文档链接：<http://opentsdb.net/docs/build/html/index.html>

目录

# Documentation for OpenTSDB 2.3

Welcome to OpenTSDB 2.3, the scalable, distributed time series database. We recommend that you start with the [User Guide](http://opentsdb.net/docs/build/html/user_guide/index.html) then test your understanding with an [Installation](http://opentsdb.net/docs/build/html/installation.html) and read on the [HTTP API](http://opentsdb.net/docs/build/html/api_http/index.html) if you need to develop against it.

# Contents

* [What's New](http://opentsdb.net/docs/build/html/new.html)
  + [3.X (Planned)](http://opentsdb.net/docs/build/html/new.html#x-planned)
  + [2.4 (Planned)](http://opentsdb.net/docs/build/html/new.html#planned)
  + [2.3](http://opentsdb.net/docs/build/html/new.html#id1)
  + [2.2](http://opentsdb.net/docs/build/html/new.html#id2)
  + [2.1](http://opentsdb.net/docs/build/html/new.html#id3)
  + [2.0](http://opentsdb.net/docs/build/html/new.html#id4)
* [Installation](http://opentsdb.net/docs/build/html/installation.html)
  + [Runtime Requirements](http://opentsdb.net/docs/build/html/installation.html#runtime-requirements)
  + [Installation](http://opentsdb.net/docs/build/html/installation.html#id1)
  + [Upgrading from 1.x](http://opentsdb.net/docs/build/html/installation.html#upgrading-from-1-x)
  + [Upgrading from 2.x to a Later 2.x](http://opentsdb.net/docs/build/html/installation.html#upgrading-from-2-x-to-a-later-2-x)
  + [Downgrading](http://opentsdb.net/docs/build/html/installation.html#downgrading)
* [User Guide](http://opentsdb.net/docs/build/html/user_guide/index.html)
  + [Configuration](http://opentsdb.net/docs/build/html/user_guide/configuration.html)
  + [Writing Data](http://opentsdb.net/docs/build/html/user_guide/writing/index.html)
  + [Querying or Reading Data](http://opentsdb.net/docs/build/html/user_guide/query/index.html)
  + [Rollup And Pre-Aggregates](http://opentsdb.net/docs/build/html/user_guide/rollups.html)
  + [UIDs and TSUIDs](http://opentsdb.net/docs/build/html/user_guide/uids.html)
  + [Metadata](http://opentsdb.net/docs/build/html/user_guide/metadata.html)
  + [Trees](http://opentsdb.net/docs/build/html/user_guide/trees.html)
  + [GUI](http://opentsdb.net/docs/build/html/user_guide/guis/index.html)
  + [Plugins](http://opentsdb.net/docs/build/html/user_guide/plugins.html)
  + [Stats](http://opentsdb.net/docs/build/html/user_guide/stats.html)
  + [Definitions](http://opentsdb.net/docs/build/html/user_guide/definitions.html)
  + [Storage](http://opentsdb.net/docs/build/html/user_guide/backends/index.html)
  + [CLI Tools](http://opentsdb.net/docs/build/html/user_guide/cli/index.html)
  + [Utilities](http://opentsdb.net/docs/build/html/user_guide/utilities/index.html)
  + [Logging](http://opentsdb.net/docs/build/html/user_guide/logging.html)
  + [Troubleshooting](http://opentsdb.net/docs/build/html/user_guide/troubleshooting.html)
* [HTTP API](http://opentsdb.net/docs/build/html/api_http/index.html)
  + [Overview](http://opentsdb.net/docs/build/html/api_http/index.html#overview)
  + [Version 1.X to 2.x](http://opentsdb.net/docs/build/html/api_http/index.html#version-1-x-to-2-x)
  + [Serializers](http://opentsdb.net/docs/build/html/api_http/index.html#serializers)
  + [Authentication/Permissions](http://opentsdb.net/docs/build/html/api_http/index.html#authentication-permissions)
  + [Response Codes](http://opentsdb.net/docs/build/html/api_http/index.html#response-codes)
  + [Errors](http://opentsdb.net/docs/build/html/api_http/index.html#errors)
  + [Verbs](http://opentsdb.net/docs/build/html/api_http/index.html#verbs)
  + [API Versioning](http://opentsdb.net/docs/build/html/api_http/index.html#api-versioning)
  + [Query String Vs. Body Content](http://opentsdb.net/docs/build/html/api_http/index.html#query-string-vs-body-content)
  + [Compressed Requests](http://opentsdb.net/docs/build/html/api_http/index.html#compressed-requests)
  + [CORS](http://opentsdb.net/docs/build/html/api_http/index.html#cors)
  + [Documentation](http://opentsdb.net/docs/build/html/api_http/index.html#documentation)
  + [Deprecated API](http://opentsdb.net/docs/build/html/api_http/index.html#deprecated-api)
  + [API Endpoints](http://opentsdb.net/docs/build/html/api_http/index.html#api-endpoints)
* [Telnet Style API](http://opentsdb.net/docs/build/html/api_telnet/index.html)
  + [put](http://opentsdb.net/docs/build/html/api_telnet/put.html)
  + [rollup](http://opentsdb.net/docs/build/html/api_telnet/rollup.html)
  + [histogram](http://opentsdb.net/docs/build/html/api_telnet/histogram.html)
  + [stats](http://opentsdb.net/docs/build/html/api_telnet/stats.html)
  + [version](http://opentsdb.net/docs/build/html/api_telnet/version.html)
  + [help](http://opentsdb.net/docs/build/html/api_telnet/help.html)
  + [dropcaches](http://opentsdb.net/docs/build/html/api_telnet/dropcaches.html)
  + [diediedie](http://opentsdb.net/docs/build/html/api_telnet/diediedie.html)
* [Development](http://opentsdb.net/docs/build/html/development/index.html)
  + [Guidelines](http://opentsdb.net/docs/build/html/development/index.html#guidelines)
  + [Git Repository](http://opentsdb.net/docs/build/html/development/index.html#git-repository)
  + [Details](http://opentsdb.net/docs/build/html/development/index.html#details)
* [Additional Resources](http://opentsdb.net/docs/build/html/resources.html)
  + [Monitoring](http://opentsdb.net/docs/build/html/resources.html#monitoring)
  + [Docker Images](http://opentsdb.net/docs/build/html/resources.html#docker-images)
  + [Front Ends](http://opentsdb.net/docs/build/html/resources.html#front-ends)
  + [Utilities](http://opentsdb.net/docs/build/html/resources.html#utilities)
  + [Clients](http://opentsdb.net/docs/build/html/resources.html#clients)
  + [References to OpenTSDB](http://opentsdb.net/docs/build/html/resources.html#references-to-opentsdb)
  + [Statistical Analysis Tools](http://opentsdb.net/docs/build/html/resources.html#statistical-analysis-tools)

