In Documentation

Release 0.1

Stan Seibert

May 06, 2014

Contents

Natural Log (ln for short) is a time series database with a REST API. The database records time-varying scalars, arrays, and generic binary data at irregularly-spaced intervals, and supports time-oriented queries on that data. It is primarly intended for logging numeric data, rather than string messages like syslog.

Warning: Natural Log is currently practicing "Documentation Driven Development," so none of the code exists to do any of this yet.

The key feature of Natural Log is its support for *resampling queries*. In such queries, the server takes the raw, irregularly-spaced time series and returns a new time series with equally spaced intervals of (approximately) the requested size. This can be used to reduce a time series sampled very finely to one sampled much more coarsely without having to transfer a large amount of data to a plotting client. Moreover, two time series *(t,x)* and *(t,y)* can be more easily joined to create an *(x,y)* series if the time series first can be resampled to the same points in time.

Different resampling strategies are required for different applications, so Natural Log allows both the reduction (used to combine raw points) and interpolation (used between reduced points) strategies to be selected as part of the query. See *Reduction Strategies* and *Interpolation Strategies* for more information.

CHAPTER 1

Contents

1.1 Server Installation and Setup

The fastest way to install Natural Log and its dependencies is with virtualenv/pip:

```
virtualenv ln_env
source ln_env/bin/activate
cd ln_env
pip install ln
```

The included REST server is built with the Flask microframework.

1.1.1 Configuring the Server

Once Natural Log is installed, you need to create a server configuration file, which uses the JSON format. Here is an example `ln_local.json` config file for a local test server we will use elsewhere in the documentation:

```
{
    "host" : "127.0.0.1",
    "port" : 6283,
    "url_base" : "http://localhost:6823/",

    "resampling_intervals" : [1, 60, 3600, 86400],

    "storage" : {
                    "backend" : "memory"
                }
}
```

The required fields in the server configuration are:

Field Name	JSON Type	Description
`host`	String	IP address to which to bind
`port`	Number	Port number to which to listen
`url_base`	String	Base URL for this server. Used to construct URLs for responses.
`resampling_interva`	List of numbers	The server will resample data with these intervals (in seconds).
`storage`	Object	An object describing the storage backend to use. See *Storage Backends* for more details.

1.1.2 Storage Backends

Natural Log supports multiple storage backends as a way to experiment with different ways to store time series data. Each backend and its configuration parameters are described below.

In-memory Backend

This backend stores all data in memory, so **memory usage will grow without bound** and **all data is lost when the server is shutdown**. The in-memory backend is intended for testing and development purposes only. Do not ever use this backend on a production deployment! The storage configuration fields are:

Field Name	JSON Type	Description
`backend`	String	Set to `memory`

SQLite

This backend stores the data in an SQLite file. Data is saved to disk and SQLite's ACID guarantees make this a production-worthy option. It is not particularly fast.

Field Name	JSON Type	Description
`backend`	String	Set to `sqlite`
`filename`	String	Name of sqlite file on disk.

1.1.3 Starting the Server

Before starting the Natural Log server, we first need to initialize the storage backend:

```
$ ln-server -c ln_local.json init
Natural Log 0.1
Initializing "memory" storage backend...
Done.
```

Then we can start the server:

```
$ ln-server -c ln_local.json start
Natural Log 0.1
Opening "memory" storage backend...
Listening on 127.0.0.1:6283
Base URL is http://localhost:6283/
```

1.2 Data Series

A data series is a typed quantity that changes over time. Each data series has a name, a data type, a default reduction strategy, and a default interpolation strategy. Most data series are scalar types, such as integers and floating point values, but can also be array types or binary blobs. Data series may also optionally have a unit, a description, or application-specific metadata.

1.2.1 Data Series Attributes

Series Names

The name of a series can be any sequence of alphanumeric characters, underscores, and periods. We suggest the convention of organizing data series hierarchically with levels separated by periods. For example, one could organize a set of temperature sensors this way:

```
servers.node01.cpu_temp
servers.node01.disk_temp
servers.node02.cpu_temp
servers.node02.disk_temp
```

Data Types

The type of a data series is used to determine the storage format for data in memory and on disk. To save space, use a type appropriate to the range and precision of the data series.

Scalar Types

Data series with scalar types record a single number per update. The valid scalar types are:

```
int8
int16
int32
int64
float32
float64
```

Note that the number at the end of the type refers to its length in *bits*.

