MetalPipe Documentation

Release 0.1

Zachary Ernst

Mar 29, 2019

Contents:

MetalPipe is a set of modules for building configuration-driven, efficient ETL pipelines with a minimum amount of custom code.

CHAPTER 1

Overview

MetalPipe is a package of classes and functions that help you write consistent, efficient, configuration-driven ETL pipelines in Python. It is open-source and as simple as possible (but not simpler).

This overview tells you why MetalPipe exists, and how it can help you escape from ETL hell.

1.1 Why is it?

Tolstoy said that every happy family is the same, but every unhappy family is unhappy in its own way. ETL pipelines are unhappy families.

Why are they so unhappy? Every engineer who does more than one project involving ETL eventually goes through the same stages of ETL grief. First, they think it's not so bad. Then they do another project and discover that they have to rewrite very similar code. Then they think, "Surely, I could have just written a few library functions and reused that code, saving lots of time." But when they try to do this, they discover that although their ETL projects are very similar, they are just different enough that their code isn't reusable. So they resign themselves to rewriting code over and over again. The code is unreliable, difficult to maintain, and usually poorly tested and documented because it's such a pain to write in the first place. The task of writing ETL pipelines is so lousy that engineering best practices tend to go out the window because the engineer has better things to do.

1.2 What is it?

MetalPipe is an ETL framework for the real world. It aims to provide structure and consistency to your ETL pipelines, while still allowing you to write bespoke code for all of the weird little idiosyncratic features of your data. It is opinionated without being bossy.

The overall idea of MetalPipe is simple. On the surface, it looks a lot like streaming frameworks such as Spark or Storm. You hook up various tasks in a directed graph called a "pipeline". The pipeline ingests data from or more places, transforms it, and loads the data somewhere else. But it differs from Spark-like systems in important ways:

1. It is agnostic between stream and batch. Batches of data can be turned into streams and vice-versa.

2. It is lightweight, requiring no specialized infrastructure or network configuration.
3. Its built-in functionality is specifically designed for ETL tasks.
4. It is meant to accommodate 90% of your ETL needs entirely by writing configuration files.

1.3 What isn't it?

There are many things that MetalPipe is not:

1. It is not a Big Data(tm) tool. If you're handling petabytes of data, you do not want to use MetalPipe.
2. It is not suitable for large amounts of computation. If you need to use dataframes to calculate lots of complex statistical information in real-time, this is not the tool for you.

Basically, MetalPipe deliberately makes two trade-offs: (1) it gives up Big Data(tm) for simplicity; and (2) it gives up being a general-purpose analytic tool in favor of being very good at ETL.

1.4 MetalPipe pipelines

An ETL pipeline in MetalPipe is a series of nodes connected by queues. Data is generated or processed in each node, and the output is placed on a queue to be picked up by downstream nodes.

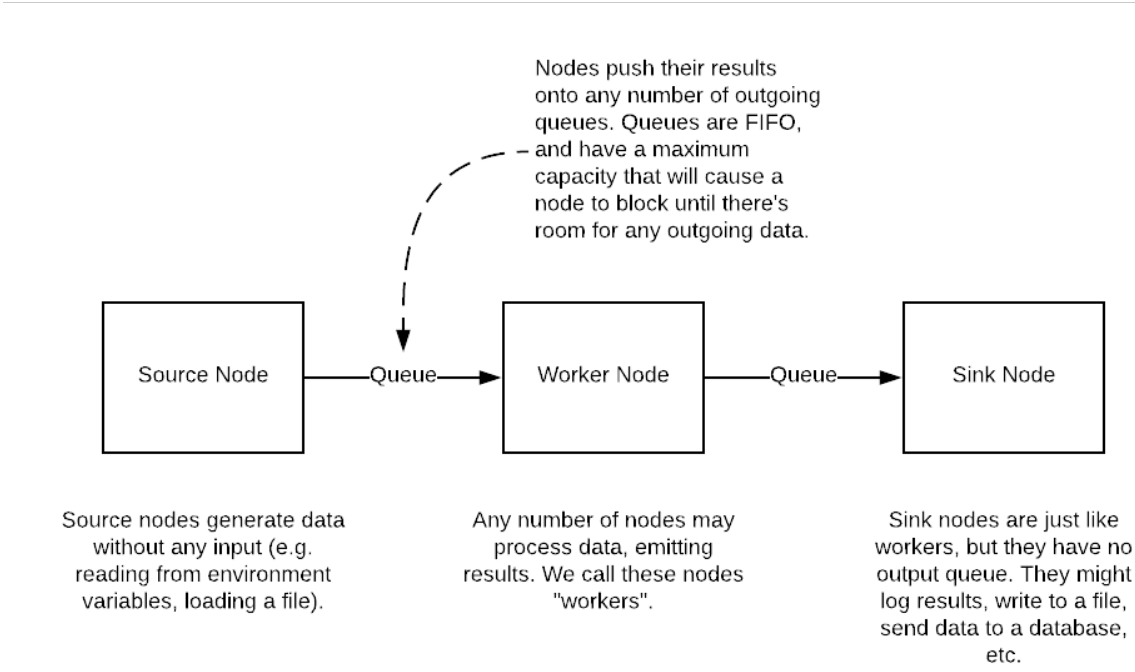

Fig. 1: Very high-level view of a MetalPipe pipeline

For the sake of convenience, we distinguish between three types of nodes (although there's no real difference in their use or implementation):

1. Source nodes. These are nodes that generate data and send it to the rest of the pipeline. They might, for example, read data from an external data source such as an API endpoint or a database.

2. Worker nodes. The workers process data by picking up messages from their incoming queues. Their output is placed onto any number of outgoing queues to be further processed by downstream nodes.
3. Sink nodes. These are worker nodes with no outgoing queue. They will typically perform tasks such as inserting data into a database or generating statistics to be sent somewhere outside the pipeline.

All pipelines are implemented in pure Python (version >=3.5). Each node is instantiated from a class that inherits from the `MetalNode` class. Queues are never instantiated directly by the user; they are created automatically whenever two nodes are linked together.

There is a large (and growing) number of specialized `MetalNode` subclasses, each geared toward a specific task. Such tasks include:

1. Querying a table in a SQL database and sending the results downstream.
2. Making a request to a REST API, paging through the responses until there are no more results.
3. Ingesting individual messages from an upstream node and batching them together into a single message, or doing the reverse.
4. Reading environment variables.
5. Watching a directory for new files and sending the names of those files down the pipeline when they appear.
6. Filtering messages, letting them through the pipeline only if a particular test is passed.

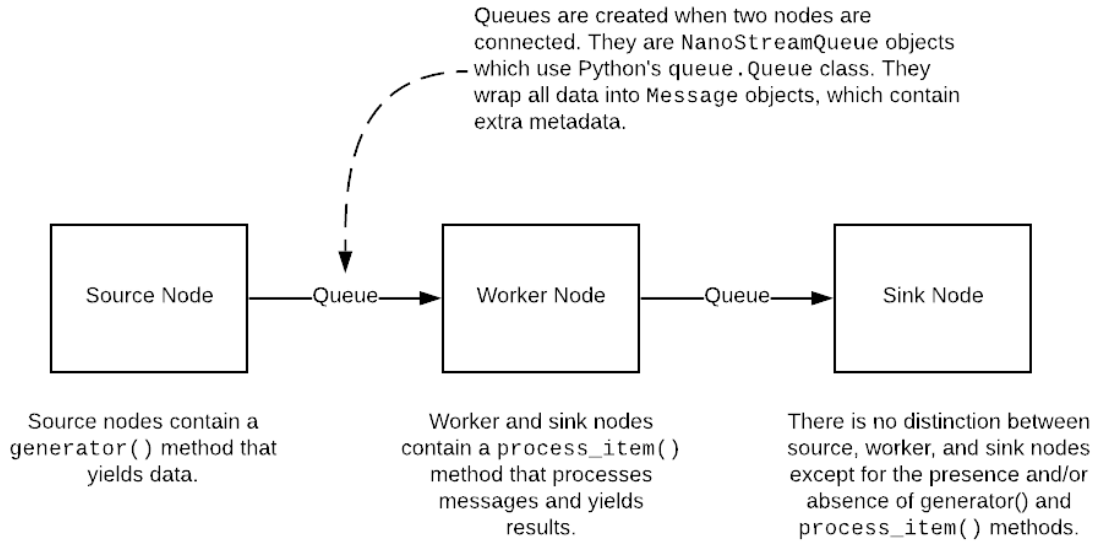

Fig. 2: Somewhat high-level view of a MetalPipe pipeline

All results and messages passed among the nodes must be dictionary-like objects. By default, messages retain any keys and values that were created by upstream nodes.

The goal is for MetalPipe to be "batteries included", with built-in `MetalNode` subclasses for every common ETL task. But because ETL pipelines generally have something weird going on somewhere, MetalNode makes it easy to roll your own node classes.

Nodes are defined in code by instantiating classes that inherit from `MetalNode`. Upon instantiation, the constructor takes the same set of keyword arguments as you see in the configuration. Nodes are linked together by the `>` operator, as in `node_1 > node_2`. After the pipeline has been built in this way, it is started by calling `node.global_start()` on any of the nodes in the pipeline.