# User Guide

These pages serve as a user and administration guide. We highly recommend that you start with the **writing** and [Querying or Reading Data](http://opentsdb.net/docs/build/html/user_guide/query/index.html) sections to understand how OpenTSDB handles its core purpose of storing and serving time series information. Then follow up with the [Quick Start](http://opentsdb.net/docs/build/html/user_guide/quickstart.html) section to play around with getting some data into your OpenTSDB instance. Finally follow up with the other pages for details on the other features of OpenTSDB.

* [Configuration](http://opentsdb.net/docs/build/html/user_guide/configuration.html)
* [Writing Data](http://opentsdb.net/docs/build/html/user_guide/writing/index.html)
* [Querying or Reading Data](http://opentsdb.net/docs/build/html/user_guide/query/index.html)
* [Rollup And Pre-Aggregates](http://opentsdb.net/docs/build/html/user_guide/rollups.html)
* [UIDs and TSUIDs](http://opentsdb.net/docs/build/html/user_guide/uids.html)
* [Metadata](http://opentsdb.net/docs/build/html/user_guide/metadata.html)
* [Trees](http://opentsdb.net/docs/build/html/user_guide/trees.html)
* [GUI](http://opentsdb.net/docs/build/html/user_guide/guis/index.html)
* [Plugins](http://opentsdb.net/docs/build/html/user_guide/plugins.html)
* [Stats](http://opentsdb.net/docs/build/html/user_guide/stats.html)
* [Definitions](http://opentsdb.net/docs/build/html/user_guide/definitions.html)
* [Storage](http://opentsdb.net/docs/build/html/user_guide/backends/index.html)
* [CLI Tools](http://opentsdb.net/docs/build/html/user_guide/cli/index.html)
* [Utilities](http://opentsdb.net/docs/build/html/user_guide/utilities/index.html)
* [Logging](http://opentsdb.net/docs/build/html/user_guide/logging.html)
* [Troubleshooting](http://opentsdb.net/docs/build/html/user_guide/troubleshooting.html)

