Hadoopy Documentation

Release .0.6.0

Brandyn White

July 19, 2016

Contents

CHAPTER 1

Tutorial

1.1 Installing Hadoopy

The best way to get Hadoopy is off of the github.

Github Clone

```
git clone https://github.com/bwhite/hadoopy.git
cd hadoopy
sudo python setup.py install
```

PIP installation

```
sudo pip install -e git+https://github.com/bwhite/hadoopy#egg=hadoopy
```

This guide assumes you have a functional Hadoop cluster with a supported version of Hadoop. If you need help with this see the cluster guide

1.2 Putting Data on HDFS

As Hadoop is used to process large amounts of data, being able to easily put data on HDFS (Hadoop Distributed File System) is essential. The primary way to do this using Hadoopy is with the hadoopy.writetb command which takes an iterator of key/value pairs and puts them in a SequenceFile encoded as TypedBytes. This file can be used directly as input to Hadoopy jobs and they maintain their Python types due to the TypedBytes encoding.

1.3 Anatomy of Hadoopy Jobs

Each job consists of a Map task and an optional Reduce task. If a Reduce task is present then there is also an optional Combine task. A job is often specified in a single Python (.py) file with a call to hadoopy.run. The hadoopy.run command adds several command line options to the script and takes as arguments the map/reduce/combine classes/functions. The function form is appropriate for small tasks that don't hold state (i.e., information is maintained between calls); however, when a task is large enough to spread across several functions, require initial setup, or hold state between jobs, the class form should be used. All that is necessary to use a map/reduce/combine task is to provide it to the hadoopy.run function. All tasks return an iterator of Key/Value pairs, the most common way to do this is by using the python "yield" statement to produce a generator. The combiner and reducer take a key along with an iterator of values corresponding to that key as input. Below are examples of map-only, map/reduce, and map/combine/reduce jobs that all act as an identify function (i.e., the same key/value pairs that are input are in the final output).

While using generators is the most common way to make Hadoopy tasks, as long as a task returns an iterator of Key/Value pairs or None (useful if the task doesn't output when it is called) it will work.

The class form also provides an additional capability that is useful in more advanced design patterns. A class can specify a "close" method that is called after all inputs are provided to the task. This often leads to map/reduce tasks that return None when called, but produce meaningful output when close is called. Below is an example of an identity Map task that buffers all the inputs and outputs them during close. Note this is to demonstrate the concept and would generally be a bad idea as the Map tasks would run out of memory when given large inputs.

1.4 Running Jobs

Now that we have a Hadoopy job, we would like to execute it on an existing Hadoop cluster. To launch a job, hadoopy builds the necessary command line arguments to call the "hadoop" command with. The command that is constructed is shown when the job is launched (depending on the logging level enabled). The hadoopy.launch command requires an hdfs input, hdfs output, and script path (in order). It sends the python script with the job and it is executed on the cluster. That means that everything the job script needs in terms of Python version, Python packages (e.g., numpy), C libraries (e.g., lapack), and utilities (e.g., ffmpeg) must already reside on the server. If this sounds difficult (i.e., ensuring that all machines have identical libraries/packages) or impossible (e.g., you have no admin access to the cluster) then please continue reading to the "hadoopy.launch_frozen" command. Additional files (including .py files) can be included in the "files" keyword argument (they will all be placed in the local directory of the job).

1.5 Writing Jobs

While each job is different I'll describe a common process for designing and writing them. This process is for an entire workflow which may consist of multiple jobs and intermingled client-side processing. The first step is to identify what you are trying to do as a series of steps (very important), you then start by identifying parallelism. Is your data entirely independent (i.e., embarrassingly parallel)? If so then use a Map-only job. Does your problem involve a "join"? If so then use a Map/Reduce job. It helps if you think in extremes about your data. Maybe you are using a small test set now, what if you were using a TB of data?

One of the most important things to get comfortable with is what data should be input to a job, and what data should be included as side-data. Side-data is data that each job has access to and doesn't come as input to the job. This is important because it enables many ways of factoring your problem. Something to watch out for is making things "too scalable" in that you are developing jobs that have constant memory and time requirements (i.e., O(1)) but end up not using your machines efficiently. A warning sign is when the majority of your time is spent in the Shuffle phase (i.e., copying/sorting data before the Reducer runs), at that point you should consider if there is a way to utilize side-data, a combiner (with a preference for in-mapper combiners), or computation on the local machine to speed the task up. Side-data may be a trained classifier (e.g., face detector), configuration parameters (e.g., number of iterations), and small data (e.g., normalization factor, cluster centers).

Four ways of providing side data (in recommended order) are

- Files that are copied to the local directory of your job (using the "files" argument in the launchers)
- Environmental variables accessibile through os.environ (using the "cmdenvs" argument in the launchers)
- Python scripts (can be stored as a global string, useful with launch_frozen as it packages up imported .py files)
- HDFS paths (using hadoopy.readtb)

1.6 Getting data from HDFS

After you've run your Hadoop jobs you'll eventually want to get something back from HDFS. The most effective way of doing this in Hadoopy is using the hadoopy.readtb command which provides an iterator over Key/Value pairs in a SequenceFile. Below is an example of how to read data of HDFS and store each key/value pair as a file with name as the key and value as the file (assumes unique keys).

CHAPTER 2

Hadoop Cluster Setup

2.1 Supported Versions

Any version of Hadoop that has built in TypedBytes support (Apache 0.21+, CDH2+) should work with all features (if not please submit a bug report). The version that currently has the most testing is CDH3, and CDH2/4 have also been tested. Hadoopy can be used with text input/output (see the text guide guide) in all Hadoop distributions that support streaming; however, this input/output method is not recommended as TypedBytes handles serialization, is unambiguous (i.e., no need to worry about binary data with tabs or new lines), and is more efficient.

