streamparse Documentation

Release 3.7.1

Parsely

Aug 29, 2017

Contents

streamparse lets you run Python code against real-time streams of data. Integrates with Apache Storm.

CHAPTER 1

Quickstart

Dependencies

Java and Clojure

To run local and remote computation clusters, streamparse relies upon a JVM technology called Apache Storm. The integration with this technology is lightweight, and for the most part, you don't need to think about it.

However, to get the library running, you'll need

1. JDK 7+, which you can install with apt-get, homebrew, or an installler; and
2. lein, which you can install from the Leiningen project page or github
3. Apache Storm development environment, which you can install from the Storm project page

   You will need to have at least Apache Storm version 0.10.0 to cooperate with Streamparse.

Confirm that you have `lein` installed by running:

```
> lein version
```

You should get output similar to this:

```
Leiningen 2.3.4 on Java 1.7.0_55 Java HotSpot(TM) 64-Bit Server VM
```

Confirm that you have `storm` installed by running:

```
> storm version
```

You should get output similar to this:

```
Running: java -client -Ddaemon.name= -Dstorm.options= -Dstorm.home=/opt/apache-storm-
↪1.0.1 -Dstorm.log.dir=/opt/apache-storm-1.0.1/logs -Djava.library.path=/usr/local/
↪lib:/opt/local/lib:/usr/lib -Dstorm.conf.file= -cp /opt/apache-storm-1.0.1/lib/
↪reflectasm-1.10.1.jar:/opt/apache-storm-1.0.1/lib/kryo-3.0.3.jar:/opt/apache-storm-
↪1.0.1/lib/log4j-over-slf4j-1.6.6.jar:/opt/apache-storm-1.0.1/lib/clojure-1.7.0.jar:/
↪opt/apache-storm-1.0.1/lib/log4j-slf4j-impl-2.1.jar:/opt/apache-storm-1.0.1/lib/
↪servlet-api-2.5.jar:/opt/apache-storm-1.0.1/lib/disruptor-3.3.2.jar:/opt/apache-
↪storm-1.0.1/lib/objenesis-2.1.jar:/opt/apache-storm-1.0.1/lib/storm-core-1.0.1.jar:/
↪opt/apache-storm-1.0.1/lib/slf4j-api-1.7.7.jar:/opt/apache-storm-1.0.1/lib/storm-
↪rename-hack-1.0.1.jar:/opt/apache-storm-1.0.1/lib/log4j-api-2.1.jar:/opt/apache-
↪storm-1.0.1/lib/log4j-core-2.1.jar:/opt/apache-storm-1.0.1/lib/minlog-1.3.0.jar:/
↪opt/apache-storm-1.0.1/lib/asm-5.0.3.jar:/opt/apache-storm-1.0.1/conf org.apache.
↪storm.utils.VersionInfo
Storm 1.0.1
URL https://git-wip-us.apache.org/repos/asf/storm.git -r␣
↪b5c16f919ad4099e6fb25f1095c9af8b64ac9f91
Branch (no branch)
Compiled by tgoetz on 2016-04-29T20:44Z
From source with checksum 1aea9df01b9181773125826339b9587e
```

If `lein` isn't installed, follow these directions to install it.

If `storm` isn't installed, follow these directions.

Once that's all set, you install streamparse using `pip`:

```
> pip install streamparse
```

Your First Project

When working with streamparse, your first step is to create a project using the command-line tool, `sparse`:

```
> sparse quickstart wordcount

Creating your wordcount streamparse project...
    create    wordcount
    create    wordcount/.gitignore
    create    wordcount/config.json
    create    wordcount/fabfile.py
    create    wordcount/project.clj
    create    wordcount/README.md
    create    wordcount/src
    create    wordcount/src/bolts/
    create    wordcount/src/bolts/__init__.py
    create    wordcount/src/bolts/wordcount.py
    create    wordcount/src/spouts/
    create    wordcount/src/spouts/__init__.py
    create    wordcount/src/spouts/words.py
    create    wordcount/topologies
    create    wordcount/topologies/wordcount.py
    create    wordcount/virtualenvs
    create    wordcount/virtualenvs/wordcount.txt
Done.
```

Try running your topology locally with:

```
> cd wordcount
  sparse run
```

The quickstart project provides a basic wordcount topology example which you can examine and modify. You can inspect the other commands that `sparse` provides by running:

```
> sparse -h
```

If you see an error like:

```
Local Storm version, 1.0.1, is not the same as the version in your project.clj, 0.10.
↪0. The versions must match.
```

You will have to edit your wordcount/project.clj file and change Apache Storm library version to match the one you have installed.

Project Structure

streamparse projects expect to have the following directory layout:

File/Folder	Contents
config.json	Configuration information for all of your topologies.
fabfile.py	Optional custom fabric tasks.
project.clj	leiningen project file (can be used to add external JVM dependencies).
src/	Python source files (bolts/spouts/etc.) for topologies.
tasks.py	Optional custom invoke tasks.
topologies/	Contains topology definitions written using the *Topology DSL*.
virtualenvs/	Contains pip requirements files used to install dependencies on remote Storm servers.

Defining Topologies

Storm's services are Thrift-based and although it is possible to define a topology in pure Python using Thrift. For details see *Topology DSL*.

Let's have a look at the definition file created by using the `sparse quickstart` command.

```
"""
Word count topology
"""

from streamparse import Grouping, Topology

from bolts.wordcount import WordCountBolt
from spouts.words import WordSpout


class WordCount(Topology):
    word_spout = WordSpout.spec()
    count_bolt = WordCountBolt.spec(inputs={word_spout: Grouping.fields('word')},
                                    par=2)
```

In the `count_bolt` bolt, we've told Storm that we'd like the stream of input tuples to be grouped by the named field `word`. Storm offers comprehensive options for stream groupings, but you will most commonly use a **shuffle** or **fields** grouping:

- **Shuffle grouping**: Tuples are randomly distributed across the bolt's tasks in a way such that each bolt is guaranteed to get an equal number of tuples. This is the default grouping if no other is specified.
- **Fields grouping**: The stream is partitioned by the fields specified in the grouping. For example, if the stream is grouped by the "user-id" field, tuples with the same "user-id" will always go to the same task, but tuples with different "user-id"'s may go to different tasks.

There are more options to configure with spouts and bolts, we'd encourage you to refer to our *Topology DSL* docs or Storm's Concepts for more information.

Spouts and Bolts

The general flow for creating new spouts and bolts using streamparse is to add them to your `src` folder and update the corresponding topology definition.

Let's create a spout that emits sentences until the end of time:

```
import itertools

from streamparse.spout import Spout


class SentenceSpout(Spout):
    outputs = ['sentence']

    def initialize(self, stormconf, context):
        self.sentences = [
            "She advised him to take a long holiday, so he immediately quit work and␣
↪took a trip around the world",
            "I was very glad to get a present from her",
            "He will be here in half an hour",
            "She saw him eating a sandwich",
        ]
        self.sentences = itertools.cycle(self.sentences)

    def next_tuple(self):
        sentence = next(self.sentences)
        self.emit([sentence])

    def ack(self, tup_id):
        pass  # if a tuple is processed properly, do nothing

    def fail(self, tup_id):
        pass  # if a tuple fails to process, do nothing
```

The magic in the code above happens in the `initialize()` and `next_tuple()` functions. Once the spout enters the main run loop, streamparse will call your spout's `initialize()` method. After initialization is complete, streamparse will continually call the spout's `next_tuple()` method where you're expected to emit tuples that match whatever you've defined in your topology definition.

Now let's create a bolt that takes in sentences, and spits out words:

```
import re

from streamparse.bolt import Bolt

class SentenceSplitterBolt(Bolt):
    outputs = ['word']

    def process(self, tup):
        sentence = tup.values[0]  # extract the sentence
        sentence = re.sub(r"[,.;!\?]", "", sentence)  # get rid of punctuation
        words = [[word.strip()] for word in sentence.split(" ") if word.strip()]
        if not words:
            # no words to process in the sentence, fail the tuple
            self.fail(tup)
            return

        for word in words:
            self.emit([word])
        # tuple acknowledgement is handled automatically
```

The bolt implementation is even simpler. We simply override the default `process()` method which streamparse calls when a tuple has been emitted by an incoming spout or bolt. You are welcome to do whatever processing you would like in this method and can further emit tuples or not depending on the purpose of your bolt.