# Writing Data

You may want to jump right in and start throwing data into your TSD, but to really take advantage of OpenTSDB's power and flexibility, you may want to pause and think about your naming schema（约束）. After you've done that, you can proceed to pushing data over the Telnet or HTTP APIs, or use an existing tool with OpenTSDB support such as 'tcollector'.

## Naming Schema

Many metrics administrators（监控项管理员） are used to supplying a single name for their time series. For example, systems administrators used to RRD-style systems may name their time series webserver01.sys.cpu.0.user. The name tells us that the time series is recording the amount of time in user space for cpu 0 on webserver01. This works great if you want to retrieve just the user time for that cpu core on that particular web server later on.

But what if the web server has 64 cores and you want to get the average time across all of them? Some systems allow you to specify a wild card（扩展面） such as webserver01.sys.cpu.\*.user that would read all 64 files and aggregate the results. Alternatively, you could record a new time series called webserver01.sys.cpu.user.all that represents the same aggregate but you must now write '64 + 1' different time series. What if you had a thousand web servers and you wanted the average cpu time for all of your servers? You could craft a wild card query like \*.sys.cpu.\*.user and the system would open all 64,000 files, aggregate the results and return the data. Or you setup a process to pre-aggregate the data and write it to webservers.sys.cpu.user.all.

OpenTSDB handles things a bit differently by introducing the idea of 'tags'. Each time series still has a 'metric' name, but it's much more generic（一般的）, something that can be shared by many unique time series. Instead, the uniqueness（唯一的） comes from a combination of tag key/value pairs（结合tag的key和value的数据对） that allows for flexible queries with very fast aggregations.

**Note**

Every time series in OpenTSDB must have at least one tag.

Take the previous example where the metric was webserver01.sys.cpu.0.user. In OpenTSDB, this may become sys.cpu.user host=webserver01, cpu=0. Now if we want the data for an individual core（一个核心core）, we can craft a query like sum:sys.cpu.user{host=webserver01,cpu=42}. If we want all of the cores, we simply drop the cpu tag and ask for sum:sys.cpu.user{host=webserver01}. This will give us the aggregated results for all 64 cores. If we want the results for all 1,000 servers, we simply request sum:sys.cpu.user. The underlying data schema will store all of the sys.cpu.user time series next to each other so that aggregating the individual values is very fast and efficient. OpenTSDB was designed to make these aggregate queries as fast as possible since most users start out at a high level, then drill down（训练） for detailed information.

### Aggregations

While the tagging system（标签系统） is flexible, some problems can arise if you don't understand the querying side of OpenTSDB, hence the need for some forethought（预先考虑）. Take the example query above: sum:sys.cpu.user{host=webserver01}. We recorded 64 unique time series for webserver01, one time series for each of the CPU cores. When we issued that query, all of the time series for metric sys.cpu.user with the tag host=webserver01 were retrieved, averaged, and returned as one series of numbers. Let's say the resulting average was 50 for timestamp 1356998400. Now we were migrating from another system to OpenTSDB and had a process that pre-aggregated（预先聚合） all 64 cores so that we could quickly get the average value and simply wrote a new time series sys.cpu.user host=webserver01. If we run the same query, we'll get a value of 100 at 1356998400. What happened? OpenTSDB aggregated all 64 time series and the pre-aggregated time series to get to that 100. In storage, we would have something like this:

sys.cpu.user host=webserver01 1356998400 50

sys.cpu.user host=webserver01,cpu=0 1356998400 1

sys.cpu.user host=webserver01,cpu=1 1356998400 0

sys.cpu.user host=webserver01,cpu=2 1356998400 2

sys.cpu.user host=webserver01,cpu=3 1356998400 0

...

sys.cpu.user host=webserver01,cpu=63 1356998400 1

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. With the query sum:sys.cpu.user{host=webserver01}, we would include sys.cpu.user host=webserver01,cpu=0 as well as sys.cpu.user host=webserver01,cpu=0,manufacturer=Intel, sys.cpu.userhost=webserver01,foo=bar and sys.cpu.user host=webserver01,cpu=0,datacenter=lax,department=ops. The moral（经验） of this example is: be careful with your naming schema.