2.2 launch vs. launch_frozen

An important consideration is whether you want to use launch or launch_frozen primarily to start jobs. The preferred method is launch as there is very little startup overhead; however, it requires each node of the cluster to have Python, Hadoopy, and all libraries that you use in your job. The benefit of launch_frozen is that you can use it on clusters that don't even have Python, that you have no administrative access to, or don't want to have to maintain (e.g., ephemeral EC2 clusters to run a job). If you want to use launch, then be prepared to have some method for keeping the cluster homogeneous and up-to-date (e.g., using puppet or fabric). If you use launch_frozen, then you have a 15+ second penalty each time PyInstaller freezes your script which is mitigated in large workflows as repeated called to launch_frozen for the same script only use PyInstaller once by default (overrided by setting cache=False). This can be further mitagated by manually maintaining PyInstaller paths on HDFS and setting the frozen_tar_path argument (useful if the jobs are not modified often). Ideally if the PyInstaller time can be brought down to a second or a more aggressive caching stategy is used, launch_frozen would be the preferred method in nearly all cases (we are currently working on this).

2.3 Whirr

The easiest way to install Hadoop on EC2 is to use whirr.

CHAPTER 3

Example Projects

3.1 Compute Vector Statistics

Compute statistics for different vector groups. The first job is a basic implementation, the second uses the “In-mapper” combine design pattern to minimize the amount of data sent to the reducer(s) during the shuffle phase.

Below is the driver and test script for the above jobs.

3.2 Resize Images

3.3 Face Finder

3.4 Get frames from Video

Programming Documentation

This section contains library documentation for Hadoopy.

4.1 Job Launchers (Start Hadoopy Jobs)

`hadoopy.launch`(*in_name*, *out_name*, *script_path*[, *partitioner=False*, *files=()*, *jobconfs=()*, *cmdenvs=()*, *copy_script=True*, *wait=True*, *hstreaming=None*, *name=None*, *use_typedbytes=True*, *use_seqoutput=True*, *use_autoinput=True*, *add_python=True*, *config=None*, *pipe=True*, *python_cmd="python"*, *num_mappers=None*, *num_reducers=None*, *script_dir=''*, *remove_ext=False*, ***kw*])

Run Hadoop given the parameters

Parameters

- **`in_name`** – Input path (string or list)
- **`out_name`** – Output path
- **`script_path`** – Path to the script (e.g., script.py)
- **`partitioner`** – If True, the partitioner is the value.
- **`files`** – Extra files (other than the script) (iterator). NOTE: Hadoop copies the files into working directory
- **`jobconfs`** – Extra jobconf parameters (iterator of strings or dict)
- **`cmdenvs`** – Extra cmdenv parameters (iterator of strings or dict)
- **`libjars`** – Extra jars to include with the job (iterator of strings).
- **`input_format`** – Custom input format (if set overrides use_autoinput)
- **`output_format`** – Custom output format (if set overrides use_seqoutput)
- **`copy_script`** – If True, the script is added to the files list.
- **`wait`** – If True, wait till the process is completed (default True) this is useful if you want to run multiple jobs concurrently by using the 'process' entry in the output.
- **`hstreaming`** – The full hadoop streaming path to call.
- **`name`** – Set the job name to this (default None, job name is the script name)
- **`use_typedbytes`** – If True (default), use typedbytes IO.

- **`use_seqoutput`** – True (default), output sequence file. If False, output is text. If output_format is set, this is not used.
- **`use_autoinput`** – If True (default), sets the input format to auto. If input_format is set, this is not used.
- **`remove_output`** – If True, output directory is removed if it exists. (defaults to False)
- **`add_python`** – If true, use 'python script_name.py'
- **`config`** – If a string, set the hadoop config path
- **`pipe`** – If true (default) then call user code through a pipe to isolate it and stop bugs when printing to stdout. See project docs.
- **`python_cmd`** – The python command to use. The default is "python". Can be used to override the system default python, e.g. python_cmd = "python2.6"
- **`num_mappers`** – The number of mappers to use (i.e., jobconf mapred.map.tasks=num_mappers).
- **`num_reducers`** – The number of reducers to use (i.e., jobconf mapred.reduce.tasks=num_reducers).
- **`script_dir`** – Where the script is relative to working dir, will be prefixed to script_path with a / (default '' is current dir)
- **`remove_ext`** – If True, remove the script extension (default False)
- **`check_script`** – If True, then copy script and .py(c) files to a temporary directory and verify that it can be executed. This catches the majority of errors related to not included locally imported files. (default True)
- **`make_executable`** – If True, ensure that script is executable and has a #! line at the top.
- **`required_files`** – Iterator of files that must be specified (verified against the files argument)
- **`required_cmdenvs`** – Iterator of cmdenvs that must be specified (verified against the cmdenvs argument)

Return type Dictionary with some of the following entries (depending on options)

Returns freeze_cmds: Freeze command(s) ran

Returns frozen_tar_path: HDFS path to frozen file

Returns hadoop_cmds: Hadoopy command(s) ran

Returns process: subprocess.Popen object

Returns output: Iterator of (key, value) pairs

Raises subprocess.CalledProcessError: Hadoop error.

Raises OSError: Hadoop streaming not found.

Raises TypeError: Input types are not correct.

Raises ValueError: Script not found or check_script failed

`hadoopy.`**`launch_frozen`**(*in_name*, *out_name*, *script_path*[, *frozen_tar_path=None*, *temp_path='_hadoopy_temp'*, *partitioner=False*, *wait=True*, *files=()*, *jobconfs=()*, *cmdenvs=()*, *hstreaming=None*, *name=None*, *use_typedbytes=True*, *use_seqoutput=True*, *use_autoinput=True*, *add_python=True*, *config=None*, *pipe=True*, *python_cmd="python"*, *num_mappers=None*, *num_reducers=None*, ***kw*])

Freezes a script and then launches it.

This function will freeze your python program, and place it on HDFS in 'temp_path'. It will not remove it afterwards as they are typically small, you can easily reuse/debug them, and to avoid any risks involved with removing the file.