If your `process()` method completes without raising an Exception, streamparse will automatically ensure any emits you have are anchored to the current tuple being processed and acknowledged after `process()` completes.

If an Exception is raised while `process()` is called, streamparse automatically fails the current tuple prior to killing the Python process.

Failed Tuples

In the example above, we added the ability to fail a sentence tuple if it did not provide any words. What happens when we fail a tuple? Storm will send a "fail" message back to the spout where the tuple originated from (in this case `SentenceSpout`) and streamparse calls the spout's `fail()` method. It's then up to your spout implementation to decide what to do. A spout could retry a failed tuple, send an error message, or kill the topology. See *Dealing With Errors* for more discussion.

Bolt Configuration Options

You can disable the automatic acknowleding, anchoring or failing of tuples by adding class variables set to false for: `auto_ack`, `auto_anchor` or `auto_fail`. All three options are documented in `streamparse.bolt.Bolt`.

Example:

```
from streamparse.bolt import Bolt

class MyBolt(Bolt):

    auto_ack = False
    auto_fail = False

    def process(self, tup):
        # do stuff...
        if error:
```

```
            self.fail(tup)   # perform failure manually
        self.ack(tup)   # perform acknowledgement manually
```

Handling Tick Tuples

Ticks tuples are built into Storm to provide some simple forms of cron-like behaviour without actually having to use cron. You can receive and react to tick tuples as timer events with your python bolts using streamparse too.

The first step is to override `process_tick()` in your custom Bolt class. Once this is overridden, you can set the storm option `topology.tick.tuple.freq.secs=<frequency>` to cause a tick tuple to be emitted every `<frequency>` seconds.

You can see the full docs for `process_tick()` in `streamparse.bolt.Bolt`.

Example:

```
from streamparse.bolt import Bolt

class MyBolt(Bolt):

    def process_tick(self, freq):
        # An action we want to perform at some regular interval...
        self.flush_old_state()
```

Then, for example, to cause `process_tick()` to be called every 2 seconds on all of your bolts that override it, you can launch your topology under `sparse run` by setting the appropriate -o option and value as in the following example:

```
$ sparse run -o "topology.tick.tuple.freq.secs=2" ...
```

Remote Deployment

Setting up a Storm Cluster

See Storm's Setting up a Storm Cluster.

Submit

When you are satisfied that your topology works well via testing with:

```
> sparse run -d
```

You can submit your topology to a remote Storm cluster using the command:

```
sparse submit [--environment <env>] [--name <topology>] [-dv]
```

Before submitting, you have to have at least one environment configured in your project's `config.json` file. Let's create a sample environment called "prod" in our `config.json` file:

```
{
    "serializer": "json",
    "topology_specs": "topologies/",
```

```
    "virtualenv_specs": "virtualenvs/",
    "envs": {
        "prod": {
            "user": "storm",
            "nimbus": "storm1.my-cluster.com",
            "workers": [
                "storm1.my-cluster.com",
                "storm2.my-cluster.com",
                "storm3.my-cluster.com"
            ],
            "log": {
                "path": "/var/log/storm/streamparse",
                "file": "pystorm_{topology_name}_{component_name}_{task_id}_{pid}.log
↪",
                "max_bytes": 100000,
                "backup_count": 10,
                "level": "info"
            },
            "use_ssh_for_nimbus": true,
            "virtualenv_root": "/data/virtualenvs/"
        }
    }
}
```

We've now defined a `prod` environment that will use the user `storm` when deploying topologies. Before submitting the topology though, streamparse will automatically take care of instaling all the dependencies your topology requires. It does this by sshing into everyone of the nodes in the `workers` config variable and building a virtualenv using the the project's local `virtualenvs/<topology_name>.txt` requirements file.

This implies a few requirements about the user you specify per environment:

1. Must have ssh access to all servers in your Storm cluster
2. Must have write access to the `virtualenv_root` on all servers in your Storm cluster

streamparse also assumes that virtualenv is installed on all Storm servers.

Once an environment is configured, we could deploy our wordcount topology like so:

```
> sparse submit
```

Seeing as we have only one topology and environment, we don't need to specify these explicitly. streamparse will now:

1. Package up a JAR containing all your Python source files
2. Build a virtualenv on all your Storm workers (in parallel)
3. Submit the topology to the `nimbus` server

Disabling & Configuring Virtualenv Creation

If you do not have ssh access to all of the servers in your Storm cluster, but you know they have all of the requirements for your Python code installed, you can set `"use_virtualenv"` to `false` in `config.json`.

If you have virtualenvs on your machines that you would like streamparse to use, but not update or manage, you can set `"install_virtualenv"` to `false` in `config.json`.

If you would like to pass command-line flags to virtualenv, you can set `"virtualenv_flags"` in `config.json`, for example:

```
"virtualenv_flags": "-p /path/to/python"
```

Note that this only applies when the virtualenv is created, not when an existing virtualenv is used.

If you would like to share a single virtualenv across topologies, you can set `"virtualenv_name"` in `config.json` which overrides the default behaviour of using the topology name for virtualenv. Updates to a shared virtualenv should be done after shutting down topologies, as code changes in running topologies may cause errors.

Using unofficial versions of Storm

If you wish to use streamparse with unofficial versions of storm (such as the HDP Storm) you should set `:repositories` in your `project.clj` to point to the Maven repository containing the JAR you want to use, and set the version in `:dependencies` to match the desired version of Storm.

For example, to use the version supplied by HDP, you would set `:repositories` to:

```
:repositories {"HDP Releases" "http://repo.hortonworks.com/content/
repositories/releases"}
```

Local Clusters

Streamparse assumes that your Storm cluster is not on your local machine. If it is, such as the case with VMs or Docker images, change `"use_ssh_for_nimbus"` in `config.json` to `false`.

Setting Submit Options in config.json

If you frequently use the same options to `sparse submit` in your project, you can set them in `config.json` using the `options` key in your environment settings. For example:

```
{
    "topology_specs": "topologies/",
    "virtualenv_specs": "virtualenvs/",
    "envs": {
        "vagrant": {
            "user": "vagrant",
            "nimbus": "streamparse-box",
            "workers": [
                "streamparse-box"
            ],
            "virtualenv_root": "/data/virtualenvs",
            "options": {
                "topology.environment": {
                    "LD_LIBRARY_PATH": "/usr/local/lib/"
                }
            }
        }
    }
}
```

You can also set the `--worker` and `--acker` parameters in `config.json` via the `worker_count` and `acker_count` keys in your environment settings.

```
{
    "topology_specs": "topologies/",
```

```
    "virtualenv_specs": "virtualenvs/",
    "envs": {
        "vagrant": {
            "user": "vagrant",
            "nimbus": "streamparse-box",
            "workers": [
                "streamparse-box"
            ],
            "virtualenv_root": "/data/virtualenvs",
            "acker_count": 1,
            "worker_count": 1
        }
    }
}
```

Logging

The Storm supervisor needs to have access to the `log.path` directory for logging to work (in the example above, `/var/log/storm/streamparse`). If you have properly configured the `log.path` option in your config, streamparse will use the value for the `log.file` option to set up log files for each Storm worker in this path. The filename can be customized further by using certain named placeholders. The default filename is set to:

```
pystorm_{topology_name}_{component_name}_{task_id}_{pid}.log
```

Where:

- `topology_name`: is the `topology.name` variable set in Storm
- `component_name`: is the name of the currently executing component as defined in your topology definition file (.clj file)
- `task_id`: is the task ID running this component in the topology
- `pid`: is the process ID of the Python process

streamparse uses Python's `logging.handlers.RotatingFileHandler` and by default will only save 10 1 MB log files (10 MB in total), but this can be tuned with the `log.max_bytes` and `log.backup_count` variables.

The default logging level is set to `INFO`, but if you can tune this with the `log.level` setting which can be one of critical, error, warning, info or debug. **Note** that if you perform `sparse run` or `sparse submit` with the `--debug` set, this will override your `log.level` setting and set the log level to debug.

When running your topology locally via `sparse run`, your log path will be automatically set to `/path/to/your/streamparse/project/logs`.

