Pydoop: a Python MapReduce and HDFS API for Hadoop

Simone Leo        Gianluigi Zanetti

Distributed Computing Group
CRS4 – Cagliari (Italy)

MAPREDUCE ’10
Outline

1. Motivation
2. Architecture
3. Examples
4. Conclusions and Future Work
Java: Hadoop is all you need
C/C++: APIs for both MR and HDFS are supported by Hadoop Pipes and included in the Hadoop distribution
Python: current solutions fail to meet all requirements of nontrivial apps
  - Reuse existing modules, including C/C++ extensions
  - NumPy / SciPy for numerical computation
  - Specialized components (RecordReader/Writer, Partitioner . . . )
  - HDFS access
## Hadoop-Integrated Solutions

### Hadoop Streaming
- text protocol: cannot process arbitrary data streams
- can only write mapper and reducer scripts (no RecordReader, etc.)
- awkward programming style (read key/value pairs from stdin, write them to stdout)

### Jython
- incomplete standard library
- most third-party packages are only compatible with CPython
- cannot use C/C++ extensions
- typically one or more releases behind CPython
Third Party Solutions

- Dumbo
- Disco
- Octopy
- Happy
- Hadoop-based
- Jython

Hadoop-based: same limitations as Streaming (Dumbo) and Jython (Happy), except for ease of use

Other implementations: good if you have your own cluster
  - Hadoop is the most widespread implementation

Leo, Zanetti
Pydoop: a Python MapReduce and HDFS API for Hadoop
Our Solution: Pydoop

- Access to most MR components, including RecordReader, RecordWriter and Partitioner
- Get configuration, set counters and report status via context objects
- Framework is similar to the Java one: you define classes, framework instantiates them and calls their methods
- CPython: use any module
- HDFS API
## Summary of Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Streaming</th>
<th>Jython</th>
<th>Pydoop</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ Ext</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Standard Lib</td>
<td>Full</td>
<td>Partial</td>
<td>Full</td>
</tr>
<tr>
<td>MR API</td>
<td>No*</td>
<td>Full</td>
<td>Partial</td>
</tr>
<tr>
<td>Java-like FW</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>HDFS</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

(*) you can only write the map and reduce parts as executable scripts.
Outline

1. Motivation
2. Architecture
3. Examples
4. Conclusions and Future Work
Hadoop Pipes

- Communication with Java framework via persistent sockets
- The C++ app provides a factory used by the framework to create MR components
- Providing Mapper and Reducer is mandatory
Integration of Pydoop with C++

- **Integration with Pipes:**
  - Method calls flow from the framework through the C++ and the Pydoop API, ultimately reaching user-defined methods.
  - Results are wrapped by Boost and returned to the framework.

- **Integration with HDFS:**
  - Function calls initiated by Pydoop.
  - Results wrapped and returned as Python objects to the app.
HDFS: Command Line Examples

```python
>>> import os
>>> from pydoop.hdfs import hdfs
>>> fs = hdfs("localhost", 9000)
>>> fs.open_file("f", os.O_WRONLY).write(open("f").read())

>>> from pydoop.hdfs import hdfs
>>> for f in fs.list_directory("temp"):  
...     print f["name"].rsplit("/",1)[-1],
...     if f["kind"] == "file":
...         print f["size"]
...     else:
...         print "(%d)" % len(fs.list_directory(f["name"]))

dir (0)
file1 512000
file2 614400
file3 455680
```
from pydoop.hdfs import hdfs

def treewalker(fs, root_info):
    yield root_info
    if root_info["kind"] == "directory":
        for info in fs.list_directory(root_info["name"]):
            for item in treewalker(fs, info): yield item

def usage_by_bs(fs, root):
    stats = {}
    root_info = fs.get_path_info(root)
    for info in treewalker(fs, root_info):
        if info["kind"] == "directory": continue
        bs = int(info["block_size"])
        size = int(info["size"])
        stats[bs] = stats.get(bs, 0) + size
    return stats

def main(argv):
    fs = hdfs("localhost", 9000)
    root = fs.working_directory()
    for k, v in usage_by_bs(fs, root).iteritems():
        print "%.1f %d" % (k/float(2**20), v)
    fs.close()
from pydoop.pipes import Mapper, Reducer, Factory, runTask

class WordCountMapper(Mapper):
    def map(self, context):
        words = context.getInputValue().split()
        for w in words:
            context.emit(w, "1")

class WordCountReducer(Reducer):
    def reduce(self, context):
        s = 0
        while context.nextValue():
            s += int(context.getInputValue())
            context.emit(context.getInputKey(), str(s))

runTask(Factory(WordCountMapper, WordCountReducer))