### Time Series Cardinality（基数）

A critical aspect of any naming schema is to consider the cardinality of your time series. Cardinality is defined as the number of unique items in a set. In OpenTSDB's case, this means the number of items associated with a metric, i.e. all of the possible tag name and value combinations, as well as the number of unique metric names, tag names and tag values. Cardinality is important for two reasons outlined below.

**Limited Unique IDs (UIDs)**

There is a limited number of unique IDs to assign for each metric, tag name and tag value. By default there are just over 16 million possible IDs per type. If, for example, you ran a very popular web service and tried to track the IP address of clients as a tag, e.g. web.app.hits clientip=38.26.34.10, you may quickly run into the UID assignment limit as there are over 4 billion possible IP version 4 addresses. Additionally, this approach would lead to creating a very sparse（稀疏的，分散的） time series as the user at address 38.26.34.10 may only use your app sporadically（偶尔的，零星的）, or perhaps never again from that specific address.

The UID limit is usually not an issue, however. A tag value is assigned a UID that is completely disassociated from its tag name. If you use numeric identifiers for tag values, the number is assigned a UID once and can be used with many tag names. For example, if we assign a UID to the number 2, we could store timeseries with the tag pairs cpu=2, interface=2, hdd=2 and fan=2 while consuming only 1 tag value UID (2) and 4 tag name UIDs (cpu, interface, hdd and fan).

If you think that the UID limit may impact you, first think about the queries that you want to execute. If we look at the web.app.hits example above, you probably only care about the total number of hits to your service and rarely need to drill down to a specific IP address. In that case, you may want to store the IP address as an annotation. That way you could still benefit from low cardinality but if you need to, you could search the results for that particular IP using external scripts. (Note: Support for annotation queries is expected in a future version of OpenTSDB.)

If you desperately need more than 16 million values, you can increase the number of bytes that OpenTSDB uses to encode UIDs from 3 bytes up to a maximum of 8 bytes. This change would require modifying the value in source code, recompiling, deploying your customized code to all TSDs which will access this data, and maintaining this customization across all future patches and releases.

**Warning**

It is possible that your situation requires this value to be increased. If you choose to modify this value, you must start with fresh data and a new UID table. Any data written with a TSD expecting 3-byte UID encoding will be incompatible with this change, so ensure that all of your TSDs are running the same modified code and that any data you have stored in OpenTSDB prior to making this change has been exported to a location where it can be manipulated by external tools. See the TSDB.java file for the values to change.

**Query Speed**

Cardinality also affects query speed a great deal, so consider the queries you will be performing frequently and optimize your naming schema for those. OpenTSDB creates a new row per time series per hour. If we have one host with a single core that emits one time series sys.cpu.user host=webserver01,cpu=0 with data written every second for 1 day, that would result in 24 rows of data or 86,400 data points. However if we have 8 possible CPU cores for that host, now we have 192 rows and 691,200 data points. This looks good because we can get easily a sum or average of CPU usage across all cores by issuing a query like start=1d-ago&m=avg:sys.cpu.user{host=webserver01}. The query will iterate over all 192 rows and aggregate the data into a single time series.

However what if we have 20,000 hosts, each with 8 cores? Now we will have 3.8 million rows and 1.728 billion data points per day due to a high cardinality of host values. Queries for the average core usage on host webserver01 will be slower as it must pick out 192 rows out of 3.8 million. (However with OpenTSDB 2.2, you can use the explicit tags feature to specify cpu=\* and the fuzzy filter will kick in to help skip those unnecessary rows quicker.)

The benefits of this schema are that you have very deep granularity in your data, e.g., storing usage metrics on a per-core basis. You can also easily craft a query to get the average usage across all cores an all hosts: start=1d-ago&m=avg:sys.cpu.user. However queries against that particular metric will take longer as there are more rows to sift through. This is common amongst all databases and is not OpenTSDB's problem alone.