The code corresponding to the configuration file above would look like this:

```
# Define the nodes using the various subclasses of MetalNode
get_environment_variables =
GetEnvironmentVariables(
    environment_variables=['API_KEY', 'API_USER_ID'])
print_variables = PrinterOfThings(prepend='Environment variables: ')

# The '>' operator can also be chained, as in:
# node_1 > node_2 > node_3 > ...
get_environment_variables > print_variables

# Run the pipeline. This command will not block.
get_environment_variables.global_start()
```

1.5 Rolling your own `MetalNode` class

If there are no built-in `MetalNode` classes suitable for your ETL pipeline, it is easy to write your own.

For example, suppose you want to create a source node for your pipeline that simply emits a user-defined string every few seconds forever. The user would be able to specify the string and the number of seconds to pause after each message has been sent. The class could be defined like so:

```
class FooEmitter(MetalNode):  # inherit from MetalNode
    '''
    Sends ``self.output_string`` every ``self.interval`` seconds.
    '''
    def __init__(self, output_string='', interval=1, **kwargs):
        self.output_string = output_string
        self.interval = interval
        super(FooEmitter, self).__init__()  # Must call the `MetalNode` __init__

    def generator(self):
        while True:
            time.sleep(self.interval)
            yield self.output_string  # Output must be yielded, not returned
```

Let's look at each part of this class.

The first thing to note is that the class inherits from `MetalNode` – this is the mix-in class that gives the node all of its functionality within the MetalPipe framework.

The `__init__` method should take only keyword arguments, not positional arguments. This restriction is to guarantee that the configuration files have names for any options that are specified in the pipeline. In the `__init__` function, you should also be sure to accept `**kwargs`, because options that are common to all `MetalNode` objects are expected to be there.

After any attributes have been defined, the `__init__` method **must** invoke the parent class's constructor through the use of the `super` function. Be sure to pass the `**kwargs` argument into the function as shown in the example.

If the node class is intended to be used as a source node, then you need to define a `generator` method. This method can be virtually anything, so long as it sends its output via a `yield` statement.

If you need to define a worker node (that is, a node that accepts input from a queue), you will provide a `process_item` method instead of a `generator`. But the structure of that method is the same, with the single exception that you will have access to a `__message__` attribute which contains the incoming message data. The structure of a typical `process_item` method is shown in the figure.

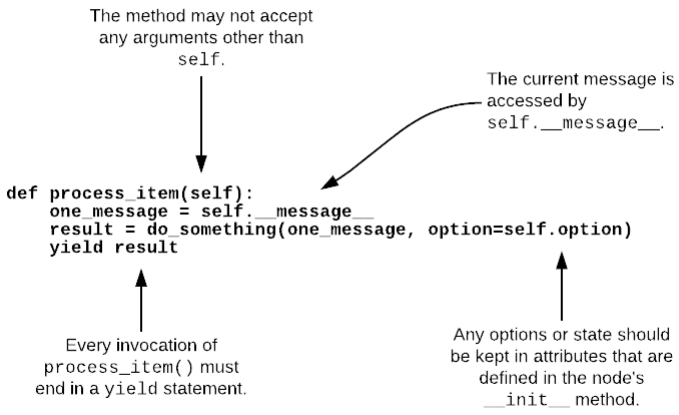

Fig. 3: A typical `process_item` method for `MetalNode` objects

For example, let's suppose you want to create a node that is passed a string as a message, and returns `True` if the message has an even number of characters, `False` otherwise. The class definition would look like this:

```
class MessageLengthTester(MetalNode):
    def __init__(self):
        # No particular initialization required in this example
        super(MessageLengthTester, self).__init__()

    def process_item(self):
        if len(self.__message__) % 2 == 0:
            yield True
        else:
            yield False
```

1.6 Composing and configuring `MetalNode` objects

Warning: The code described in this section is experimental and very unstable. It would be bad to use it for anything important.

Let's suppose you've worked very hard to create the pipeline from the last example. Now, your boss says that another engineering team wants to use it, but they want to rename parameters and "freeze" the values of certain other parameters to specific values. Once that's done, they want to use it as just one part of a more complicated `MetalPipe` pipeline.

This can be accomplished using a configuration file. When `MetalPipe` parses the configuration file, it will dynamically create the desired class, which can be instantiated and used as if it were a single node in another pipeline.

The configuration file is written in YAML, and it would look like this:

```
name: FooMessageTester

nodes:
  - name: foo_generator
    class FooEmitter
    frozen_arguments:
      message: foobar
    arg_mapping:
      interval: foo_interval
  - name: length_tester
    class: MessageLengthTester
    arg_mapping: null
```

With this file saved as (e.g.) `foo_message.yaml`, the following code will create a `FooMessageTester` class and instantiate it:

```
foo_message_config = yaml.load(open('./foo_message.yaml', 'r').read())
class_factory(foo_message_config)
# At this point, there is now a `FooMessageTester` class
foo_node = FooMessageTester(foo_interval=1)
```

You can now use `foo_node` just as you would any other node. So in order to run it, you just do:

```
foo_node.global_start()
```

Because `foo_node` is just another node, you can insert it into a larger pipeline and reuse it. For example, suppose that other engineering team wants to add a `PrinterOfThings` to the end of the pipeline. They'd do this:

```
printer = PrinterOfThings()
foo_node > printer
```

CHAPTER 2

Quickstart

This explains how to install MetalPipe, create a simple configuration file, and execute a pipeline.

2.1 Install MetalPipe

MetalPipe is installed in the usual way, with pip:

```
pip install metalpipe
```

To test your installation, try typing

```
metalpipe --help
```

If MetalPipe is installed correctly, you should see a help message.

2.2 Write a configuration file

You use MetalPipe by (1) writing a configuration file that describes your pipeline, and (2) running the `metalpipe` command, specifying the location of your configuration file. MetalPipe will read the configuration, create the pipeline, and run it.

The configuration file is written in YAML. It has three parts:

1. A list of global variables (optional)
2. The nodes and their options (required)
3. A list of edges connecting those nodes to each other.

This is a simple configuration file. If you want to, you can copy it into a file called `sample_config.yaml`:

```
---
pipeline_name: Sample MetalPipe configuration
pipeline_description: Reads some environment variables and prints them

nodes:
  get_environment_variables:
    class: GetEnvironmentVariables
    summary: Gets all the necessary environment variables
    options:
      environment_variables:
        - API_KEY
        - API_USER_ID

  print_variables:
    class: PrinterOfThings
    summary: Prints the environment variables to the terminal
    options:
      prepend: "Environment variables: "

paths:
  -
    - get_environment_variables
    - print_variables
```

2.3 Run the pipeline

If you've installed MetalPipe and copied this configuration into `sample_config.yaml`, then you can execute the pipeline:

```
metalpipe run --filename sample_config.yaml
```

The output should look like this (you might also see some log messages):

```
Environment variables:
{'API_USER_ID': None, 'API_KEY': None}
```

The MetalPipe pipeline has found the values of two environment variables (`API_KEY` and `API_USER_ID`) and printed them to the terminal. If those environmet variables have not been set, their values will be `None`. But if you were to set any of them, their values would be printed.

CHAPTER 3

Configuration file structure

A configuration file starts with two top-level options, `pipeline_name` and `pipeline_description`. These are optional, and are only used for the user's convenience.

Below those are two sections: `nodes` and `paths`. Each `nodes` section contains one or more blocks that always have this form:

```
do_something:
  class: node class
  summary: optional string describing what this node does
  options:
    option_1: value of this option
    option_2: value of another option
```

Let's go through this one line at a time.

Each node block describes a single node in the MetalPipe pipeline. A node must be given a name, which can be any arbitrary string. This should be a short, descriptive string describing its action, such as `get_environment_variables` or `parse_json`, for example. We encourage you to stick to a clear naming convention. We like nodes to have names of the form `verb_noun` (as in `print_name`).

MetalPipe contains a number of node classes, each of which is designed for a specific type of ETL task. In the sample configuration, we've used the built-in classes `GetEnvironmentVariables` and `PrinterOfThings`; these are the value following `class`. You can also roll your own node classes (we'll describe how to do this later in the documentation).

Next is a set of keys and values for the various options that are supported by that class. Because each node class does something different, the options are different as well. In the sample configuration, the `GetEnvironmentVariables` node class requires a list of environment variables to retrieve, so as you would expect, we specify that list under the `environment_variables` option. The various options are explained in the documentation for each class. In addition to the options that are specific to each node, there are also options that are common to every type of node. These will be explained later.

The structure of the pipeline is given in the `paths` section, which contains a list of lists. Each list is a set of nodes that are to be linked together in order. In our example, the `paths` value says that `get_environment_variables` will send its output to `print_variables`. Paths can be arbitrarily long.

