DataModelv2

Under Construction

This page is still under construction, please click on the following link the see the current version:

Current Data Model Explanation

Please do not put this page on the main wiki page until completed. If you have any questions regarding this page then please contact me on ronald (at) sodeso dot nl

Introduction

Cassandra has a data model that is far different from normal relational databases, instead of having schemas, tables and column the data model consists of a structure of lists and maps.

When we start to look from the highest level we have clusters, clusters are physical machines operating together and forming a logical Cassandra instance. A cluster can contain several keyspaces. A keyspace is a group consisting of various ColumnFamilies, in general an application uses a single Keyspace. ColumnFamilies consists of rows which in turn consists of multiple values (Columns) per row.

A ColumnFamily comes in two flavors, the first one we already described, which is a ColumnFamily which has columns. The second ColumnFamily contains SuperColumns where the SuperColumns contain a list of Columns. If it becomes confusing just read on, it will become clearer.

So to recap, a ColumnFamily can contain either a list of Columns or a list of SuperColumns.

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).

```
-- REMARK: ---
From my experience, comparing concepts to those in a relational database pretty consistently confuses people.
This section intermixes the "structure of lists and maps" approach with relational db comparisons
"ColumnFamilies can be compared to a table in a relational database.", which is probably worse still.

I also think it's best to avoid referring to column families as containers.''

-- SOLUTION: ---
Changed the above description, there is no reference to a relation database anymore
```

Columns

A Column consists of a name, value and a timestamp.

```
--- REMARK: ---
This wording suggests that Tuple is a synonym for Column (which is not true).

--- SOLUTION: ---
Removed the synonym
```

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.

Model representation:

Column	
name	Binary
value	Binary
timestamp	i64

Data representation:

Column		
name	**value**	**timestamp**
"firstname"	"Ronald"	1270073054

ColumnFamily

A column family is a container for columns, analogous to the table in a relational system. You define column families in your storage-conf.xml file, and cannot modify them (or add new column families) without restarting your Cassandra process. A column family holds an ordered list of columns, which you can reference by the column name.

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, and UUID (lexical or time).

An example configuration of an Authors ColumnFamily using the UTF-8 sorting implementation would be:

<Keyspaces>
<Keyspace Name="Blog">
<ColumnFamily CompareWith="UTF8Type" Name="Authors"/>
</Keyspace>
</Keyspaces>

In Cassandra, each column family 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.

```
--- REMARK: ---
IMO, you should avoid implementation details unless they are really relevant, as it distracts, (i.e. "each
column family is stored in a separate file").

--- SOLUTION: ---
Nice remark, this should indeed be covered in a separate topic about the storage itself.
```

The row key is what determines what machine data is stored on. A key can be used for several column families at the same time, this does however not imply that the 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 client. Also, the columns can be different between the two column families. 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 relational database world. 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. On the other hand, a number of methods within the Thrift interface make use of this functionality, for example the batch_insert and batch_mutate make it possible to insert or modify data in multiple ColumnFamilies at the same time, as long as the key for the different ColumnFamilies are the same.

Model representation:

ColumnFamily	
key	**list**
binary	1 .. * Columns

Data representation:

ColumnFamily			
key	**Columns**		
1	**name**	**value**	**timestamp**
	"firstname"	"Ronald"	1270073054
	"lastname"	"Mathies"	1270073054
	"birthday"	"01/01/1978"	1270073054
2	**name**	**value**	**timestamp**
	"firstname"	"John"	1270084021
	"lastname"	"Steward"	1270084021
	"birthday"	"01/01/1982"	1270084021

As you can see in this example, we have a ColumnFamily containing two rows identified by the keys "1" and "2", each row has a number of columns, in this example we have the columns, *firstname*, *lastname* and *birthday* for each row.

SuperColumn

A SuperColumn is very similar to a ColumnFamily, it consists of a key and a list of Columns.

Model representation:

SuperColumn	
key	**list**
binary	1 .. * Columns

Data representation:

SuperColumn			
key	**Columns**		
1	**name**	**value**	**timestamp**
	"firstname"	"Ronald"	1270073054
	"lastname"	"Mathies"	1270073054
	"birthday"	"01/01/1978"	1270073054
2	**name**	**value**	**timestamp**
	"firstname"	"John"	1270084021
	"lastname"	"Steward"	1270084021
	"birthday"	"01/01/1982"	1270084021

As you can see it looks the same as a ColumnFamily, the only difference is the usage, a SuperColumn is used within a ColumnFamily, so it adds an extra layer in your data structure, instead of having only a row which consists of a key and a list of columns we can now have a row which consists of a key and a list of super columns which by itself has keys and per key a list of columns.

ColumnFamily containing SuperColumns

A ColumnFamily which contains SuperColumns isn't that much different from a ColumnFamily containing Columns, instead of having a row consisting of Columns we have rows consisting of SuperColumns.

The following example defines a super column family in your storage-conf.xml:

```
-- REMARK: ---
IMO, the term "SuperColumnFamily" should die.

-- SOLUTION: ---
And it's dead, i've removed it everywhere and rephrased the sentences to make it clear.
```

An example configuration of an Authors ColumnFamily using the UTF-8 sorting implementation would be:

<Keyspaces>
<Keyspace Name="Blog">
<ColumnFamily ColumnType="Super" CompareWith="UTF8Type" CompareSubcolumnsWith="UTF8Type" Name="Posts"/>
</Keyspace>
</Keyspaces>

The ColumnType tells Cassandra that the Posts columns family is a ColumnFamily containing SuperColumns, the CompareSubcolumnsWith attribute defines the sorting behavior of the keys of the super columns.

Model representation:

ColumnFamily	
key	**list**
binary	1 .. * SuperColumns

Data representation:

ColumnFamily	
Key	**SuperColumns**

"my-new-guitar"	**key**	**Columns**		
	post	**name**	**value**	**timestamp**
		"subject"	"My new guitar"	1270073054
		"body"	"a lot of text"	1270073054
		"created"	"01/01/2010"	1270073054
	tags	**name**	**value**	**timestamp**
		"tag0"	"Guitar"	1270084021
		"tag1"	"Instrument"	1270084021
"guitar-lessons"	**key**	**Columns**		
	post	**name**	**value**	**timestamp**
		"subject"	"Guitar lessons"	1270073054
		"body"	"a lot of text"	1270073054
		"created"	"01/03/2010"	1270073054
	tags	**name**	**value**	**timestamp**
		"tag0"	"Guitar"	1270084021
		"tag1"	"Instrument"	1270084021
		"tag2"	"Lessons"	1270084021

This example shows that we have two blog postings with the keys "my-new-guitar" and "guitar-lessons", per row we have two SuperColumns, one for the post information and one for the tag information. Per SuperColumn we have a number of Columns. For the post SuperColumn we have the *subject*, *body* and *created* columns. For the tags SuperColumn we have the *tag0*, *tag1* and sometimes a *tag2* Column.

Keyspaces

A keyspace is the first dimension of the Cassandra hash, and is the container for the ColumnFamilies. 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. In most cases you will have one Keyspace for an application.

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.

Arin Sarkissian of Digg has an excellent post detailing Cassandra's data model with highly illustrative examples. Ronald Mathies has some nice postings for when you want to use Cassandra with Java, if you have read this then you can start from part three Installing and using Apache Cassandra With Java Part 3 (Data model 2).

Furter reading

See the CassandraLimitations page for other things to keep in mind when designing a model. See the Thrift API page to see what methods are available and how to use them. See the StorageConfiguration page for more information on how to configure the storage-conf.xml file.

Author

Ronald Mathies