Parameters

- **`in_name`** – Input path (string or list)
- **`out_name`** – Output path
- **`script_path`** – Path to the script (e.g., script.py)
- **`frozen_tar_path`** – If not None, use this path to a previously frozen archive. You can get such a path from the return value of this function, it is particularly helpful in iterative programs.
- **`cache`** – If True (default) then use previously frozen scripts. Cache is stored in memory (not persistent).
- **`temp_path`** – HDFS path that we can use to store temporary files (default to _hadoopy_temp)
- **`partitioner`** – If True, the partitioner is the value.
- **`wait`** – If True, wait till the process is completed (default True) this is useful if you want to run multiple jobs concurrently by using the 'process' entry in the output.
- **`files`** – Extra files (other than the script) (iterator). NOTE: Hadoop copies the files into working directory
- **`jobconfs`** – Extra jobconf parameters (iterator)
- **`cmdenvs`** – Extra cmdenv parameters (iterator)
- **`hstreaming`** – The full hadoop streaming path to call.
- **`name`** – Set the job name to this (default None, job name is the script name)
- **`use_typedbytes`** – If True (default), use typedbytes IO.
- **`use_seqoutput`** – True (default), output sequence file. If False, output is text.
- **`use_autoinput`** – If True (default), sets the input format to auto.
- **`config`** – If a string, set the hadoop config path
- **`pipe`** – If true (default) then call user code through a pipe to isolate it and stop bugs when printing to stdout. See project docs.
- **`python_cmd`** – The python command to use. The default is "python". Can be used to override the system default python, e.g. python_cmd = "python2.6"
- **`num_mappers`** – The number of mappers to use, i.e. the argument given to 'numMapTasks'. If None, then do not specify this argument to hadoop streaming.
- **`num_reducers`** – The number of reducers to use, i.e. the argument given to 'numReduceTasks'. If None, then do not specify this argument to hadoop streaming.
- **`check_script`** – If True, then copy script and .py(c) files to a temporary directory and verify that it can be executed. This catches the majority of errors related to not included locally imported files. The default is False when using launch_frozen as the freeze process packages local files.

Return type Dictionary with some of the following entries (depending on options)

Returns freeze_cmds: Freeze command(s) ran

Returns frozen_tar_path: HDFS path to frozen file

Returns hadoop_cmds: Hadoopy command(s) ran

Returns process: subprocess.Popen object

Returns output: Iterator of (key, value) pairs

Raises subprocess.CalledProcessError: Hadoop error.

Raises OSError: Hadoop streaming not found.

Raises TypeError: Input types are not correct.

Raises ValueError: Script not found

`hadoopy.`**`launch_local`**(*in_name*, *out_name*, *script_path*[, *max_input=-1*, *files=()*, *cmdenvs=()*, *pipe=True*, *python_cmd='python'*, *remove_tempdir=True*, ***kw*])

A simple local emulation of hadoop

This doesn't run hadoop and it doesn't support many advanced features, it is intended for simple debugging. The input/output uses HDFS if an HDFS path is given. This allows for small tasks to be run locally (primarily while debugging). A temporary working directory is used and removed.

Support

- Environmental variables
- Map-only tasks
- Combiner
- Files
- Pipe (see below)
- Display of stdout/stderr
- Iterator of KV pairs as input or output (bypassing HDFS)

Parameters

- **`in_name`** – Input path (string or list of strings) or Iterator of (key, value). If it is an iterator then no input is taken from HDFS.
- **`out_name`** – Output path or None. If None then output is not placed on HDFS, it is available through the 'output' key of the return value.
- **`script_path`** – Path to the script (e.g., script.py)
- **`poll`** – If not None, then only attempt to get a kv pair from kvs if when called, poll returns True.
- **`max_input`** – Maximum number of Mapper inputs, None (default) then unlimited.
- **`files`** – Extra files (other than the script) (iterator). NOTE: Hadoop copies the files into working directory
- **`cmdenvs`** – Extra cmdenv parameters (iterator)
- **`pipe`** – If true (default) then call user code through a pipe to isolate it and stop bugs when printing to stdout. See project docs.
- **`python_cmd`** – The python command to use. The default is "python". Can be used to override the system default python, e.g. python_cmd = "python2.6"

- **`remove_tempdir`** – If True (default), then rmtree the temporary dir, else print its location. Useful if you need to see temporary files or how input files are copied.
- **`identity_mapper`** – If True, use an identity mapper, regardless of what is in the script.
- **`num_reducers`** – If 0, don't run the reducer even if one exists, else obey what is in the script.

Return type Dictionary with some of the following entries (depending on options)

Returns freeze_cmds: Freeze command(s) ran

Returns frozen_tar_path: HDFS path to frozen file

Returns hadoop_cmds: Hadoopy command(s) ran

Returns process: subprocess.Popen object

Returns output: Iterator of (key, value) pairs

Raises subprocess.CalledProcessError: Hadoop error.

Raises OSError: Hadoop streaming not found.

Raises TypeError: Input types are not correct.

Raises ValueError: Script not found

4.2 Task functions (Usable inside Hadoopy jobs)

`hadoopy.`**`run`**(*mapper=None*, *reducer=None*, *combiner=None*, ***kw*)

Hadoopy entrance function

This is to be called in all Hadoopy job's. Handles arguments passed in, calls the provided functions with input, and stores the output.

TypedBytes are used if the following is True os.environ['stream_map_input'] == 'typedbytes'

It is *highly* recommended that TypedBytes be used for all non-trivial tasks. Keep in mind that the semantics of what you can safely emit from your functions is limited when using Text (i.e., no t or n). You can use the base64 module to ensure that your output is clean.

If the HADOOPY_CHDIR environmental variable is set, this will immediately change the working directory to the one specified. This is useful if your data is provided in an archive but your program assumes it is in that directory.