If you wanted to send the environment variables down two different execution paths, you add another list to the `paths`, like so:

```
paths:
  -
    - get_environment_variables
    - print_variables
  -
    - get_environment_variables
    - do_something_else
    - and_then_do_this
```

With this set of `paths`, the pipeline looks like a very simple tree, with `get_environment_variables` at the root, which branches to `print_variables` and `do_something_else`.

When you have written the configuration file, you're ready to use the MetalPipe CLI. It accepts a command, followed by some options. As of now, the commands it accepts are `run`, which executes the pipeline, and `draw`, which generates a diagram of the pipeline. The relevant command(s) are:

```
python metalpipe_cli.py [run | draw] --filename my_sample_config.yaml
```

The `metalpipe` command can generate a pdf file containing a drawing of the pipeline, showing the flow of data through the various nodes. Just speciy `draw` instead of `run` to generate the diagram. For our simple little pipeline, we get this:

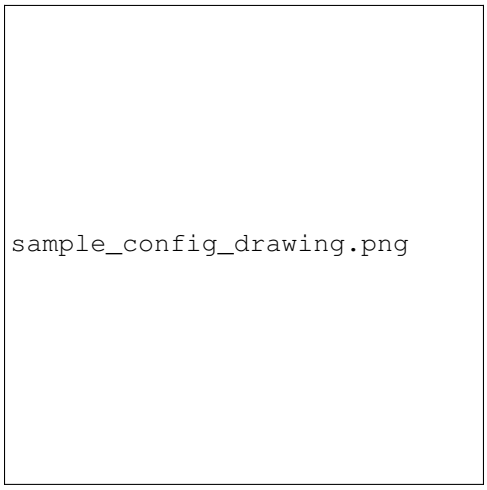

Fig. 1: The pipeline drawing for the simple configuration example

It is also possible to skip using the configuration file and define your pipelines directly in code. In general, it's better to use the configuration file for a variety of reasons, but you always have the option of doing this in Python.

CHAPTER 4

Node and Data Lifecycle

This section describes what's happening under the hood in a `MetalPipe` data pipeline. Most people won't need to read this section. But if you're planning on writing custom classes that inherit from `MetalNode`, this will be helpful.

4.1 The Node Lifecycle

The `MetalNode` class is where the crucial work happens in a pipeline. The lifecycle of a `MetalNode` object comprises several steps.

4.1.1 Instantiating the node and pipeline

Recall that when a node is defined in a configuration file, the definition looks like this:

```
my_node:
  class: MyMetalNodeClass
  options:
    an_option: foo
    another_option: bar
```

The code for any `MetalNode` subclass has an `__init__` method that has the following form:

```
class MyMetalNodeClass(MetalNode):
    def __init__(self, an_option=None, another_option=None, **kwargs):
        self.an_option = an_option
        self.another_option = another_option
        super(MyMetalNodeClass, self).__init__(**kwargs)
```

As you can see, the keyword arguments directly correspond to the keys under the `options` key in the configuration file. When the configuration file is read by the command-line tool, the class is instantiated and the options are converted to keyword arguments to be passed to the constructor. Keyword arguments will typically be a combination of options that are specific to that class and options that are inherited by any subclass of `MetalNode`.

Instantiating the class does not create any input or output queues. That happens only when two nodes are hooked together. In python code, you can hook up two or more nodes by using the > operator, as in:

```
node_1 > node_2 > node_3
```

In a configuration file, this is accomplished with the `paths` key, like so:

```
paths:
  -
    - node_1
    - node_2
    - node_3
```

4.1.2 Starting the node

To do.

4.1.3 Processing data in the pipeline

To do.

4.1.4 Shutting down normally

To do.

4.1.5 Shutting down due to error

To do.

4.2 The data journey

REVISE THIS

MetalPipe pipelines are sets of `MetalNode` objects connected by `MetalPipeQueue` objects. Think of each `MetalNode` as a vertex in a directed graph, and each `MetalPipeQueue` as a directed edge.

There are two types of `MetalNode` objects. A "source" is a `MetalNode` that does not accept incoming data from another `MetalNode`. A "processor" is any `MetalNode` that is not a "source". Note that there is nothing in the class definition or object that distinguishes between these two – the only difference is that processors have a `process_item` method, and sources have a `generator` method. Other than that, they are identical.

The data journey begins with one or more source nodes. When a source node is started (by calling its `start` method), a new thread is created and the node's `generator` method is executed inside the thread. As results from the `generator` method are yielded, they are placed on each outgoing `MetalPipeQueue` to be picked up by one or more processors downstream.

The data from the source's `generator` is handled by the `MetalPipeQueue` object. At its heart, the `MetalPipeQueue` is simply a class which has a Python `Queue.queue` object as an attribute. The reason we don't simply use Python `Queue` objects is because the `MetalPipeQueue` contains some logic that's useful. In particular:

1. It wraps the data into a `MetalPipeMessage` object, which also holds useful metadata including a UUID, the ID of the node that generated the data, and a timestamp.
2. If the `MetalPipeQueue` receives data that is simply a `None` object, then it is skipped.

CHAPTER 5

Monitoring

MetalPipe lets you easily monitor your pipeline, identify bottlenecks, and help diagnose failures.

5.1 Logging table

While a pipeline is being executed, a table of information will periodically be logged (at the INFO logging level). Each row provide diagnostic information about a single node in the pipeline. This is a typical example:

Node	Class	Received	Sent	Queued	Status	Time
aggregate_identities	AggregateValues	0	0	0	running	0:02:47.052141
batch_contacts	BatchMessages	249	249	2	running	0:02:48.814964
batch_signature_events	BatchMessages	89	89	0	running	0:02:51.238849
batch_unmatched_emails	BatchMessages	0	0	0	running	0:02:53.395140
batch_vid_email_rows	BatchMessages	0	0	0	running	0:02:56.635160
collect_form_signature_record	Remapper	89	89	0	running	0:02:56.653631
collect_vid_list	AggregateValues	0	0	0	running	0:02:54.603117
contacts_epoch_to_timestamp	SimpleTransforms	324	324	128	running	0:02:51.213190
contacts_printer	PrinterOfThings	0	0	0	running	0:02:54.006477
contacts_timestamp_to_redshift	SimpleTransforms	253	253	70	running	0:02:48.780311
datetime_to_milliseconds	SimpleTransforms	1	1	0	success	0:01:33.860897
email_event_epoch_to_timestamp	SimpleTransforms	0	0	0	success	0:01:55.427740
email_event_timestamp_to_redshift	SimpleTransforms	0	0	0	success	0:01:57.558140
engagements_printer	PrinterOfThings	0	0	0	success	0:02:06.284718
filter_null_email_addresses	Filter	0	0	0	running	0:02:55.841321
find_email_address	SimpleTransforms	0	0	0	running	0:02:57.126499
forms_printer	PrinterOfThings	1	0	0	success	0:02:13.640672
generate_email_api_query	SimpleTransforms	0	0	0	running	0:02:57.704351
generic_printer	PrinterOfThings	0	0	0	running	0:02:57.474036
get_contact_for_email	HttpGetRequest	0	0	0	running	0:02:57.691576

Fig. 1: The logging table provides information about each node in a running pipeline.

We'll go through each column of the table.

The `Node` column contains the name of a node. This is the name that was given in the configuration file as a top-level key in the `nodes` section.

If the name is printed in red (as in `contacts_epoch_to_timestamp` in the example), then the node is a "bottleneck". In order to identify bottlenecks, MetalPipe periodically polls each node to determine if (1) it input queue is full and (2) its output queue is not full. If those conditions are frequently met, then the node is identified as a bottleneck.

Note that being a bottleneck is not necessarily a sign of inefficiency. For any sufficiently long-running pipeline, it is very likely that some node will happen to be the slowest, and it will be considered a bottleneck.

The `Class` column simply gives the class of the `MetalNode` object, which tells you what function it is performing.

The `Received`, `Sent`, and `Queued` columns tell you how many messages are at various stages of processing. The `Received` number indicate how many messages have been procesed by the node, including any message that is currently being procesed. `Sent` gives how many messages have been output by this node. Finally, `Queued` is the number of messages that are on that nodes incoming queue(s). If there are several incoming queues, then this number is the sum. Note that for a source node, the value of `Received` will always be zero, and for any sink node, the value of `Sent` will be zero.

The `Status` column has three possible values: `running`, `success`, and `error`. Here, `success` means that the node has completed its work and has terminated without raising an error. A node is considered to be done with its work when its parent nodes (if any) have completed, its incoming queues are all empty, and it is not processing any messages. An `error` is indicated whenever a node raises an Exception. When this happens, the entire pipeline is shut down automatically. These status messages are colored yellow, green, and red respectively.

Finally `Time` is the total amount of time the node has spent running. When it is in a non-running state (either `success` or `error`), the clock stops.

CHAPTER 6

Treehorn

Treehorn is a set of classes for manipulating dictionary- and list-like objects in a declarative style. It is meant to be useful for the sort of tasks required for ETL, such as extracting structured data from JSON objects.

6.1 Using Treehorn

Treehorn allows you to search for information in a dictionary- or list-like object by specifying conditions. Structures that match those conditions can be returned, or they can be labeled. If they are labeled, you can use those labels to build more complex searches later, or retrieve the data. The style of Treehorn is somewhat like JQuery and similar languages that are good for manipulating tree-like data structures such as web pages.