Array Types

An array type is an n-dimensional array of values with identical scalar type. Array types are a more compact way to record values that are typically accessed as a group, such as a list of numbered channels or a histogram. Array types are specified with a scalar type followed by a shape specification in brackets. The shape specification gives the size of each dimension separated by commas. Examples:

```
int16[100]
float32[2,4]
int32[10,10,2]
```

Note: Individual array elements cannot be recorded or queried separately. If you frequently need to access array elements one at a time, consider splitting the elements into separately named data series.

Blob Types

A generic binary "blob" type is provided for variable length binary data which cannot be reasonably represented as a scalar or an array. Blob data have very limited reduction and interpolation options, so scalar and array types should be used when practical. Individual blob values for a data series will be given unique server URLs. A blob type specification includes a MIME type which is returned to the HTTP client when the blob is retrieved. Examples:

```
blob:image/png
blob:application/pdf
blob:binary/octet-stream
```

Default Resampling Strategies

The handling of data points when resampling the time series in a query is controlled by a *reduction* and an *interpolation* strategy. The reduction strategy specifies how to combine multiple points when they fall into a time interval, and the interpolation strategy specifies how to use adjacent intervals to set the value of a time interval when it contains no points. These methods are described further in *Reduction Strategies* and *Interpolation Strategies*.

Although any strategy can be selected in the query (except in the case of blob types), many data series naturally fit only one pair of resampling strategies. For this reason, when creating a data series, a default reduction and interpolation strategy must be selected.

Series Description, Unit and Metadata

The description of a data series is an optional free-form text field that can be displayed in user-facing interfaces to Natural Log. The unit describes the physical unit (kg, m, deg F, etc) of the values recorded for the data series. The unit is not used for calculation purposes, but may also be shown to users in interfaces to Natural Log data.

The metadata field is another free-form text field that can be used for specific applications. Unlike the description field, the metadata field should not be shown directly to users. As an example, an array series that holds the bins of a histogram can use the metadata field to store the bin boundaries of the histogram for use by display applications.

1.2.2 Creating a Data Series

We assume that you have the Natural Log server running as described in *Configuring the Server*. The examples below use the Requests module for Python to make the REST API calls:

```
import requests
url_base = 'http://localhost:6283/'
```

To create a data series, we make a dictionary of the desired attributes and send it to the *create* URL. Here is an example of creating a temperature data series:

```
t = {
    'name' : 'node01.cpu_temp',
    'type' : 'float32',
    'reduction' : 'mean',
    'interpolation' : 'linear',
    'description' : 'Temperature of CPU in node01',
    'unit' : 'deg F'
}

r = requests.post(url_base + 'create/', data=t)
assert r.status_code == 200  # Check for success
```

An integer commit counter:

```
t = {
    'name' : 'commits',
    'type' : 'int8',
    'reduction' : 'sum',
    'interpolation' : 'middle',
    'description' : 'Number of commits to repository',
}

r = requests.post(url_base + 'create/', data=t)
assert r.status_code == 200  # Check for success
```

An array data series:

```
t = {
    'name' : 'channel_crc_errors',
    'type' : 'int32[100]',
    'reduction' : 'sum',
    'interpolation' : 'previous',
    'description' : 'Number of CRC errors for each data channel.',
}

r = requests.post(url_base + 'create/', data=t)
assert r.status_code == 200  # Check for success
```

And finally, a blob data series:

```
t = {
    'name' : 'cameras.entrance',
    'type' : 'blob:image/jpeg',
    'reduction' : 'middle',
    'interpolation' : 'none',
    'description' : 'Webcam aimed at lab entrance'
}

r = requests.post(url_base + 'create/', data=t)
assert r.status_code == 200  # Check for success
```

1.3 Recording Data

Natural Log is designed as an append-only database. The values for a given data source must be recorded in chronological order, and cannot be modified later. We may relax these restrictions eventually, but for now inserting, removing or modify points requires replaying points from one data source into another one.

The examples in this section assume the Natural Log server is running with the configuration described in *Configuring the Server* and the the data sources described in *create-source*. We continue to use the Requests module:

```
import requests
url_base = 'http://localhost:6283/'
```

1.3.1 Server-side Timestamping

Recording data is done by POST requests to the *data/[source name]/* URL. If no time is given, then the time of the request is used as the time of the data point. This avoids the need to worry about time synchronization on data collection clients.

