OpenTSDB Current Status

1) Hadoop Status
    - Login to Ambari dashboard: http://us0789lnxp.america.apci.com:8180/#/login
    - Restart ambari agent on each node if necessary: service ambari-agent restart
    - On the ‘Services’ tab, check status of all services across the cluster
    - Resolve any issues and restart all services affected
2) Hbase Status
    - Use ambari services tab to check status of HBase Master and Region Servers
    - Resolve any issues and restart HBase service
    - Ssh to a compute node and run the ‘hbase shell’ command
3) Review opentsdb install
    - Ssh to compute903 and check the status of opentsdb: service opentsdb status
    - The endpoint http://us0789lnxp.america.apci.com:4242/api/stats shows performance information about your opentsdb instance
4) Schema Definition
    a. Overview
    By default, OpenTSDB handles all the concerns related to the design of a schema for HBase, a column-oriented key-value data store which utilizes HDFS and MapReduce functionality. Issues like row-key selection, data locality and aggregation through hashing and bucketing are all handled under the hood.
    b. Requirements
    Every time series data point requires the following data:

    metric - A generic name for the time series such as sys.cpu.user, stock.quote or env.probe.temp.
    timestamp - A Unix/POSIX Epoch timestamp in seconds or milliseconds defined as the number of seconds that have elapsed since January 1st, 1970 at 00:00:00 UTC time. Only positive timestamps are supported at this time.
    value - A numeric value to store at the given timestamp for the time series. This may be an integer or a floating point value.
    tag(s) - A key/value pair consisting of a tagk (the key) and a tagv (the value). Each data point must have at least one tag.

    c. Tags
    OpenTSDB introduced the idea of 'tags'. Each time series still has a 'metric' name, but it's much more generic, something that can be shared by many unique time series.

    Instead, the uniqueness comes from a combination of tag key/value pairs that allows for flexible queries with very fast aggregations.

Metric	TimeStamp	Value	Tag1name=Tag1value
plants.baytown	1542000001	354.33	(tagname=H2PI5010C-PV)

Note: This is how we currently query from mongodb

We could add the hostname as an additional tag if we needed to reuse plant names

Metric	TimeStamp	Value	Tag1=value	Tag2=value
Plants.baytown	1542000001	354.33	tagname=H2PI5010C	server=US36393

d. Query

Metric	, tag1
Plants.baytown	tagname=H2PI5010C-PV

Now if we want the data for an individual tag, we can craft a query like sum:plants.baytown{tagname=H2PI5010C-PV}.

If we want all tags from all plants
sum:plant.baytown

the underlying data schema will store all of the plants.baytown time series next to each other so that aggregating the individual values is very fast and efficient.

OpenTSDB will automatically aggregate all of the time series for the metric in a query if no tags are given.

If one or more tags are defined, the aggregate will 'include all' time series that match on that tag, regardless of other tags.

5) Data Infusion
6) API
7) Front-end Application

Data Infusion

Backup

- Need review openTSDB export utilities to dump database to backup file
- Create script to export data regularly and add to cron job

Data Infusion

- Download script to manually backfill date ranges for instances of error handling, data corruption, etc.
- Get mysql up to date and read/write from mysql to openTSDB
- Batch Downloader that runs nightly as a cron job

API
- NodeJS read/write access

Frontend Application
- Benchmarking/timing utilities
- Trend Visualization

Batch Imports
Let's imagine that you have a cron job that crunches gigabytes of application logs every day or every hour to extract profiling data. For instance, you could be logging the time taken to process a request and your cron job would compute an average for every 30 second window. Maybe you're particularly interested in 2 types of requests handled by your application, so you'll compute separate averages for those requests, and an another average for every other request type. So your cron job may produce an output file that looks like this:

1288900000 42 foo
1288900000 51 bar
1288900000 69 other
1288900030 40 foo
1288900030 59 bar
1288900030 80 other
The first column is a timestamp, the second is the average latency for that 30 second window, and the third is the type of request we're talking about. If you run your cron job on a day worth of logs, you'll end up with 8640 such lines. In order to import those into OpenTSDB, you need to adjust your cron job slightly to produce its output in the following format:

myservice.latency.avg 1288900000 42 reqtype=foo
myservice.latency.avg 1288900000 51 reqtype=bar
myservice.latency.avg 1288900000 69 reqtype=other
myservice.latency.avg 1288900030 40 reqtype=foo
myservice.latency.avg 1288900030 59 reqtype=bar
myservice.latency.avg 1288900030 80 reqtype=other

Notice we're simply associating each data point with the name of a metric (myservice.latency.avg) and naming the tag that represents the request type. If each server has its own logs and you process them separately, you may want to add another tag to each line like the host=foo tag we saw in the previous section. This way you'll be able to plot the latency of

each server individually, in addition to your average latency across the board and/or per request type. In order to import a data file in the format above (metric timestamp value tags) simply run the following command:

./tsdb import your-file

If your data file is large, consider gzip'ing it first. This can be as simple as piping the output of your cron job to gzip -9 >output.gz instead of writing directly to a file. The import command is able to read gzip'ed files and it greatly helps performance for large batch imports.