Data Intensive Computing – Handout 6

Hadoop

Hadoop 1.2.1 is installed in `/HADOOP` directory. The JobTracker web interface is available at `http://dlrc:50030`, the NameNode web interface is available at `http://dlrc:50070`.

To conveniently access the web interfaces remotely, you can use `ssh -D localhost:2020 -p 11422 ufallab.ms.mff.cuni.cz` to open SOCKS5 proxy server forwarding requests to the remove site, and use it as a proxy server in a browser, for example as `chromium --proxy-server=socks://localhost:2020`.

Simple Tokenizer

We will commonly need to split given text into words (called *tokens*). You can do so easily by using function `wordpunct_tokenize` from `nltk.tokenize` package, i.e. using the following import line at the beginning of you program:

```
from nltk.tokenize import wordpunct_tokenize
```

Dumbo

Dumbo is (one of several) Python API to Hadoop. It utilizes Hadoop Streaming, as most Hadoop language bindings.

Overview of features:

- always a mapper, maybe reducer and combiner
- mapper, reducer and combiner can be implemented as
  - simple method
  - class with `__init__`, `__call__` and possibly `cleanup`. Note that both `__call__` and `cleanup` must return either a list or a generator returning the results. Especially, if you do not want to return anything, you have to return `[]` (that is Dumbo bug as it could be fixes easily).
- parameters through `self.params` and `-param`
- passing files using `-file` or `-cachefile`
- counters
- multiple iterations with any non-circular dependencies

Simple Grep Example

All mentioned examples are available in `/home/straka/examples`.

`grep_ab.py`

```
def mapper(key, value):
  if key.startswith("Ab"):
    yield key, value.replace("\n", " ")

if __name__ == "__main__":
  import dumbo
  dumbo.run(mapper)
```

grep_ab.sh

```
dumbo start grep_ab.py -input /data/wiki/cs-medium -output
   /users/$USER/grep_ab -outputformat text -overwrite yes
```

Running Dumbo

Run above script using `dumbo start script.py options`. Available options:

- `-input input_HDFS_path`: input path to use
- `-inputformat [auto|sequencefile|text|keyvaluetext]`: input format to use, `auto` is default
- `-output output_HDFS_path`: output path to use
- `-output [sequencefile|text]`: output format to use, `sequencefile` is default
- `-nummaptasks n`: set number of map tasks to given number
- `-numreducetasks n`: set number of reduce tasks to given number. Zero is allowed (and default if no reducer is specified) and only mappers are executed in that case.
- `-file local_file`: file to be put in the dir where the python program gets executed
- `-cachefile HDFS_path#link_name`: create a link `link_name` in the dir where the python program gets executed
- `-param name=value`: param available in the Python script as `self.params["name"]`
- `-hadoop hadoop_prefix`: default is `/HADOOP`
- `-name hadoop_job_name`: default is `script.py`
- `-mapper hadoop_mapper`: Java class to use as mapper instead of mapper in script.py
- `-reducer hadoop_reducer`: Java class to use as reducer instead of reducer in script.py

Dumbo HDFS Commands

- `dumbo cat HDFS_path [-ascode=yes]`: convert to text and print given file
- `dumbo ls HDFS_path`
- `dumbo exists HDFS_path`
- `dumbo rm HDFS_path`
- `dumbo put local_path HDFS_path`
- `dumbo get HDFS_path local_path`

Grep Example

Parameters can be passed to mappers and reducers using `-param name=value` Dumbo option and accessed using `self.params` dictionary. Also note the class version of the mapper, using constructor `__init__` and mapper method `__call__`. Reducers can be implemented similarly.

grep.py

```
import re

class Mapper:
  def __init__(self):
    self.re = re.compile(self.params.get("pattern", ""))
```

```
  def __call__(self, key, value):
    if self.re.search(key):
      yield key, value.replace("\n", " ")

if __name__ == "__main__":
  import dumbo
  dumbo.run(Mapper)
```

grep.sh

```
dumbo start grep.py -input /data/wiki/cs-medium -output /users/$USER/grep
   -outputformat text -param pattern="^H" -overwrite yes
```

Simple Word Count

Reducers are similar to mappers, and can be specified also either using a method or a class. An optional combiner (third parameter of `dumbo.run`) can be specified too.

wordcount.py

```
from nltk.tokenize import wordpunct_tokenize

def mapper(key, value):
  for word in wordpunct_tokenize(value.decode('utf-8')):
    yield word, 1

def reducer(key, values):
  yield key, sum(values)

if __name__ == "__main__":
  import dumbo
  dumbo.run(mapper, reducer, reducer)
```

wordcount.sh

```
dumbo start wordcount.py -input /data/wiki/cs-medium -output
   /users/$USER/wordcount -outputformat text -overwrite yes
```

Efficient Word Count Example

More efficient word count is obtained when counts in the processed block are stored in an associative array (more efficient version of local reducer). To that end, `cleanup` method can be used nicely.

