num\_node = 4

for n in range(num\_node):

print int(2\*\*127 / num\_node \* n)

Once everything is configured and the nodes are running, use the bin/nodetool ring utility to verify a properly connected cluster. For example:

eevans@achilles:‾$ bin/nodetool -host 192.168.0.10 -p 7199 ring

Address DC Rack Status State Load Owns Token

127605887595351923798765477786913079296

192.168.0.10 DC1 r1 Up Normal 17.3 MB 25.00% 0

192.168.0.11 DC1 r1 Up Normal 17.4 MB 25.00% 42535295865117307932921825928971026432

192.168.0.12 DC1 r1 Up Normal 37.2 MB 25.00% 85070591730234615865843651857942052864

192.168.0.13 DC1 r1 Up Normal 24.55 MB 25.00% 127605887595351923798765477786913079296