We'll explain Treehorn by stepping through an example of how we would extract data from the following JSON blob:

```
{
    "source": "users",
    "hash": "Ch8KFgjQj67igOnVto4BELHgwMD7iNfjkQEYlrfjtZAt",
    "events": [
        {
            "appName": "mobileapp",
            "browser": {
                "name": "Google Chrome",
                "version": []
            },
            "duration": 0,
            "created": 1550596005797,
            "location": {
                "country": "United States",
                "state": "Massachusetts",
                "city": "Boston"
            },
            "id": "af6de71b",
            "smtpId": null,
            "portalId": 537105,
```

(continues on next page)

(continued from previous page)


```
            "email": "alice@gmail.com",
            "sentBy": {
                "id": "befa29c9",
                "created": 1550518557458
            },
            "type": "OPEN",
            "filteredEvent": false,
            "deviceType": "COMPUTER"
        },
        {
            "appName": "desktopapp",
            "browser": {
                "name": "Firefox",
                "version": []
            },
            "duration": 0,
            "created": 1550596005389,
            "location": {
                "country": "United States",
                "state": "New York",
                "city": "New York"
            },
            "id": "12aadd80",
            "smtpId": null,
            "portalId": 537105,
            "email": "bob@gmail.com",
            "sentBy": {
                "id": "2cd1e257",
                "created": 1550581974777
            },
            "type": "OPEN",
            "filteredEvent": false,
            "deviceType": "COMPUTER"
        }
    ]
}
```


As you can see, this JSON blob is similar to a typical response from a REST API (in fact, this is actually an example from a real REST API, with all personal information deleted).

Let's suppose you need to extract the email address and corresponding city name for each entry in `events`. This example is simple enough that you might not see the usefulness of Treehorn, but it's complex enough to get a sense of how Treehorn works. Later, we'll look at circumstances where Treehorn's declarative style is especially useful.

There are four classes that are important for Treehorn:

1. `Conditions` – These are classes that test a particular location in a tree (e.g. a dictionary) for some condition. Examples of useful conditions are being a dictionary with a certain key, being a non-empty list, having an integer value, and so on.
2. `Traversal` – These classes move throughout a tree, recursively applying tests to each node that they visit. Traversals can be upward (toward the root) or downward (toward the leaves).
3. `Label` – These are nothing more than strings that are attached to particular locations in the tree. Typically, we apply a label to locations in the tree that match particular conditions.
4. `Relations` – Finally, this class represents n-tuples of locations in the tree. For example, if an email address is present in the tree, and the user's city is present, a `Relation` can be used to denote that the person with that email address lives in that city.

The workflow for a typical Treehorn query is that we (1) define some conditions (such as being an email address field); (2) traverse the tree, searching for locations that match those conditions; (3) label those locations; and (4) define a relationship from those labels, which we can use to extract the right information. We'll gradually build up a query by adding each of these steps one at a time.

6.1.1 Condition objects

For this example, let's suppose you've loaded the JSON into a dictionary, like so:

```
import json

with open('./sample_api_response.json', 'r') as infile:
    api_response = json.load(infile)
```

Let's extract the email addresses and corresonding cities for each user in the API response. First, we create a couple of `Condition` objects using the built-in class `HasKey`:

```
has_email_key = HasKey('email')
has_city_key = HasKey('city')
```

The `HasKey` class is a subclass of `MeetsCondition`, all of which are callable and return `True` or `False`. For example, you could do the following:

```
d = {'email': 'myemail.com', 'name': 'carol'}
has_email_key(d)  # Returns True
has_city_key(d)   # Returns False
```

What if you want to test for two conditions on a single node? `MeetsCondition` objects can be combined into larger boolean expressions using `&`, `|`, and `~` like so:

```
(has_email_key & has_city_key)(d)     # Returns False
(has_email_key & ~ has_city_key)(d)   # Returns True
(has_email_key | has_city_key)(d)     # Returns True
```

6.1.2 Traversal objects

`MeetsCondition` objects aren't very useful unless they're combined with traversals. There are two types of traversal classes: `GoUp` and `GoDown`. Each takes a `MeetsCondition` object as a parameter. For example, if you want to search from the root of the tree for every location that is a dictionary with the `email` key, the traversal is:

```
find_email = GoDown(condition=has_email_key)  # or GoDown(condition=HasKey('email'))
```

Similarly for finding places with a `city` key:

```
find_city = GoDown(condition=has_city_key)  # or GoDown(condition=HasKey('city'))
```

If you want to retrieve all of `find_city`'s matches, you can use its `matches` method, which will yield each match:

```
for match in has_email_key.matches(api_response):
    print(match)
```

which will yield:

```
{'id': 'af6de71b', 'portalId': 537105, 'location': {'state': 'Massachusetts', 'city':
↪'Boston', 'country': 'United States'}, 'type': 'OPEN', 'sentBy': {'id': 'befa29c9',
↪'created': 1550518557458}, 'appName': 'mobileapp', 'duration': 0,'smtpId': None,
↪'deviceType': 'COMPUTER', 'created': 1550596005797, 'email': 'alice@gmail.com',
↪'browser': {'version': [], 'name': 'Google Chrome'}, 'filteredEvent': False}
{'id': '12aadd80', 'portalId': 537105, 'location': {'state': 'New York', 'city': 'New␣
↪York', 'country': 'United States'}, 'type': 'OPEN', 'sentBy': {'id': '2cd1e257',
↪'created': 1550581974777}, 'appName': 'desktopapp', 'duration': 0, 'smtpId': None,
↪'deviceType': 'COMPUTER', 'created': 1550596005389, 'email': 'bob@gmail.com',
↪'browser': {'version': [], 'name': 'Firefox'}, 'filteredEvent': False}
```

Examining each of the dictionaries, we see that they do in fact contain an `email` key. Note that the traversal does **not** return the email string itself – we asked only for the dictionary containing the key. This is by design, as we will see soon.

Because we want to retrieve not only the email addresses but also the cities, we need another traversal. Each of the two dictionaries containing the `email` key also have a subdictionary that contains a `city` key. So we need a second traversal to get that subdictionary. In other words, retrieving the data we need is a two-step process:

1. Starting at the root of the tree, we traverse downward until we find a dictionary with the `email` key.
2. From each of those dictionaries, we go down until we find a dictionary with the `city` key.

Any non-trivial ETL task involving nested dictionary-like objects will require multi-stage traversals like this one. So Treehorn allows you to chain traversals together using the > operator:

```
chained_traversal = find_email > find_city
```

The `chained_traversal` says, in effect, "Go down into the tree and find every node that has an `email` key. Then, from each of those, continue to go down until you find a node that contains a `city` key. In pseudo-code:

```
For each node_1 starting at the root:
    if node_1 has ``email`` key:
        for each node_2 starting at node_1:
            if node_2 has ``city`` key:
                return
```

So far, we have set set up multi-stage searches for nodes in a tree that satisfy various conditions. Next, we have to extract the right data from those searches. This is where the `Label` and `Relation` classes come into play.

6.1.3 Labels

When nodes are identified that satisfy certain conditions, we will want to label those nodes so that we can extract data from them later. The mechanism for doing this is to use a "label".

Continuing the example, let's use the labels "email" and "city" to mark the respective nodes in the two-stage traversal. We do so by adding a label to the traversal chain. Recall that in the previous section, we wrote:

```
chained_traversal = find_email > find_city
```

whereas we now have:

```
chained_traversal = find_email + 'email' > find_city + 'city'
```

We use + to add a label, and the label is just a string. Under the hood, Treehorn is instantiating a `Label` object, but ordinarily, you shouldn't have to do that directly.

6.1.4 Relations

Lastly, we define a `Relation` object to extract the data from our search. In this example, we might think of the search as returning data about people who live in a certain city. So we might name the `Relation` "FROM_CITY".We'll want to extract the value of the `email` key from the node labeled with `email`, and similarly with the `city` node. This is accomplished by adding a little more syntax:

```
Relation('FROM_CITY') == (
    (find_email + 'email')['email'] > (find_city + 'city')['city'])
```

After executing that statement, Treehorn will create an object named `FROM_CITY`, which can be called on a dictionary to yield the information we want, like so:

```
for email_city in FROM_CITY(api_response):
    print(email_city)
```

which will give us:

```
{'city': 'Boston', 'email': 'alice@gmail.com'}
{'city': 'New York', 'email': 'bob@gmail.com'}
```

Voila!

6.2 Summing up

Normally, ETL pipelines that extract data from dictionary-like objects involve a lot of loops and hard-coded keypaths. To accomplish the simple task of extracting emails and city names from our sample JSON blob, we'd probably hard-code paths for each specific key and value, and then we'd loop over various levels in the dictionary. This has several disadvantages:

1. It leads to brittle code. If the JSON blob changes structure in even very small ways, the hard-coded paths become obsolete and have to be rewritten.
2. The code is difficult to understand and debug. Given a whole bunch of nested loops and hard-coded keypaths, it's very difficult to understand the intent of the code. Errors have to be found by painstakingly stepping through the execution.
3. It is very difficult to accommodate JSON blobs with variable structure. Some JSON blobs returned from APIs have unpredictable levels of nesting, for example. Therefore, keypaths cannot be hard-coded and recursive searches have to be written, which are inefficient and difficult to debug.

