streamparse Documentation

Release 2.2.0.dev

Parsely

November 22, 2015

Contents

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

CHAPTER 1

Quickstart

1.1 Dependencies

1.1.1 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 project's page or github

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
```

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

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

```
> pip install streamparse
```

1.2 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/tasks.py
    create    wordcount/topologies
    create    wordcount/topologies/wordcount.clj
    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
```

1.3 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 Clojure DSL for Storm.
virtualenvs/	Contains pip requirements files in order to install dependencies on remote Storm servers.

1.4 Defining Topologies

Storm's services are Thrift-based and although it is possible to define a topology in pure Python using Thrift, it introduces a host of additional dependencies which are less than trivial to setup for local development. In addition, it turns out that using Clojure to define topologies, still feels fairly Pythonic, so the authors of streamparse decided this was a good compromise.

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

```
(ns wordcount
  (:use     [streamparse.specs])
  (:gen-class))

(defn wordcount [options]
   [
    ;; spout configuration
    {"word-spout" (python-spout-spec
          options
```

```
            "spouts.words.WordSpout"
            ["word"]
            )
    }
    ;; bolt configuration
    {"count-bolt" (python-bolt-spec
            options
            {"word-spout" :shuffle}
            "bolts.wordcount.WordCounter"
            ["word" "count"]
            :p 2
            )
    }
  ]
)
```

The first block of code we encounter effectively states "import the Clojure DSL functions for Storm." By convention, use the same name for the namespace (`ns`) and function (`defn`) as the basename of the file ("wordcount"), though these are not strictly required.

```
(ns wordcount
  (:use     [streamparse.specs])
  (:gen-class))
```

The next block of code actually defines the topology and stores it into a function named "wordcount".

```
(defn wordcount [options]
   [
    ;; spout configuration
    {"word-spout" (python-spout-spec
            options
            "spouts.words.WordSpout"
            ["word"]
            )
    }
    ;; bolt configuration
    {"count-bolt" (python-bolt-spec
            options
            {"word-spout" :shuffle}
            "bolts.wordcount.WordCounter"
            ["word" "count"]
            :p 2
            )
    }
  ]
)
```

It turns out, the name of the function doesn't matter much; we've used `wordcount` above, but it could just as easily be `bananas`. What is important, is that **the function must return an array with only two dictionaries and take one argument**, and that the last function in the file is the DSL spec (i.e. do not add a `defn` below this function).

The first dictionary holds a named mapping of all the spouts that exist in the topology, the second holds a named mapping of all the bolts. The `options` argument contains a mapping of topology settings.

An additional benefit of defining topologies in Clojure is that we're able to mix and match the types of spouts and bolts. In most cases, you may want to use a pure Python topology, but you could easily use JVM-based spouts and bolts or even spouts and bolts written in other languages like Ruby, Go, etc.

Since you'll most often define spouts and bolts in Python however, we'll look at two important functions provided by streamparse: `python-spout-spec` and `python-bolt-spec`.

When creating a Python-based spout, we provide a name for the spout and a definition of that spout via `python-spout-spec`:

```
{"sentence-spout-1" (python-spout-spec
                     ;; topology options passed in
                     options
                     ;; name of the python class to ``run``
                     "spouts.SentenceSpout"
                     ;; output specification, what named fields will this spout emit?
                     ["sentence"]
                     ;; configuration parameters, can specify multiple
                     :p 2)
 "sentence-spout-2" (shell-spout-spec
                     options
                     "spouts.OtherSentenceSpout"
                     ["sentence"])}
```

In the example above, we've defined two spouts in our topology: `sentence-spout-1` and `sentence-spout-2` and told Storm to run these components. `python-spout-spec` will use the `options` mapping to get the path to the python executable that Storm will use and streamparse will run the class provided. We've also let Storm know exactly what these spouts will be emitting, namely a single field called `sentence`.

You'll notice that in `sentence-spout-1`, we've passed an optional map of configuration parameters `:p 2`, which sets the spout to have 2 Python processes. This is discussed in *Parallelism and Workers*.