New in version 3.0.0.

CHAPTER 2

Topologies

Storm topologies are described as a Directed Acyclic Graph (DAG) of Storm components, namely *bolts* and *spouts*.

Topology DSL

To simplify the process of creating Storm topologies, streamparse features a Python Topology DSL. It lets you specify topologies as complex as those you can in Java or Clojure, but in concise, readable Python.

Topology files are located in `topologies` in your streamparse project folder. There can be any number of topology files for your project in this directory.

- `topologies/my_topology.py`
- `topologies/my_other_topology.py`
- `topologies/my_third_topology.py`
- ...

A valid *`Topology`* may only have *`Bolt`* and *`Spout`* attributes.

Simple Python Example

The first step to putting together a topology, is creating the bolts and spouts, so let's assume we have the following bolt and spout:

```
from collections import Counter

from redis import StrictRedis

from streamparse import Bolt


class WordCountBolt(Bolt):
```

```
    outputs = ['word', 'count']

    def initialize(self, conf, ctx):
        self.counter = Counter()
        self.total = 0

    def _increment(self, word, inc_by):
        self.counter[word] += inc_by
        self.total += inc_by

    def process(self, tup):
        word = tup.values[0]
        self._increment(word, 10 if word == "dog" else 1)
        if self.total % 1000 == 0:
            self.logger.info("counted %i words", self.total)
        self.emit([word, self.counter[word]])


class RedisWordCountBolt(Bolt):
    def initialize(self, conf, ctx):
```

```
from itertools import cycle

from streamparse import Spout


class WordSpout(Spout):
    outputs = ['word']

    def initialize(self, stormconf, context):
        self.words = cycle(['dog', 'cat', 'zebra', 'elephant'])

    def next_tuple(self):
        word = next(self.words)
        self.emit([word])
```

One important thing to note is that we have added an `outputs` attribute to these classes, which specify the names of the output fields that will be produced on their `default` streams. If we wanted to specify multiple streams, we could do that by specifying a list of *`Stream`* objects.

Now let's hook up the bolt to read from the spout:

```
"""
Word count topology (in memory)
"""

from streamparse import Grouping, Topology

from bolts import WordCountBolt
from spouts import WordSpout


class WordCount(Topology):
    word_spout = WordSpout.spec()
    count_bolt = WordCountBolt.spec(inputs={word_spout: Grouping.fields('word')},
                                    par=2)
```

Note: Your project's `src` directory gets added to `sys.path` before your topology is imported, so you should use absolute imports based on that.

As you can see, *`streamparse.Bolt.spec()`* and *`streamparse.Spout.spec()`* methods allow us to specify information about the components in your topology and how they connect to each other. Their respective docstrings outline all of the possible ways they can be used.

Java Components

The topology DSL fully supports JVM-based bolts and spouts via the *`JavaBolt`* and *`JavaSpout`* classes.

Here's an example of how we would use the Storm Kafka Spout:

```
"""
Pixel count topology
"""

from streamparse import Grouping, JavaSpout, Topology

from bolts.pixel_count import PixelCounterBolt
from bolts.pixel_deserializer import PixelDeserializerBolt


class PixelCount(Topology):
    pixel_spout = JavaSpout.spec(name="pixel-spout",
                                 full_class_name="pixelcount.spouts.PixelSpout",
                                 args_list=[],
                                 outputs=["pixel"])
    pixel_deserializer = PixelDeserializerBolt.spec(name='pixel-deserializer-bolt',
                                                    inputs=[pixel_spout])
    pixel_counter = PixelCounterBolt.spec(name='pixel-count-bolt',
                                          inputs={pixel_deserializer:
                                                  Grouping.fields('url')},
                                          config={"topology.tick.tuple.freq.secs": 1})
```

One limitation of the Thrift interface we use to send the topology to Storm is that the constructors for Java components can only be passed basic Python data types: *bool*, *bytes*, *float*, *int*, and *str*.

Note: If you are passing strings as constructor arguments to Java components via the `args_list` parameter, you must use *unicode* literals to do so in Python 2. Otherwise, Storm will raise an exception.

Components in Other Languages

If you have components that are written in languages other than Java or Python, you can have those as part of your topology as well—assuming you're using the corresponding multi-lang library for that language.

To do that you just need to use the *`streamparse.ShellBolt.spec()`* and *`streamparse.ShellSpout.spec()`* methods. They take `command` and `script` arguments to specify a binary to run and its string-separated arguments.

Multiple Streams

To specify that a component has multiple output streams, instead of using a list of strings for `outputs`, you must specify a list of *`Stream`* objects, as shown below.

```
class FancySpout(Spout):
    outputs = [Stream(fields=['good_data'], name='default'),
               Stream(fields=['bad_data'], name='errors')]
```

To select one of those streams as the input for a downstream *`Bolt`*, you simply use `[]` to specify the stream you want. Without any stream specified, the `default` stream will be used.

```
class ExampleTopology(Topology):
    fancy_spout = FancySpout.spec()
    error_bolt = ErrorBolt.spec(inputs=[fancy_spout['errors']])
    process_bolt = ProcessBolt.spec(inputs=[fancy_spout])
```

Groupings

By default, Storm uses a `SHUFFLE` grouping to route tuples to particular executors for a given component, but you can also specify other groupings by using the appropriate *`Grouping`* attribute. The most common grouping is probably the *`fields()`* grouping, which will send all the tuples with the same value for the specified fields to the same executor. This can be seen in the prototypical word count topology:

```
"""
Word count topology (in memory)
"""

from streamparse import Grouping, Topology

from bolts import WordCountBolt
from spouts import WordSpout


class WordCount(Topology):
    word_spout = WordSpout.spec()
    count_bolt = WordCountBolt.spec(inputs={word_spout: Grouping.fields('word')},
                                    par=2)
```

Topology Cycles

On rare occassions, you may want to create a cyclical topology. This may not seem easily done with the current topology DSL, but there is a workaround you can use: manually declaring a temporary lower-level *:class:~streamparse.thrift.GlobalStreamId* that you can refer to in multiple places.

The following code creates a *`Topology`* with a cycle between its two Bolts.

```
from streamparse.thrift import GlobalStreamId

# Create a reference to B's output stream before we even declare Topology
b_stream = GlobalStreamId(componentId='b_bolt', streamId='default')

class CyclicalTopology(Topology):
    some_spout = SomeSpout.spec()
    # Include our saved stream in your list of inputs for A
```

```
    a_bolt = A.spec(name="A", inputs=[some_spout, b_stream])
    # Have B get input from A like normal
    b_bolt = B.spec(name="B", inputs=[a_bolt])
```

Topology-Level Configuration

If you want to set a config option for all components in your topology, like `topology.environment`, you can do that by adding a `config` class attribute to your *`Topology`* that is a *dict* mapping from option names to their values. For example:

```
class WordCount(Topology):
    config = {'topology.environment': {'LD_LIBRARY_PATH': '/usr/local/lib/'}}
    ...
```

Running Topologies

What Streamparse Does

When you run a topology either locally or by submitting to a cluster, streamparse will

1. Bundle all of your code into a JAR
2. Build a Thrift Topology struct out of your Python topology definition.
3. Pass the Thrift Topology struct to Nimbus on your Storm cluster.

If you invoked streamparse with `sparse run`, your code is executed directly from the `src/` directory.

If you submitted to a cluster with `sparse submit`, streamparse uses `lein` to compile the `src` directory into a jar file, which is run on the cluster. Lein uses the `project.clj` file located in the root of your project. This file is a standard lein project file and can be customized according to your needs.

Dealing With Errors

When detecting an error, bolt code can call its *`fail()`* method in order to have Storm call the respective spout's *`fail()`* method. Known error/failure cases result in explicit callbacks to the spout using this approach.

Exceptions which propagate without being caught will cause the component to crash. On `sparse run`, the entire topology will stop execution. On a running cluster (i.e. `sparse submit`), Storm will auto-restart the crashed component and the spout will receive a *`fail()`* call.

If the spout's fail handling logic is to hold back the tuple and not re-emit it, then things will keep going. If it re-emits it, then it may crash that component again. Whether the topology is tolerant of the failure depends on how you implement failure handling in your spout.

