HBase Schema

Data Table Schema

All OpenTSDB data points are stored in a single, massive table, named `tsdb` by default. This is to take advantage of HBase's ordering and region distribution. All values are stored in the `t` column family.

Row Key - Row keys are byte arrays comprised of an optional salt, the metric UID, a base timestamp and the UID for tagk/v pairs: `[salt]<metric_uid><timestamp><tagk1><tagv1>[...<tagkN><tagvN>]`. By default, UIDs are encoded on 3 bytes.

With salting enabled (as of OpenTSDB 2.2) the first byte (or bytes) are a hashed salt ID to better distribute data across multiple regions and/or region servers.

The timestamp is a Unix epoch value in seconds encoded on 4 bytes. Rows are broken up into hour increments, reflected by the timestamp in each row. Thus each timestamp will be normalized to an hour value, e.g. *2013-01-01 08:00:00*. This is to avoid stuffing too many data points in a single row as that would affect region distribution. Also, since HBase sorts on the row key, data for the same metric and time bucket, but with different tags, will be grouped together for efficient queries.

Some example unsalted row keys, represented as hex are:

```
00000150E22700000001000001
00000150E22700000001000001000002000004
00000150E22700000001000002
00000150E22700000001000003
00000150E23510000001000001
00000150E23510000001000001000002000004
00000150E23510000001000002
00000150E23510000001000003
00000150E24320000001000001
00000150E24320000001000001000002000004
00000150E24320000001000002
00000150E24320000001000003
```

where:

```
00000150E22700000001000001
'----''------''----''----'
metric  time   tagk  tagv
```

This represents a single metric but four time series across three hours. Note how there is one time series with two sets of tags:

```
00000150E22700000001000001000002000004
'----''------''----''----''----''----'
metric  time   tagk  tagv  tagk  tagv
```

Tag names (tagk) are sorted alphabetically before storage, so the "host" tag will always appear first in the row key/TSUID ahead of "owner".

Data Point Columns

By far the most common column are data points. These are the actual values recorded when data is sent to the TSD for storage.

Column Qualifiers - The qualifier is comprised of 2 or 4 bytes that encode an offset from the row's base time and flags to determine if the value is an integer or a decimal value. Qualifiers encode an offset from the row base time as well as the format and length of the data stored.

Columns with 2 byte qualifiers have an offset in seconds. The first 12 bits of the qualifier represent an integer that is a delta from the timestamp in the row key. For example, if the row key is normalized to

`1292148000` and a data point comes in for `1292148123`, the recorded delta will be `123`. The last 4 bits are format flags

Columns with 4 byte qualifiers have an offset in milliseconds. The first 4 *bits* of the qualifier will always be set to `1` or `F` in hex. The next 22 bits encode the offset in milliseconds as an unsigned integer. The next 2 bits are reserved and the final 4 bits are format flags.

The last 4 bits of either column type describe the data stored. The first bit is a flag that indicates whether or not the value is an integer or floating point. A value of 0 indicates an integer, 1 indicates a float. The last 3 bits indicate the length of the data, offset by 1. A value of `000` indicates a 1 byte value while `010` indicates a 2 byte value. The length must reflect a value of 1, 2, 4 or 8. Anything else indicates an error.

For example, `0100` means the column value is an 8 byte, signed integer. `1011` indicates the column value is a 4 byte floating point value So the qualifier for the data point at `1292148123` with an integer value of 4294967296 would have a qualifier of `0000011110110100` or `07B4` in hex.

Column Values - 1 to 8 bytes encoded as indicated by the qualifier flag.

Compactions

If compactions have been enabled for a TSD, a row may be compacted after it's base hour has passed or a query has run over the row. Compacted columns simply squash all of the data points together to reduce the amount of overhead consumed by disparate data points. Data is initially written to individual columns for speed, then compacted later for storage efficiency. Once a row is compacted, the individual data points are deleted. Data may be written back to the row and compacted again later.

> **Note**
>
> The OpenTSDB compaction process is entirely separate in scope and definition than the HBase idea of compactions.

Column Qualifiers - The qualifier for a compacted column will always be an even number of bytes and is simply a concatenation of the qualifiers for every data point that was in the row. Since we know each data point qualifier is 2 bytes, it's simple to split this up. A qualifier in hex with 2 data points may look like `07B407D4`.