Here are some common means of dealing with cardinality:

**Pre-Aggregate** - In the example above with sys.cpu.user, you generally care about the average usage on the host, not the usage per core. While the data collector may send a separate value per core with the tagging schema above, the collector could also send one extra data point such as sys.cpu.user.avg host=webserver01. Now you have a completely separate timeseries that would only have 24 rows per day and with 20K hosts, only 480K rows to sift through. Queries will be much more responsive for the per-host average and you still have per-core data to drill down to separately.

**Shift to Metric** - What if you really only care about the metrics for a particular host and don't need to aggregate across hosts? In that case you can shift the hostname to the metric. Our previous example becomes sys.cpu.user.websvr01 cpu=0. Queries against this schema are very fast as there would only be 192 rows per day for the metric. However to aggregate across hosts you would have to execute multiple queries and aggregate outside of OpenTSDB. (Future work will include this capability).

### Naming Conclusion

When you design your naming schema, keep these suggestions in mind:

* Be consistent with your naming to reduce duplication. Always use the same case for metrics, tag names and values.
* Use the same number and type of tags for each metric. E.g. don't store my.metric host=foo and my.metric datacenter=lga.
* Think about the most common queries you'll be executing and optimize your schema for those queries
* Think about how you may want to drill down when querying
* Don't use too many tags, keep it to a fairly small number, usually up to 4 or 5 tags (By default, OpenTSDB supports a maximum of 8 tags).

## Data Specification

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.

### Timestamps

Data can be written to OpenTSDB with second or millisecond resolution. Timestamps must be integers and be no longer than 13 digits (See first [NOTE] below). Millisecond timestamps must be of the format 1364410924250where the final three digits represent the milliseconds. Applications that generate timestamps with more than 13 digits (i.e., greater than millisecond resolution) must be rounded to a maximum of 13 digits before submitting or an error will be generated.

Timestamps with second resolution are stored on 2 bytes while millisecond resolution are stored on 4. Thus if you do not need millisecond resolution or all of your data points are on 1 second boundaries, we recommend that you submit timestamps with 10 digits for second resolution so that you can save on storage space. It's also a good idea to avoid mixing second and millisecond timestamps for a given time series. Doing so will slow down queries as iteration across mixed timestamps takes longer than if you only record one type or the other. OpenTSDB will store whatever you give it.

**Note**

When writing to the telnet interface, timestamps may optionally be written in the form 1364410924.250, where three digits representing the milliseconds are placed after a period. Timestamps sent to the /api/put endpoint over HTTP must be integers and may not have periods. Data with millisecond resolution can only be extracted via the /api/query endpoint or CLI command at this time. See **query/index** for details.

**Note**

Providing millisecond resolution does not necessarily mean that OpenTSDB supports write speeds of 1 data point per millisecond over many time series. While a single TSD may be able to handle a few thousand writes per second, that would only cover a few time series if you're trying to store a point every millisecond. Instead OpenTSDB aims to provide greater measurement accuracy and you should generally avoid recording data at such a speed, particularly for long running time series.

### Metrics and Tags

The following rules apply to metric and tag values:

* Strings are case sensitive, i.e. "Sys.Cpu.User" will be stored separately from "sys.cpu.user"
* Spaces are not allowed
* Only the following characters are allowed: a to z, A to Z, 0 to 9, -, \_, ., / or Unicode letters (as per the specification)

Metric and tags are not limited in length, though you should try to keep the values fairly short.

### Integer Values

If the value from a put command is parsed without a decimal point (.), it will be treated as a signed integer. Integers are stored, unsigned, with variable length encoding so that a data point may take as little as 1 byte of space or up to 8 bytes. This means a data point can have a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). Integers cannot have commas or any character other than digits and the dash (for negative values). For example, in order to store the maximum value, it must be provided in the form 9223372036854775807.