As hadoop streaming relies on stdin/stdout/stderr for communication, anything that outputs on them in an unexpected way (especially stdout) will break the pipe on the Java side and can potentially cause data errors. To fix this problem, hadoopy allows file descriptors (integers) to be provided to each task. These will be used instead of stdin/stdout by hadoopy. This is designed to combine with the 'pipe' command.

To use the pipe functionality, instead of using *your_script.py map* use *your_script.py pipe map* which will call the script as a subprocess and use the read_fd/write_fd command line arguments for communication. This isolates your script and eliminates the largest source of errors when using hadoop streaming.

The pipe functionality has the following semantics stdin: Always an empty file stdout: Redirected to stderr (which is visible in the hadoop log) stderr: Kept as stderr read_fd: File descriptor that points to the true stdin write_fd: File descriptor that points to the true stdout

Command Interface

The command line switches added to your script (e.g., script.py) are

python script.py *map* (read_fd) (write_fd) Use the provided mapper, optional read_fd/write_fd.

python script.py *reduce* (read_fd) (write_fd) Use the provided reducer, optional read_fd/write_fd.

python script.py *combine* (read_fd) (write_fd) Use the provided combiner, optional read_fd/write_fd.

python script.py *freeze* <tar_path> <-Z add_file0 -Z add_file1...> Freeze the script to a tar file specified by <tar_path>. The extension may be .tar or .tar.gz. All files are placed in the root of the tar. Files specified with -Z will be added to the tar root.

python script.py info Prints a json object containing 'tasks' which is a list of tasks which can include 'map', 'combine', and 'reduce'. Also contains 'doc' which is the provided documentation through the doc argument to the run function. The tasks correspond to provided inputs to the run function.

Specification of mapper/reducer/combiner

Input Key/Value Types

For TypedBytes/SequenceFileInputFormat, the Key/Value are the decoded TypedBytes

For TextInputFormat, the Key is a byte offset (int) and the Value is a line without the newline (string)

Output Key/Value Types

For TypedBytes, anything Pickle-able can be used

For Text, types are converted to string. Note that neither may contain t or n as these are used in the encoding. Output is keytvaluen

Expected arguments

mapper(key, value) or mapper.map(key, value)

reducer(key, values) or reducer.reduce(key, values)

combiner(key, values) or combiner.reduce(key, values)

Optional methods

func.configure(): Called before any input read. Returns None.

func.close(): Called after all input read. Returns None or Iterator of (key, value)

Expected return

None or Iterator of (key, value)

Parameters

- **`mapper`** – Function or class following the above spec
- **`reducer`** – Function or class following the above spec
- **`combiner`** – Function or class following the above spec
- **`doc`** – If specified, on error print this and call sys.exit(1)

`hadoopy.`**`status`**(*msg*[, *err=None*])

Output a status message that is displayed in the Hadoop web interface

The status message will replace any other, if you want to append you must do this yourself.

Parameters

- **`msg`** – String representing the status message
- **`err`** – Func that outputs a string, if None then sys.stderr.write is used (default None)

`hadoopy.counter(group, counter[, amount=1, err=None])`

Output a counter update that is displayed in the Hadoop web interface

Counters are useful for quickly identifying the number of times an error occurred, current progress, or coarse statistics.

Parameters

- **group** – Counter group
- **counter** – Counter name
- **amount** – Value to add (default 1)
- **err** – Func that outputs a string, if None then sys.stderr.write is used (default None)

4.3 HDFS functions (Usable locally and in Hadoopy jobs)

`hadoopy.readtb(paths[, ignore_logs=True, num_procs=10])`

Read typedbytes sequence files on HDFS (with optional compression).

By default, ignores files who's names start with an underscore '_' as they are log files. This allows you to cat a directory that may be a variety of outputs from hadoop (e.g., _SUCCESS, _logs). This works on directories and files. The KV pairs may be interleaved between files (they are read in parallel).

Parameters

- **paths** – HDFS path (str) or paths (iterator)
- **num_procs** – Number of reading procs to open (default 1)
- **java_mem_mb** – Integer of java heap size in MB (default 256)
- **ignore_logs** – If True, ignore all files who's name starts with an underscore. Defaults to True.

Returns An iterator of key, value pairs.

Raises IOError: An error occurred reading the directory (e.g., not available).

`hadoopy.writetb(path, kvs)`

Write typedbytes sequence file to HDFS given an iterator of KeyValue pairs

Parameters

- **path** – HDFS path (string)
- **kvs** – Iterator of (key, value)
- **java_mem_mb** – Integer of java heap size in MB (default 256)

Raises IOError: An error occurred while saving the data.

`hadoopy.abspath(path)`

Return the absolute path to a file and canonicalize it

Path is returned without a trailing slash and without redundant slashes. Caches the user's home directory.

Parameters **path** – A string for the path. This should not have any wildcards.

Returns Absolute path to the file

Raises **IOError** – If unsuccessful `hadoopy.`**`ls`**`(`*path*`)`

List files on HDFS.

Parameters **`path`** – A string (potentially with wildcards).

Return type A list of strings representing HDFS paths.

Raises IOError: An error occurred listing the directory (e.g., not available).

`hadoopy.`**`get`**`(`*hdfs_path*, *local_path*`)`

Get a file from hdfs

Parameters

- **`hdfs_path`** – Destination (str)
- **`local_path`** – Source (str)

Raises IOError: If unsuccessful

`hadoopy.`**`put`**`(`*local_path*, *hdfs_path*`)`

Put a file on hdfs

Parameters

- **`local_path`** – Source (str)
- **`hdfs_path`** – Destination (str)

Raises IOError: If unsuccessful

`hadoopy.`**`rmr`**`(`*path*`)`

Remove a file if it exists (recursive)

Parameters **`path`** – A string (potentially with wildcards).

Raises **`IOError`** – If unsuccessful

`hadoopy.`**`isempty`**`(`*path*`)`

Check if a path has zero length (also true if it's a directory)

Parameters **`path`** – A string for the path. This should not have any wildcards.