Common approaches are to:

- Append errant tuples to some sort of error log or queue for manual inspection later, while letting processing continue otherwise.
- Attempt 1 or 2 retries before considering the tuple a failure, if the error was likely an transient problem.
- Ignore the failed tuple, if appropriate to the application.

Parallelism and Workers

In general, use the ‘‘par‘‘ “parallelism hint” parameter per spout and bolt in your configuration to control the number of Python processes per component.

Reference: Understanding the Parallelism of a Storm Topology

Storm parallelism entities:

- A *worker process* is a JVM, i.e. a Java process.
- An *executor* is a thread that is spawned by a worker process.
- A *task* performs the actual data processing. (To simplify, you can think of it as a Python callable.)

Spout and bolt specs take a `par` keyword to provide a parallelism hint to Storm for the number of executors (threads) to use for the given spout/bolt; for example, `par=2` is a hint to use two executors. Because streamparse implements spouts and bolts as independent Python processes, setting `par=N` results in N Python processes for the given spout/bolt.

Many streamparse applications will need only to set this parallelism hint to control the number of resulting Python processes when tuning streamparse configuration. For the underlying topology workers, streamparse sets a default of 2 workers, which are independent JVM processes for Storm. This allows a topology to continue running when one worker process dies; the other is around until the dead process restarts.

Both `sparse run` and `sparse submit` accept a `-p N` command-line flag to set the number of topology workers to N. For convenience, this flag also sets the number of Storm's underlying messaging reliability *acker bolts* to the same N value. In the event that you need it (and you understand Storm ackers), use the `-a` and `-w` command-line flags instead of `-p` to control the number of acker bolts and the number of workers, respectively. The `sparse` command does not support Storm's rebalancing features; use `sparse submit -f -p N` to kill the running topology and redeploy it with N workers.

Note that the underlying Storm thread implementation, LMAX Disruptor, is designed with high-performance inter-thread messaging as a goal. Rule out Python-level issues when tuning your topology:

- bottlenecks where the number of spout and bolt processes are out of balance
- serialization/deserialization overhead of more data emitted than you need
- slow routines/callables in your code

CHAPTER 3

API

Tuples

class `streamparse.`**`Tuple`**(*id*, *component*, *stream*, *task*, *values*)
Storm's primitive data type passed around via streams.

Variables

- **`id`** – the ID of the Tuple.
- **`component`** – component that the Tuple was generated from.
- **`stream`** – the stream that the Tuple was emitted into.
- **`task`** – the task the Tuple was generated from.
- **`values`** – the payload of the Tuple where data is stored.

You should never have to instantiate an instance of a *`streamparse.Tuple`* yourself as streamparse handles this for you prior to, for example, a *`streamparse.Bolt`*'s `process()` method being called.

None of the emit methods for bolts or spouts require that you pass a *`streamparse.Tuple`* instance.

Components

Both *`streamparse.Bolt`* and *`streamparse.Spout`* inherit from a common base-class, `streamparse.storm.component.Component`. It extends pystorm's code for handling Multi-Lang IPC between Storm and Python and adds suport for our Python *Topology DSL*.

Spouts

Spouts are data sources for topologies, they can read from any data source and emit tuples into streams.

class streamparse.**Spout**(*input_stream=<open file '<stdin>', mode 'r'>, output_stream=<open file '<stdout>', mode 'w'>, rdb_signal=u'SIGUSR1', serializer=u'json'*)

Bases: pystorm.spout.Spout, streamparse.storm.spout.ShellSpout

pystorm Spout with streamparse-specific additions

ack(*tup_id*)

Called when a bolt acknowledges a Tuple in the topology.

Parameters **tup_id** (*str*) – the ID of the Tuple that has been fully acknowledged in the topology.

activate()

Called when the Spout has been activated after being deactivated.

Note: This requires at least Storm 1.1.0.

---

deactivate()

Called when the Spout has been deactivated.

Note: This requires at least Storm 1.1.0.

---

emit(*tup*, *tup_id=None*, *stream=None*, *direct_task=None*, *need_task_ids=False*)

Emit a spout Tuple message.

Parameters

- **tup** (*list or tuple*) – the Tuple to send to Storm, should contain only JSON-serializable data.
- **tup_id** (*str*) – the ID for the Tuple. Leave this blank for an unreliable emit.
- **stream** (*str*) – ID of the stream this Tuple should be emitted to. Leave empty to emit to the default stream.
- **direct_task** (*int*) – the task to send the Tuple to if performing a direct emit.
- **need_task_ids** (*bool*) – indicate whether or not you'd like the task IDs the Tuple was emitted (default: False).

Returns None, unless need_task_ids=True, in which case it will be a list of task IDs that the Tuple was sent to if. Note that when specifying direct_task, this will be equal to [direct_task].

fail(*tup_id*)

Called when a Tuple fails in the topology

A spout can choose to emit the Tuple again or ignore the fail. The default is to ignore.

Parameters **tup_id** (*str*) – the ID of the Tuple that has failed in the topology either due to a bolt calling fail() or a Tuple timing out.

initialize(*storm_conf*, *context*)

Called immediately after the initial handshake with Storm and before the main run loop. A good place to initialize connections to data sources.

Parameters

- **storm_conf** (*dict*) – the Storm configuration for this component. This is the configuration provided to the topology, merged in with cluster configuration on the worker node.

- **context** (*dict*) – information about the component's place within the topology such as: task IDs, inputs, outputs etc.

is_heartbeat(*tup*)

Returns Whether or not the given Tuple is a heartbeat

log(*message*, *level=None*)
Log a message to Storm optionally providing a logging level.

Parameters

- **message** (*str*) – the log message to send to Storm.
- **level** (*str*) – the logging level that Storm should use when writing the `message`. Can be one of: trace, debug, info, warn, or error (default: `info`).

Warning: This will send your message to Storm regardless of what level you specify. In almost all cases, you are better of using `Component.logger` and not setting `pystorm.log.path`, because that will use a `pystorm.component.StormHandler` to do the filtering on the Python side (instead of on the Java side after taking the time to serialize your message and send it to Storm).

next_tuple()
Implement this function to emit Tuples as necessary.

This function should not block, or Storm will think the spout is dead. Instead, let it return and pystorm will send a noop to storm, which lets it know the spout is functioning.

raise_exception(*exception*, *tup=None*)
Report an exception back to Storm via logging.

Parameters

- **exception** – a Python exception.
- **tup** – a `Tuple` object.

read_handshake()
Read and process an initial handshake message from Storm.

read_message()
Read a message from Storm via serializer.

report_metric(*name*, *value*)
Report a custom metric back to Storm.

Parameters

- **name** – Name of the metric. This can be anything.
- **value** – Value of the metric. This is usually a number.

Only supported in Storm 0.9.3+.

run()
Main run loop for all components.

Performs initial handshake with Storm and reads Tuples handing them off to subclasses. Any exceptions are caught and logged back to Storm prior to the Python process exiting.

> **Warning:** Subclasses should **not** override this method.

send_message(*message*)
Send a message to Storm via stdout.

classmethod spec(*name=None*, *par=None*, *config=None*)
Create a `ShellBoltSpec` for a Python Spout.

This spec represents this Spout in a *Topology*.

Parameters

- **name** (*str*) – Name of this Spout. Defaults to name of *Topology* attribute this is assigned to.
- **par** (*int*) – Parallelism hint for this Spout. For Python Components, this works out to be the number of Python processes running it in the the topology (across all machines). See *Parallelism and Workers*.

  > **Note:** This can also be specified as an attribute of your `Spout` subclass.

- **config** (*dict*) – Component-specific config settings to pass to Storm.

  > **Note:** This can also be specified as an attribute of your `Spout` subclass.

> **Note:** This method does not take a `outputs` argument because `outputs` should be an attribute of your `Spout` subclass.

class streamparse.**ReliableSpout**(**args*, ***kwargs*)
Bases: `pystorm.spout.ReliableSpout`, `streamparse.storm.spout.Spout`

pystorm ReliableSpout with streamparse-specific additions

ack(*tup_id*)
Called when a bolt acknowledges a Tuple in the topology.

Parameters **tup_id** (*str*) – the ID of the Tuple that has been fully acknowledged in the topology.

activate()
Called when the Spout has been activated after being deactivated.