Here is an example that populate the temperature data source with some random values by pausing between each point:

```
import time, random
post_url = url_base + 'data/node01.cpu_temp/'
for i in xrange(10):
    v = { 'value' : str(random.uniform(80, 120)) }
    r = requests.post(post_url, data=v)
    assert r.status_code == requests.codes.ok  # Check for errors
```

1.3.2 Client-side Timestamping

A timestamp can also be provided with the data point. The restriction is that each data source is append-only, so data must always be sent to the database in chronological order.

To avoid ambiguity, the interface of Natural Log describes times in ISO 8601 format, and times are always in UTC. Here is an example:

```
post_url = url_base + 'data/commits/'
for i in xrange(1, 15):
    v = {
        'time'  : '2013-07-%02d 12:00:31.503' % i,
        'value' : '1'
    }
    r = requests.post(post_url, data=v)
    assert r.status_code == requests.codes.ok  # Check for errors
```

1.3.3 Array Data Sources

Sources with array data types are updated in the same way, just with value set to a JSON string representation of a list. Here is an example with the CRC error data source:

```
import json
post_url = url_base + 'data/channel_crc_errors/'
channel_data = [ 1 for i in xrange(100) ]
v = { 'value' : json.dumps(channel_data) }
r = requests.post(post_url, data=v)
assert r.status_code == requests.codes.ok  # Check for errors
```

Multi-dimensional arrays can be represented with nested lists of lists.

1.3.4 Blob Data Sources

Blob data sources are handled slightly differently in order to transmit binary data. We have to use a multipart form encoding to hold the value field. In Requests, this is very straightforward:

```
# First fetch Google's Euler doodle image
r = requests.get('http://lh5.ggpht.com/Npa8E2JNHZHzrfQCutzmqqxD3WyQiiLibcaAvR4rR0hEs7LJDY-ahWf5SRBN5
assert r.status_code == requests.codes.ok
image = r.content

# Next, upload
post_url = url_base + 'data/cameras.entrance/'
import datetime
t = { 'time' : datetime.utcnow().isoformat(' ') }
v = { 'value' : image }
r = requests.post(post_url, data=t, files=v)
assert r.status_code = requests.codes.ok
```

1.4 Querying Data

The query interface is the key distinguishing feature of Natural Log. All queries return a *resampled, uniformly spaced time series* based upon the irregularly spaced raw points recorded in the database. Raw points can also be retrieved directly from the database, but that is not the primary usage model.

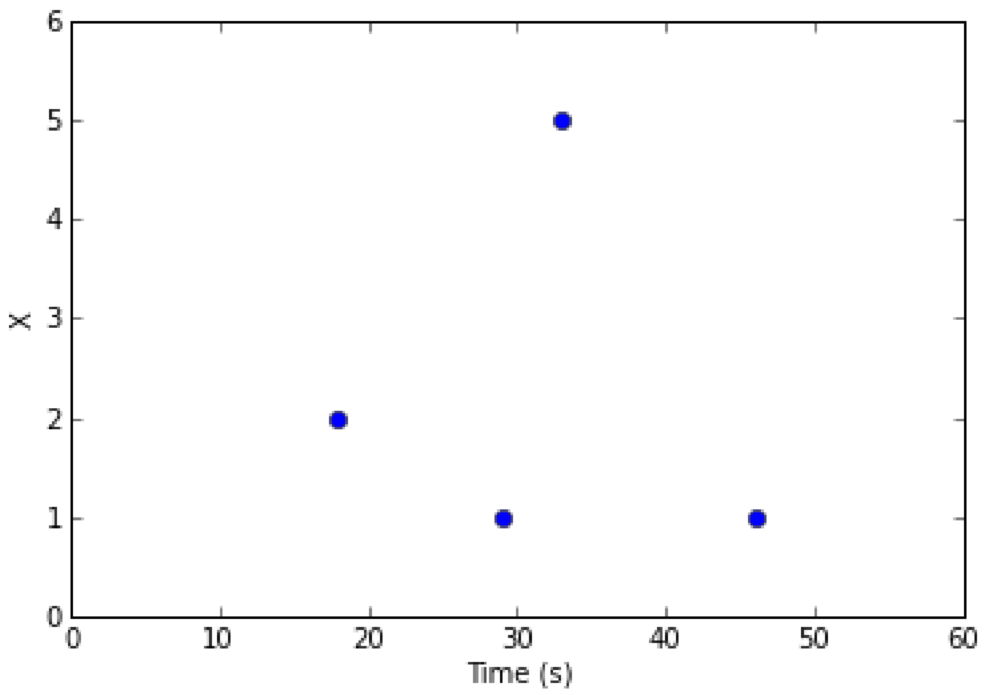
The basic components of a query are:

- List of data series names
- Optional resampling strategies for each data series. (If no resampling strategies are given, the default for the data series is used.)
- Desired time of first point
- Desired time of last point
- Desired number of points (must be >= 2)

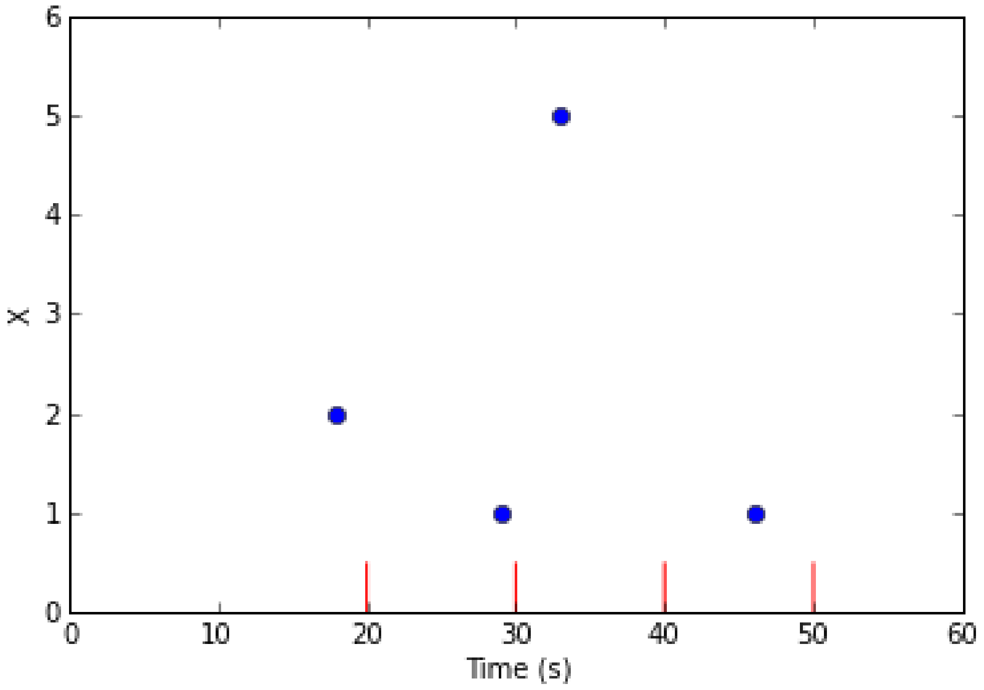
The result of the query is a 1D array of sample times and a 2D array with the resampled value of each data series (column) at each sample time (row). To enable storage backends to return results quickly, the actual start and end times, as well as the number of points, returned from the query may be slightly different than the requested values. How close the query result will match the request depends on the storage backend configuration.

