Data Intensive Computing – Handout 2

Training Cluster

You can login to the training cluster *dlrc* via machine `ufallab.ms.mff.cuni.cz` and port 11422, i.e., using `ssh -p 11422 ufallab.ms.mff.cuni.cz`.

All machines in the cluster use same architecture and share `/dlrc_share` directory.

The cluster consists of master `dlrc` and 5 nodes `dlrc-node1` to `dlrc-node5`. Each node has 2 cores and 4GB ram, but four SGE jobs are allowed to run simultaneously (only because otherwise there would be too little slots; the slot number may be even increased later).

Running Browser with Proxy to the Cluster

To run a browser, which can access master *and* the cluster nodes, run:

```
(chromium --proxy-server=socks://localhost:2020& \
ssh -ND 2020 -p 11422 ufallab.ms.mff.cuni.cz)
```

HDFS Distributed Filesystem

The distributed filesystem HDFS is installed on the cluster.

You can browse the filesystem using `dlrc:50070`.

The HDFS filesystem can be manipulated using `hadoop fs` command, which you can shorten to `dfs`. It prints nice usage when executed without arguments, some simple commands are:

- `dfs -ls /`
- `dfs -mkdir /home/straka/test`
- `dfs -put .bashrc /home/straka/test`
- `dfs -get /home/straka/test .`
- `dfs -rm -r /home/straka/test`

Every user has directory `/home/username` which they can write to.

Wikipedia Data

Compressed and structured Wikipedia data is available in the HDFS:

- `/data/wiki/cs`: Czech Wikipedia data (Sep 2009), 124k articles.
- `/data/wiki/cs-medium`: Subset of Czech Wikipedia data, 5307 articles.
- `/data/wiki/cs-small`: Subset of Czech Wikipedia data, 13 articles.
- `/data/wiki/en`: English Wikipedia data (Sep 2009), 2.9M articles.

All files are so called *sequence files*, every record is a pair *(article name, article text)*.

Apache Spark

Apache Spark (available from `spark.apache.org`) is a fast and general engine for large-scale data processing, with API available in Scala, Java, Python and R.

Python Example

`wordcount.py`

```python
import sys
if len(sys.argv) < 3:
    print >>sys.stderr, "Usage: %s input output" % sys.argv[0]
    exit(1)
input = sys.argv[1]
output = sys.argv[2]

from pyspark import SparkContext

sc = SparkContext()
(sc.textFile(input, 3*sc.defaultParallelism)
  .flatMap(lambda line: line.split())
  .map(lambda token: (token, 1))
  .reduceByKey(lambda x,y: x + y)
  .sortBy(lambda (word,count): count, ascending=False)
  .saveAsTextFile(output))
```

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

Running locally using all available threads

```
spark-submit wordcount.py /dlrc_share/data/wiki/cs-text-small wc-output
```

Running on the cluster

- `spark-qrsh 2 1G`: Start interactive session on the cluster, with 2 additional Apache Spark workers, each using 1GB of memory.
- `spark-qrsh 2 1G spark-submit wordcount.py \`
  `/dlrc_share/data/wiki/cs-text-medium wc-output`: Start interactive session on the cluster, with 2 additional Apache Spark workers, each using 1GB of memory, run wordcount.py and stop the cluster after it finished.
- `spark-qsub 2 1G spark-submit wordcount.py \`
  `/dlrc_share/data/wiki/cs-text-medium wc-output`: Start the wordcount.py script on the cluster, with 2 additional Apache Spark workers, each using 1GB of memory. The command ends immediately.

Running interactive Python shell on the cluster

- `spark-qrsh 1 1G ipython`: Start interactive Python shell on the cluster, with `sc` variable initialized to the `SparkContext`. One additional Apache Spark worker with 1GB memory is spawned.