Returns True if the path has zero length, False otherwise.

`hadoopy.`**`isdir`**`(`*path*`)`

Check if a path is a directory

Parameters **`path`** – A string for the path. This should not have any wildcards.

Returns True if the path is a directory, False otherwise.

`hadoopy.`**`exists`**`(`*path*`)`

Check if a file exists.

Parameters **`path`** – A string for the path. This should not have any wildcards.

Returns True if the path exists, False otherwise.

CHAPTER 5

Text Input

When working with Hadoopy it is preferable to work with SequenceFiles encoded with TypedBytes; however, it can operate on raw text input too (e.g., put text data directly on HDFS). This section explores the use of raw text with Hadoopy. Note that this is distinct from Hadoop Streaming's communication method with Hadoopy (i.e., Text or TypedBytes), see internals for more discussion.

5.1 Putting Text Data on HDFS

As Hadoopy jobs can take pure text as input, another option is to make a large line-oriented textfile. The main drawback of this is that you have to do your own encoding (especially avoiding using newline/tab characters in binary data). A conservative way to do this is to pick any encoding you want (e.g., JSON, Python Pickles, Protocol Buffers, etc.) and base64 encode the result. In the following example, we duplicate the behavior of the writetb method by base64 encoding the path and binary contents. The two are distinguishable because of the tab separator and each pair is on a separate line.

CHAPTER 6

Background

This section contains links to relevent background material that is often useful when working with Hadoopy.

6.1 Python Generators

Python Generators are often used in Hadoopy jobs, below is an example of generators in Python.

6.2 Using C Code in Python (Cython and Ctypes Example)

There are a variety of methods of calling C code from Python. Two simple methods are using CTypes (a library that comes with python that can call functions in shared libraries) and Cython (a language that mixes Python and C that enables easily integrating them). The recommended method is using Cython as it can build a self contained shared object (.so) which simplifies deployment as launch frozen is able to determine that it is being used and bring it along. When using ctypes it can be difficult to ensure that the python executable can find the correct libraries and they must be manually included with the job. The following example shows Python, Numpy, C (Ctypes), and C (Cython) usage.

6.3 Cython

The core of Hadoopy is written in the Cython language which is effectively Python with types but that is compiled to C. This enables the code to be easily maintained but perform at native speeds.

6.4 MapReduce

http://code.google.com/edu/parallel/mapreduce-tutorial.html

6.5 OpenCV

http://docs.opencv.org/ http://opencv.willowgarage.com/wiki/

6.6 Hadoop

http://hadoop.apache.org/

CHAPTER 7

Hadoopy Internals

This section is for understanding how Hadoopy works.

7.1 Streaming

Hadoopy uses the Hadoop Streaming mechanism to run jobs and communicate with Hadoop.

7.2 TypedBytes

Hadoopy makes exensive use of the TypedBytes encoding. It is used to communicate between Hadoop Streaming and Hadoopy, the alternative is a line/tab delimited key/value system (the old style) which leaves serialization up to you and you have to avoid using those characters.

7.3 PyInstaller

The launch_frozen command uses PyInstaller (included in the source tree) to package up all Python scripts (.py) and shared libraries (.so) into a .tar file consisting of a self-contained executable (the same name as your script without .py) and the shared libraries. This means that launch_frozen allows you to launch jobs on clusters without Python or any libraries that your job needs; however, there is a 15+ second overhead to do this (there are tricks to minimizing this effect).

7.4 Detailed Functional Walkthrough

To explain how Hadoopy works, we will now walk through a job and discuss some of the behind-the-scenes details of Hadoopy.

Python Source (fully documented version in wc.py)

```
"""Hadoopy Wordcount Demo"""
import hadoopy

def mapper(key, value):
    for word in value.split():
        yield word, 1
```

```
def reducer(key, values):
    accum = 0
    for count in values:
        accum += int(count)
    yield key, accum

if __name__ == "__main__":
    hadoopy.run(mapper, reducer, doc=__doc__)
```

Command line test (run without args, it prints the docstring and quits because of doc=__doc__)

```
$ python wc.py
Hadoopy Wordcount Demo
```

Command line test (map)

```
$ echo "a b a a b c" | python wc.py map
a     1
b     1
a     1
a     1
b     1
c     1
```

Command line test (map/sort)

```
$ echo "a b a a b c" | python wc.py map | sort
a     1
a     1
a     1
b     1
b     1
c     1
```

Command line test (map/sort/reduce)

```
$ echo "a b a a b c" | python wc.py map | sort | python wc.py reduce
a     3
b     2
c     1
```

Here are a few test files

```
$ hadoop fs -ls playground/
Found 3 items
-rw-r--r--   2 brandyn supergroup     259835 2011-01-17 18:56 /user/brandyn/playground/wc-input-alice
-rw-r--r--   2 brandyn supergroup     167529 2011-01-17 18:56 /user/brandyn/playground/wc-input-alice
-rw-r--r--   2 brandyn supergroup      60638 2011-01-17 18:56 /user/brandyn/playground/wc-input-alice
```

We can also do this in Python

```
>>> import hadoopy
>>> hadoopy.ls('playground/')
['/user/brandyn/playground/wc-input-alice.tb', '/user/brandyn/playground/wc-input-alice.txt', '/user/
```