> **Note:** This requires at least Storm 1.1.0.

deactivate()
Called when the Spout has been deactivated.

> **Note:** This requires at least Storm 1.1.0.

emit(*tup*, *tup_id=None*, *stream=None*, *direct_task=None*, *need_task_ids=False*)
Emit a spout Tuple & add metadata about it to *unacked_tuples*.

In order for this to work, *tup_id* is a required parameter.

See `Bolt.emit()`.

fail(*tup_id*)
Called when a Tuple fails in the topology

A reliable spout will replay a failed tuple up to `max_fails` times.

Parameters **tup_id** (*str*) – the ID of the Tuple that has failed in the topology either due to a bolt calling `fail()` or a Tuple timing out.

initialize(*storm_conf*, *context*)
Called immediately after the initial handshake with Storm and before the main run loop. A good place to initialize connections to data sources.

Parameters

- **storm_conf** (*dict*) – the Storm configuration for this component. This is the configuration provided to the topology, merged in with cluster configuration on the worker node.
- **context** (*dict*) – information about the component's place within the topology such as: task IDs, inputs, outputs etc.

is_heartbeat(*tup*)

Returns Whether or not the given Tuple is a heartbeat

log(*message*, *level=None*)
Log a message to Storm optionally providing a logging level.

Parameters

- **message** (*str*) – the log message to send to Storm.
- **level** (*str*) – the logging level that Storm should use when writing the `message`. Can be one of: trace, debug, info, warn, or error (default: `info`).

Warning: This will send your message to Storm regardless of what level you specify. In almost all cases, you are better of using `Component.logger` and not setting `pystorm.log.path`, because that will use a `pystorm.component.StormHandler` to do the filtering on the Python side (instead of on the Java side after taking the time to serialize your message and send it to Storm).

next_tuple()
Implement this function to emit Tuples as necessary.

This function should not block, or Storm will think the spout is dead. Instead, let it return and pystorm will send a noop to storm, which lets it know the spout is functioning.

raise_exception(*exception*, *tup=None*)
Report an exception back to Storm via logging.

Parameters

- **exception** – a Python exception.
- **tup** – a `Tuple` object.

read_handshake()
Read and process an initial handshake message from Storm.

read_message()
Read a message from Storm via serializer.

report_metric(*name*, *value*)
Report a custom metric back to Storm.

Parameters

- **name** – Name of the metric. This can be anything.
- **value** – Value of the metric. This is usually a number.

Only supported in Storm 0.9.3+.

run()
Main run loop for all components.

Performs initial handshake with Storm and reads Tuples handing them off to subclasses. Any exceptions are caught and logged back to Storm prior to the Python process exiting.

Warning: Subclasses should **not** override this method.

send_message(*message*)
Send a message to Storm via stdout.

spec(*name=None*, *par=None*, *config=None*)
Create a `ShellBoltSpec` for a Python Spout.

This spec represents this Spout in a *`Topology`*.

Parameters

- **name** (*str*) – Name of this Spout. Defaults to name of *`Topology`* attribute this is assigned to.
- **par** (*int*) – Parallelism hint for this Spout. For Python Components, this works out to be the number of Python processes running it in the the topology (across all machines). See *Parallelism and Workers*.

  Note: This can also be specified as an attribute of your `Spout` subclass.

- **config** (*dict*) – Component-specific config settings to pass to Storm.

  Note: This can also be specified as an attribute of your `Spout` subclass.

Note: This method does not take a `outputs` argument because `outputs` should be an attribute of your `Spout` subclass.

Bolts

class `streamparse.`**Bolt**(**args*, ***kwargs*)
Bases: `pystorm.bolt.Bolt`, `streamparse.storm.bolt.ShellBolt`

pystorm Bolt with streamparse-specific additions

ack(*tup*)
Indicate that processing of a Tuple has succeeded.

Parameters **tup** (str or pystorm.component.Tuple) – the Tuple to acknowledge.

emit(*tup*, *stream=None*, *anchors=None*, *direct_task=None*, *need_task_ids=False*)
Emit a new Tuple to a stream.

Parameters

- **tup** (list or pystorm.component.Tuple) – the Tuple payload to send to Storm, should contain only JSON-serializable data.
- **stream** (*str*) – the ID of the stream to emit this Tuple to. Specify None to emit to default stream.
- **anchors** (*list*) – IDs the Tuples (or pystorm.component.Tuple instances) which the emitted Tuples should be anchored to. If auto_anchor is set to True and you have not specified anchors, anchors will be set to the incoming/most recent Tuple ID(s).
- **direct_task** (*int*) – the task to send the Tuple to.
- **need_task_ids** (*bool*) – indicate whether or not you'd like the task IDs the Tuple was emitted (default: False).

Returns None, unless need_task_ids=True, in which case it will be a list of task IDs that the Tuple was sent to if. Note that when specifying direct_task, this will be equal to [direct_task].

fail(*tup*)
Indicate that processing of a Tuple has failed.

Parameters **tup** (str or pystorm.component.Tuple) – the Tuple to fail (its id if str).

initialize(*storm_conf*, *context*)
Called immediately after the initial handshake with Storm and before the main run loop. A good place to initialize connections to data sources.

Parameters

- **storm_conf** (*dict*) – the Storm configuration for this component. This is the configuration provided to the topology, merged in with cluster configuration on the worker node.
- **context** (*dict*) – information about the component's place within the topology such as: task IDs, inputs, outputs etc.

is_heartbeat(*tup*)

Returns Whether or not the given Tuple is a heartbeat

is_tick(*tup*)

Returns Whether or not the given Tuple is a tick Tuple

log(*message*, *level=None*)
Log a message to Storm optionally providing a logging level.

Parameters

- **message** (*str*) – the log message to send to Storm.
- **level** (*str*) – the logging level that Storm should use when writing the message. Can be one of: trace, debug, info, warn, or error (default: info).

> **Warning:** This will send your message to Storm regardless of what level you specify. In almost all cases, you are better of using `Component.logger` and not setting `pystorm.log.path`, because that will use a `pystorm.component.StormHandler` to do the filtering on the Python side (instead of on the Java side after taking the time to serialize your message and send it to Storm).

process(*tup*)
Process a single Tuple `pystorm.component.Tuple` of input

This should be overridden by subclasses. `pystorm.component.Tuple` objects contain metadata about which component, stream and task it came from. The actual values of the Tuple can be accessed by calling `tup.values`.

Parameters **tup** (`pystorm.component.Tuple`) – the Tuple to be processed.

process_tick(*tup*)
Process special 'tick Tuples' which allow time-based behaviour to be included in bolts.

Default behaviour is to ignore time ticks. This should be overridden by subclasses who wish to react to timer events via tick Tuples.

Tick Tuples will be sent to all bolts in a toplogy when the storm configuration option 'topology.tick.tuple.freq.secs' is set to an integer value, the number of seconds.

Parameters **tup** (`pystorm.component.Tuple`) – the Tuple to be processed.

raise_exception(*exception*, *tup=None*)
Report an exception back to Storm via logging.

Parameters

- **exception** – a Python exception.
- **tup** – a `Tuple` object.

read_handshake()
Read and process an initial handshake message from Storm.

read_message()
Read a message from Storm via serializer.

read_tuple()
Read a tuple from the pipe to Storm.

report_metric(*name*, *value*)
Report a custom metric back to Storm.

Parameters

- **name** – Name of the metric. This can be anything.
- **value** – Value of the metric. This is usually a number.

Only supported in Storm 0.9.3+.

run()
Main run loop for all components.

Performs initial handshake with Storm and reads Tuples handing them off to subclasses. Any exceptions are caught and logged back to Storm prior to the Python process exiting.

> **Warning:** Subclasses should **not** override this method.

send_message(*message*)
Send a message to Storm via stdout.

classmethod spec(*name=None*, *inputs=None*, *par=None*, *config=None*)
Create a `ShellBoltSpec` for a Python Bolt.

This spec represents this Bolt in a *Topology*.

Parameters

- **name** (*str*) – Name of this Bolt. Defaults to name of *Topology* attribute this is assigned to.
- **inputs** – Streams that feed into this Bolt.