Column Values - The value is also a concatenation of all of the individual data points. The qualifier is split first and the flags for each data point determine if the parser consumes 4 or 8 bytes

Annotations or Other Objects

A row may store notes about the timeseries inline with the datapoints. Objects differ from data points by having an odd number of bytes in the qualifier.

Column Qualifiers - The qualifier is on 3 or 5 bytes with the first byte an ID that denotes the column as a qualifier. The first byte will always have a hex value of `0x01` for annotations (future object types will have a different prefix). The remaining bytes encode the timestamp delta from the row base time in a manner similar to a data point, though without the flags. If the qualifier is 3 bytes in length, the offset is in seconds. If the qualifier is 5 bytes in length, the offset is in milliseconds. Thus if we record an annotation at `1292148123`, the delta will be `123` and the qualifier, in hex, will be `01007B`.

Column Values - Annotation values are UTF-8 encoded JSON objects. Do not modify this value directly. The order of the fields is important, affecting CAS calls.

Append Data Points

OpenTSDB 2.2 introduced the idea of writing numeric data points to OpenTSDB using the `append` method instead of the normal `put` method. This saves space in HBase by writing all data for a row in a single column, enabling the benefits of TSD compactions while avoiding problems with reading massive amounts of data back into TSDs and re-writing them to HBase. The drawback is that the schema is incompatible with regular data points and requires greater CPU usage on HBase region servers as they perform a read, modify, write operation for each value.

Row Key - Same as regular values.

Column Qualifier - The qualifier is always the object prefix `0x05` with an offset of 0 from the base time on two bytes. E.g. `0x050000`.

Column Values - Each column value is the concatenation of original data point qualifier offsets and values in the format `<offset1><value1><offset2><value2>...<offsetN><valueN>`. Values can appear in any order and are sorted at query time (with the option to re-write the sorted result back to HBase.).

UID Table Schema

A separate, smaller table called `tsdb-uid` stores UID mappings, both forward and reverse. Two columns exist, one named `name` that maps a UID to a string and another `id` mapping strings to UIDs. Each row in the column family will have at least one of three columns with mapping values. The standard column qualifiers are:

- `metrics` for mapping metric names to UIDs
- `tagk` for mapping tag names to UIDs
- `tagv` for mapping tag values to UIDs.

The `name` family may also contain additional meta-data columns if configured.

`id` Column Family

Row Key - This will be the string assigned to the UID. E.g. for a metric we may have a value of `sys.cpu.user` or for a tag value it may be `42`.

Column Qualifiers - One of the standard column types above.

Column Value - An unsigned integer encoded on 3 bytes by default reflecting the UID assigned to the string for the column type. If the UID length has been changed in the source code, the width may vary.

`name` Column Family

Row Key - The unsigned integer UID encoded on 3 bytes by default. If the UID length has been changed in the source code, the width may be different.

Column Qualifiers - One of the standard column types above OR one of `metrics_meta`, `tagk_meta` or `tagv_meta`.

Column Value - For the standard qualifiers above, the string assigned to the UID. For a `*_meta` column, the value will be a UTF-8 encoded, JSON formatted UIDMeta Object as a string. Do not modify the column value outside of OpenTSDB. The order of the fields is important, affecting CAS calls.

UID Assignment Row

Within the `id` column family is a row with a single byte key of `\x00`. This is the UID row that is incremented for the proper column type (metrics, tagk or tagv) when a new UID is assigned. The column values are 8 byte signed integers and reflect the maximum UID assigned for each type. On assignment, OpenTSDB calls HBase's atomic increment command on the proper column to fetch a new UID.

Meta Table Schema

This table is an index of the different time series stored in OpenTSDB and can contain meta-data for each series as well as the number of data points stored for each series. Note that data will only be written to this table if OpenTSDB has been configured to track meta-data or the user creates a TSMeta object via the API. Only one column family is used, the `name` family and currently there are two types of columns, the meta column and the counter column.

Row Key

This is the same as a data point table row key without the timestamp. E.g. `<metric_uid><tagk1><tagv1>[...<tagkN><tagvN>]`. It is shared for all column types.