Lets put wc-input-alice.txt through the word counter using Hadoop. Each node in the cluster has Hadoopy installed (later we will show that it isn't necessary with launch_frozen). Note that it is using typedbytes, SequenceFiles, and the AutoInputFormat by default.

```
>>> out = hadoopy.launch('playground/wc-input-alice.txt', 'playground/out/', 'wc.py')
/\----------Hadoop Output----------/\
hadoopy: Running[hadoop jar /usr/lib/hadoop-0.20/contrib/streaming/hadoop-streaming-0.20.2+737.jar -o
11/01/17 20:22:31 WARN streaming.StreamJob: -jobconf option is deprecated, please use -D instead.
packageJobJar: [wc.py, /var/lib/hadoop-0.20/cache/brandyn/hadoop-unjar464849802654976085/] [] /tmp/st
11/01/17 20:22:32 INFO mapred.FileInputFormat: Total input paths to process : 1
11/01/17 20:22:32 INFO streaming.StreamJob: getLocalDirs(): [/var/lib/hadoop-0.20/cache/brandyn/mapre
11/01/17 20:22:32 INFO streaming.StreamJob: Running job: job_201101141644_0723
11/01/17 20:22:32 INFO streaming.StreamJob: To kill this job, run:
11/01/17 20:22:32 INFO streaming.StreamJob: /usr/lib/hadoop-0.20/bin/hadoop job  -Dmapred.job.tracker
11/01/17 20:22:32 INFO streaming.StreamJob: Tracking URL: http://node.com:50030/jobdetails.jsp?jobid=
11/01/17 20:22:33 INFO streaming.StreamJob:  map 0%  reduce 0%
11/01/17 20:22:40 INFO streaming.StreamJob:  map 50%  reduce 0%
11/01/17 20:22:41 INFO streaming.StreamJob:  map 100%  reduce 0%
11/01/17 20:22:52 INFO streaming.StreamJob:  map 100%  reduce 100%
11/01/17 20:22:55 INFO streaming.StreamJob: Job complete: job_201101141644_0723
11/01/17 20:22:55 INFO streaming.StreamJob: Output: playground/out/
\/----------Hadoop Output----------\/
```

Return value is a dictionary of the command's run, key/value iterator of the output (lazy evaluated), and other useful intermediate values.

Lets see what the output looks like.

```
>>> out = list(hadoopy.readtb('playground/out'))
>>> out[:10]
[('*', 60), ('-', 7), ('3', 2), ('4', 1), ('A', 8), ('I', 260), ('O', 1), ('a', 662), ('"I', 7), ("'A
>>> out.sort(lambda x, y: cmp(x[1], y[1]))
>>> out[-10:]
[('was', 329), ('it', 356), ('in', 401), ('said', 416), ('she', 484), ('of', 596), ('a', 662), ('to',
```

Note that the output is stored in SequenceFiles and each key/value is stored encoded as TypedBytes, by using readtb you don't have to worry about any of that (if the output was compressed it would also be decompressed here). This can also be done inside of a job for getting additional side-data off of HDFS.

What if we don't want to install python, numpy, scipy, or your-custom-code-that-always-changes on every node in the cluster? We have you covered there too. I'll remove hadoopy from all nodes except for the one executing the job.

```
$ sudo rm -r /usr/local/lib/python2.7/dist-packages/hadoopy*
```

Now it's gone

```
>>> import hadoopy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named hadoopy
```

The rest of the nodes were cleaned up the same way. We modify the command, note that we now get the output from freeze at the top

```
>>> out = hadoopy.launch_frozen('playground/wc-input-alice.txt', 'playground/out_frozen/', 'wc.py')
/\----------Hadoop Output----------/\
hadoopy: Running[hadoop jar /hadoop-0.20.2+320/contrib/streaming/hadoop-streaming-0.20.2+320.jar -out
11/07/07 21:23:23 WARN streaming.StreamJob: -jobconf option is deprecated, please use -D instead.
packageJobJar: [/tmp/hadoop/brandyn/hadoop-unjar12844/] [] /tmp/streamjob12845.jar tmpDir=null
11/07/07 21:23:24 INFO mapred.FileInputFormat: Total input paths to process : 1
11/07/07 21:23:24 INFO streaming.StreamJob: getLocalDirs(): [/scratch0/hadoop/mapred/local, /scratch1
11/07/07 21:23:24 INFO streaming.StreamJob: Running job: job_201107051032_0215
11/07/07 21:23:24 INFO streaming.StreamJob: To kill this job, run:
11/07/07 21:23:24 INFO streaming.StreamJob: /hadoop-0.20.2+320/bin/hadoop job  -Dmapred.job.tracker=n
```

```
11/07/07 21:23:24 INFO streaming.StreamJob: Tracking URL: http://node.com:50030/jobdetails.jsp?jobid=
11/07/07 21:23:25 INFO streaming.StreamJob:  map 0%  reduce 0%
11/07/07 21:23:31 INFO streaming.StreamJob:  map 100%  reduce 0%
11/07/07 21:23:46 INFO streaming.StreamJob:  map 100%  reduce 100%
11/07/07 21:23:49 INFO streaming.StreamJob: Job complete: job_201107051032_0215
11/07/07 21:23:49 INFO streaming.StreamJob: Output: playground/out_frozen/
\/----------Hadoop Output----------\/
```

And lets check the output

```
>>> out = list(hadoopy.readtb('playground/out_frozen'))
>>> out[:10]
[('*', 60), ('-', 7), ('3', 2), ('4', 1), ('A', 8), ('I', 260), ('O', 1), ('a', 662), ('"I', 7), ("'A
>>> out.sort(lambda x, y: cmp(x[1], y[1]))
>>> out[-10:]
[('was', 329), ('it', 356), ('in', 401), ('said', 416), ('she', 484), ('of', 596), ('a', 662), ('to',
```

We can also generate a tar of the frozen script (useful when working with Oozie). Note the ‘wc’ is not wc.py, it is actually a self contained executable.

```
$ python wc.py freeze wc.tar
$ tar -tf wc.tar
_codecs_cn.so
readline.so
strop.so
cPickle.so
time.so
_collections.so
operator.so
zlib.so
_codecs_jp.so
grp.so
_codecs_kr.so
_codecs_hk.so
_functools.so
_json.so
math.so
libbz2.so.1.0
libutil.so.1
unicodedata.so
array.so
_bisect.so
libz.so.1
_typedbytes.so
_random.so
_main.so
cStringIO.so
_codecs_tw.so
libncurses.so.5
datetime.so
_struct.so
_weakref.so
fcntl.so
_heapq.so
wc
_io.so
select.so
_codecs_iso2022.so
_locale.so
```

```
itertools.so
binascii.so
bz2.so
libpython2.7.so.1.0
_multibytecodec.so
```