1.4.1 How Resampling Works

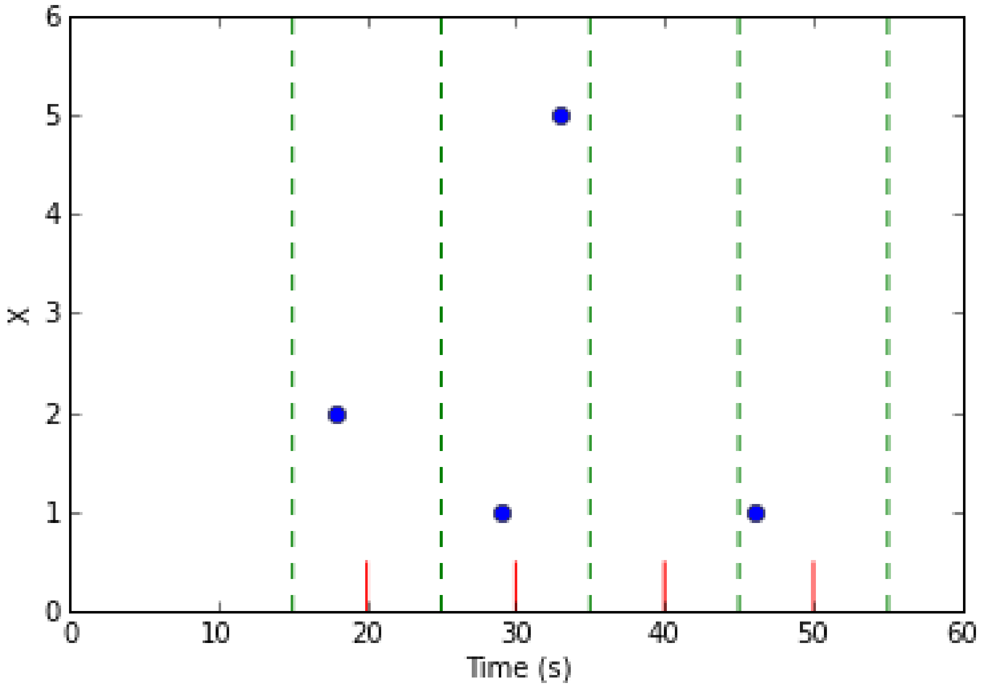
Let's suppose we have a time-ordered series of raw data points (floats, in this case) called *X*:



A query comes in asking for 4 data points from t=20 to t=50 seconds. First, the storage backend decides if it needs to adjust this request for performance reasons. Assuming it makes no changes to the request, it will need to decide the value of the data series at t=20, 30, 40, and 50 seconds, indicated by the red lines:



To estimate the value of the data series at the given times, the query engine groups the raw data points into equally-spaced bins around each of the evaluation times:



In this example, there are some bins with multiple raw points, and some bins with no points.

Warning: To ensure that raw data points can only appear in one bin, bin intervals are defined to be *half open*. If a raw point equals the value of the lower bin boundary, it is included in the bin, but if it equals the upper bin boundary, is is not included in the bin.

Natural Log uses the following rules to decide the value of each evaluation point:

1. If the bin contains 1 raw point, that value in the bin is the value of that raw point.
2. If the bin contains multiple raw points, the *reduction strategy* is used to combine the raw points into a single value.
3. If the bin contains 0 raw points, the *interpolation strategy* is used to generate a value from adjacent bins.

In the above example, suppose the reduction strategy is `mean` and the interpolation strategy is `linear`, the result of the query will be:

t	*x*
20	2.0
30	3.0
40	4.0
50	1.0

Reduction Strategies

A reduction strategy determines how raw values from a data series should be combined when they fall into the same interval during resampling.

`closest`

Select the point closest to the center of the time interval. This is the only allowed option for blob data.

`sum`

Add up the values of all points in the time interval.

`mean`

Add the values of all points in the time interval and divide by the number of points. Note that each point is weighted equally, regardless of the spacing of points in the time interval.

`min`

Report the minimum value in the interval.

`max`

Report the maximum value in the interval.

Interpolation Strategies

An interpolation strategy determines the value of the data series during resampling when no point falls into the interval.

`none`

Return a JSON `null` for entries where no data point is present. This is the only option allowed for blob data.

`zero`

Return zero (or an array of zeros) for entries where no data point is present. This option is not allowed for blob data.

`previous`

Return the value of the previous non-empty time interval. Note that if there is no such interval (due to no data points within the query range), then `null` is returned.

`linear`

Returns a linear interpolation between the non-empty time intervals preceeding and following this time interval. If non-empty time intervals cannot be found before and after this time interval, `null` is returned.

1.4.2 Making a Query

The examples in this section assume the Natural Log server is running with the configuration described in *Configuring the Server*. We continue to use the Requests module:

```
import requests
url_base = 'http://localhost:6283/'
```

For this example, let's make two new data series, `temperature` and `humidity`:

```
t = {
    'name' : 'temperature',
    'type' : 'float32',
    'reduction' : 'mean',
    'interpolation' : 'linear',
    'description' : 'Outside Temperature',
    'unit' : 'deg C'
}

r = requests.post(url_base + 'create/', data=t)
assert r.status_code == 200  # Check for success

t = {
    'name' : 'humidity',
    'type' : 'float32',
    'reduction' : 'mean',
    'interpolation' : 'linear',
    'description' : 'Relative humidity',
    'unit' : '%'
}
```

```
r = requests.post(url_base + 'create/', data=t)
assert r.status_code == 200  # Check for success
```