Creating bolts is very similar and uses the `python-bolt-spec` function:

```
{"sentence-splitter" (python-bolt-spec
                      ;; topology options passed in
                      options
                      ;; inputs, where does this bolt recieve it's tuples from?
                      {"sentence-spout-1" :shuffle
                       "sentence-spout-2" :shuffle}
                      ;; class to run
                      "bolts.SentenceSplitter"
                      ;; output spec, what tuples does this bolt emit?
                      ["word"]
                      ;; configuration parameters
                      :p 2)
 "word-counter" (python-bolt-spec
                 options
                 ;; recieves tuples from "sentence-splitter", grouped by word
                 {"sentence-splitter" ["word"]}
                 "bolts.WordCounter"
                 ["word" "count"])
 "word-count-saver" (python-bolt-spec
                     ;; topology options passed in
                     options
                     {"word-counter" :shuffle}
                     "bolts.WordSaver"
                     ;; does not emit any fields
                     [])}
```

In the example above, we define 3 bolts by name `sentence-splitter`, `word-counter` and `word-count-saver`. Since bolts are generally supposed to process some input and optionally produce some output, we have to tell Storm where a bolts inputs come from and whether or not we'd like Storm to use any stream grouping on the tuples from the input source.

In the `sentence-splitter` bolt, you'll notice that we define two input sources for the bolt. It's completely fine to add multiple sources to any bolts.

In the `word-counter` 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.
- **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 Storm's Concepts for more information.

1.5 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):

    def initialize(self, stormconf, context):
        self.sentences = [
            "She advised him to take a long holiday, so he immediately quit work and took a trip arou
            "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):

    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

        self.emit_many(words)
        # 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.

In the `SentenceSplitterBolt` above, we have decided to use the `emit_many()` method instead of `emit()` which is a bit more efficient when sending a larger number of tuples to Storm.

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.

1.5.1 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.

1.5.2 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
```

1.5.3 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" ...
```

1.6 Remote Deployment

1.6.1 Setting up a Storm Cluster

See Storm's Setting up a Storm Cluster.

1.6.2 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:

```
{
    "library": "",
    "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",
                "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

1.6.3 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 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.

1.6.4 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"}
```

1.6.5 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`.

1.6.6 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 automatically set up a log files on each Storm worker in this path using the following filename convention:

```
streamparse_<topology_name>_<component_name>_<task_id>_<process_id>.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
- `process_id`: 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`.

CHAPTER 2

Topologies

2.1 Clojure Quick Reference Guide

Topologies in streamparse are defined using Clojure. Here is a quick overview so you don't get lost.

Function definitions `(defn fn-name [options] expressions)` defines a function called `fn-name` that takes `options` as an argument and evaluates each of the `expressions`, treating the last evaluated expression as the return value for a function.

Keyword arguments In Clojure, keyword arguments are specified using paired-up positional arguments. Thus `:p 2` is the `p` keyword set to value `2`.

List `[val1 val2 ... valN]` defines a list of N values.

Map `{"key-1" val1 "key-2" val2 ... "key-N" valN}` is a mapping of key-value pairs.

Comments Anything after `;;` is a line comment.

For Python programmers, Clojure can be a little tricky in that whitespace is not significant, and `,` is treated as whitespace. This means `[val1 val2]` and `[val1, val2]` are identical lists. Function definitions can similarly take up multiple lines.

```
(defn fn-name [options]
    expression1
    expression2
    ;; ...
    expressionN
    ;; the value of expressionN is the returned value
)
```

2.2 Topology Files

A topology file describes your topology in terms of Directed Acyclic Graph (DAG) of Storm components, namely *bolts* and *spouts*. It uses the Clojure DSL for this, along with some utility functions streamparse provides.

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.clj
- topologies/my-other-topology.clj
- topologies/my-third-topology.clj

So on and so forth.

A sample `my-topology.clj`, would start off importing the streamparse Clojure DSL functions.