  Two forms of this are acceptable:

  1. A *dict* mapping from `ComponentSpec` to *Grouping*.
  2. A *list* of *Stream* or `ComponentSpec`.
- **par** (*int*) – Parallelism hint for this Bolt. For Python Components, this works out to be the number of Python processes running it in the the topology (across all machines). See *Parallelism and Workers*.

  Note: This can also be specified as an attribute of your `Bolt` subclass.
- **config** (*dict*) – Component-specific config settings to pass to Storm.

  Note: This can also be specified as an attribute of your `Bolt` subclass.

Note: This method does not take a `outputs` argument because `outputs` should be an attribute of your `Bolt` subclass.

class `streamparse.`**BatchingBolt**(**args*, ***kwargs*)
Bases: `pystorm.bolt.BatchingBolt`, `streamparse.storm.bolt.Bolt`

pystorm BatchingBolt with streamparse-specific additions

ack(*tup*)
Indicate that processing of a Tuple has succeeded.

Parameters **tup** (`str` or `pystorm.component.Tuple`) – the Tuple to acknowledge.

emit(*tup*, ***kwargs*)
Modified emit that will not return task IDs after emitting.

See `pystorm.component.Bolt` for more information.

Returns `None`.

fail(*tup*)
Indicate that processing of a Tuple has failed.

Parameters **tup** (`str` or `pystorm.component.Tuple`) – the Tuple to fail (its `id` if `str`).

group_key(*tup*)
Return the group key used to group Tuples within a batch.

By default, returns None, which put all Tuples in a single batch, effectively just time-based batching. Override this to create multiple batches based on a key.

Parameters **tup** (`pystorm.component.Tuple`) – the Tuple used to extract a group key

Returns Any `hashable` value.

initialize(*storm_conf*, *context*)
Called immediately after the initial handshake with Storm and before the main run loop. A good place to initialize connections to data sources.

Parameters

- **storm_conf** (*dict*) – the Storm configuration for this component. This is the configuration provided to the topology, merged in with cluster configuration on the worker node.
- **context** (*dict*) – information about the component's place within the topology such as: task IDs, inputs, outputs etc.

is_heartbeat(*tup*)

Returns Whether or not the given Tuple is a heartbeat

is_tick(*tup*)

Returns Whether or not the given Tuple is a tick Tuple

log(*message*, *level=None*)
Log a message to Storm optionally providing a logging level.

Parameters

- **message** (*str*) – the log message to send to Storm.
- **level** (*str*) – the logging level that Storm should use when writing the `message`. Can be one of: trace, debug, info, warn, or error (default: `info`).

> **Warning:** This will send your message to Storm regardless of what level you specify. In almost all cases, you are better of using `Component.logger` and not setting `pystorm.log.path`, because that will use a `pystorm.component.StormHandler` to do the filtering on the Python side (instead of on the Java side after taking the time to serialize your message and send it to Storm).

process(*tup*)
Group non-tick Tuples into batches by `group_key`.

> **Warning:** This method should **not** be overriden. If you want to tweak how Tuples are grouped into batches, override `group_key`.

process_batch(*key*, *tups*)
Process a batch of Tuples. Should be overridden by subclasses.

Parameters

- **key** (*hashable*) – the group key for the list of batches.
- **tups** (*list*) – a *list* of `pystorm.component.Tuple` s for the group.

process_batches()
Iterate through all batches, call process_batch on them, and ack.

Separated out for the rare instances when we want to subclass BatchingBolt and customize what mechanism causes batches to be processed.

process_tick(*tick_tup*)
Increment tick counter, and call `process_batch` for all current batches if tick counter exceeds `ticks_between_batches`.

See `pystorm.component.Bolt` for more information.

> **Warning:** This method should **not** be overriden. If you want to tweak how Tuples are grouped into batches, override `group_key`.

raise_exception(*exception*, *tup=None*)
Report an exception back to Storm via logging.

Parameters

- **exception** – a Python exception.
- **tup** – a `Tuple` object.

read_handshake()
Read and process an initial handshake message from Storm.

read_message()
Read a message from Storm via serializer.

read_tuple()
Read a tuple from the pipe to Storm.

report_metric(*name*, *value*)
Report a custom metric back to Storm.

Parameters

- **name** – Name of the metric. This can be anything.
- **value** – Value of the metric. This is usually a number.

Only supported in Storm 0.9.3+.

run()
Main run loop for all components.

Performs initial handshake with Storm and reads Tuples handing them off to subclasses. Any exceptions are caught and logged back to Storm prior to the Python process exiting.

> **Warning:** Subclasses should **not** override this method.

send_message(*message*)
Send a message to Storm via stdout.

spec(*name=None*, *inputs=None*, *par=None*, *config=None*)
Create a `ShellBoltSpec` for a Python Bolt.

This spec represents this Bolt in a *Topology*.

Parameters

- **name** (*str*) – Name of this Bolt. Defaults to name of *Topology* attribute this is assigned to.
- **inputs** – Streams that feed into this Bolt.

  Two forms of this are acceptable:

  1. A *dict* mapping from `ComponentSpec` to *Grouping*.
  2. A *list* of *Stream* or `ComponentSpec`.
- **par** (*int*) – Parallelism hint for this Bolt. For Python Components, this works out to be the number of Python processes running it in the the topology (across all machines). See *Parallelism and Workers*.

  Note: This can also be specified as an attribute of your `Bolt` subclass.
- **config** (*dict*) – Component-specific config settings to pass to Storm.

  Note: This can also be specified as an attribute of your `Bolt` subclass.

Note: This method does not take a `outputs` argument because `outputs` should be an attribute of your `Bolt` subclass.

class `streamparse.`**`TicklessBatchingBolt`**`(*args, **kwargs)`

Bases: `pystorm.bolt.TicklessBatchingBolt`, `streamparse.storm.bolt.BatchingBolt`

pystorm TicklessBatchingBolt with streamparse-specific additions

`ack``(tup)`

Indicate that processing of a Tuple has succeeded.

Parameters **`tup`** (`str` or `pystorm.component.Tuple`) – the Tuple to acknowledge.

`emit``(tup, **kwargs)`

Modified emit that will not return task IDs after emitting.

See `pystorm.component.Bolt` for more information.

Returns `None`.

`fail``(tup)`

Indicate that processing of a Tuple has failed.

Parameters **`tup`** (`str` or `pystorm.component.Tuple`) – the Tuple to fail (its `id` if `str`).

`group_key``(tup)`

Return the group key used to group Tuples within a batch.

By default, returns None, which put all Tuples in a single batch, effectively just time-based batching. Override this to create multiple batches based on a key.

Parameters **`tup`** (`pystorm.component.Tuple`) – the Tuple used to extract a group key

Returns Any `hashable` value.

`initialize``(storm_conf, context)`

Called immediately after the initial handshake with Storm and before the main run loop. A good place to initialize connections to data sources.

Parameters

- **storm_conf** (*dict*) – the Storm configuration for this component. This is the configuration provided to the topology, merged in with cluster configuration on the worker node.
- **context** (*dict*) – information about the component's place within the topology such as: task IDs, inputs, outputs etc.

is_heartbeat(*tup*)

Returns Whether or not the given Tuple is a heartbeat

is_tick(*tup*)

Returns Whether or not the given Tuple is a tick Tuple

log(*message*, *level=None*)
Log a message to Storm optionally providing a logging level.

Parameters

- **message** (*str*) – the log message to send to Storm.
- **level** (*str*) – the logging level that Storm should use when writing the `message`. Can be one of: trace, debug, info, warn, or error (default: `info`).

Warning: This will send your message to Storm regardless of what level you specify. In almost all cases, you are better of using `Component.logger` and not setting `pystorm.log.path`, because that will use a `pystorm.component.StormHandler` to do the filtering on the Python side (instead of on the Java side after taking the time to serialize your message and send it to Storm).

process(*tup*)
Group non-tick Tuples into batches by `group_key`.

Warning: This method should **not** be overriden. If you want to tweak how Tuples are grouped into batches, override `group_key`.

process_batch(*key*, *tups*)
Process a batch of Tuples. Should be overridden by subclasses.

Parameters

- **key** (*hashable*) – the group key for the list of batches.
- **tups** (*list*) – a *list* of `pystorm.component.Tuple` s for the group.

process_batches()
Iterate through all batches, call process_batch on them, and ack.