The approach taken by Treehorn alleviates some of this pain. For example, the `GoDown` traversal doesn't care how many levels down in the tree it must search; so it is often able to cope with inconsistent structures (within reason) without any code changes. It's also much easier to understand. You can tell from glancing at the code that the intention is to search for a dictionary with a key, and then search from there for lower-level dictionaries with another key, and return the results. Treehorn is also more efficient than writing loops and keypaths because all of its evaluations are lazy – it doesn't hold partial results in memory any longer than necessary because everything is yielded by generators.

CHAPTER 7

API Documentation

7.1 Node module

The `node` module contains the `MetalNode` class, which is the foundation for MetalPipe.

class metalpipe.node.**AggregateValues**(*values=False*, *tail_path=None*, ***kwargs*)

Bases: *metalpipe.node.MetalNode*

Does that.

process_item()

Default no-op for nodes.

class metalpipe.node.**BatchMessages**(*batch_size=None*, *batch_list=None*, *counter=0*, *timeout=5*, ***kwargs*)

Bases: *metalpipe.node.MetalNode*

cleanup()

If there is any cleanup (closing files, shutting down database connections), necessary when the node is stopped, then the node's class should provide a `cleanup` method. By default, the method is just a logging statement.

process_item()

Default no-op for nodes.

class metalpipe.node.**CSVReader**(**args*, ***kwargs*)

Bases: *metalpipe.node.MetalNode*

process_item()

Default no-op for nodes.

class metalpipe.node.**CSVToDictionaryList**(***kwargs*)

Bases: *metalpipe.node.MetalNode*

process_item()

Default no-op for nodes.

class metalpipe.node.**ConstantEmitter**(*thing=None*, *max_loops=5*, *delay=0.5*, ***kwargs*)
Bases: *metalpipe.node.MetalNode*

Send a thing every n seconds

generator()

class metalpipe.node.**CounterOfThings**(**args*, *batch=False*, *get_runtime_attrs=<function no_op>*, *get_runtime_attrs_args=None*, *get_runtime_attrs_kwargs=None*, *runtime_attrs_destinations=None*, *input_mapping=None*, *retain_input=True*, *throttle=0*, *keep_alive=True*, *max_errors=0*, *max_messages_received=None*, *name=None*, *input_message_keypath=None*, *key=None*, *messages_received_counter=0*, *prefer_existing_value=False*, *messages_sent_counter=0*, *post_process_function=None*, *post_process_keypath=None*, *summary=''*, *fixturize=False*, *post_process_function_kwargs=None*, *output_key=None*, *break_test=None*, ***kwargs*)
Bases: *metalpipe.node.MetalNode*

bar__init__(**args*, *start=0*, *end=None*, ***kwargs*)

generator()
Just start counting integers

class metalpipe.node.**DynamicClassMediator**(**args*, ***kwargs*)
Bases: *metalpipe.node.MetalNode*

get_sink()

get_source()

hi()

sink_list()

source_list()

class metalpipe.node.**Filter**(*test=None*, *test_keypath=None*, *value=True*, **args*, ***kwargs*)
Bases: *metalpipe.node.MetalNode*

Applies tests to each message and filters out messages that don't pass

Built-in tests: key_exists value_is_true value_is_not_none

Example

{'test': 'key_exists', 'key': mykey}

process_item()
Default no-op for nodes.

class metalpipe.node.**FunctionOfMessage**(*function_name*, **args*, ***kwargs*)
Bases: *metalpipe.node.MetalNode*

process_item()
Default no-op for nodes.

class metalpipe.node.**GetEnvironmentVariables**(*mappings=None*, *environment_variables=None*, ***kwargs*)

Bases: *metalpipe.node.MetalNode*

This node reads environment variables and stores them in the message.

The required keyword argument for this node is `environment_variables`, which is a list of – you guessed it! – environment variables. By default, they will be read and stored in the outgoing message under keys with the same names as the environment variables. E.g. `FOO_VAR` will be stored in the message `{"FOO_BAR": whatever}`.

Optionally, you can provide a dictionary to the `mappings` keyword argument, which maps environment variable names to new names. E.g. if `mappings = {"FOO_VAR": "bar_var"}`, then the value of `FOO_VAR` will be stored in the message `{"bar_var": whatever}`.

If the environment variable is not defined, then its value will be set to `None`.

Parameters

- **mappings** (*dict*) – An optional dictionary mapping environment variable names to new names.
- **environment_variables** (*list*) – A list of environment variable names.

generator()

process_item()
Default no-op for nodes.

class metalpipe.node.**InsertData**(*overwrite=True*, *overwrite_if_null=True*, *value_dict=None*, ***kwargs*)

Bases: *metalpipe.node.MetalNode*

process_item()
Default no-op for nodes.

class metalpipe.node.**LocalDirectoryWatchdog**(*directory='.'*, *check_interval=3*, ***kwargs*)

Bases: *metalpipe.node.MetalNode*

generator()

class metalpipe.node.**LocalFileReader**(**args*, ***kwargs*)

Bases: *metalpipe.node.MetalNode*

process_item()
Default no-op for nodes.

class metalpipe.node.**MetalNode**(**args*, *batch=False*, *get_runtime_attrs=<function no_op>*, *get_runtime_attrs_args=None*, *get_runtime_attrs_kwargs=None*, *runtime_attrs_destinations=None*, *input_mapping=None*, *retain_input=True*, *throttle=0*, *keep_alive=True*, *max_errors=0*, *max_messages_received=None*, *name=None*, *input_message_keypath=None*, *key=None*, *messages_received_counter=0*, *prefer_existing_value=False*, *messages_sent_counter=0*, *post_process_function=None*, *post_process_keypath=None*, *summary=''*, *fixturize=False*, *post_process_function_kwargs=None*, *output_key=None*, *break_test=None*, ***kwargs*)

Bases: `object`

The foundational class of *MetalPipe*. This class is inherited by all nodes in a computation graph.

Order of operations: 1. Child class `__init__` function 2. `MetalNode` `__init__` function 3. `preflight_function` (Specified in initialization params) 4. `setup` 5. start

These methods have the following intended uses:

1. `__init__` Sets attribute values and calls the `MetalNode` `__init__` method.
2. `get_runtime_attrs` Sets any attribute values that are to be determined at runtime, e.g. by checking environment variables or reading values from a database. The `get_runtime_attrs` should return a dictionary of attributes -> values, or else `None`.
3. `setup` Sets the state of the `MetalNode` and/or creates any attributes that require information available only at runtime.

Parameters

- **`send_batch_markers`** – If `True`, then a `BatchStart` marker will be sent when a new input is received, and a `BatchEnd` will be sent after the input has been processed. The intention is that a number of items will be emitted for each input received. For example, we might emit a table row-by-row for each input.
- **`get_runtime_attrs`** – A function that returns a dictionary-like object. The keys and values will be saved to this `MetalNode` object's attributes. The function is executed one time, upon starting the node.
- **`get_runtime_attrs_args`** – A tuple of arguments to be passed to the `get_runtime_attrs` function upon starting the node.
- **`get_runtime_attrs_kwargs`** – A dictionary of kwargs passed to the `get_runtime_attrs` function.
- **`runtime_attrs_destinations`** – If set, this is a dictionary mapping the keys returned from the `get_runtime_attrs` function to the names of the attributes to which the values will be saved.
- **`throttle`** – For each input received, a delay of `throttle` seconds will be added.
- **`keep_alive`** – If `True`, keep the node's thread alive after everything has been processed.
- **`name`** – The name of the node. Defaults to a randomly generated hash. Note that this hash is not consistent from one run to the next.
- **`input_mapping`** – When the node receives a dictionary-like object, this dictionary will cause the keys of the dictionary to be remapped to new keys.
- **`retain_input`** – If `True`, then combine the dictionary-like input with the output. If keys clash, the output value will be kept.
- **`input_message_keypath`** – Read the value in this keypath as the content of the incoming message.

`add_edge`(*target*, ***kwargs*)

Create an edge connecting *self* to *target*.

This method instantiates the `MetalPipeQueue` object that connects the nodes. Connecting the nodes together consists in (1) adding the queue to the other's `input_queue_list` or `output_queue_list` and (2) setting the queue's `source_node` and `target_node` attributes.

Parameters **`target`** (`MetalNode`) – The node to which `self` will be connected.

`all_connected`(*seen=None*)

Returns all the nodes connected (directly or indirectly) to `self`. This allows us to loop over all the nodes in a pipeline even if we have a handle on only one. This is used by `global_start`, for example.

Parameters **seen** (*set*) – A set of all the nodes that have been identified as connected to `self`.

Returns

All the nodes connected to `self`. This includes `self`.