### Floating Point Values

If the value from a put command is parsed with a decimal point (.) it will be treated as a floating point value. Currently all floating point values are stored on 4 bytes, single-precision, with support for 8 byte double-precision in 2.4 and later. Floats are stored in IEEE 754 floating-point "single format" with positive and negative value support. Infinity and Not-a-Number values are not supported and will throw an error if supplied to a TSD. See [Wikipedia](https://en.wikipedia.org/wiki/IEEE_floating_point)and the [Java Documentation](http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2.3) for details.

**Note**

Because OpenTSDB only supports floating point values, it is not suitable for storing measurements that require exact values like currency. This is why, when storing a value like 15.2 the database may return 15.199999809265137.

### Ordering

Unlike other solutions, OpenTSDB allows for writing data for a given time series in any order you want. This enables significant flexibility in writing data to a TSD, allowing for populating current data from your systems, then importing historical data at a later time.

### Duplicate Data Points

Writing data points in OpenTSDB is generally idempotent within an hour of the original write. This means you can write the value 42 at timestamp 1356998400 and then write 42 again for the same time and nothing bad will happen. However if you have compactions enabled to reduce storage consumption and write the same data point after the row of data has been compacted, an exception may be returned when you query over that row. If you attempt to write two different values with the same timestamp, a duplicate data point exception may be thrown during query time. This is due to a difference in encoding integers on 1, 2, 4 or 8 bytes and floating point numbers. If the first value was an integer and the second a floating point, the duplicate error will always be thrown. However if both values were floats or they were both integers that could be encoded on the same length, then the original value may be overwritten if a compaction has not occurred on the row.

In most situations, if a duplicate data point is written it is usually an indication that something went wrong with the data source such as a process restarting unexpectedly or a bug in a script. OpenTSDB will fail "safe" by throwing an exception when you query over a row with one or more duplicates so you can down the issue.

With OpenTSDB 2.1 you can enable last-write-wins by setting the tsd.storage.fix\_duplicates configuration value to true. With this flag enabled, at query time, the most recent value recorded will be returned instead of throwing an exception. A warning will also be written to the log file noting a duplicate was found. If compaction is also enabled, then the original compacted value will be overwritten with the latest value.

## Input Methods

There are currently three main methods to get data into OpenTSDB: Telnet API, HTTP API and batch import from a file. Alternatively you can use a tool that provides OpenTSDB support, or if you're extremely adventurous, use the Java library.

**Warning**

Don't try to write directly to the underlying storage system, e.g. HBase. Just don't. It'll get messy quickly.

**Note**

If the tsd.mode is set to ro instead of rw, the TSD will not accept data points through RPC calls. Telnet style calls will throw an exception and calls to the HTTP endpoint will return a 404 error. However it is still possible to write via the JAVA API when the mode is set to read only.

### Telnet

The easiest way to get started with OpenTSDB is to open up a terminal or telnet client, connect to your TSD and issue a put command and hit 'enter'. If you are writing a program, simply open a socket, print the string command with a new line and send the packet. The telnet command format is:

put <metric> <timestamp> <value> <tagk1=tagv1[ tagk2=tagv2 ...tagkN=tagvN]>

For example:

put sys.cpu.user 1356998400 42.5 host=webserver01 cpu=0

Each put can only send a single data point. Don't forget the newline character, e.g. \n at the end of your command.

**Note**

The Telnet method of writing is discouraged as it doesn't provide a way of determining which data points failed to write due to formatting or storage errors. Instead use the HTTP API.

### Http API

As of version 2.0, data can be sent over HTTP in formats supported by 'Serializer' plugins. Multiple, un-related data points can be sent in a single HTTP POST request to save bandwidth. See the **../api\_http/put** for details.

### Batch Import

If you are importing data from another system or you need to backfill historical data, you can use the import CLI utility. See **cli/import** for details.

## Write Performance

OpenTSDB can scale to writing millions of data points per 'second' on commodity servers with regular spinning hard drives. However users who fire up a VM with HBase in stand-alone mode and try to slam millions of data points at a brand new TSD are disappointed when they can only write data in the hundreds of points per second. Here's what you need to do to scale for brand new installs or testing and for expanding existing systems.

