Data Intensive Computing – Handout 10

Spark

According to homepage:

Apache Spark is a fast and general engine for large-scale data processing.

Spark is available either in Scala (and therefore in Java) or in Python.

Python Example

`spark/wordcount.py`

```
from nltk.tokenize import wordpunct_tokenize
from pyspark import SparkConf, SparkContext
import sys

if len(sys.argv) < 4:
    print >>sys.stderr, "Usage: %s master input output" % sys.argv[0]
    exit(1)

sc = SparkContext(sys.argv[1], "Word count")
file = sc.textFile(sys.argv[2])
counts = file.flatMap(lambda x: wordpunct_tokenize(x)) \
             .map(lambda x: (x, 1)) \
             .reduceByKey(lambda x,y: x + y)
counts.map(lambda (x,y) : "%s\t%d" % (x, y)) \
      .saveAsTextFile(sys.argv[3])
```

The example is available in `/home/straka/spark/examples`*.*

Running locally using single thread:

- `pyspark wordcount.py local /dlrc_share/data/wiki/cs-text-small wc-output`

Running locally using 4 threads:

- `pyspark wordcount.py "local[4]" /dlrc_share/data/wiki/cs-text-small wc-output`

Running on cluster:

- `pyspark wordcount.py spark://dlrc-headnode:7077 \`
  `hdfs://dlrc-headnode/data/wiki-txt/cs hdfs://dlrc-headnode/users/straka/wc-output`
- by default, gets 4 cores
- by default, reduces into 2 parts

Monitoring Job Status

- `dlrc-headnode:8080` interface of the cluster
- `dlrc-headnode:4040` interface of running application (4041, 4042 if multiple, see job output or go through the cluster interface
- 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`.

Running Python Jobs

- `pyspark file`: run given file
- `ipyspark-local`: run interpreter using one local thread
- `ipyspark-local-n 3`: run interpreter using three local threads
- `ipyspark-cluster`: run interpreter using the cluster

Scala

Compiled language compatible with Java, statically typed, object-functional language.

Running Scala Interpreter

- `spark-scala-local`: run interpreter using one local thread
- `spark-scala-local-n 3`: run interpreter using three local threads
- `spark-scala-cluster`: run interpreter using the cluster

Running Scala Programs

- `spark-scalac file.scala`: compile the given file into jar
- `spark-jar file.jar class`: run given class in supplied jar file

Scala Example

spark/wordcount.scala

```scala
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object Main {
  def main(args: Array[String]) {
    if (args.length <3) sys.error("Usage: master input output")

    val sc = new SparkContext(args(0), "Word count",
        System.getenv("SPARK_HOME"),
        SparkContext.jarOfClass(this.getClass).toSeq)
    val file = sc.textFile(args(1))
    val counts = file.flatMap(line => line.split(" "))
                     .map(word => (word, 1))
                     .reduceByKey(_ + _)
    counts.saveAsTextFile(args(2))
  }
}
```

The example is available in `/home/straka/spark/examples`.

PySpark Reference

Available online on `http://spark.apache.org/docs/latest/api/pyspark/index.html`.

`SparkContext`

- `__init__(self, master=None, appName=None, sparkHome=None, pyFiles=None, environment=None, batchSize=1024, serializer=PickleSerializer(), conf=None)`: Create a new SparkContext.
- `accumulator(self, value, accum_param=None)`: Create an Accumulator with the given initial value, using a given AccumulatorParam helper object to define how to add values of the data type if provided.
- `addFile(self, path)`: Add a file to be downloaded with this Spark job on every node.
- `addPyFile(self, path)`: Add a .py or .zip dependency for all tasks to be executed on this SparkContext in the future.
- `broadcast(self, value)`: Broadcast a read-only variable to the cluster, returning a Broadcast object for reading it in distributed functions.
- `clearFiles(self)`: Clear the job's list of files added by addFile or addPyFile so that they do not get downloaded to any new nodes.
- `defaultParallelism(self)`: Default level of parallelism to use when not given by user (e.g. for reduce tasks)
- `parallelize(self, c, numSlices=None)`: Distribute a local Python collection to form an RDD.
- `setCheckpointDir(self, dirName)`: Set the directory under which RDDs are going to be checkpointed.
- `textFile(self, name, minSplits=None)`: Read a text file from HDFS, a local file system (available on all nodes), or any Hadoop-supported file system URI, and return it as an RDD of Strings.
- `union(self, rdds)`: Build the union of a list of RDDs.
- `__del__(self)`, `stop(self)`: Shut down the SparkContext.