Return type (set of `MetalNode`)

broadcast(*broadcast_message*)
Puts the message into all the input queues for all connected nodes.

cleanup()
If there is any cleanup (closing files, shutting down database connections), necessary when the node is stopped, then the node's class should provide a `cleanup` method. By default, the method is just a logging statement.

draw_pipeline()
Draw the pipeline structure using graphviz.

global_start(*prometheus=False*, *pipeline_name=None*, *max_time=None*, *fixturize=False*)
Starts every node connected to `self`. Mainly, it: 1. calls `start()` on each node #. sets some global variables #. optionally starts some experimental code for monitoring

input_queue_size
Return the total number of items in all of the queues that are inputs to this node.

is_sink
Tests whether the node is a sink or not, i.e. whether there are no outputs from the node.

Returns `True` if the node has no output nodes, `False` otherwise.

Return type (bool)

is_source
Tests whether the node is a source or not, i.e. whether there are no inputs to the node.

Returns `True` if the node has no inputs, `False` otherwise.

Return type (bool)

kill_pipeline()

log_info(*message=''*)

logjam
Returns the logjam score, which measures the degree to which the node is holding up progress in downstream nodes.

We're defining a logjam as a node whose input queue is full, but whose output queue(s) is not. More specifically, we poll each node in the `monitor_thread`, and increment a counter if the node is a logjam at that time. This property returns the percentage of samples in which the node is a logjam. Our intention is that if this score exceeds a threshold, the user is alerted, or the load is rebalanced somehow (not yet implemented).

Returns Logjam score

Return type (float)

pipeline_finished

process_item(**args*, ***kwargs*)
Default no-op for nodes.

setup()
For classes that require initialization at runtime, which can't be done when the class's __init__ function is called. The MetalNode base class's setup function is just a logging call.

It should be unusual to have to make use of setup because in practice, initialization can be done in the __init__ function.

start()
Starts the node. This is called by MetalNode.global_start().

The node's main loop is contained in this method. The main loop does the following:

1. records the timestamp to the node's started_at attribute.
2. calls get_runtime_attrs (TODO: check if we can deprecate this)
3. calls the setup method for the class (which is a no-op by default)
4. if the node is a source, then successively yield all the results of the node's generator method, then exit.
5. if the node is not a source, then loop over the input queues, getting the next message. Note that when the message is pulled from the queue, the MetalPipeQueue yields it as a dictionary.
6. gets either the content of the entire message if the node has no key attribute, or the value of message[self.key].
7. remaps the message content if a remapping dictionary has been given in the node's configuration
8. calls the node's process_item method, yielding back the results. (Note that a single input message may cause the node to yield zero, one, or more than one output message.)
9. places the results into each of the node's output queues.

stream()
Called in each MetalNode thread.

terminate_pipeline(*error=False*)
This method can be called on any node in a pipeline, and it will cause all of the nodes to terminate if they haven't stopped already.

Parameters **error** (*bool*) – Not yet implemented.

thread_monitor(*max_time=None*)
This function loops over all of the threads in the pipeline, checking that they are either finished or running. If any have had an abnormal exit, terminate the entire pipeline.

time_running
Return the number of wall-clock seconds elapsed since the node was started.

wait_for_pipeline_finish()

class metalpipe.node.**NothingToSeeHere**
Bases: object

Vacuous class used as a no-op message type.

class metalpipe.node.**PrinterOfThings**(**args*, ***kwargs*)
Bases: *metalpipe.node.MetalNode*

process_item()
Default no-op for nodes.

class metalpipe.node.**RandomSample**(*sample=0.1*)
Bases: *metalpipe.node.MetalNode*

Lets through only a random sample of incoming messages. Might be useful for testing, or when only approximate results are necessary.

process_item()
Default no-op for nodes.

class metalpipe.node.**Remapper**(*mapping=None*, ***kwargs*)
Bases: *metalpipe.node.MetalNode*

process_item()
Default no-op for nodes.

class metalpipe.node.**SequenceEmitter**(*sequence*, **args*, *max_sequences=1*, ***kwargs*)
Bases: *metalpipe.node.MetalNode*

Emits `sequence` `max_sequences` times, or forever if `max_sequences` is `None`.

generator()
Emit the sequence `max_sequences` times.

process_item()
Emit the sequence `max_sequences` times.

class metalpipe.node.**Serializer**(*values=False*, **args*, ***kwargs*)
Bases: *metalpipe.node.MetalNode*

Takes an iterable thing as input, and successively yields its items.

process_item()
Default no-op for nodes.

class metalpipe.node.**SimpleTransforms**(*missing_keypath_action='ignore'*, *starting_path=None*, *transform_mapping=None*, *target_value=None*, *keypath=None*, ***kwargs*)
Bases: *metalpipe.node.MetalNode*

process_item()
Default no-op for nodes.

class metalpipe.node.**StreamMySQLTable**(**args*, *host='localhost'*, *user=None*, *table=None*, *password=None*, *database=None*, *port=3306*, *to_row_obj=False*, *send_batch_markers=True*, ***kwargs*)
Bases: *metalpipe.node.MetalNode*

generator()

get_schema()

setup()
For classes that require initialization at runtime, which can't be done when the class's `__init__` function is called. The `MetalNode` base class's setup function is just a logging call.

It should be unusual to have to make use of `setup` because in practice, initialization can be done in the `__init__` function.

class metalpipe.node.**StreamingJoin**(*window=30*, *streams=None*, **args*, ***kwargs*)
Bases: *metalpipe.node.MetalNode*

Joins two streams on a key, using exact match only. MVP.

process_item()

class metalpipe.node.**SubstituteRegex**(*match_regex=None*, *substitute_string=None*, **args*, ***kwargs*)

Bases: *metalpipe.node.MetalNode*

process_item()

Default no-op for nodes.

class metalpipe.node.**TimeWindowAccumulator**(**args*, ***kwargs*)

Bases: *metalpipe.node.MetalNode*

Every N seconds, put the latest M seconds data on the queue.

class metalpipe.node.**bcolors**

Bases: object

This class holds the values for the various colors that are used in the tables that monitor the status of the nodes.

BOLD = '\x1b[1m'

ENDC = '\x1b[0m'

FAIL = '\x1b[91m'

HEADER = '\x1b[95m'

OKBLUE = '\x1b[94m'

OKGREEN = '\x1b[92m'

UNDERLINE = '\x1b[4m'

WARNING = '\x1b[93m'

metalpipe.node.**class_factory**(*raw_config*)

metalpipe.node.**get_node_dict**(*node_config*)

metalpipe.node.**kwarg_remapper**(*f*, ***kwarg_mapping*)

metalpipe.node.**no_op**(**args*, ***kwargs*)

No-op function to serve as default get_runtime_attrs.

metalpipe.node.**template_class**(*class_name*, *parent_class*, *kwargs_remapping*, *frozen_arguments_mapping*)

7.2 Civis-specific node types

This is where any classes specific to the Civis API live.

class metalpipe.node_classes.civis_nodes.**CivisSQLExecute**(**args*, *sql=None*, *civis_api_key=None*, *civis_api_key_env_var='CIVIS_API_KEY'*, *database=None*, *dummy_run=False*, *query_dict=None*, *returned_columns=None*, ***kwargs*)

Bases: *metalpipe.node.MetalNode*

Execute a SQL statement and return the results.

process_item()

Execute a SQL statement and return the result.

class metalpipe.node_classes.civis_nodes.**CivisToCSV**(**args*, *sql=None*, *civis_api_key=None*, *civis_api_key_env_var='CIVIS_API_KEY'*, *database=None*, *dummy_run=False*, *query_dict=None*, *returned_columns=None*, *include_headers=True*, *delimiter=','*, ***kwargs*)

Bases: *metalpipe.node.MetalNode*

Execute a SQL statement and return the results via a CSV file.

process_item()
Execute a SQL statement and return the result.

class metalpipe.node_classes.civis_nodes.**EnsureCivisRedshiftTableExists**(*on_failure='exit'*, *table=None*, *schema=None*, *database=None*, *columns=None*, *block=True*, ***kwargs*)

Bases: *metalpipe.node.MetalNode*

generator()

process_item()
Default no-op for nodes.

class metalpipe.node_classes.civis_nodes.**FindValueInRedshiftColumn**(*on_failure='exit'*, *table=None*, *database=None*, *schema=None*, *column=None*, *choice='max'*, ***kwargs*)

Bases: *metalpipe.node.MetalNode*

generator()

process_item()
Default no-op for nodes.

class metalpipe.node_classes.civis_nodes.**SendToCivis**(**args*, *civis_api_key=None*, *civis_api_key_env_var='CIVIS_API_KEY'*, *database=None*, *schema=None*, *existing_table_rows='append'*, *include_columns=None*, *dummy_run=False*, *block=False*, *max_errors=0*, *table=None*, *via_staging_table=False*, *columns=None*, *staging_table=None*, *remap=None*, *recorded_tables={}*, ***kwargs*)

Bases: *metalpipe.node.MetalNode*

cleanup()
Check if we're using staging tables. If so, copy the staging table into the production table. TODO: options for merge, upsert, append, drop

full_table_name

monitor_futures()

process_item()
Accept a bunch of dictionaries mapping column names to values.

setup()
Check if we're using staging tables and create the table if necessary.

