ONZRA

# Reporting with Cassandra

Tuesday January 3, 2013

**PREPARED BY:** Jose Avila III

6565 Sunset Blvd #518; Hollywood, CA 90028

T 323-476-1331 E JOSE@ONZRA.COM W WWW.ONZRA.COM

Quite commonly the need arises to analyze data over time and provide quick and easy access to those statistics via a dashboard. Providing access to real time stats can be difficult as the quantity of data being analyzed grows. This article will cover one potential solution for storing, and querying statistical data with Apache Cassandra for real time report generation.

Using an SQL database is commonly the first choice for this task, but this is only practical provided you are dealing with small amounts of data. At ONZRA, have repeatedly observed attempts to implement reporting dashboards with MySQL databases, many of which have been discouraging. These attempts have range from databases that have grown so large that queries take minutes to generate reports, to customers that spin up a hundreds of individual servers, each to handle a portion of their dataset., SQL has shown to be problem prone when scaled up for use with extremely large data sets. Apache Cassandra, on the other hand, is a much better fit for large scale operations.

# Why Cassandra?

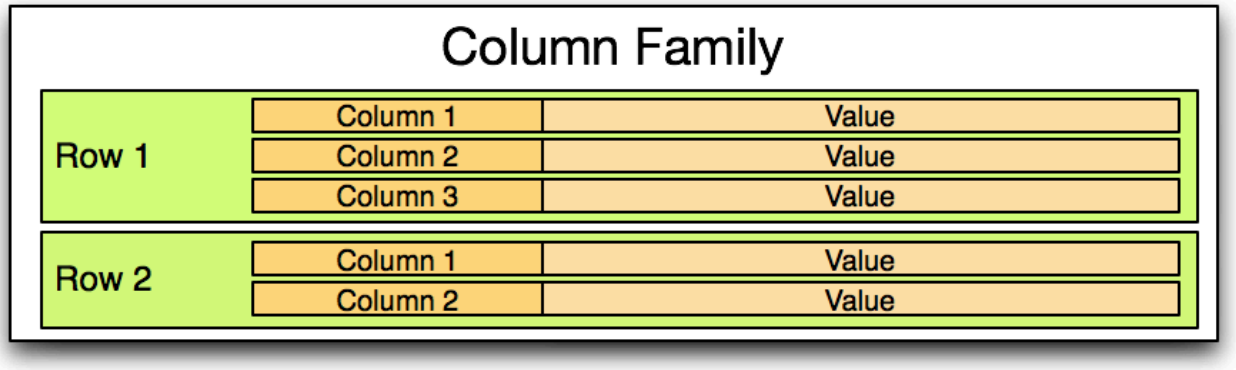
Cassandra is a NOSQL database that will scale horizontally as you add nodes to your cluster. Cassandra was designed to be non-centralized so there is no single bottle neck that would lead to the database cluster becoming over strained. It is extremely efficient with its reads. Given a key to lookup, Cassandra internally knows exactly which node in the cluster owns the “key” and from there can pull the row off of disk in very few hard drive seeks. With some planning, the data is stored how you want it retrieved; you can have reporting queries that are measured in milliseconds as opposed to seconds. Cassandra’s write path is also extremely efficient. All of these features make Cassandra is a good solution for storing reporting data for display in real time dashboards.

## Data Storage in Cassandra

In Cassandra, data is stored in Column Families. Within a column family you will have multiple rows each identified by a unique key, These rows will have multiple columns, each identified a name and containing value. As seen by figure 1, the data within can be thought of as a multi-leveled dictionary of data. For example {‘Row 1’:{‘Column 1’:‘Value’},...}.

*Figure 1. Representation of data stored within Cassandra.*

The columns within the rows are stored in order based on the name of the column. Apache Cassandra uses different "Comparators" to determine this sort order. There are various




comparators that are built in; for example, BytesType (the default), DateType, UTF8Type, LongType, etc. While ByteType compares the values based on the hexadecimal bytes in the string, the LongType compares values based on the 8 byte long value. As a result you would see different ordering for a row with columns named 12, 2, and 11 with a comparator of UTF8Type then you would with a comparator of LongType. Having the proper ordering of the columns is critical for the reporting implementation, otherwise you could get stats from outside of your requested time frames in your results.