Lets open it up and try it out

```
$ tar -xf wc.py
$ ./wc
Hadoopy Wordcount Demo
$ python wc.py
Hadoopy Wordcount Demo
$ hexdump -C wc | head -n5
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  80 41 40 00 00 00 00 00  |..>......A@.....|
00000020  40 00 00 00 00 00 00 00  50 04 16 00 00 00 00 00  |@.......P.......|
00000030  00 00 00 00 40 00 38 00  09 00 40 00 1d 00 1c 00  |....@.8...@.....|
00000040  06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
```

7.5 Pipe Hopping: Using Stdout/Stderr in Hadoopy Jobs

Hadoop streaming implements the standard Mapper/Reducer classes and simply opens 3 pipes to a streaming program (stdout, stderr, and stdin). The first issue is how is data encoded? The standard is to separate keys and values with a tab and each key/value pair with a newline; however, this is really a bad way to have to work as you have to ensure that your output never contains tabs or newlines. Moreover, serializing everything to an escaped string is inefficient and tends to hurt interoperability of jobs as everyone has their own solution to encoding. The solution (part of CDH2+) is to use TypedBytes which is an encoding format for basic types (int, float, dictionary, list, string, etc.) which is fast, standardized, and simple. Hadoopy has its own implementation and it is particularly fast.

TypedBytes doesn't solve the issue of client code outputting to stdout, it actually makes it worse as the resulting output is interpreted as TypedBytes which can have very complex effects. Most Hadoop streaming programs have to meticulously avoid printing to stdout as it will interfere with the connection to Hadoop streaming. Hadoopy uses a technique I refer to as 'pipe hopping' where a launcher remaps the stdin/stdout of the client program to be null and stderr respectively, and communication happens over file descriptors which correspond to the true stdout/stdin that Hadoop streaming communicates with. This is transparent to the user but the end result is more useful error messages when exceptions are thrown (as opposed to generic Java errors) and the ability to use print statements like normal. This is a general solution to the problem and if other library writers (for python or other languages) would like a minimum working example of this technique I have one available.

This technique is on by default and can be disabled by passing pipe=False to the launch command of your choice.

7.6 Script Info

You can determine if a job provides map/reduce/combine functionality and get its documention by using 'info'. This is also used internally by Hadoopy to automatically enable/disable the reducer/combiner. The task values are set when the corresponding slots in hadoopy.run are filled.

```
>>> python wc.py info
{"doc": "Hadoopy Wordcount Demo", "tasks": ["map", "reduce"]}
```

CHAPTER 8

Benchmarks

8.1 readtb/writetb

8.2 launch_frozen (PyInstaller)

8.3 TypedBytes

The majority of the time spent by Hadoopy (and Dumbo) is in the TypedBytes conversion code. This is a simple binary serialization format that covers standard types with the ability to use Pickle for types not natively supported. We generate a large set of test vectors (using the tb_make_test.py script), that have primatives, containers, and a uniform mix (GrabBag). The idea is that by factoring out the types, we can easily see where optimization is needed. Each element is read from stdin, then written to stdout. Outside of the timing all of the values are compared to ensure that the final written values are the same. Four methods are compared: Hadoopy TypedBytes (speed_hadoopy.py), Hadoopy TypedBytes file interface (speed_hadoopyfp.py), TypedBytes (speed_tb.py), and cTypedBytes (speed_tbc.py). All columns are in seconds except for ratio. The ratio is min(TB, cTB) / HadoopyFP (e.g., 7 means HadoopyFP is 7 times faster).

Filename	Hadoopy	HadoopyFP	TB	cTB	Ratio
double_100k.tb	0.148790	0.119961	0.904720	0.993845	7.541784
float_100k.tb	0.145637	0.118920	0.883124	0.992447	7.426198
gb_100k.tb	4.638573	4.011934	25.577765	16.515563	4.116609
bool_100k.tb	0.171327	0.150975	0.942188	0.542741	3.594907
dict_50.tb	0.394323	0.364878	1.649921	1.225979	3.359970
tuple_50.tb	0.370037	0.413579	1.546317	1.241491	3.001823
byte_100k.tb	0.183307	0.164549	0.894184	0.487520	2.962767
list_50.tb	0.355870	0.370738	1.529233	1.092422	2.946614
int_100k.tb	0.234842	0.193235	0.922423	0.526160	2.722903
long_100k.tb	0.761289	0.640638	1.727951	1.957162	2.697234
bytes_100_10k.tb	0.069889	0.069375	0.147470	0.096838	1.395862
string_100_10k.tb	0.106642	0.104784	0.157907	0.106571	1.017054
string_10k_10k.tb	6.392013	6.527343	6.494607	6.949912	0.994985
bytes_10k_10k.tb	3.073718	3.123196	3.039668	3.100858	0.973256
string_1k_10k.tb	0.742198	0.719119	0.686382	0.676537	0.940786
bytes_1k_10k.tb	0.379785	0.370314	0.329728	0.339387	0.890401
gb_single.tb	0.045760	0.042701	0.038656	0.034925	0.817896

CHAPTER 9

Local Hadoop Execution

9.1 Mixing Local and Hadoop Computation

Using hadoopy.readtb along with hadoopy.writetb allows for trivial integration of local computation with Hadoop computation. An architecture design pattern that I've found useful is to have your job launcher machine to have a very large amount of memory (compared to the cluster nodes). This is useful because there are jobs that can be written in a constant memory formulation, but use excessive disk/network IO (e.g., training very large classifiers) and can be performed once on this node and then the result can be used by the Hadoop nodes. A mock example of this is shown below.