Note that we have to return `[]` in `__call__`. That is causes by the fact that Dumbo iterates over results of a `__call__` invocation and unfortunately does not handle `None` return value.

wc_effective.py

```
from nltk.tokenize import wordpunct_tokenize

class Mapper:
  def __init__(self):
    self.counts = {}
  def __call__(self, key, value):
    for word in wordpunct_tokenize(value.decode('utf-8')):
      self.counts[word] = self.counts.get(word, 0) + 1
    return [] # Method __call__ has to return the (key, value) pairs.
              # Unfortunately, NoneType is not handled in Dumbo.
```

```
  def cleanup(self):
    for word_count in self.counts.iteritems():
      yield word_count

class Reducer:
  def __call__(self, key, values):
    yield key, sum(values)

if __name__ == "__main__":
  import dumbo
  dumbo.run(Mapper, Reducer)
```

wc_effective.sh

```
dumbo start wc_effective.py -input /data/wiki/cs-medium -output
   /users/$USER/wc_effective -outputformat text -overwrite yes
```

Word Count with Counters

User counters can be collected using Hadoop using `self.counters` object.

wc_counters.py

```
from nltk.tokenize import wordpunct_tokenize

def mapper(key, value):
  for word in wordpunct_tokenize(value.decode('utf-8')):
    yield word, 1

class Reducer:
  def __call__(self, key, values):
    total = sum(values)
    counter = "Key occurrences " + (str(total) if total < 10 else "10 or more")
    self.counters[counter] += 1
    yield key, total

if __name__ == "__main__":
  import dumbo
  dumbo.run(mapper, Reducer)
```

wc_counters.sh

```
dumbo start wc_counters.py -input /data/wiki/cs-medium -output
   /users/$USER/wc_counters -outputformat text -overwrite yes
```

Word Count using Stop List

Sometimes customization using `-param` is not enough, instead a whole file should be used to customize the mapper or reducer. Consider for example case where word count should ignore given words. This task can be solved by using `-param` to specify file with words to ignore and by `-file` or `-cachefile` to distribute the file in question with the computation.

wc_excludes.py

```
from nltk.tokenize import wordpunct_tokenize

class Mapper:
  def __init__(self):
```

```
    file = open(self.params["excludes"],"r")
    self.excludes = set(line.strip() for line in file)
    file.close()

  def __call__(self, key, value):
    for word in wordpunct_tokenize(value.decode('utf-8')):
      if not (word in self.excludes):
        yield word, 1

def reducer(key, values):
  yield key, sum(values)

if __name__ == "__main__":
  import dumbo
  dumbo.run(Mapper, reducer, reducer)
```

wc_excludes.sh

```
dumbo start wc_excludes.py -input /data/wiki/cs-medium -output
   /users/$USER/wc_excludes -outputformat text -param excludes=stoplist.txt
   -file stoplist.txt -overwrite yes
```

Multiple Iterations Word Count

Dumbo can execute multiple iterations of MapReduce. In the following artifical example, we first create lower case variant of values and then filter out words not matching given pattern and count their occurrences.

wc_2iterations.py

```
from nltk.tokenize import wordpunct_tokenize
import re

class LowercaseMapper:
  def __call__(self, key, value):
    yield key, value.decode('utf-8').lower().encode('utf-8')

class GrepMapper:
  def __init__(self):
    self.re = re.compile(self.params.get("pattern", ""))

  def __call__(self, key, value):
    for word in wordpunct_tokenize(value.decode('utf-8')):
      if self.re.search(word):
        yield word, 1

def reducer(key, values):
  yield key, sum(values)

def runner(job):
  job.additer(LowercaseMapper)
  job.additer(GrepMapper, reducer)

if __name__ == "__main__":
    import dumbo
    dumbo.main(runner)
```

wc_2iterations.sh

```
dumbo start wc_2iterations.py -input /data/wiki/cs-medium -output
   /users/$USER/wc_2iterations -outputformat text -param pattern=h
   -overwrite yes
```

Non-Trivial Dependencies Between Iterations

The MapReduce iterations can depend on output of arbitrary iterations (as long as the dependencies do not form a cycle, of course). This can be specifies using `input` parameter to `additer` as follows.