- All communication happens through the context
  - Mapper/Reducer: get/emit key/value pairs
  - All components: get jobconf, update status and counters
```

class WordCountMapper(Mapper):
    def __init__(self, context):
        super(WordCountMapper, self).__init__(context)
        context.setStatus("initializing")
        self.inputWords = context.getCounter(WC, INPUT_WORDS)

    def map(self, context):
        words = context.getInputValue().split()
        for w in words:
            context.emit(w, "1")
        context.incrementCounter(self.inputWords, len(words))

class WordCountReducer(Reducer):
    def __init__(self, context):
        super(WordCountReducer, self).__init__(context)
        self.outputWords = context.getCounter(WC, OUTPUT_WORDS)

    def reduce(self, context):
        s = 0
        while context.nextValue():
            s += int(context.getInputValue())
        context.emit(context.getInputKey(), "str(s)")
        context.incrementCounter(self.outputWords, 1)
```
class WordCountReader(RecordReader):

def __init__(self, context):
    super(WordCountReader, self).__init__()
    self.isplit = InputSplit(context.getInputSplit())
    self.host, self.port, self.fpath = split_hdfs_path(self.isplit.filename)
    self.fs = hdfs(self.host, self.port)
    self.file = self.fs.open_file(self.fpath, os.O_RDONLY)
    self.file.seek(self.isplit.offset)
    self.bytes_read = 0
    if self.isplit.offset > 0:
        discarded = self.file.readline()  # read by previous reader
        self.bytes_read += len(discarded)

def next(self):
    # @return: (got_record, key, value)
    if self.bytes_read > self.isplit.length: return (False, '', '')
    key = struct.pack('>q', self.isplit.offset + self.bytes_read)
    record = self.file.readline()
    if record == '': return (False, '', '')
    self.bytes_read += len(record)
    return (True, key, record)

def getProgress(self):
    return min(float(self.bytes_read)/self.isplit.length, 1.0)
class WordCountWriter(RecordWriter):

    def __init__(self, context):
        super(WordCountWriter, self).__init__(context)
        jc = context.getJobConf()
        jc_configure_int(self, jc, "mapred.task.partition", "part")
        jc_configure(self, jc, "mapred.work.output.dir", "outdir")
        jc_configure(self, jc, "mapred.textoutputformat.separator", "sep", "\t")
        outfn = "%s/part-%05d" % (self.outdir, self.part)
        host, port, fpath = split_hdfs_path(outfn)
        self.fs = hdfs(host, port)
        self.file = self.fs.open_file(fpath, os.O_WRONLY)

    def emit(self, key, value):
        self.file.write("%s%s%s\n" % (key, self.sep, value))

class WordCountPartitioner(Partitioner):

    def partition(self, key, numOfReduces):
        reducer_id = (hash(key) & sys.maxint) % numOfReduces
        return reducer_id
Comparison: slower than Java/C++, as expected

- Cluster: 48 nodes with 2 dual core 1.8 GHz Opterons, 4 GB RAM
- App: WordCount on 20 GB of random English text
  - Dataset: uniform sampling from a spell checker list
  - Each run: 192 mappers, 90 reducers; 5 iterations
- Combiner = Reducer
Comparison: much better than Streaming

Cluster: 48 nodes with 2 dual core 1.8 GHz Opterons, 4 GB RAM

App: WordCount on 20 GB of random English text
  - Dataset: uniform sampling from a spell checker list
  - Each run: 192 mappers, 90 reducers; 5 iterations
  - No Combiner

Leo, Zanetti
Pydoop: a Python MapReduce and HDFS API for Hadoop
Pydoop Usage at CRS4

- **Motivation**
- **Architecture**
- **Examples**
- **Conclusions and Future Work**

**Pydoop: a Python MapReduce and HDFS API for Hadoop**

- **XML-RPC**
- **Bio Suite**
- **MR lib**
- **Short Read**
- **Alignment**
- **MR lib**
- **Pedigree**
- **Analysis**
- **MR lib**
- **Genotyping**
- **comlib**

**Target:**
- 5 full genomes per week
- 1.5K families
- 800K SNPs
- GWAS on 10K individuals
Conclusions

- Pydoop vs Jython:
  - Pydoop is CPython
  - No significant performance difference

- Pydoop vs Streaming:
  - Java-like API
  - Access to most MR components
  - Process any data type
  - HDFS access
  - Better performance
Current/future Work

- Pydoop is under active development
  - Continuous improvements are made, often arising from production application needs
- We are planning to add a more “pythonic” interface
  - Property access for keys/values
  - Python-style iterators

```python
class WordCountReducer(Reducer):
    def reduce(self, context):
        context.emit(context.input_key, str(sum(context.itervalues())))
```
Thank you for your time!

http://pydoop.sourceforge.net