`SparkConf`

- `__init__(self, loadDefaults=True, _jvm=None)`: Create a new Spark configuration.
- `contains(self, key)`: Does this configuration contain a given key?
- `get(self, key, defaultValue=None)`: Get the configured value for some key, or return a default otherwise.
- `getAll(self)`: Get all values as a list of key-value pairs.
- `set(self, key, value)`: Set a configuration property.
  - `spark.cores.max`: Ask for this amount of cores (default 4).
  - `spark.default.parallelism`: Default number of partitions created by shuffle operations (`groupByKey`, `reduceByKey`, etc).
  - `spark.executor.memory`: Ask for this amount of memory per core (default 512m).
- `setAll(self, pairs)`: Set multiple parameters, passed as a list of key-value pairs.

- `setAppName(self, value)`: Set application name.
- `setExecutorEnv(self, key=None, value=None, pairs=None)`: Set an environment variable to be passed to executors.
- `setMaster(self, value)`: Set master URL to connect to.
- `setSparkHome(self, value)`: Set path where Spark is installed on worker nodes.
- `toDebugString(self)`: Returns a printable version of the configuration, as a list of key=value pairs, one per line.

`RDD`

- `cache(self)`: Persist this RDD with the storage level `MEMORY_ONLY`.
- `cartesian(self, other)`: Return the Cartesian product of this RDD and another one, that is, the RDD of all pairs of elements (a, b) where a is in self and b is in other.
- `checkpoint(self)`: Mark this RDD for checkpointing.
- `coalesce(self, numPartitions, shuffle=False)`: Return a new RDD that is reduced into 'numPartitions' partitions.
- `cogroup(self, other, numPartitions=None)`: For each key k in self or other, return a resulting RDD that contains a tuple with the list of values for that key in self as well as other.
- `collect(self)`: Return a list that contains all of the elements in this RDD.
- `collectAsMap(self)`: Return the key-value pairs in this RDD to the master as a dictionary.
- `combineByKey(self, createCombiner, mergeValue, mergeCombiners, numPartitions=None)`: Generic function to combine the elements for each key using a custom set of aggregation functions.
- `context(self)`: The SparkContext that this RDD was created on.
- `count(self)`: Return the number of elements in this RDD.
- `countByKey(self)`: Count the number of elements for each key, and return the result to the master as a dictionary.
- `countByValue(self)`: Return the count of each unique value in this RDD as a dictionary of (value, count) pairs.
- `distinct(self)`: Return a new RDD containing the distinct elements in this RDD.
- `filter(self, f)`: Return a new RDD containing only the elements that satisfy a predicate.
- `first(self)`: Return the first element in this RDD.
- `flatMap(self, f, preservesPartitioning=False)`: Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.
- `flatMapValues(self, f)`: Pass each value in the key-value pair RDD through a flatMap function without changing the keys; this also retains the original RDD's partitioning.
- `fold(self, zeroValue, op)`: Aggregate the elements of each partition, and then the results for all the partitions, using a given associative function and a neutral "zero value."

- `foldByKey(self, zeroValue, func, numPartitions=None)`: Merge the values for each key using an associative function "func" and a neutral "zeroValue" which may be added to the result an arbitrary number of times, and must not change the result (e.g., 0 for addition, or 1 for multiplication.).
- `foreach(self, f)`: Applies a function to all elements of this RDD.
- `getCheckpointFile(self)`: Gets the name of the file to which this RDD was checkpointed
- `getStorageLevel(self)`: Get the RDD's current storage level.
- `glom(self)`: Return an RDD created by coalescing all elements within each partition into a list.
- `groupBy(self, f, numPartitions=None)`: Return an RDD of grouped items.
- `groupByKey(self, numPartitions=None)`: Group the values for each key in the RDD into a single sequence.
- `groupWith(self, other)`: Alias for cogroup.
- `isCheckpointed(self)`: Return whether this RDD has been checkpointed or not
- `join(self, other, numPartitions=None)`: Return an RDD containing all pairs of elements with matching keys in self and other.
- `keyBy(self, f)`: Creates tuples of the elements in this RDD by applying f.
- `leftOuterJoin(self, other, numPartitions=None)`: Perform a left outer join of self and other.
- `map(self, f, preservesPartitioning=False)`: Return a new RDD by applying a function to each element of this RDD.
- `mapPartitions(self, f, preservesPartitioning=False)`: Return a new RDD by applying a function to each partition of this RDD.
- `mapPartitionsWithIndex(self, f, preservesPartitioning=False)`: Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.
- `mapPartitionsWithSplit(self, f, preservesPartitioning=False)`: Deprecated: use mapPartitionsWithIndex instead.
- `mapValues(self, f)`: Pass each value in the key-value pair RDD through a map function without changing the keys; this also retains the original RDD's partitioning.
- `mean(self)`: Compute the mean of this RDD's elements.
- `name(self)`: Return the name of this RDD.
- `partitionBy(self, numPartitions, partitionFunc=hash)`: Return a copy of the RDD partitioned using the specified partitioner.
- `persist(self, storageLevel)`: Set this RDD's storage level to persist its values across operations after the first time it is computed.
- `pipe(self, command, env={})`: Return an RDD created by piping elements to a forked external process.
- `reduce(self, f)`: Reduces the elements of this RDD using the specified commutative and associative binary operator.
- `reduceByKey(self, func, numPartitions=None)`: Merge the values for each key using an associative reduce function.