Cassandra also has the ability to query the database for ranges of data. For example, if your columns are named 1, 2, 3, 4, and 5 in row R1 you can ask Cassandra to give you R1 columns 2-4. When Cassandra does the internal lookup of this range, it uses the column's comparator to determine what results to send you. Below is the format for a range query using CQL. *Note: In this example we are using the CQL Shell as the CLI does not have the ability to run range queries.*

```
$ cqlsh
cqlsh> use reporting;
cqlsh:reporting> select 2..4 from utilization where key='R1';
 2 | 3 | 4
```

```
---+---+---
 1 | 1 | 1
```

*Code Sample 1: Range Query*

Each column also has an associated value. Just as a column name has a “comparator”, values have “validators”. The validator identifies the data type of the value being stored. Cassandra has plenty of built in validators, for example: BytesType, UTF8Type, IntegerType, DateType, DoubleType, etc. One very unique validator we will use is the CounterColumnType. This type lets ApacheCassandra know the data should be validated as a “Counter”. Counters are a special column type that allow you to increment or decrement the value stored within that column in one smooth operation. In the example below you will see the creation and manipulation of the counters. *Note: this feature was added in Cassandra version 0.8.0.*

```
create column family counterCF with default_validation_class=CounterColumnType
INCR counterCF['key']['column'] BY 1;
```

*Code Sample2: Counters Schema / Counter Increment*

# Real World Example

Lets say, for example, that “Company A” has a blogging service. They provide their customers with credentials for accessing an API. This API has 4 methods: post/list, post/add, post/edit and post/delete. Company A limits what API calls you can make on a free account, so they want to provide a real time reporting dashboard for each customer. This allows the customers to view their API utilization and determine if they are approaching the maximum allowed use.

The company decides to provide a Method Utilization graph and a Method Comparison graph. The method utilization graph (Figure 2) displays how popular a particular method call was over a period of time. In this particular graph, the X-Axis represents the Time Frame for which the stats apply, whereas the Y-Axis refers to how often the event occurred.

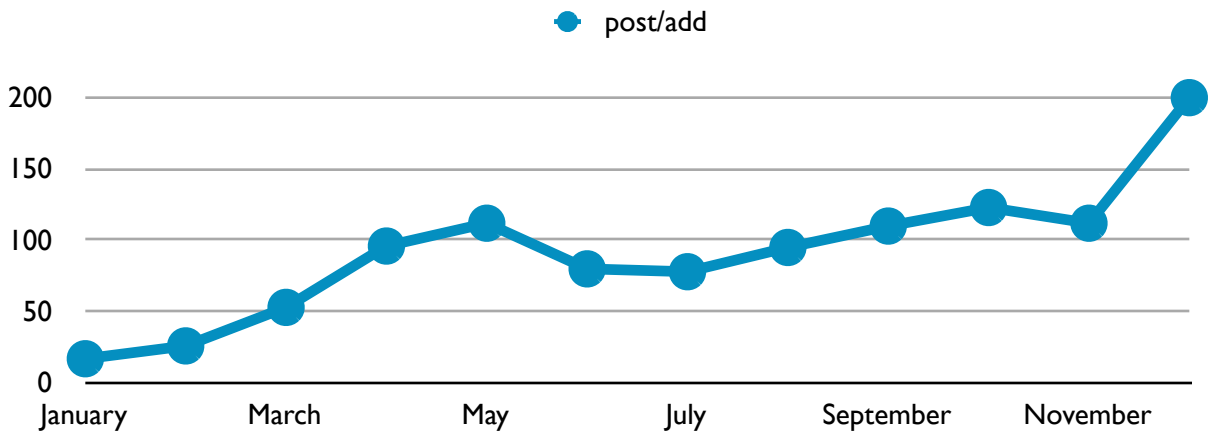



*Figure 2. Trending of "post/add" Method over time*

The Method Comparison graph (Figure 3) shows how often a particular method was called within a specified time frame.

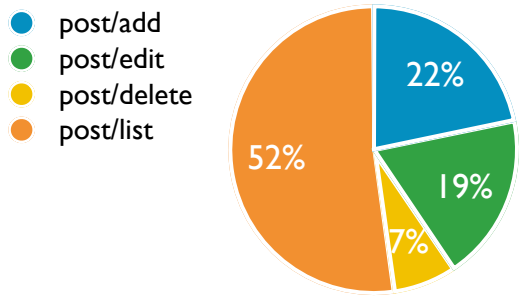



*Figure 3. API Method Comparison Graph for a specified time window*

Both of the graphs above are easily generated in milliseconds using Cassandra as the data backend, and both require only one simple lookup and one row hit to accomplish the task.