7.3 Data structures module

Data types (e.g. Rows, Records) for ETL.

class metalpipe.utils.data_structures.**BOOL**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.DataType*, *metalpipe.utils.data_structures.IntermediateTypeSystem*

python_cast_function
alias of builtins.bool

class metalpipe.utils.data_structures.**DATETIME**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.DataType*, *metalpipe.utils.data_structures.IntermediateTypeSystem*

python_cast_function()

class metalpipe.utils.data_structures.**DataSourceTypeSystem**

Bases: object

Information about mapping one type system onto another contained in the children of this class.

static convert(*obj*)
Override this method if something more complicated is necessary.

static type_mapping(**args*, ***kwargs*)

class metalpipe.utils.data_structures.**DataType**(*value*, *original_type=None*, *name=None*)

Bases: object

Each DataType gets a python_cast_function, which is a function.

intermediate_type = None

python_cast_function = None

to_intermediate_type()

Convert the DataType to an IntermediateDataType using its class's intermediate_type attribute.

to_python()

type_system

Just for convenience to make the type system an attribute.

class metalpipe.utils.data_structures.**FLOAT**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.DataType*, *metalpipe.utils.data_structures.IntermediateTypeSystem*

python_cast_function

alias of builtins.float

class metalpipe.utils.data_structures.**INTEGER**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.DataType*, *metalpipe.utils.data_structures.IntermediateTypeSystem*

python_cast_function

alias of builtins.int

exception metalpipe.utils.data_structures.**IncompatibleTypesException**

Bases: Exception

class metalpipe.utils.data_structures.**IntermediateTypeSystem**

Bases: *metalpipe.utils.data_structures.DataSourceTypeSystem*

Never instantiate this by hand.

class metalpipe.utils.data_structures.**MYSQL_BOOL**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.DataType*, *metalpipe.utils.data_structures.MySQLTypeSystem*

intermediate_type

alias of *BOOL*

python_cast_function

alias of builtins.bool

class metalpipe.utils.data_structures.**MYSQL_DATE**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.DataType*, *metalpipe.utils.data_structures.MySQLTypeSystem*

intermediate_type

alias of *DATETIME*

python_cast_function()

class metalpipe.utils.data_structures.**MYSQL_ENUM**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.DataType*, *metalpipe.utils.data_structures.MySQLTypeSystem*

intermediate_type
alias of *STRING*

python_cast_function
alias of builtins.str

class metalpipe.utils.data_structures.**MYSQL_INTEGER**
Bases: type

class metalpipe.utils.data_structures.**MYSQL_INTEGER0**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 0

class metalpipe.utils.data_structures.**MYSQL_INTEGER1**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 1

class metalpipe.utils.data_structures.**MYSQL_INTEGER10**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 10

class metalpipe.utils.data_structures.**MYSQL_INTEGER1024**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 1024

class metalpipe.utils.data_structures.**MYSQL_INTEGER11**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 11

class metalpipe.utils.data_structures.**MYSQL_INTEGER12**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 12

class metalpipe.utils.data_structures.**MYSQL_INTEGER128**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 128

class metalpipe.utils.data_structures.**MYSQL_INTEGER13**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 13

class metalpipe.utils.data_structures.**MYSQL_INTEGER14**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 14

class metalpipe.utils.data_structures.**MYSQL_INTEGER15**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 15

class metalpipe.utils.data_structures.**MYSQL_INTEGER16**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 16

class metalpipe.utils.data_structures.**MYSQL_INTEGER16384**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 16384

class metalpipe.utils.data_structures.**MYSQL_INTEGER17**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 17

class metalpipe.utils.data_structures.**MYSQL_INTEGER18**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 18

class metalpipe.utils.data_structures.**MYSQL_INTEGER19**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 19

class metalpipe.utils.data_structures.**MYSQL_INTEGER2**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 2

class metalpipe.utils.data_structures.**MYSQL_INTEGER20**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 20

class metalpipe.utils.data_structures.**MYSQL_INTEGER2048**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 2048

class metalpipe.utils.data_structures.**MYSQL_INTEGER21**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 21

class metalpipe.utils.data_structures.**MYSQL_INTEGER22**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 22

class metalpipe.utils.data_structures.**MYSQL_INTEGER23**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 23

class metalpipe.utils.data_structures.**MYSQL_INTEGER24**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 24

class metalpipe.utils.data_structures.**MYSQL_INTEGER25**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 25

class metalpipe.utils.data_structures.**MYSQL_INTEGER256**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 256

class metalpipe.utils.data_structures.**MYSQL_INTEGER26**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 26

class metalpipe.utils.data_structures.**MYSQL_INTEGER27**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 27

class metalpipe.utils.data_structures.**MYSQL_INTEGER28**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 28

class metalpipe.utils.data_structures.**MYSQL_INTEGER29**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 29

class metalpipe.utils.data_structures.**MYSQL_INTEGER3**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 3

class metalpipe.utils.data_structures.**MYSQL_INTEGER30**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 30

class metalpipe.utils.data_structures.**MYSQL_INTEGER31**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 31

class metalpipe.utils.data_structures.**MYSQL_INTEGER32**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 32

class metalpipe.utils.data_structures.**MYSQL_INTEGER32768**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 32768

class metalpipe.utils.data_structures.**MYSQL_INTEGER4**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 4

class metalpipe.utils.data_structures.**MYSQL_INTEGER4096**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 4096

class metalpipe.utils.data_structures.**MYSQL_INTEGER5**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 5

class metalpipe.utils.data_structures.**MYSQL_INTEGER512**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 512

class metalpipe.utils.data_structures.**MYSQL_INTEGER6**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 6

class metalpipe.utils.data_structures.**MYSQL_INTEGER64**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 64

class metalpipe.utils.data_structures.**MYSQL_INTEGER7**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 7

class metalpipe.utils.data_structures.**MYSQL_INTEGER8**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 8

class metalpipe.utils.data_structures.**MYSQL_INTEGER8192**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 8192

class metalpipe.utils.data_structures.**MYSQL_INTEGER9**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_INTEGER_BASE*

max_length = 9

class metalpipe.utils.data_structures.**MYSQL_INTEGER_BASE**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.DataType*, *metalpipe.utils.data_structures.MySQLTypeSystem*

intermediate_type

alias of *INTEGER*

python_cast_function

alias of builtins.int

class metalpipe.utils.data_structures.**MYSQL_VARCHAR**

Bases: type

class metalpipe.utils.data_structures.**MYSQL_VARCHAR0**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 0

class metalpipe.utils.data_structures.**MYSQL_VARCHAR1**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 1

class metalpipe.utils.data_structures.**MYSQL_VARCHAR10**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 10

class metalpipe.utils.data_structures.**MYSQL_VARCHAR1024**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 1024

class metalpipe.utils.data_structures.**MYSQL_VARCHAR11**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 11

class metalpipe.utils.data_structures.**MYSQL_VARCHAR12**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 12

class metalpipe.utils.data_structures.**MYSQL_VARCHAR128**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 128

class metalpipe.utils.data_structures.**MYSQL_VARCHAR13**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 13

class metalpipe.utils.data_structures.**MYSQL_VARCHAR14**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 14

class metalpipe.utils.data_structures.**MYSQL_VARCHAR15**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 15

class metalpipe.utils.data_structures.**MYSQL_VARCHAR16**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 16

class metalpipe.utils.data_structures.**MYSQL_VARCHAR16384**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 16384

class metalpipe.utils.data_structures.**MYSQL_VARCHAR17**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 17

class metalpipe.utils.data_structures.**MYSQL_VARCHAR18**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 18

class metalpipe.utils.data_structures.**MYSQL_VARCHAR19**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 19

class metalpipe.utils.data_structures.**MYSQL_VARCHAR2**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 2

class metalpipe.utils.data_structures.**MYSQL_VARCHAR20**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 20

class metalpipe.utils.data_structures.**MYSQL_VARCHAR2048**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 2048

class metalpipe.utils.data_structures.**MYSQL_VARCHAR21**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 21

class metalpipe.utils.data_structures.**MYSQL_VARCHAR22**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 22

class metalpipe.utils.data_structures.**MYSQL_VARCHAR23**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 23

class metalpipe.utils.data_structures.**MYSQL_VARCHAR24**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 24

class metalpipe.utils.data_structures.**MYSQL_VARCHAR25**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 25

class metalpipe.utils.data_structures.**MYSQL_VARCHAR256**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 256

class metalpipe.utils.data_structures.**MYSQL_VARCHAR26**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 26

class metalpipe.utils.data_structures.**MYSQL_VARCHAR27**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 27

class metalpipe.utils.data_structures.**MYSQL_VARCHAR28**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 28

class metalpipe.utils.data_structures.**MYSQL_VARCHAR29**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 29

class metalpipe.utils.data_structures.**MYSQL_VARCHAR3**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 3

class metalpipe.utils.data_structures.**MYSQL_VARCHAR30**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 30

class metalpipe.utils.data_structures.**MYSQL_VARCHAR31**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 31

class metalpipe.utils.data_structures.**MYSQL_VARCHAR32**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 32

class metalpipe.utils.data_structures.**MYSQL_VARCHAR32768**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 32768

class metalpipe.utils.data_structures.**MYSQL_VARCHAR4**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 4

class metalpipe.utils.data_structures.**MYSQL_VARCHAR4096**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 4096

class metalpipe.utils.data_structures.**MYSQL_VARCHAR5**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 5

class metalpipe.utils.data_structures.**MYSQL_VARCHAR512**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 512

class metalpipe.utils.data_structures.**MYSQL_VARCHAR6**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 6

class metalpipe.utils.data_structures.**MYSQL_VARCHAR64**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 64

class metalpipe.utils.data_structures.**MYSQL_VARCHAR7**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 7

class metalpipe.utils.data_structures.**MYSQL_VARCHAR8**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 8

class metalpipe.utils.data_structures.**MYSQL_VARCHAR8192**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 8192

class metalpipe.utils.data_structures.**MYSQL_VARCHAR9**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.MYSQL_VARCHAR_BASE*

max_length = 9

class metalpipe.utils.data_structures.**MYSQL_VARCHAR_BASE**(*value*, *original_type=None*, *name=None*)