### UID Assignment

The first sticking point folks run into is ''uid assignment''. Every string for a metric, tag key and tag value must be assigned a UID before the data point can be stored. For example, the metric sys.cpu.user may be assigned a UID of 000001 the first time it is encountered by a TSD. This assignment takes a fair amount of time as it must fetch an available UID, write a UID to name mapping and a name to UID mapping, then use the UID to write the data point's row key. The UID will be stored in the TSD's cache so that the next time the same metric comes through, it can find the UID very quickly.

Therefore, we recommend that you 'pre-assign' UID to as many metrics, tag keys and tag values as you can. If you have designed a naming schema as recommended above, you'll know most of the values to assign. You can use the CLI tools **cli/mkmetric**, **cli/uid** or the HTTP API **../api\_http/uid/index** to perform pre-assignments. Any time you are about to send a bunch of new metrics or tags to a running OpenTSDB cluster, try to pre-assign or the TSDs will bog down a bit when they get the new data.

**Note**

If you restart a TSD, it will have to lookup the UID for every metric and tag so performance will be a little slow until the cache is filled.

### Random Metric UID Assignment

With 2.2 you can randomly assign UIDs to metrics for better region server write distribution. Because metric UIDs are located at the start of the row key, if a new set of busy metric are created, all writes for those metric will be on the same server until the region splits. With random UID generation enabled, the new metrics will be distributed across the key space and likely to wind up in different regions on different servers.

Random metric generation can be enabled or disabled at any time by modifying the tsd.core.uid.random\_metrics flag and data is backwards compatible all the way back to OpenTSDB 1.0. However it is recommended that you pre-split your TSDB data table according to the full metric UID space. E.g. if you use the default UID size in OpenTSDB, UIDs are 3 bytes wide, thus you can have 16,777,215 values. If you already have data in your TSDB table and choose to enable random UIDs, you may want to create new regions.

When generating random IDs, TSDB will try up to 10 times to assign a UID without a collision. Thus as the number of assigned metrics increases so too will the number of collisions and the likely hood that a data point may be dropped due to retries. If you enable random IDs and keep adding more metrics then you may want to increase the number of bytes on metric UIDs. Note that the UID change is not backwards compatible so you have to create a new table and migrate your old data.

### Salting

In 2.2 salting is supported to greatly increase write distribution across region servers. When enabled, a configured number of bytes are prepended to each row key. Each metric and combination of tags is then hashed into one "bucket", the ID of which is written to the salt bytes. Distribution is improved particularly for high-cardinality metrics (those with a large number of tag combinations) as the time series are split across the configured bucket count, thus routed to different regions and different servers. For example, without salting, a metric with 1 million series will be written to a single region on a single server. With salting enabled and a bucket size of 20, the series will be split across 20 regions (and 20 servers if the cluster has that many hosts) where each region has 50,000 series.

**Warning**

Because salting modifies the storage format, you cannot enable or disable salting at whim. If you have existing data, you must start a new data table and migrate data from the old table into the new one. Salted data cannot be read from previous versions of OpenTSDB.

To enable salting you must modify the config file parameter tsd.storage.salt.width and optionally tsd.storage.salt.buckets. We recommend setting the salt width to 1 and determine the number of buckets based on a factor of the number of region servers in your cluster. Note that at query time, the TSD will fire tsd.storage.salt.buckets number of scanners to fetch data. The proper number of salt buckets must be determined through experimentation as at some point query performance may suffer due to having too many scanners open and collating the results. In the future the salt width and buckets may be configurable but we didn't want folks changing settings on accident and losing data.

### Appends

Also in 2.2, writing to HBase columns via appends is now supported. This can improve both read and write performance in that TSDs will no longer maintain a queue of rows to compact at the end of each hour, thus preventing a massive read and re-write operation in HBase. However due to the way appends operate in HBase, an increase in CPU utilization, store file size and HDFS traffic will occur on the region servers. Make sure to monitor your HBase servers closely.

At read time, only one column is returned per row similar to post-TSD-compaction rows. However note that if the tsd.storage.repair\_appends is enabled, then when a column has duplicates or out of order data, it will be re-written to HBase. Also columns with many duplicates or ordering issues may slow queries as they must be resolved before answering the caller.

Appends can be enabled and disabled at any time. However versions of OpenTSDB prior to 2.2 will skip over appended values.