- `reduceByKeyLocally(self, func)`: Merge the values for each key using an associative reduce function, but return the results immediately to the master as a dictionary.
- `repartition(self, numPartitions)`: Return a new RDD that has exactly numPartitions partitions.
- `rightOuterJoin(self, other, numPartitions=None)`: Perform a right outer join of self and other.
- `sample(self, withReplacement, fraction, seed)`: Return a sampled subset of this RDD (relies on numpy and falls back on default random generator if numpy is unavailable).
- `sampleStdev(self)`: Compute the sample standard deviation of this RDD's elements (which corrects for bias in estimating the standard deviation by dividing by N-1 instead of N).
- `sampleVariance(self)`: Compute the sample variance of this RDD's elements (which corrects for bias in estimating the variance by dividing by N-1 instead of N).
- `saveAsTextFile(self, path)`: Save this RDD as a text file, using string representations of elements.
- `setName(self, name)`: Assign a name to this RDD.
- `sortByKey(self, ascending=True, numPartitions=None, keyfunc=lambda x: x)`: Sorts this RDD, which is assumed to consist of (key, value) pairs.
- `stats(self)`: Return a StatCounter object that captures the mean, variance and count of the RDD's elements in one operation.
- `stdev(self)`: Compute the standard deviation of this RDD's elements.
- `subtract(self, other, numPartitions=None)`: Return each value in self that is not contained in other.
- `subtractByKey(self, other, numPartitions=None)`: Return each (key, value) pair in self that has no pair with matching key in other.
- `sum(self)`: Add up the elements in this RDD.
- `take(self, num)`: Take the first num elements of the RDD.
- `takeOrdered(self, num, key=None)`: Get the N elements from a RDD ordered in ascending order or as specified by the optional key function.
- `takeSample(self, withReplacement, num, seed)`: Return a fixed-size sampled subset of this RDD (currently requires numpy).
- `toDebugString(self)`: A description of this RDD and its recursive dependencies for debugging.
- `top(self, num)`: Get the top N elements from a RDD.
- `union(self, other)`: Return the union of this RDD and another one.
- `unpersist(self)`: Mark the RDD as non-persistent, and remove all blocks for it from memory and disk.
- `variance(self)`: Compute the variance of this RDD's elements.
- `zip(self, other)`: Zips this RDD with another one, returning key-value pairs with the first element in each RDD second element in each RDD, etc.
- `__add__(self, other)`: Return the union of this RDD and another one.

StatCounter

- `copy(self)`
- `count(self)`
- `mean(self)`
- `merge(self, value)`
- `mergeStats(self, other)`
- `sampleStdev(self)`
- `sampleVariance(self)`
- `stdev(self)`
- `sum(self)`
- `variance(self)`

StorageLevel

- `DISK_ONLY`
- `DISK_ONLY_2`
- `MEMORY_AND_DISK`
- `MEMORY_AND_DISK_2`
- `MEMORY_AND_DISK_SER`
- `MEMORY_AND_DISK_SER_2`
- `MEMORY_ONLY`
- `MEMORY_ONLY_2`
- `MEMORY_ONLY_SER`
- `MEMORY_ONLY_SER_2`

BroadCast

- `value(self)`: Value of broadcasted variable.

Accumulator

- `add(self, term)`: Adds a term to this accumulator's value
- `value(self, value)`: Sets the accumulator's value; only usable in driver program
- `__iadd__(self, term)`: The += operator; adds a term to this accumulator's value
- `__repr__(self)`: repr(x)
- `__str__(self)`: str(x)

SparkFiles

- `get(cls, filename)`: Get the absolute path of a file added through `SparkContext.addFile()`.
- `getRootDirectory(cls)`: Get the root directory that contains files added through `SparkContext.ad`