9.2 Launch Local (execute jobs w/o Hadoop)

CHAPTER 10

Hadoopy Flow: Automatic Job-Level Parallization (Experimental)

Hadoopy flow is experimental and is maintained out of branch at https://github.com/bwhite/hadoopy_flow. It is under active development.

Once you get past the wordcount examples and you have a few scripts you use regularly, the next level of complexity is managing a workflow of jobs. The simplest way of doing this is to put a few sequential launch statements in a python script and run it. This is fine for simple workflows but you miss out on two abilities: re-execution of previous workflows by re-using outputs (e.g., when tweaking one job in a flow) and parallel execution of jobs in a flow. I've had some fairly complex flows and previously the best solution I could find was using Oozie with a several thousand line XML file. Once setup, this ran jobs in parallel and re-execute the workflow by skipping previous nodes; however, it is another server you have to setup and making that XML file takes a lot of the fun out of using Python in the first place (it could be more code than your actual task). While Hadoopy is fully compatible with Oozie, it certainly seems lacking for the kind of short turn-around scripts most users want to make.

In solving this problem, our goal was to avoid specifying the dependencies (often as a DAG) as they are inherent in the code itself. Hadoopy Flow solves both of these problems by keeping track of all HDFS outputs your program intends to create and following your program order. By doing this, if we see a 'launch' command we run it in a 'greenlet', note the output path of the job, and continue with the rest of the program. If none of the job's inputs depend on any outputs that are pending (i.e., outputs that will materialize from previous jobs/hdfs commands) then we can safely start the job. This is entirely safe because if the program worked before Hadoopy Flow, then it will work now as those inputs must exist as nothing prior to the job could have created it. When a job completes, we notify dependent jobs/hdfs commands and if all of their inputs are available they are executed. The same goes for HDFS commands such as readtb and writetb (most but not all HDFS commands are supported, see Hadoopy Flow for more info). If you try to read from a file that another job will eventually output to but it hasn't finished yet, then the execution will block at that point until the necessary data is available.

So it sounds pretty magical, but it wouldn't be worth it if you have to rewrite all of your code. To use Hadoopy Flow, all that you have to do is add 'import hadoopy_flow' before you import Hadoopy, and it will automatically parallelize your code. It monkey patches Hadoopy (i.e., wraps the calls at run time) and the rest of your code can be unmodified. All of the code is just a few hundred lines in one file, if you are familiar with greenlets then it might take you 10 minutes to fully understand it (which I recommend if you are going to use it regularly).

Re-execution is another important feature that Hadoopy Flow addresses and it does so trivially. If after importing Hadoopy Flow you use 'hadoopy_flow.USE_EXISTING = True', then when paths already exist we simply skip the task/command that would have output to them. This is useful if you run a workflow, a job crashes, fix the bug, delete the bad job's output, and re-run the workflow. All previous jobs will be skipped and jobs that don't have their outputs on HDFS are executed like normal. This simple addition makes iterative development using Hadoop a lot more fun and effective as tweaks generally happen at the end of the workflow and you can easily waste hours recomputing results or hacking your workflow apart to short circuit it.

CHAPTER 11

Hadoopy Helper: Useful Hadoopy tools (Experimental)

Hadoopy Helper support is experimental and is maintained out of branch at https://github.com/bwhite/hadoopy_helper. It is under active development.

CHAPTER 12

HBase Integration (Experimental)

Hadoopy HBase support is experimental and is maintained out of branch at https://github.com/bwhite/hadoopy_hbase. It is under active development.

CHAPTER 13

Hadoopy/Java Integration

CHAPTER 14

Cookbook

This section is a collection of tips/tricks that are useful when working with Hadoopy.

14.1 Enabling Verbose Output

14.2 Map/Reduce Output Compression

14.3 Tweakable Jobconfs

14.4 Status/Counters in Jobs

14.5 Accessing Jobconfs Inside Jobs

14.6 Using Writetb to Write Multiple Parts

14.7 Reverse SSH Tunnel

14.8 Timing Sections of Code

14.9 Skipping Jobs in a Large Workflow

14.10 Randomly Sampling Key/Value Pairs

Visit https://github.com/bwhite/hadoopy/ for the source.

Relevant blog posts

- http://brandynwhite.com/hadoopy-cython-based-mapreduce-library-for-py

CHAPTER 15

About

Hadoopy is a Python wrapper for Hadoop Streaming written in Cython. It is simple, fast, and readily hackable. It has been tested on 700+ node clusters. The goals of Hadoopy are

- Similar interface as the Hadoop API (design patterns usable between Python/Java interfaces)
- General compatibility with dumbo to allow users to switch back and forth
- Usable on Hadoop clusters without Python or admin access
- Fast conversion and processing
- Stay small and well documented
- Be transparent with what is going on
- Handle programs with complicated .so's, ctypes, and extensions
- Code written for hack-ability
- Simple HDFS access (e.g., reading, writing, ls)
- Support (and not replicate) the greater Hadoop ecosystem (e.g., Oozie, whirr)

Killer Features (unique to Hadoopy)

- Automated job parallelization 'auto-oozie' available in the hadoopy flow project (maintained out of branch)
- Local execution of unmodified MapReduce job with launch_local
- Read/write sequence files of TypedBytes directly to HDFS from python (readtb, writetb)
- Allows printing to stdout and stderr in Hadoop tasks without causing problems (uses the 'pipe hopping' technique, both are available in the task's stderr)
- Works on clusters without any extra installation, Python, or any Python libraries (uses Pyinstaller that is included in this source tree)

Additional Features

- Works on OS X
- Critical path is in Cython
- Simple HDFS access (readtb and ls) inside Python, even inside running jobs
- Unit test interface
- Reporting using status and counters (and print statements! no need to be scared of them in Hadoopy)
- Supports design patterns in the Lin&Dyer book

- Typedbytes support (very fast)
- Oozie support