TSMeta Column

These columns store UTF-8 encoded, JSON formatted objects similar to UIDMeta objects. The qualifier is always `ts_meta`. Do not modify these column values outside of OpenTSDB or it may break CAS calls.

Counter Column

These columns are atomic incrementers that count the number of data points stored for a time series. The qualifier is `ts_counter` and the value is an 8 byte signed integer.

Tree Table Schema

This table behaves as an index, organizing time series into a hierarchical structure similar to a file system for use with tools such as Graphite or other dashboards. A tree is defined by a set of rules that process a TSMeta object to determine where in the hierarchy, if at all, a time series should appear.

Each tree is assigned a Unique ID consisting of an unsigned integer starting with `1` for the first tree. All rows related to a tree are prefixed with this ID encoded as a two byte array. E.g. `\x00\x01` for UID `1`.

Row Key

Tree definition rows are keyed with the ID of the tree on two bytes. Columns pertaining to the tree definition, as well as the root branch, appear in this row. Definitions are generated by the user.

Two special rows may be included. They are keyed on `<tree ID>\x01` for the `collisions` row and `<tree ID>\x02` for the `not matched` row. These are generated during tree processing and will be described later.

The remaining rows are branch and leaf rows containing information about the hierarchy. The rows are keyed on `<tree ID><branch ID>` where the `branch ID` is a concatenation of hashes of the branch display names. For example, if we have a flattened branch `dal.web01.myapp.bytes_sent` where each branch name is separated by a period, we would have 3 levels of branching. `dal`, `web01` and `myapp`. The leaf would be named `bytes_sent` and links to a TSUID. Hashing each branch name in Java returns a 4 byte integer and converting to hex for readability yields:

- `dal` = x00x01x83x8F
- `web01` = x06xBCx4Cx55
- `myapp` = x06x38x7CxF5

If this branch belongs to tree `1`, the row key for `dal` would be `\x00\x01\x00\x01\x83\x8F`. The branch for `myapp` would be `\x00\x01\x00\x01\x83\x8F\x06\xBC\x4C\x55\x06\x38\x7C\xF5`. This schema allows for navigation by providing a row key filter using a prefix including the tree ID and current branch level and a wild-card to match any number of child branch levels (usually only one level down).

Tree Column

A Tree is defined as a UTF-8 encoded JSON object in the `tree` column of a tree row (identified by the tree's ID). The object contains descriptions and configuration settings for processing time series through the tree. Do not modify this object outside of OpenTSDB as it may break CAS calls.

Rule Column

In the tree row there are 0 or more rule columns that define a specific processing task on a time series. These columns are also UTF-8 encoded JSON objects and are modified with CAS calls. The qualifier id of the format `rule:<level>:<order>` where `<level>` is the main processing order of a rule in the set (starting at 0) and `order` is the processing order of a rule (starting at 0) within a given level. For example `rule:1:0` defines a rule at level 1 and order 0.

Tree Collision Column

If collision storage is enabled for a tree, a column is recorded for each time series that would have created a leaf that was already created for a previous time series. These columns are used to debug rule sets and only appear rin the collision row for a tree. The qualifier is of the format `tree_collision:<tsuid>` where the TSUID is a byte array representing the time series identifier. This allows for a simple `getRequest` call to determine if a particular time series did not appear in a tree due to a collision. The value of a colission column is the byte array of the TSUID that was recorded as a leaf.

Not Matched Column

Similar to collisions, when enabled for a tree, a column can be recorded for each time series that failed to match any rules in the rule set and therefore, did not appear in the tree. These columns only appear in the not matched row for a tree. The qualifier is of the format `tree_not_matched:<TSUID>` where the TSUID is a byte array representing the time series identifier. The value of a not matched column is the byte array of the TSUID that failed to match a rule.

Branch Column

Branch columns have the qualifier `branch` and contain a UTF-8 JSON encoded object describing the current branch and any child branches that may exist. A branch column may appear in any row except the collision or not matched columns. Branches in the tree definition row are the `root` branch and link to the first level of child branches. These links are used to traverse the heirarchy.

Leaf Column

Leaves are mappings to specific time series and represent the end of a hierarchy. Leaf columns have a qualifier format of `leaf:<TSUID>` where the TUID is a byte array representing the time series identifier. The value of a leaf is a UTF-8 encoded JSON object describing the leaf. Leaves may appear in any row other than the collision or not matched rows.