wc_dag.py

```
from nltk.tokenize import wordpunct_tokenize
import re

class LowercaseMapper:
  def __call__(self, key, value):
    yield key, value.decode('utf-8').lower().encode('utf-8')

class FilterMapper1:
  def __init__(self):
    self.re = re.compile(self.params.get("pattern1", ""))

  def __call__(self, key, value):
    for word in wordpunct_tokenize(value.decode('utf-8')):
      if self.re.search(word):
        yield word, 1

class FilterMapper2:
  def __init__(self):
    self.re = re.compile(self.params.get("pattern2", ""))

  def __call__(self, key, value):
    for word in wordpunct_tokenize(value.decode('utf-8')):
      if self.re.search(word):
        yield word, 1

class IdentityMapper:
  def __call__(self, key, value):
    yield key, value

def reducer(key, values):
  yield key, sum(values)

def runner(job):
  lowercased = job.additer(LowercaseMapper) # implicit input = job.root
  filtered1 = job.additer(FilterMapper1, input = lowercased)
  filtered2 = job.additer(FilterMapper2, input = lowercased)
  job.additer(IdentityMapper, reducer, input = [filtered1, filtered2])

if __name__ == "__main__":
    import dumbo
    dumbo.main(runner)
```

wc_dag.sh

```
dumbo start wc_dag.py -input /data/wiki/cs-medium -output
   /users/$USER/wc_dag -outputformat text -param pattern1=h -param
   pattern2=i -overwrite yes
```

Execute Hadoop Locally

Using `dumbo-local` instead of `dumbo`, you can run the Hadoop computation locally using one mapper and one reducer. The standard error of the Python script is available in that case.

Wikipedia Data

The Wikipedia Data available from `/dlrc_share/data/wiki/` are available also in HDFS:

- `/data/wiki/cs`: Czech Wikipedia data (Sep 2009), 195MB, 124k articles
- `/data/wiki/en`: English Wikipedia data (Sep 2009), 4.9GB, 2.9M articles

All data is stored in a record-compressed sequence files, with article names as keys and article texts as values, in UTF-8 encoding.

Tasks

Solve the following tasks. Solution for each task is a Dumbo Python source processing the Wikipedia source data and producing required results.

Tasks From Previous Seminars

Task	*Points*	*Description*
`dumbo_unique_words`	2	Create a list of unique words used in the articles using Dumbo. Convert them to lowercase to ignore case. Use the `wordpunct_tokenize` as a tokenizer.
`article_initials`	2	Run a Dumbo job which uses counters to count the number of articles according to their first letter, ignoring the case and merging all non-Czech initials.
`dumbo_inverted_index`	2	Compute inverted index in Dumbo – for every lowercased word from the articles, compute *(article name, ascending positions of occurrences as word indices)* pairs. Use the `wordpunct_tokenize` as a tokenizer. The output should be a file with one word on a line in the following format: `word \t articleName \t spaceSeparatedOccurrences...` You will get 2 additional points if the articles will be numbered using consecutive integers. In that case, the output is ascending *(article id, ascending positions of occurrences as word indices)* pairs, together with a file containing list of articles representing this mapping (the article on line i is the article with id i).
`no_references`	3	An article A is said to reference article B, if it contains B as a token (ignoring case). Run a Dumbo job which counts for each article how many references there exists for the given article (summing all references in a single article). You will get one extra point if the result is sorted by the number of references (you are allowed to use 1 reducer in the sorting phase). Use the `wordpunct_tokenize` as a tokenizer.

Tasks From Previous Seminars

Task	*Points*	*Description*
`dumbo_wordsim_index`	4	In order to implement word similarity search, compute for each form with at least three occurrences all *contexts* in which it occurs, including their number of occurrences. List the contexts in ascending order. Given N (either 1, 2, 3 or 4), the *context* of a form occurrence is N forms preceding this occurrence and N forms following this occurrence (ignore sentence boundaries, use empty words when article boundaries are reached). Use the `wordpunct_tokenize` as a tokenizer. The output should be a file with one form on a line in the following format: `form \t context \t counts...`
`dumbo_wordsim_find`	4	Let S be given natural number. Using the index created in `dumbo_wordsim_index`, find for each form S most similar forms. The similarity of two forms is computed using *cosine similarity* as $\frac{C_A \cdot C_B}{\lvert C_A \rvert \cdot \lvert C_B \rvert}$, where C_F is a vector of occurrences of form F contexts. The output should be a file with one form on a line in the following format: `form \t most similar form \t cosine similarity...`

New Tasks

Task	*Points*	*Description*
`dumbo_anagrams`	2	Two words are anagrams if one is a letter permutation of the other (ignoring case). For a given input, find all anagram classes that contain at least A words. Output each anagram class on a separate line. Use the `wordpunct_tokenize` as a tokenizer.
`dumbo_kmeans`	6	Implement K-means clustering algorithm as described on `http://en.wikipedia.org/wiki/K-means_clustering#Standard_algorithm`. The user specifies number of iterations and the program run specified number of K-means clustering algorithm iterations. You can use the following training data. The key of the data is point ID, the value of the data is list of integral coordinates. The coordinates in one input naturally have the same dimension. *HDFS path* | *Points* | *Dimension* | *Clusters* `/data/points/small` | 10000 | 50 | 50 `/data/points/medium` | 100000 | 100 | 100 `/data/points/large` | 500000 | 200 | 200 You will get 2 additional points if the algorithm stops when none of the centroid positions change more than given ε.