And let's fill them with some hourly data:

```
post_url = url_base + 'data/temperature/'
for i in xrange(1, 15):
    v = {
        'time'  : '2013-07-24 %02d:00:00' % i,
        'value' : 25.0 + 0.5 * i
    }
    r = requests.post(post_url, data=v)
    assert r.status_code == requests.codes.ok  # Check for errors

post_url = url_base + 'data/humidity/'
for i in xrange(1, 15):
    v = {
        'time'  : '2013-07-24 %02d:00:00' % i,
        'value' : 55.0 - 0.2 * i
    }
    r = requests.post(post_url, data=v)
    assert r.status_code == requests.codes.ok  # Check for errors
```

Now we can create a query for the database that resamples to 4 points and send it:

```
q = {
    'selectors' : ['temperature', 'humidity'],
    'first' : '2013-07-24 01:00:00'
    'last'  : '2013-07-24 15:00:00'
    'npoints' : 4,
}

r = requests.post(post_url, data=t)
assert r.status_code == requests.codes.ok  # Check for errors
result = r.json()
```

Depending on the particular backend settings, the contents of `result` could be something like:

```
{
    'times'  : ['2013-07-24 01:00:00',
                '2013-07-24 05:00:00',
                '2013-07-24 09:00:00',
                '2013-07-24 14:00:00'],
    'values' : [
        [ 25.75, 54.7],
        [ 27.25, 54.1],
        [ 29.25, 53.3],
        [ 31.25, 52.5]
        ]
}
```

If a series is an array type, the entries in the `values` array will be lists (possibly nested, if the array has multiple dimensions), and if the series is a blob type, the value will be a string containing a URL to retrieve the appropriate blob from the Natural Log server.

The default strategies for the series (`mean` and `linear`) were used in the above query, but we can also decide to override them in the selector using the forms `name:reduction` (leaving the interpolation strategy to be the default) or `name:reduction:interpolation` (overriding both strategies).

For example, this query:

```
q = {
    'selectors' : ['temperature:closest', 'humidity:min'],
    'first' : '2013-07-24 01:00:00'
    'last'  : '2013-07-24 15:00:00'
    'npoints' : 4,
}

r = requests.post(post_url, data=t)
assert r.status_code == requests.codes.ok  # Check for errors
result = r.json()
```

returns the following result:

```
{
    'times' : ['2013-07-24 01:00:00',
               '2013-07-24 05:00:00',
               '2013-07-24 09:00:00',
               '2013-07-24 14:00:00'],
    'values' : [
        [ 25.5, 54.6],
        [ 27.5, 53.8],
        [ 29.5, 53.0],
        [ 31.5, 52.2]
        ]
}
```

1.4.3 Continuous Queries

Rather than polling for new data, you can subscribe to changes and receive updates pushed from the server via server-sent events (SSE). In this case, the results are sent as a series of (time, value) pairs rather than a list of times and a list of values. This mode is selected by leaving the parameter *last* out of the query.

1.5 REST API Reference

1.5.1 `GET /`

Get a list of data series.

URL Options

None

Response

Format: *JSON*

Field name	Type	Description
`names`	List of strings	Names of all data series known to this server

1.5.2 `POST /create`

Create a new data series.

Request

Format: *Form encoded*

Field name	Type	Description
`name`	String	Name of this data series. Example: `servers.node01.cpu_temp`
`type`	String	Type of this data series. See *Data Types* for more information.
`reduction`	String	Name of reduction strategy. See *Reduction Strategies* for more information.
`interpolatio`	String	Name of interpolation strategy. See *Interpolation Strategies* for more information.
`unit`	String	Unit of measure. May be empty string.
`description`	String	Description of data series. May be empty string.
`metadata`	String	Application-specific metadata about this series. Unlike `description`, this is not intended to be shown to users.

1.5.3 `GET /data/[series name]`

Get the original values recorded for this data series, without resampling.

URL Options

Parameter Name	Description
`offset`	Index number of data point to start with.
`limit`	Maximum number of points to return. Server may impose a smaller maximum.

If `offset` and `limit` are not set in the URL, then by default the server will return the last recorded value for the data series.

Response (200)

Format: *JSON*

Field name	Type	Description
`times`	List of strings	List of ISO 8601 timestamps for all values.
`values`	List of ??	List of recorded values for this data series.
`resume`	Number (optional)	If maximum # of returned values reached, this is the value to pass to the `offset` parameter on the next `GET` call to continue.