Rollup Tables Schema

As of OpenTSDB 2.4 is the concept of rollup and pre-aggregation tables. While TSDB does a great job of storing raw values as long as you want, querying for wide timespans across massive amounts of raw data can slow queries to a crawl and potentially OOM a JVM. Instead, individual time series can be rolled up (or downsampled) by time and stored as separate values that allow for scanning much wider timespans at a lower resolution. Additionally, for metrics with high cardinalities, pre-aggregate groups can be stored to improve query speed dramatically.

There are three types of rolled up data:

- Rollup - This is a downsampled value across time for a single time series. It's similar to using a downsampler in query where the time series may have a data point every minute but is downsampled to a data point every hour using the `sum` aggregation. In that case, the resulting rolled up value is the sum of 60 values. E.g. if the value for each 1 minute data point is `1` then the resulting rollup value would be `60`.
- Pre-Aggregate - For a metric with high cardinality (many unique tag values), scanning for all of the series can be costly. Take a metric `system.interface.bytes.out` where there are 10,000 hosts spread across 5 data centers. If users often look at the total data output by data center (the query would look similar to aggregation = sum and data_center = *) then pre-calculating the sum across each data center would result in 5 data points being fetched per time period from storage instead of 10K. The resulting pre-aggregate would have a different tag set than the raw time series. In the example above, each series would likely have a `host` tag along with a `data_center` tag. After pre-aggregation, the `host` tag would be dropped, leaving only the `data_center` tag.
- Rolled-up Pre-Aggregate - Pre-aggregated data can also be rolled up on time similar to raw time series. This can improve query speed for wide time spans over pre-aggregated data.

Configuration

TODO - Settle on a config. Rollup configs consist of a table name, interval span and rollup interval. Raw pre-aggs can be stored in the data table or rollup tables as well.

Pre-Aggregate Schema

In OpenTSDB's implementation, a new, users configurable tag is added to all time series when rollups are enabled. The default key is `_aggegate` with a value of `raw` or an aggregation function. The tag is used to differentiate pre-aggregated data from raw (original) values. Therefore pre-aggregated data is stored in the same manner as original time series and can either be written to the original data table or stored in a separate table for greater query performance.

Rollup Schema

Rolled up data must be stored in a separate table from the raw data as to avoid existing schema conflicts and to allow for more performant queries.

Row Key - The row key for rollups is in the same format as the original data table.

Column Qualifier - Columns are different for rolled up data and consist of `<aggregation_function>:<time offset><type + length>` where the aggregation function is an upper-case string consisting of the function name used to generate the rollup and time offset is an offset from the row base time and the type + length describes the column value encoding.

- Aggregation Function - This is the name of a function such as `SUM`, `COUNT`, `MAX` or `MIN`.
- Time Offset - This is an offset based on the rollup table config, generally on 2 bytes. The offset is not a specific number of seconds or minutes from the base, instead it's the index of an interval of an offset. For example, if the table is configured to store 1 day of data at a resolution of 1 hour per row, then the base timestamp of the row key will align on daily boundaries (on Unix epoch timestamps). Then there would be a potential of 24 offsets (1 for each hour in the day) for the row. A data point at midnight for the given day would have an offset of 0 whereas the 23:00 hour value would have an offset of 22. Since rollup timestamps are aligned to time boundaries, qualifiers can save a fair amount of space.
- Type and Length - Similar to the original data table, the last 4 bits of each offset byte array contains the encoding of the data value including it's length and whether or not it's a floating point value.

An example column qualifier for the daily 1 hour interval table looks like:

```
SUM:0010
'-''---'
agg offset
```

Where the aggregator is `SUM`, the offset is `1` and the length is 1 byte of an integer value.

Column Value - The values are the same as in the main data table.

TODOs - Some further work that's needed:

- Compactions/Appends - The current schema does not support compacted or append data types. These can be implemented by denoting a single column per aggregation function (e.g. `SUM`, `COUNT`) and storing the offsets and values in the column value similar to the main data table.
- Additional Data Types - Currently only numeric data points are written to the pre-agg and rollup tables. We need to support rolling up annotations and other types of data.