```
(ns my-topology
  (:use     [streamparse.specs])
  (:gen-class))
```

Notice the `my-topology` matches the name of the file. The next line is the import of the streamparse utility functions.

You could optionally avoid all of the streamparse-provided helper functions and import your own functions or the Clojure DSL for Storm directly.

```
(ns my-topology
  (:use     [backtype.storm.clojure])
  (:gen-class))
```

In the next part of the file, we setup a topology definition, also named `my-topology` (matching the `ns` line and filename). This definition is actually a Clojure function that takes the topology options as a single map argument. This function returns a list of 2 maps – a spout map, and a bolt map. These two maps define the DAG that is your topology.

```
(defn my-topology [options]
   [
    ;; spout configuration
    {"my-python-spout" (python-spout-spec
          ;; topology options passed in
          options
          ;; python class to run
          "spouts.myspout.MySpout"
          ;; output specification, what named fields will this spout emit?
          ["data"]
          ;; configuration parameters, can specify multiple or none at all
          )
   }


    ;; bolt configuration
    {"my-python-bolt" (python-bolt-spec
          ;; topology options pased in
          options
          ;; inputs, where does this bolt receive its tuples from?
          {"my-python-spout" :shuffle}
          ;; python class to run
          "bolts.mybolt.MyBolt"
          ;; output specification, what named fields will this spout emit?
          ["data" "date"]
          ;; configuration parameters, can specify multiple or none at all
          :p 2
          )
    }
  ]
)
```

2.3 Shell Spouts and Bolts

The Clojure DSL provides the `shell-bolt-spec` and `shell-spout-spec` functions to handle bolts in non-JVM languages.

The `shell-spout-spec` takes at least 2 arguments:

1. The command line program to run (as a list of arguments)
2. A list of the named fields the spout will output
3. Any optional keyword arguments

```
"my-shell-spout" (shell-spout-spec
    ;; Command to run
    ["python" "spout.py"]
    ;; output specification, what named fields will this spout emit?
    ["data"]
    ;; configuration parameters, can specify multiple or none at all
    :p 2
)
```

The `shell-bolt-spec` takes at least 3 arguments:

1. A map of the input spouts and their groupings
2. The command line program to run (as a list of arguments)
3. A list of the named fields the spout will output
4. Any optional keyword arguments

```
"my-shell-bolt" (shell-bolt-spec
    ;; input spouts and their groupings
    {"my-shell-spout" :shuffle}
    ;; Command to run
    ["bash" "mybolt.sh"]
    ;; output specification, what named fields will this spout emit?
    ["data"]
    ;; configuration parameters, can specify multiple or none at all
    :p 2
)
```

2.4 Python Spouts and Bolts

The example topology above, and the `sparse quickstart wordcount` project utilizes the `python-spout-spec` and `python-bolt-spec` provided by the `streamparse.specs` import statement.

`(python-spout-spec ...)` and `(python-bolt-spec ...)` are just convenience functions provided by streamparse for creating topology components. They are simply wrappers around `(shell-spout-spec ...)` and `(shell-bolt-spec ...)`.

The `python-spout-spec` takes at least 3 arguments:

1. `options` - the topology options array passed in
2. The full path to the class to run. `spouts.myspout.MySpout` is actually the `MySpout` class in `src/spouts/myspout.py`
3. A list of the named fields the spout will output
4. Any optional keyword arguments, such as parallelism `:p 2`

The `python-bolt-spec` takes at least 4 arguments:

1. `options` - the topology options array passed in

2. A map of the input spouts and their groupings (See below)
3. The full path to the class to run. `bolts.mybolt.MyBolt` is actually the `MyBolt` class in `src/bolts/mybolt.py`
4. A list of the named fields the spout will output
5. Any optional keyword arguments, such as parallelism `:p 2`

Parallelism is further discussed in *Parallelism and Workers*.

2.4.1 Groupings

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.
- **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.

2.4.2 Streams

Topologies support multiple streams when routing tuples between components. The `emit()` method takes an optional *stream* argument to specify the stream ID. For example:

```
self.emit([term, timestamp, lookup_result], stream='index')
self.emit([term, timestamp, lookup_result], stream='topic')
```

The topology definition can include these stream IDs to route between components, and a component can specify more than one stream. Example with the Clojure DSL:

```
"lookup-bolt" (python-bolt-spec
    options
    {"search-bolt" :shuffle}
    "birding.bolt.TwitterLookupBolt"
    {"index" ["url" "timestamp" "search_result"]
     "topic" ["url" "timestamp" "search_result"]}
    :p 2
    )
"elasticsearch-index-bolt" (python-bolt-spec
    options
    {["lookup-bolt" "index"] ["url" "timestamp" "search_result"]}
    "birding.bolt.ElasticsearchIndexBolt"
    []
    :p 1
    )
"result-topic-bolt" (python-bolt-spec
    options
    {["lookup-bolt" "index"] ["url" "timestamp" "search_result"]
     ["lookup-bolt" "topic"] ["url" "timestamp" "search_result"]}
    "birding.bolt.ResultTopicBolt"
    []
    :p 1
    )
```

Storm sets a default stream ID of `"default"`, as described in its doc on Streams:

> Every stream is given an id when declared. Since single-stream spouts and bolts are so common, ... the stream is given the default id of "default".

2.5 Running Topologies

2.5.1 What Streamparse Does

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

1. Compile your .clj topology file
2. Execute the Clojure code by invoking your topology function, passing it the `options` map
3. Get the DAG defined by the topology and pass it into the Storm Java interop classes like StormSubmitter and LocalCluster
4. Run/submit your topology

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.

2.5.2 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.

2.6 Parallelism and Workers

In general, use the :p "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 `:p` keyword to provide a parallelism hint to Storm for the number of executors (threads) to use for the given spout/bolt; for example, `:p 2` is a hint to use two executors. Because streamparse implements spouts and bolts as independent Python processes, setting `:p 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 sumbit` 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 Storm's underlying 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

3.1 Tuples

class pystorm.component.**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 *pystorm.component.Tuple* yourself as streamparse handles this for you prior to, for example, a *pystorm.bolt.Bolt*'s process() method being called.

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

3.2 Components

Both *pystorm.bolt.Bolt* and *pystorm.spout.Spout* inherit from a common base-class, *pystorm.component.Component*. It handles the basic Multi-Lang IPC between Storm and Python.

class pystorm.component.**Component**(*input_stream=<open file '<stdin>', mode 'r'>*, *output_stream=<open file '<stdout>', mode 'w'>*, *rdb_signal=10*)
Base class for spouts and bolts which contains class methods for logging messages back to the Storm worker process.

Variables

- **input_stream** – The file-like object to use to retrieve commands from Storm. Defaults to sys.stdin.
- **output_stream** – The file-like object to send messages to Storm with. Defaults to sys.stdout.
- **topology_name** – The name of the topology sent by Storm in the initial handshake.

- **task_id** – The numerical task ID for this component, as sent by Storm in the initial handshake.
- **component_name** – The name of this component, as sent by Storm in the initial handshake.
- **debug** – A `bool` indicating whether or not Storm is running in debug mode. Specified by the *topology.debug* Storm setting.
- **storm_conf** – A `dict` containing the configuration values sent by Storm in the initial handshake with this component.
- **context** – The context of where this component is in the topology. See the Storm Multi-Lang protocol documentation for details.
- **pid** – An `int` indicating the process ID of this component as retrieved by `os.getpid()`.
- **logger** – A logger to use with this component.

Note: Using `Component.logger` combined with the `pystorm.component.StormHandler` handler is the recommended way for logging messages from your component. If you use `Component.log` instead, the logging messages will *always* be sent to Storm, even if they are `debug` level messages and you are running in production. Using `pystorm.component.StormHandler` ensures that you will instead have your logging messages filtered on the Python side and only have the messages you actually want logged serialized and sent to Storm.

---

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

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.
- **tup_id** (*str*) – the ID for the Tuple. If omitted by a *pystorm.spout.Spout*, this emit will be unreliable.
- **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. This is only passed by *pystorm.bolt.Bolt*.
- **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: `True`).

Returns a `list` of task IDs that the Tuple was sent to. Note that when specifying direct_task, this will be equal to `[direct_task]`. If you specify `need_task_ids=False`, this function will return `None`.

static **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` with a `pystorm.component.StormHandler`, because the filtering will happen on the Python side (instead of on the Java side after taking the time to serialize your message and send it to Storm).

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, reconstruct newlines appropriately.

All of Storm's messages (for either bolts or spouts) should be of the form:

```
'<command or task_id form prior emit>\nend\n'
```

Command example, an incoming Tuple to a bolt:

```
'{ "id": "-6955786537413359385",  "comp": "1", "stream": "1", "task": 9, "tuple": ["snow whi
```

Command example for a spout to emit its next Tuple:

```
'{"command": "next"}\nend\n'
```

Example, the task IDs a prior emit was sent to:

```
'[12, 22, 24]\nend\n'
```

The edge case of where we read `''` from `input_stream` indicating EOF, usually means that communication with the supervisor has been severed.

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.

3.2.1 Spouts

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

class pystorm.spout.**Spout**(*input_stream=<open file '<stdin>', mode 'r'>*, *output_stream=<open file '<stdout>', mode 'w'>*, *rdb_signal=10*)

Bases: *pystorm.component.Component*

Base class for all pystorm spouts.

For more information on spouts, consult Storm's Concepts documentation.

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.

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

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: `True`).

Returns a `list` of task IDs that the Tuple was sent to. Note that when specifying direct_task, this will be equal to `[direct_task]`. If you specify `need_task_ids=False`, this function will return `None`.

emit_many(*tuples*, *stream=None*, *tup_ids=None*, *direct_task=None*, *need_task_ids=True*)

Emit multiple tuples.

Parameters

- **tuples** (*list*) – a `list` of multiple Tuple payloads to send to Storm. All Tuples should contain only JSON-serializable data.
- **stream** (*str*) – the ID of the stream to emit these Tuples to. Specify `None` to emit to default stream.
- **tup_ids** (*list*) – the ID for the Tuple. Leave this blank for an unreliable emit.
- **tup_ids** – IDs for each of the Tuples in the list. Omit these for an unreliable emit.
- **direct_task** (*int*) – indicates 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: `True`).

Deprecated since version 2.0.0: Just call `Spout.emit()` repeatedly instead.

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 spout. 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` with a `pystorm.component.StormHandler`, because the filtering will happen 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, reconstruct newlines appropriately.

All of Storm's messages (for either bolts or spouts) should be of the form:

```
'<command or task_id form prior emit>\nend\n'
```

Command example, an incoming Tuple to a bolt:

```
'{ "id": "-6955786537413359385",  "comp": "1", "stream": "1", "task": 9, "tuple": ["snow whi
```

Command example for a spout to emit its next Tuple:

```
'{"command": "next"}\nend\n'
```

Example, the task IDs a prior emit was sent to:

```
'[12, 22, 24]\nend\n'
```

The edge case of where we read `''` from `input_stream` indicating EOF, usually means that communication with the supervisor has been severed.

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.

3.2.2 Bolts

class pystorm.bolt.**Bolt**(*input_stream=<open file '<stdin>', mode 'r'>, output_stream=<open file '<stdout>', mode 'w'>, rdb_signal=10*)
Bases: *pystorm.component.Component*

The base class for all pystorm bolts.

For more information on bolts, consult Storm's Concepts documentation.

Variables

- **auto_anchor** – A `bool` indicating whether or not the bolt should automatically anchor emits to the incoming Tuple ID. Tuple anchoring is how Storm provides reliability, you can read more about Tuple anchoring in Storm's docs. Default is `True`.
- **auto_ack** – A `bool` indicating whether or not the bolt should automatically acknowledge Tuples after `process()` is called. Default is `True`.
- **auto_fail** – A `bool` indicating whether or not the bolt should automatically fail Tuples when an exception occurs when the `process()` method is called. Default is `True`.

Example:

```
from pystorm.bolt import Bolt

class SentenceSplitterBolt(Bolt):

    def process(self, tup):
        sentence = tup.values[0]
        for word in sentence.split(" "):
            self.emit([word])
```

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=True*)
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: `True`).

Returns a `list` of task IDs that the Tuple was sent to. Note that when specifying direct_task, this will be equal to `[direct_task]`. If you specify `need_task_ids=False`, this function will return `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`).

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 bolt. 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

static **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` with a `pystorm.component.StormHandler`, because the filtering will happen 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, reconstruct newlines appropriately.

All of Storm's messages (for either bolts or spouts) should be of the form:

```
'<command or task_id form prior emit>\nend\n'
```

Command example, an incoming Tuple to a bolt:

```
'{ "id": "-6955786537413359385",  "comp": "1", "stream": "1", "task": 9, "tuple": ["snow whi
```

Command example for a spout to emit its next Tuple:

```
'{"command": "next"}\nend\n'
```

Example, the task IDs a prior emit was sent to:

```
'[12, 22, 24]\nend\n'
```

The edge case of where we read `''` from `input_stream` indicating EOF, usually means that communication with the supervisor has been severed.

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.

class pystorm.bolt.**BatchingBolt**(**args*, ***kwargs*)

Bases: *pystorm.bolt.Bolt*

A bolt which batches Tuples for processing.

Batching Tuples is unexpectedly complex to do correctly. The main problem is that all bolts are single-threaded. The difficult comes when the topology is shutting down because Storm stops feeding the bolt Tuples. If the bolt is blocked waiting on stdin, then it can't process any waiting Tuples, or even ack ones that were asynchronously written to a data store.

This bolt helps with that by grouping Tuples received between tick Tuples into batches.

To use this class, you must implement `process_batch`. `group_key` can be optionally implemented so that Tuples are grouped before `process_batch` is even called.

Variables

- **auto_anchor** – A `bool` indicating whether or not the bolt should automatically anchor emits to the incoming Tuple ID. Tuple anchoring is how Storm provides reliability, you can read more about Tuple anchoring in Storm's docs. Default is `True`.
- **auto_ack** – A `bool` indicating whether or not the bolt should automatically acknowledge Tuples after `process_batch()` is called. Default is `True`.
- **auto_fail** – A `bool` indicating whether or not the bolt should automatically fail Tuples when an exception occurs when the `process_batch()` method is called. Default is `True`.
- **ticks_between_batches** – The number of tick Tuples to wait before processing a batch.

Example:

```
from pystorm.bolt import BatchingBolt

class WordCounterBolt(BatchingBolt):

    ticks_between_batches = 5

    def group_key(self, tup):
        word = tup.values[0]
        return word  # collect batches of words

    def process_batch(self, key, tups):
        # emit the count of words we had per 5s batch
        self.emit([key, len(tups)])
```

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 bolt. 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` with a `pystorm.component.StormHandler`, because the filtering will happen 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, reconstruct newlines appropriately.

All of Storm's messages (for either bolts or spouts) should be of the form:

```
'<command or task_id form prior emit>\nend\n'
```

Command example, an incoming Tuple to a bolt:

```
'{ "id": "-6955786537413359385",  "comp": "1", "stream": "1", "task": 9, "tuple": ["snow whi
```

Command example for a spout to emit its next Tuple:

```
'{"command": "next"}\nend\n'
```

Example, the task IDs a prior emit was sent to:

```
'[12, 22, 24]\nend\n'
```

The edge case of where we read `''` from `input_stream` indicating EOF, usually means that communication with the supervisor has been severed.

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.

CHAPTER 4

Developing Streamparse

4.1 Lein

Install Leiningen according to the instructions in the quickstart.

4.2 Using Local Clojure Interop Library

You can tell `lein` to point directly at streamparse's Clojure repo and use the code there for all of the interop commands, so that you can test changes while developing.

To do this, add a directory called `checkouts` and symlink it up:

```
mkdir checkouts
cd checkouts
ln -s ../../../streamparse/jvm streamparse
cd ..
```

Now, comment out the `com.parsely/streamparse` dependency in `project.clj`. It will now pick up the Clojure commands from your local repo. So, now you can tweak and change them!

4.3 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.

4.4 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 v0.9.2-incubating
```

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/0.9.2-incubating
```

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)

5.1 General Questions

5.1.1 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*.

5.1.2 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.

5.1.3 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.

5.1.4 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 both a `tasks.py` in that directory as well as `fabric.py`. In either of these files, 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 `tasks.py` file that sends a message to IRC after a topology is successfully submitted to prod.

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

from invoke import task, run
from streamparse.ext.invoke import *


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

5.1.5 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. See Remote Deployment for more information.

5.1.6 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.

5.2 Errors While Running streamparse

- *I received an "InvalidClassException" while submitting my topology, what do I do?*

5.2.1 I received an "InvalidClassException" while submitting my topology, what do I do?

If the Storm version dependency you specify in your `project.clj` file is different from the version of Storm running on your cluster, you'll get an error in Storm's logs that looks something like the following when you submit your topology:

```
2014-07-07 02:30:27 b.s.d.executor [INFO] Finished loading executor __acker:[2 2]
2014-07-07 02:30:27 b.s.d.executor [INFO] Preparing bolt __acker:(2)
2014-07-07 02:30:27 b.s.d.executor [INFO] Prepared bolt __acker:(2)
2014-07-07 02:30:27 b.s.d.executor [INFO] Loading executor count-bolt:[4 4]
2014-07-07 02:30:27 b.s.d.worker [ERROR] Error on initialization of server mk-worker
```

```
java.lang.RuntimeException: java.io.InvalidClassException: backtype.storm.task.ShellBolt; local class
    at backtype.storm.utils.Utils.deserialize(Utils.java:93) ~[storm-core-0.9.2-incubating.jar:0.9.2-
    at backtype.storm.utils.Utils.getSetComponentObject(Utils.java:235) ~[storm-core-0.9.2-incubating
    at backtype.storm.daemon.task$get_task_object.invoke(task.clj:73) ~[storm-core-0.9.2-incubating.j
    at backtype.storm.daemon.task$mk_task_data$fn__3061.invoke(task.clj:180) ~[storm-core-0.9.2-incub
    at backtype.storm.util$assoc_apply_self.invoke(util.clj:816) ~[storm-core-0.9.2-incubating.jar:0.
    at backtype.storm.daemon.task$mk_task_data.invoke(task.clj:173) ~[storm-core-0.9.2-incubating.jar
    at backtype.storm.daemon.task$mk_task.invoke(task.clj:184) ~[storm-core-0.9.2-incubating.jar:0.9.
    at backtype.storm.daemon.executor$mk_executor$fn__5510.invoke(executor.clj:321) ~[storm-core-0.9.
    at clojure.core$map$fn__4207.invoke(core.clj:2485) ~[clojure-1.5.1.jar:na]
```

Check to ensure the version of Storm in your `project.clj` file matches the Storm version running on your cluster, then try resubmitting your topology.

```
(defproject my-project "0.0.1-SNAPSHOT"
  :source-paths ["topologies"]
  :resource-paths ["_resources"]
  :target-path "_build"
  :min-lein-version "2.0.0"
  :jvm-opts ["-client"]
  :dependencies  [[org.apache.storm/storm-core "0.9.4"] ;; this should match your Storm cluster
                  [com.parsely/streamparse "0.0.4-SNAPSHOT"]]
  :jar-exclusions     [#"log4j\.properties" #"backtype" #"trident" #"META-INF" #"meta-inf" #"\.yaml"]
  :uberjar-exclusions [#"log4j\.properties" #"backtype" #"trident" #"META-INF" #"meta-inf" #"\.yaml"]
)
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

S

T