In this example we will say we want to trend at three different time frames: hourly, daily, and monthly. This would allow for the following use cases:

* Provide a method utilization graph at hourly groupings.

* Provide a method utilization graph at daily groupings.

* Provide a method utilization graph at monthly groupings.

* Provide a method comparison graph for a specific hour

* Provide a method comparison graph for a specific day

* Provide a method comparison graph for a specific month

# Technical Implementation

In this example all of the column families will be created within a reporting keyspace. the following cassandra-cli code sample describes how you can create that keyspace.

```
$ cassandra-cli -host localhost -port 9160
[default@unknown] CREATE KEYSPACE reporting;
[default@unknown] USE reporting;
```

*Code Sample 3: Creating the reporting keyspace.*

## Graph 1 - Method Utilization

For the Method Utilization graph (figure 2) we want to report on the API queries that are made over a range of specified time frames. In this case, the row key will be the item we are trending, the column name will be the time frame, and the column value will be the quantity of occurrences.

For the row key, we will use a key format of “<Account>:<Method>”. In the case of trending the “post/add” method for account 5 we would have a row key of “5:post/add”. All statistics for this particular account / method will exist within this row. *Note: In this particular example we are using “:” as a separator between the account and method; however, other separators could be used if need be.*

The column will represent the time frame the stat is applicable for, and will use a LongType comparator. The names of the columns will be in the format of “YYYYMMDDHH”, “YYYYMMDD”, and “YYYYMM” for hourly, daily, and monthly time frames.

```
$ cassandra-cli -host localhost -port 9160
[default@unknown] USE reporting;
[default@reporting] create column family utilization
with key_validation_class=UTF8Type
and comparator = LongType
```

```
and default_validation_class=CounterColumnType;
```

*Code Sample 4: Utilization Schema*

When logging an API call for Account 5 to the post/add Method on January 02, 2013 5PM a counter for each time frame would be incremented.

- Row “5:post/add” Column “2013010217” to increment the hourly counter
- Row “5:post/add” Column “20130102” to increment the daily counter
- Row “5:post/add” Column “201301” to increment the monthly counter

This allows us to run range queries for a series of timeframes. In order to run a report on the calls to “post/add” for January 1st through January 2nd broken down by hour, Cassandra would be queried for Row “5:post/add” Columns “2013010100” - “2013010223”. To run the same date range grouped by day Cassandra would be queried for Row “5:post/add” Columns “20130101” - “20130102”. By storing data in this means with the LongType comparator, there will never be overlapping time scopes when making our range queries.

Note that responses returned from Cassandra may be missing keys. For example if “post/add” was never called for customer 5 on January 1st, then we would be missing a column for that date all together. One should assume all time frames that do not have a column returned in the result set have a count of 0.

## Graph 2 - Method Comparison

For the Method Comparison graph (displayed in figure 3) we want to report on the API queries that are made during a specific frame by a specific account. This gives us the ability to present a pie chart where each slice is a method call. In this case, the row key will be the item we are trending, the column name will be the method, and the column value will be the quantity of occurrences.

For the row key, we will use a format of “<Account>:<Time Frame>”. Time frame will be in the format of “YYYYMMDDHH”, “YYYYMMDD”, and “YYYYMM” for hourly, daily, and monthly time frames. For example, the row key for account 5 covering the API calls made during

January of 2013 would be “5:201301”. The column names will be the API calls that were made. For example: “post/add” or “post/edit”. These columns will be implemented as counters, and the values will be the counts.

```
$ cassandra-cli -host localhost -port 9160
[default@unknown] USE reporting;
[default@reporting] create column family comparison
with key_validation_class=UTF8Type
and comparator = UTF8Type
and default_validation_class=CounterColumnType;
```

*Code Sample 5: Comparison Schema*

If an API call was logged for Account 5 to the post/add Method on January 02, 2013 5PM the following counters would be incremented:

- Row “5: 2013010217” Column “post/add” to increment the hourly counter
- Row “5: 20130102” Column “post/add” to increment the daily counter
- Row “5: 201301” Column “post/add” to increment the monthly counter

This allows us to easily query for a particular time frame and return the counts of all API calls that were made during that time frame.