Separated out for the rare instances when we want to subclass BatchingBolt and customize what mechanism causes batches to be processed.

process_tick(*tick_tup*)
Just ack tick tuples and ignore them.

raise_exception(*exception*, *tup=None*)
Report an exception back to Storm via logging.

Parameters

- **exception** – a Python exception.
- **tup** – a `Tuple` object.

read_handshake()
Read and process an initial handshake message from Storm.

read_message()
Read a message from Storm via serializer.

read_tuple()
Read a tuple from the pipe to Storm.

report_metric(*name*, *value*)
Report a custom metric back to Storm.

Parameters

- **name** – Name of the metric. This can be anything.
- **value** – Value of the metric. This is usually a number.

Only supported in Storm 0.9.3+.

run()
Main run loop for all components.

Performs initial handshake with Storm and reads Tuples handing them off to subclasses. Any exceptions are caught and logged back to Storm prior to the Python process exiting.

> **Warning:** Subclasses should **not** override this method.

send_message(*message*)
Send a message to Storm via stdout.

spec(*name=None*, *inputs=None*, *par=None*, *config=None*)
Create a `ShellBoltSpec` for a Python Bolt.

This spec represents this Bolt in a *`Topology`*.

Parameters

- **name** (*str*) – Name of this Bolt. Defaults to name of *`Topology`* attribute this is assigned to.
- **inputs** – Streams that feed into this Bolt.

  Two forms of this are acceptable:

  1. A *dict* mapping from `ComponentSpec` to *`Grouping`*.
  2. A *list* of *`Stream`* or `ComponentSpec`.
- **par** (*int*) – Parallelism hint for this Bolt. For Python Components, this works out to be the number of Python processes running it in the the topology (across all machines). See *Parallelism and Workers*.

  > **Note:** This can also be specified as an attribute of your `Bolt` subclass.
- **config** (*dict*) – Component-specific config settings to pass to Storm.

Note: This can also be specified as an attribute of your `Bolt` subclass.

Note: This method does not take a `outputs` argument because `outputs` should be an attribute of your `Bolt` subclass.

Logging

class `streamparse.`**`StormHandler`**(*serializer*)
Bases: `logging.Handler`

Handler that will send messages back to Storm.

Initialize handler

Parameters **`serializer`** – The serializer of the component this handler is being used for.

`emit`(*record*)
Emit a record.

If a formatter is specified, it is used to format the record. If exception information is present, it is formatted using traceback.print_exception and sent to Storm.

Topology DSL

class `streamparse.`**`Topology`**
Class to define a Storm topology in a Python DSL.

class `streamparse.`**`Grouping`**
A Grouping describes how Tuples should be distributed to the tasks of a Bolt listening on a particular stream.

When no Grouping is specified, it defaults to *SHUFFLE* for normal streams, and *DIRECT* for direct streams.

Variables

- **`SHUFFLE`** – Tuples are randomly distributed across the Bolt's tasks in a way such that each Bolt is guaranteed to get an equal number of Tuples.
- **`GLOBAL`** – The entire stream goes to a single one of the Bolt's tasks. Specifically, it goes to the task with the lowest id.
- **`DIRECT`** – This is a special kind of grouping. A stream grouped this way means that the producer of the Tuple decides which task of the consumer will receive this Tuple. Direct groupings can only be declared on streams that have been declared as direct streams. Tuples emitted to a direct stream must be emitted using the the *direct_task* parameter to the *`streamparse.Bolt.emit()`* and *`streamparse.Spout.emit()`* methods.
- **`ALL`** – The stream is replicated across all the Bolt's tasks. Use this grouping with care.
- **`NONE`** – This grouping specifies that you don't care how the stream is grouped. Currently, none groupings are equivalent to shuffle groupings. Eventually though, Storm will push down Bolts with none groupings to execute in the same thread as the Bolt or Spout they subscribe from (when possible).

- **LOCAL_OR_SHUFFLE** – If the target Bolt has one or more tasks in the same worker process, Tuples will be shuffled to just those in-process tasks. Otherwise, this acts like a normal shuffle grouping.

classmethod custom_object(*java_class_name*, *arg_list*)
Tuples will be assigned to tasks by the given Java class.

classmethod custom_serialized(*java_serialized*)
Tuples will be assigned to tasks by the given Java serialized class.

classmethod fields(**fields*)
The stream is partitioned by the fields specified in the grouping.

For example, if the stream is grouped by the *user-id* field, Tuples with the same *user-id* will always go to the same task, but Tuples with different *user-id*'s may go to different tasks.

class streamparse.**Stream**(*fields=None*, *name='default'*, *direct=False*)
A Storm output stream

Parameters

- **fields** (*list* or *tuple* of *str*) – Field names for this stream.
- **name** (*str*) – Name of stream. Defaults to `default`.
- **direct** (*bool*) – Whether or not this stream is direct. Default is *False*. See `DIRECT`.

class streamparse.**JavaBolt**(*input_stream=<open file '<stdin>', mode 'r'>*, *output_stream=<open file '<stdout>', mode 'w'>*, *rdb_signal=u'SIGUSR1'*, *serializer=u'json'*)

classmethod spec(*name=None*, *serialized_java=None*, *full_class_name=None*, *args_list=None*, *inputs=None*, *par=1*, *config=None*, *outputs=None*)
Create a `JavaBoltSpec` for a Java Bolt.

This spec represents this Bolt in a *Topology*.

You must add the appropriate entries to your classpath by editing your project's `project.clj` file in order for this to work.

Parameters

- **name** (*str*) – Name of this Bolt. Defaults to name of *Topology* attribute this is assigned to.
- **serialized_java** (*bytes*) – Serialized Java code representing the class. You must either specify this, or both `full_class_name` and `args_list`.
- **full_class_name** (*str*) – Fully qualified class name (including the package name)
- **args_list** (*list* of basic data types) – A list of arguments to be passed to the constructor of this class.
- **inputs** – Streams that feed into this Bolt.

  Two forms of this are acceptable:

  1. A *dict* mapping from `ComponentSpec` to *Grouping*.
  2. A *list* of *Stream* or `ComponentSpec`.

- **par** (*int*) – Parallelism hint for this Bolt. For Python Components, this works out to be the number of Python processes running it in the the topology (across all machines). See *Parallelism and Workers*.
- **config** (*dict*) – Component-specific config settings to pass to Storm.

- **outputs** – Outputs this JavaBolt will produce. Acceptable forms are:
  1. A *list* of *Stream* objects describing the fields output on each stream.
  2. A *list* of *str* representing the fields output on the `default` stream.

class `streamparse.`**JavaSpout**(*input_stream=<open file '<stdin>', mode 'r'>*, *output_stream=<open file '<stdout>', mode 'w'>*, *rdb_signal=u'SIGUSR1'*, *serializer=u'json'*)

classmethod spec(*name=None*, *serialized_java=None*, *full_class_name=None*, *args_list=None*, *par=1*, *config=None*, *outputs=None*)

Create a `JavaSpoutSpec` for a Java Spout.

This spec represents this Spout in a *Topology*.

You must add the appropriate entries to your classpath by editing your project's `project.clj` file in order for this to work.

Parameters

- **name** (*str*) – Name of this Spout. Defaults to name of *Topology* attribute this is assigned to.
- **serialized_java** (*bytes*) – Serialized Java code representing the class. You must either specify this, or both `full_class_name` and `args_list`.
- **full_class_name** (*str*) – Fully qualified class name (including the package name)
- **args_list** (*list* of basic data types) – A list of arguments to be passed to the constructor of this class.
- **par** (*int*) – Parallelism hint for this Spout. See *Parallelism and Workers*.
- **config** (*dict*) – Component-specific config settings to pass to Storm.
- **outputs** – Outputs this JavaSpout will produce. Acceptable forms are:
  1. A *list* of *Stream* objects describing the fields output on each stream.
  2. A *list* of *str* representing the fields output on the `default` stream.

class `streamparse.`**ShellBolt**(*input_stream=<open file '<stdin>', mode 'r'>*, *output_stream=<open file '<stdout>', mode 'w'>*, *rdb_signal=u'SIGUSR1'*, *serializer=u'json'*)

A Bolt that is started by running a command with a script argument.

classmethod spec(*name=None*, *command=None*, *script=None*, *inputs=None*, *par=None*, *config=None*, *outputs=None*)

Create a `ShellBoltSpec` for a non-Java, non-Python Bolt.