### Pre-Split HBase Regions

For brand new installs you will see much better performance if you pre-split the regions in HBase regardless of if you're testing on a stand-alone server or running a full cluster. HBase regions handle a defined range of row keys and are essentially a single file. When you create the tsdb table and start writing data for the first time, all of those data points are being sent to this one file on one server. As a region fills up, HBase will automatically split it into different files and move it to other servers in the cluster, but when this happens, the TSDs cannot write to the region and must buffer the data points. Therefore, if you can pre-allocate a number of regions before you start writing, the TSDs can send data to multiple files or servers and you'll be taking advantage of the linear scalability immediately.

The simplest way to pre-split your tsdb table regions is to estimate the number of unique metric names you'll be recording. If you have designed a naming schema, you should have a pretty good idea. Let's say that we will track 4,000 metrics in our system. That's not to say 4,000 time series, as we're not counting the tags yet, just the metric names such as "sys.cpu.user". Data points are written in row keys where the metric's UID comprises the first bytes, 3 bytes by default. The first metric will be assigned a UID of 000001 as a hex encoded value. The 4,000th metric will have a UID of 000FA0 in hex. You can use these as the start and end keys in the script from the [HBase Book](http://hbase.apache.org/book/perf.writing.html) to split your table into any number of regions. 256 regions may be a good place to start depending on how many time series share each metric.

TODO - include scripts for pre-splitting.

The simple split method above assumes that you have roughly an equal number of time series per metric (i.e. a fairly consistent cardinality). E.g. the metric with a UID of 000001 may have 200 time series and 000FA0 has about 150. If you have a wide range of time series per metric, e.g. 000001 has 10,000 time series while 000FA0 only has 2, you may need to develop a more complex splitting algorithm.

But don't worry too much about splitting. As stated above, HBase will automatically split regions for you so over time, the data will be distributed fairly evenly.

### Distributed HBase

HBase will run in stand-alone mode where it will use the local file system for storing files. It will still use multiple regions and perform as well as the underlying disk or raid array will let it. You'll definitely want a RAID array under HBase so that if a drive fails, you can replace it without losing data. This kind of setup is fine for testing or very small installations and you should be able to get into the low thousands of data points per second.

However if you want serious throughput and scalability you have to setup a Hadoop and HBase cluster with multiple servers. In a distributed setup HDFS manages region files, automatically distributing copies to different servers for fault tolerance. HBase assigns regions to different servers and OpenTSDB's client will send data points to the specific server where they will be stored. You're now spreading operations amongst multiple servers, increasing performance and storage. If you need even more throughput or storage, just add nodes or disks.

There are a number of ways to setup a Hadoop/HBase cluster and a ton of various tuning tweaks to make, so Google around and ask user groups for advice. Some general recommendations include:

* Dedicate a pair of high memory, low disk space servers for the Name Node. Set them up for high availability using something like Heartbeat and Pacemaker.
* Setup Zookeeper on at least 3 servers for fault tolerance. They must have a lot of RAM and a fairly fast disk for log writing. On small clusters, these can run on the Name node servers.
* JBOD for the HDFS data nodes
* HBase region servers can be collocated with the HDFS data nodes
* At least 1 gbps links between servers, 10 gbps preferable.
* Keep the cluster in a single data center

### Multiple TSDs

A single TSD can handle thousands of writes per second. But if you have many sources it's best to scale by running multiple TSDs and using a load balancer (such as Varnish or DNS round robin) to distribute the writes. Many users colocate TSDs on their HBase region servers when the cluster is dedicated to OpenTSDB.

### Persistent Connections

Enable keep-alives in the TSDs and make sure that any applications you are using to send time series data keep their connections open instead of opening and closing for every write. See **configuration** for details.

### Disable Meta Data and Real Time Publishing

OpenTSDB 2.0 introduced meta data for tracking the kinds of data in the system. When tracking is enabled, a counter is incremented for every data point written and new UIDs or time series will generate meta data. The data may be pushed to a search engine or passed through tree generation code. These processes require greater memory in the TSD and may affect throughput. Tracking is disabled by default so test it out before enabling the feature.

2.0 also introduced a real-time publishing plugin where incoming data points can be emitted to another destination immediately after they're queued for storage. This is disabled by default so test any plugins you are interested in before deploying in production.