Response: Failure (404)

Series does not exist.

1.5.4 `GET /data/[series name]/[index]`

Get a single raw value from a series. This is primarily used to fetch the contents of a blob with the mimetype set in the response.

URL Options

None.

Response (200)

Format: Raw binary w/ mimetype

Response: Failure (404)

Series or index number does not exist.

1.5.5 `POST /data/[series name]`

Record a new value for this data series.

Request

Format: *Form encoded or multipart form encoded*

Field name	Type	Description
`time`	String (optional)	ISO 8601 timestamp for value. If omitted, the server will use the time of the POST as the time of the value.
`value`	Various	JSON-encoded new value, either as a number for scalar data series, or a list of numbers (or a list of lists of numbers, etc) for array types.

If this data series is a blob type, the request should be multipart-encoded with `value` attached as a file. The MIME type of the encoded file in the POST request will be ignore in favore of the MIME type that was specified when the data series was created.

Response: Success (200)

Format: *JSON*

Success.

Field name	Type	Description
`index`	Number	ID number of newly recorded data point. Can be used as an offset to retrieve it later.
`url`	String (optional)	If a blob data series, the URL for the newly recorded binary data.

Response: Failure (404)

Series does not exist.

Response: Failure (400)

Format: *JSON*

Failure can occur if: * The timestamp for the data point is actually before the last recorded data point (`time_order`). * The POSTed value does not match the data type of the series or has the wrong dimensions for array types.

Field name	Type	Description
`type`	String	Type of failure: "time_order", "bad_type"
`msg`	String	A short explanation of the error.

1.5.6 `GET /data/[series name]/config`

Get the configuration information for this data series.

Response (200)

Format: *JSON*

Field name	Type	Description
`name`	String	Name of this data series. Example: `servers.node01.cpu_temp`
`type`	String	Type of this data series. See *Data Types* for more information.
`reduction`	String	Name of reduction strategy. See *Reduction Strategies* for more information.
`interpolation`	String	Name of interpolation strategy. See *Interpolation Strategies* for more information.
`unit`	String	Unit of measure. May be empty string.
`description`	String	Description of data series. May be empty string.

1.5.7 `POST /data/[series name]/config`

Modify the configuration information for this data series. Only the unit and description of the series can be changed this way.

Request

Format: *Form encoded*

Field name	Type	Description
`unit`	String	Unit of measure. May be empty string.
`description`	String	Description of data series. May be empty string.
`metadata`	String	Description of data series. May be empty string.

Response: Success (200)

Format: *JSON*

Success.

Field name	Type	Description
`result`	String	Contains `ok` on success.

Response: Failure (404)

Series does not exist.

Response: Failure (400)

Format: *JSON*

Failure can only happen if the form arguments have incorrect type.

Field name	Type	Description
`result`	String	Contains `fail` on failure.
`msg`	String	A short explanation of the error.

1.5.8 `GET /query`

Resample the selected data series and return the result. The query engine may return results with slightly different first and last times, as well as a different number of points. If `last` is omitted, the request is interpreted as a *continuous query* and the requested results and any future results are pushed via a persistent server-sent events (SSE) connection.

Field name	Type	Description
`selectors`	List of strings	Names of data series to query, with optional overide of reduction and interpolation strategy. See *Making a Query* for more details.
`first`	String	ISO 8601 timestamp of desired first resampling point.
`last`	String (Optional)	ISO 8601 timestamp of desired last resampling point.
`npoints`	Number	Desired number of data points (including first and last point)

Response: Success (200)

Format: *JSON*

Success.

Field name	Type	Description
`times`	List of strings	ISO 8601 timestamps of each resampled point.
`values`	List of lists	List of resampled points. See *Making a Query* for more details.

Response (continuous query): Success (200)

Format: *text/event-stream* containing *JSON*-encoded data

Success.

The `data` section of each SSE message contains the following in JSON:

Field name	Type	Description
`time`	String	ISO 8601 timestamps of resampled point.
`value`	List	List of resampled points. See *Making a Query* for more details.

Response: Failure (400)

Format: *JSON*

Failure can happen if the selectors are incorrect, `first` is not before `last`, or `npoints` is less than 2.

Field name	Type	Description
`msg`	String	A short explanation of the error.

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*