If you want to create a spec for a Python Bolt, use `spec()`.

This spec represents this Bolt in a *Topology*.

Parameters

- **name** (*str*) – Name of this Bolt. Defaults to name of *Topology* attribute this is assigned to.
- **command** (*str*) – Path to command the Storm will execute.
- **script** (*str*) – Arguments to *command*. Multiple arguments should just be separated by spaces.

- **inputs** – Streams that feed into this Bolt.

  Two forms of this are acceptable:

  1. A *dict* mapping from `ComponentSpec` to *Grouping*.
  2. A *list* of *Stream* or `ComponentSpec`.

- **par** (*int*) – Parallelism hint for this Bolt. For shell Components, this works out to be the number of running it in the the topology (across all machines). See *Parallelism and Workers*.
- **config** (*dict*) – Component-specific config settings to pass to Storm.
- **outputs** – Outputs this ShellBolt will produce. Acceptable forms are:

  1. A *list* of *Stream* objects describing the fields output on each stream.
  2. A *list* of *str* representing the fields output on the `default` stream.

class `streamparse.`**ShellSpout**(*input_stream=<open file '<stdin>', mode 'r'>*, *output_stream=<open file '<stdout>', mode 'w'>*, *rdb_signal=u'SIGUSR1'*, *serializer=u'json'*)

classmethod **spec**(*name=None*, *command=None*, *script=None*, *par=None*, *config=None*, *outputs=None*)

Create a `ShellSpoutSpec` for a non-Java, non-Python Spout.

If you want to create a spec for a Python Spout, use `spec()`.

This spec represents this Spout in a *Topology*.

Parameters

- **name** (*str*) – Name of this Spout. Defaults to name of *Topology* attribute this is assigned to.
- **command** (*str*) – Path to command the Storm will execute.
- **script** (*str*) – Arguments to *command*. Multiple arguments should just be separated by spaces.
- **par** (*int*) – Parallelism hint for this Spout. For shell Components, this works out to be the number of processes running it in the the topology (across all machines). See *Parallelism and Workers*.
- **config** (*dict*) – Component-specific config settings to pass to Storm.
- **outputs** – Outputs this ShellSpout will produce. Acceptable forms are:

  1. A *list* of *Stream* objects describing the fields output on each stream.
  2. A *list* of *str* representing the fields output on the `default` stream.

CHAPTER 4

Developing Streamparse

Lein

Install Leiningen according to the instructions in the quickstart.

Local pip installation

In your virtualenv for this project, go into `~/repos/streamparse` (where you cloned streamparse) and simply run:

```
python setup.py develop
```

This will install a streamparse Python version into the virtualenv which is essentially symlinked to your local version.

NOTE: streamparse currently pip installs streamparse's **released** version on remote clusters automatically. Therefore, though this will work for local development, you'll need to push streamparse somewhere pip installable (or change requirements.txt) to make it pick up that version on a remote cluster.

Installing Storm pre-releases

You can clone Storm from Github here:

```
git clone git@github.com:apache/storm.git
```

There are tags available for releases, e.g.:

```
git checkout v1.0.1
```

To build a local Storm release, use:

```
mvn install
cd storm-dist/binary
mvn package
```

These steps will take awhile as they also run Storm's internal unit and integration tests.

The first line will actually install Storm locally in your maven (.m2) repository. You can confirm this with:

```
ls ~/.m2/repository/org/apache/storm/storm-core/1.0.1
```

You should now be able to change your `project.clj` to include a reference to this new release.

Once you change that, you can run:

```
lein deps :tree | grep storm
```

To confirm it is using the upgraded Clojure 1.5.1 (changed in 0.9.2), run:

```
lein repl
```

CHAPTER 5

Frequently Asked Questions (FAQ)

General Questions

Why use streamparse?

To lay your Python code out in topologies which can be automatically parallelized in a Storm cluster of machines. This lets you scale your computation horizontally and avoid issues related to Python's GIL. See *Parallelism and Workers*.

Is streamparse compatible with Python 3?

Yes, streamparse is fully compatible with Python 3 starting with version 3.3 which we use in our unit tests.

How can I contribute to streamparse?

Thanks for your interest in contributing to streamparse. We think you'll find the core maintainers great to work with and will help you along the way when contributing pull requests.

If you already know what you'd like to add to streamparse then by all means, feel free to submit a pull request and we'll review it.

If you're unsure about how to contribute, check out our open issues and find one that looks interesting to you (we particularly need help on all issues marked with the "help wanted" label).

If you're not sure how to start or have some questions, shoot us an email in the streamparse user group and we'll give you a hand.

From there, get to work on your fix and submit a pull request when ready which we'll review.

How do I trigger some code before or after I submit my topology?

After you create a streamparse project using `sparse quickstart`, you'll have a `fabfile.py` in that directory. In that file, you can specify two functions (`pre_submit` and `post_submit`) which are expected to accept three arguments:

- `topology_name`: the name of the topology being submitted
- `env_name`: the name of the environment where the topology is being submitted (e.g. `"prod"`)
- `env_config`: the relevant config portion from the `config.json` file for the environment you are submitting the topology to

Here is a sample `fabfile.py` file that sends a message to IRC after a topology is successfully submitted to prod.

```
# my_project/fabfile.py
from __future__ import absolute_import, print_function, unicode_literals

from my_project import write_to_irc


def post_submit(topo_name, env_name, env_config):
    if env_name == "prod":
        write_to_irc("Deployed {} to {}".format(topo_name, env_name))
```

How should I install streamparse on the cluster nodes?

streamparse assumes your Storm servers have Python, pip, and virtualenv installed. After that, the installation of all required dependencies (including streamparse itself) is taken care of via the *config.json* file for the streamparse project and the `sparse submit` command.

Should I install Clojure?

No, the Java requirements for streamparse are identical to that of Storm itself. Storm requires Java and bundles Clojure as a requirement, so you do not need to do any separate installation of Clojure. You just need Java on all Storm servers.

How do I deploy into a VPC?

Update your `~/.ssh/config` to use a bastion host inside your VPC for your commands:

```
Host *.internal.example.com
    ProxyCommand ssh bastion.example.com exec nc %h %p
```

If you don't have a common subdomain you'll have to list all of the hosts individually:

```
Host host1.example.com
    ProxyCommand ssh bastion.example.com exec nc %h %p
...
```

Set up your streamparse config to use all of the hosts normally (without bastion host).

How do I override SSH settings?

It is highly recommended that you just modify your `~/.ssh/config` file if you need to tweak settings for setting up the SSH tunnel to your Nimbus server, but you can also set your SSH password or port in your `config.json` by setting the `ssh_password` or `ssh_port` environment settings.

```
{
    "topology_specs": "topologies/",
    "virtualenv_specs": "virtualenvs/",
    "envs": {
        "prod": {
            "user": "somebody",
            "ssh_password": "THIS IS A REALLY BAD IDEA",
            "ssh_port": 52,
            "nimbus": "streamparse-box",
            "workers": [
                "streamparse-box"
            ],
            "virtualenv_root": "/data/virtualenvs"
        }
    }
}
```

How do I dynamically generate the worker list?

In a small cluster it's sufficient to specify the list of workers in `config.json`. However, if you have a large or complex environment where workers are numerous or short-lived, `streamparse` supports querying the nimbus server for a list of hosts.

An undefined list (empty or None) of `workers` will trigger the lookup. Explicitly defined hosts are preferred over a lookup.

Lookups are configured on a per-environment basis, so the `prod` environment below uses the dynamic lookup, while `beta` will not.

```
{
    "topology_specs": "topologies/",
    "virtualenv_specs": "virtualenvs/",
    "envs": {
        "prod": {
            "nimbus": "streamparse-prod",
            "virtualenv_root": "/data/virtualenvs"
        },
        "beta": {
            "nimbus": "streamparse-beta",
            "workers": [
                "streamparse-beta"
            ],
            "virtualenv_root": "/data/virtualenvs"
```

```
            }
        }
}
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

P

R

S

T