## Real Time Inserts

Given the above design, as each API call comes in, 6 counters would get incremented. If reporting was done on fewer time scopes, the number of incremented counters would decrease and if reporting on additional time scopes there would be additional counters to get incremented for each logged API query. Below is an example of how data would be logged for each method using the Cassandra CLI.

```
$ cassandra-cli -host localhost -port 9160
[default@unknown] USE reporting;
[default@reporting] INCR utilization['5:post/add'][2013010205] BY 1;
[default@reporting] INCR utilization['5:post/add'][20130102] BY 1;
[default@reporting] INCR utilization['5:post/add'][201301] BY 1;
```

```
[default@reporting] INCR comparison['5:2013010205']['post/add'] BY 1;
[default@reporting] INCR comparison['5:20130102']['post/add'] BY 1;
[default@reporting] INCR comparison['5:201301']['post/add'] BY 1;
```

*Code Sample: Counter Increments*

Even though 6 inserts per API query seems like a lot, Cassandra is very write efficient due to the fact that records get written directly to in memory memtables.

## Querying The Data

Pulling records out of the database is efficient, as each report is a single API query that retrieves one individual key. Below is a python example for querying for the data to generate a method utilization graph. *Note: In this example we are using the CQL Shell as the CLI does not have the ability to run range queries.*

```
$ cqlsh
cqlsh> use reporting;
cqlsh:reporting> select 20130101..20130103 from utilization where key='5:post/add';
 20130101 | 20130103
----------+----------
        2 |        5
```

*Code Sample 6: Method Utilization Query*

Below is a sample query for retrieving the method comparison results.

```
 $ cassandra-cli -host localhost -port 9160
[default@unknown] USE reporting;
[default@reporting] GET comparison['5:2013010205'];
=> (counter=post/add, value=124)
=> (counter=post/delete, value=10)
=> (counter=post/edit, value=75)
=> (counter=post/list, value=210)
Returned 4 results.
```

*Code Sample 7: Method Comparison Query*

# Additional Insights

Results do not have to be logged to Cassandra immediately as the API requests occur. Since counters allow us to increment by a value of our choice, we can store counters locally in the application, then batch push them to Cassandra in their a background process at regular intervals. This technique can also be helpful if batch processing of logs with Hadoop or another system is already happening, since we can then pipe the summary data directly to Cassandra for reporting in the dashboard.

With slight tweaking of the above example we can apply this to many different scenarios. Some examples: We could trend hits to a web page by having a row key format of "<Page>" for trending PageUtilization and a row key format of <Time Frame> for PageComparison; or if you want to trend the popularity of file downloads by country you could use a row key format of "<Download>:<Country>" for DownloadUtilization and a row key format of "<Download>:<TimeFrame>" with Column name of "<Country>" for the DownloadComparison.

# In Conclusion

Using this reporting design, we can trend data utilization over time, compare data during specific time frames, as well as generate real time reporting data for presentation within a dashboard. The most important part is to plan ahead, decide what you want to report on, then let Cassandra do the rest. This can all be done because of Cassandra's proven, fault tolerant, scaleable nature.

# Jose Avila III

Jose Avila III is a Research Director at ONZRA. Jose has spent an astounding amount of time dealing with Enterprise Architecture, Development, and Security. He has architected and overseen development of globally deployed, fault tolerant infrastructures, as well as contributed to various technology research projects. Previously, Jose was a member of NeuStar's Software Architecture Review Board providing guidance on future application development and security. Jose has also lead many of their enterprise grade development projects including their Managed Internal DNS service that was globally deployed in some of the world's largest networks. Jose has also spoken at various conferences such as OARC, and Black Hat, presented at RSA, and has given lectures at several universities hoping to bring security awareness to future developers. Jose also enjoys tequila tasting and collecting.

# ONZRA

ONZRA focuses on highly specialized enterprise grade architecture and development consulting services. ONZRA has strategic partner relationships with many of the top engineering and security minds on the planet. These are the best of breed, industry movers and shakers. You have read their books and used their tools; you may have even seen them speak at conferences around the world like Black Hat, RSA, and OARC.  With services like software architecture consulting and enterprise development to vulnerability assessments, security design and training, ONZRA is a one stop shop for the absolute best of the best when it comes to technology problems that others can’t solve.