Bases: *metalpipe.utils.data_structures.DataType*, *metalpipe.utils.data_structures.MySQLTypeSystem*

intermediate_type
alias of *STRING*

python_cast_function
alias of `builtins.str`

class metalpipe.utils.data_structures.**MySQLTypeSystem**

Bases: *metalpipe.utils.data_structures.DataSourceTypeSystem*

Each `TypeSystem` gets a `type_mapping` static method that takes a string and returns the class in the type system named by that string. For example, `int(8)` in a MySQL schema should return the `MYSQL_INTEGER8` class.

static type_mapping(*string*)
Parses the schema strings from MySQL and returns the appropriate class.

class metalpipe.utils.data_structures.**PrimitiveTypeSystem**

Bases: *metalpipe.utils.data_structures.DataSourceTypeSystem*

class metalpipe.utils.data_structures.**PythonTypeSystem**

Bases: *metalpipe.utils.data_structures.DataSourceTypeSystem*

class metalpipe.utils.data_structures.**Row**(**records*, *type_system=None*)

Bases: `object`

A collection of `DataType` objects (typed values). They are dictionaries mapping the names of the values to the `DataType` objects.

concat(*other*, *fail_on_duplicate=True*)

static from_dict(*row_dictionary*, ***kwargs*)
Creates a `Row` object form a dictionary mapping names to values.

is_empty()

keys()
For implementing the mapping protocol.

class metalpipe.utils.data_structures.**STRING**(*value*, *original_type=None*, *name=None*)
Bases: *metalpipe.utils.data_structures.DataType*, *metalpipe.utils.data_structures.IntermediateTypeSystem*

python_cast_function
alias of builtins.str

metalpipe.utils.data_structures.**all_bases**(*obj*)
Return all the class to which obj belongs.

metalpipe.utils.data_structures.**convert_to_type_system**(*obj*, *cls*)

metalpipe.utils.data_structures.**get_type_system**(*obj*)

metalpipe.utils.data_structures.**make_types**()

metalpipe.utils.data_structures.**mysql_type**(*string*)
Parses the schema strings from MySQL and returns the appropriate class.

metalpipe.utils.data_structures.**primitive_to_intermediate_type**(*thing*, *name=None*)

7.4 Network nodes module

Classes that deal with sending and receiving data across the interwebs.

class metalpipe.node_classes.network_nodes.**HttpGetRequest**(*endpoint_template=None*, *endpoint_dict=None*, *protocol='http'*, *retries=5*, *json=True*, ***kwargs*)
Bases: *metalpipe.node.MetalNode*

Node class for making simple GET requests.

process_item()
The input to this function will be a dictionary-like object with parameters to be substituted into the endpoint string and a dictionary with keys and values to be passed in the GET request.

Three use-cases: 1. Endpoint and parameters set initially and never changed. 2. Endpoint and parameters set once at runtime 3. Endpoint and parameters set by upstream messages

class metalpipe.node_classes.network_nodes.**HttpGetRequestPaginator**(*endpoint_dict=None*, *json=True*, *pagination_get_request_key=None*, *endpoint_template=None*, *additional_data_key=None*, *pagination_key=None*, *pagination_template_key=None*, *default_offset_value=''*, ***kwargs*)
Bases: *metalpipe.node.MetalNode*

Node class for HTTP API requests that require paging through sets of results.

This class handles making HTTP GET requests, determining whether there are additional results, and making additional calls if necessary. A typical case is to have an HTTP request something like this:

```
http://www.someapi.com/endpoint_name?resultpage=0
```

with a response like:

```
{"data": "something", "additional_pages": true, "next_page": 1}
```

The response contains some data, a flag `additional_pages` for determining whether there are additional results, and a parameter that gets passed to the next request for retrieving the right page of results (`next_page`). So the next GET request would be:

```
http://www.someapi.com/endpoint_name?resultpage=1
```

This process will repeat until `additional_pages` is false.

In order to use this node class, you'll need to provide arguments that tell the node where to look for the equivalent of `additional_pages` and `next_page`.

1. `endpoint_template`: The parameteried URL for the API.
2. `additional_data_key`: The keypath to the value in the API response that determines whether there are additional pages to request.
3. `pagination_key`: The keypath to the value in the API response that contains the value that would be passed to the API to retrieve the next set of values.
4. `pagination_get_request_key`: The key in the `endpoint_template` that will contain the value of the `pagination_key`.

For our simple example, the arguments would be

1. `endpoint_template: http://www.someapi.com/endpoint_name?resultpage={result_page}`
2. `additional_data_key: ["additional_pages"]`
3. `pagination_key: ["next_page"]`
4. `pagination_get_request_key: result_page`

In addition to those mandatory arguments, you can also optionally specify an `endpoint_dict`, which contains other values that will be substituted into the `endpoint_template`. For example, these APIs often have an option that controls the number of results to provide in each response, like so:

```
http://www.someapi.com/endpoint_name?results={num_results}?resultpage={result_
↪page}
```

For cases like this, the value of `endpoint_dict` is a dictionary mapping keys from the `endpoint_template` to their values. So if you wanted to have ten results per page, you would specify:

```
endpoint_dict = {"num_results": 10}
```

There can be any number of other parameters specified in the `endpoint_dict`.

If there are other keys in the `endpoint_template` that are not provided in the `endpoint_dict`, then the node will try to find them in the current message that's being processed. For example, it is common to have some kind of security token that might be given in an environment variable. If the value of that environment

variable has been provided by some upstream node and placed in the key `token`, then it would be substituted into the URL, provded that the `endpoint_template` had a place for it, such as:

```
http://www.someapi.com/endpoint_name?auth_token={token}?resultpage={result_page}
```

process_item()
Default no-op for nodes.

class metalpipe.node_classes.network_nodes.**PaginatedHttpGetRequest**(*endpoint_template=None*, *additional_data_key=None*, *pagination_key=None*, *pagination_get_request_key=None*, *protocol='http'*, *retries=5*, *default_offset_value=''*, *additional_data_test=<class 'bool'>*, *calling_node=None*)

Bases: object

For handling requests in a semi-general way that require paging through lists of results and repeatedly making GET requests.

get_with_retry(*url*, *error_on_none=True*, ***kwargs*)
Simple method for making requests from flaky endpoints.

responses()
Generator. Yields each response until empty.

7.5 MetalPipeMessage module

The `MetalPipeMesaage` encapsulates the content of each piece of data, along with some useful metadata.

class metalpipe.message.message.**MetalPipeMessage**(*message_content*)
Bases: object

A class that contains the message payloads that are queued for each `MetalPipeProcessor`. It holds the messages and lots of metadata used for logging, monitoring, etc.

7.6 Trigger module

A simple class containing no data, which is intended merely as a trigger, signaling that the downstream node should do something.

class metalpipe.message.trigger.**Trigger**(*previous_trigger_time=None*, *trigger_name=None*)

Bases: object

metalpipe.message.trigger.**hello_world**()

7.7 Batch module

We'll use markers to delimit batches of things, such as serialized files and that kind of thing.

class metalpipe.message.batch.**BatchEnd**(**args*, ***kwargs*)
Bases: object

class metalpipe.message.batch.**BatchStart**(**args*, ***kwargs*)
Bases: object

class metalpipe.message.canary.**Canary**
Bases: object

7.8 MetalPipeQueue module

These are queues that form the directed edges between nodes.

class metalpipe.node_queue.queue.**MetalPipeQueue**(*max_queue_size*, *name=None*)
Bases: object

approximately_full(*error=0.95*)

empty

get()

put(*message*, **args*, *previous_message=None*, ***kwargs*)
Places a message on the output queues. If the message is None, then the queue is skipped.

Messages are MetalPipeMessage objects; the payload of the message is message.message_content.

size()

CHAPTER 8

License

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

H

I

K

L

M

N

O

P