PageRank Example

spark/pagerank.py

```
import re, sys
from operator import add
from pyspark import SparkContext

def computeContribs(urls, rank):
    """Calculates URL contributions to the rank of other URLs."""
    num_urls = len(urls)
    for url in urls: yield (url, rank / num_urls)

def parseNeighbors(urls):
    """Parses a urls pair string into urls pair."""
    parts = re.split(r'\s+', urls)
    return parts[0], parts[1]

if len(sys.argv) < 3:
    print >> sys.stderr, "Usage: pagerank <master> <file> <number_of_iterations>"
    exit(1)

# Initialize the spark context.
sc = SparkContext(sys.argv[1], "PythonPageRank")

# Loads in input file. It should be in format of:
# URL neighbor URL
# ...
lines = sc.textFile(sys.argv[2], 1)

# Loads all URLs from input file and initialize their neighbors.
links = lines.map(lambda urls: parseNeighbors(urls))
             .distinct().groupByKey().cache()

# Initialize rank of all URLS to one.
ranks = links.map(lambda (url, neighbors): (url, 1.0))

# Calculates and updates URL ranks continuously using PageRank algorithm.
for iteration in xrange(int(sys.argv[3])):
    # Calculates URL contributions to the rank of other URLs.
    contribs = links.join(ranks).flatMap(lambda (url, (urls, rank)):
        computeContribs(urls, rank))

    # Re-calculates URL ranks based on neighbor contributions.
    ranks = contribs.reduceByKey(add).mapValues(lambda rank: rank * 0.85 + 0.15)

# Collects all URL ranks and dump them to console.
for (link, rank) in ranks.collect():
    print "%s has rank: %s." % (link, rank)
```

The example is available in `/home/straka/spark/examples`.

Tasks

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

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
```

Wikipedia Data

The textual Wikipedia Data are avilable in HDFS:

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

All data are encoded in UTF-8 and contain one particle per line. Article name is separated by `\t` character from the article content.

Task	*Points*	*Description*
`spark_unique_words`	2	Create a list of unique words used in the articles using Spark. Convert them to lowercase to ignore case.
`spark_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.
`spark_sort`	3	You are given data consisting of (31-bit integer, string data) pairs. These are available in plain text format: • `/data/numbers-txt/numbers-small`: 3MB • `/data/numbers-txt/numbers-medium`: 184MB • `/data/numbers-txt/numbers-large`: 916MB You can assume that the integers are uniformly distributed. Your task is to sort these data, comparing the key numerically and not lexicographically. The lines in the output must be the same as in the input, only in different order. Your solution should work for TBs of data. For that reason, you must use multiple machines. If your job is executed using m machines, the output consists of m files, which when concatenated would produce sorted (key, value) pairs. In other words, each of the output files contains sorted (integer, data) pairs and all keys in one file are either smaller or larger than in other file. Your solution should work for any number of machines specified.

Task	*Points*	*Description*
`spark_nonuniform_sort`	4	Improve the `spark_sort` to handle nonuniform data. You can use the following exponentially distributed data: • `/data/numbers-txt/nonuniform-small`: 3MB • `/data/numbers-txt/nonuniform-medium`: 160MB • `/data/numbers-txt/nonuniform-large`: 797MB Assume we want to produce m output files. One of the solutions is the following: • Go through the data and sample only a small fraction of the keys. • Find best $m-1$ separators using the sampled data. • Run the second pass using the computed separators.
`spark_inverted_index`	2	Compute inverted index in Spark – for every lowercased word from the articles, compute *(article name, ascending positions of occurrences as word indices)* pairs. 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).
`spark_no_references`	3	An article A is said to reference article B, if it contains B as a token (ignoring case). Run a Spark job which for each article B counts how many references for article B there exist in the whole wiki (summing references in a single article). You will get one extra point if the result is sorted by the number of references.
`spark_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). The output should be a file with one form on a line in the following format: `form \t context \t counts...`

<table>
<tr><th>Task</th><th>Points</th><th>Description</th></tr>
<tr><td><code>spark_wordsim_find</code></td><td>4</td><td>Let S be given natural number. Using the index created in <code>spark_wordsim_index</code>, find for each form S most similar forms. The similarity of two forms is computed using cosine similarity as $\frac{C_A \cdot C_B}{|C_A| \cdot |C_B|}$, 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:
<code>form \t most similar form \t cosine similarity...</code></td></tr>
<tr><td><code>spark_kmeans</code></td><td>6</td><td>Implement K-means clustering algorithm as described on <code>http://en.wikipedia.org/wiki/K-means_clustering#Standard_algorithm</code>.
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. Each line contains space separated coordinates of one points. The coordinates in one input naturally have the same dimension.
<table>
<tr><th>HDFS path</th><th>Points</th><th>Dimension</th><th>Clusters</th></tr>
<tr><td><code>/data/points-txt/small</code></td><td>10000</td><td>50</td><td>50</td></tr>
<tr><td><code>/data/points-txt/medium</code></td><td>100000</td><td>100</td><td>100</td></tr>
<tr><td><code>/data/points-txt/large</code></td><td>500000</td><td>200</td><td>200</td></tr>
</table>
You will get 2 additional points if the algorithm stops when none of the centroid positions change more than given ε.</td></tr>
</table>