Scala Example

wordcount.scala

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

object Main {
  def main(args: Array[String]) {
    if (args.length < 2) sys.error("Usage: input output")
    val (input, output) = (args(0), args(1))

    val sc = new SparkContext()
    sc.textFile(input, 3*sc.defaultParallelism)
      .flatMap(_.split("\\s"))
      .map((_,1)).reduceByKey(_+_)
      .sortBy(_._2, ascending=false)
      .saveAsTextFile(output)
  }
}
```

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

Running interactive Scala shell on the cluster

- `spark-qrsh 1 1G spark-shell`: Start interactive Scala shell on the cluster, with `sc` variable initialized to the `SparkContext`. One additional Apache Spark worker with 1GB memory is spawned.

  You can paste the following code:

  wordcount.core.scala

  ```scala
  (sc.textFile("/dlrc_share/data/wiki/cs-text-small",
      3*sc.defaultParallelism)
     .flatMap(_.split("\\s"))
     .map((_,1)).reduceByKey(_+_)
     .sortBy(_._2, ascending=false)
     .saveAsTextFile("wc-output"))
  ```

Compiling and running on the cluster

To compile `wordcount.scala` to binary JAR file, create an empty directory, copy `wordcount.scala` to the directory, add the following project file:

wordcount.sbt

```
name := "wordcount"
version := "1.0"
scalaVersion := "2.10.5"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.6.1"
```

and run `sbt package`.
You can then execute the binary on the cluster:

-
  ```
  spark-qsub 2 1G spark-submit spark-submit \
    target/scala-2.10/word_count_2.10-1.0.jar /dlrc_share/data/wiki/cs-text-medium wc
  ```
  Start the wordcount.jar binary on the cluster, with 2 additional Apache Spark workers, each using 1GB of memory. The command ends immediately.

Tasks

When needing to split given text into words (called *tokens*), use function `wordpunct_tokenize` from `nltk.tokenize` package.

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. Use HDFS `/data/wiki/{cs,en}` as input and either `/dlrc_share/data/wiki/tokenizer/{cs,en}_tokenizer` or `nltk.tokenize.wordpunct_tokenize` as a tokenizer.
`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 HDFS `/data/wiki/{cs,en}` as input and either `/dlrc_share/data/wiki/tokenizer/{cs,en}_tokenizer` or `nltk.tokenize.wordpunct_tokenize` as a tokenizer.
`sort`	3	You are given data consisting of (31-bit integer, string data) pairs. These are available in plain text format: • `/dlrc_share/data/numbers-txt/numbers-small`: 3MB • `/dlrc_share/data/numbers-txt/numbers-medium`: 184MB • `/dlrc_share/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. Obviously, do not use `sort` nor `sortByKey` method in your solution.

Task	*Points*	*Description*
`nonuniform_sort`	4	Improve the `sort` task to handle nonuniform data. You can use the following exponentially distributed data: • `/dlrc_share/data/numbers-txt/nonuniform-small`: 3MB • `/dlrc_share/data/numbers-txt/nonuniform-medium`: 160MB • `/dlrc_share/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. As there are not so many of them, they can fit in one reducer. • 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 article name \t space separated occurrences`
		`\t article name \t space separated occurences ...`
		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).
		Use HDFS `/data/wiki/{cs,en}` as input and either `/dlrc_share/data/wiki/tokenizer/{cs,en}_tokenizer` or `nltk.tokenize.wordpunct_tokenize` as a tokenizer.

`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). Use HDFS `/data/wiki/{cs,en}` as input and either `/dlrc_share/data/wiki/tokenizer/{cs,en}_tokenizer` or `nltk.tokenize.wordpunct_tokenize` as a tokenizer.

Task	*Points*	*Description*
`spark_wordsim_index`	3	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), 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 \t context \t counts ...` Use HDFS `/data/wiki/{cs,en}` as input and either `/dlrc_share/data/wiki/tokenizer/{cs,en}_tokenizer` or `nltk.tokenize.wordpunct_tokenize` as a tokenizer.
`spark_wordsim_find`	3	Let S be given natural number. Using the index created in `spark_wordsim_index`, find for each form S most similar forms. List only forms with non-zero similarity. 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: `lemma \t most similar lemma \t cosine similarity \t 2. most similar lemma \t cosine similarity ...`
`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. Each line contains space separated coordinates of one points. The coordinates in one input naturally have the same dimension.

Path in `/dlrc_share`	*Points*	*Dimension*	*Clusters*
`/data/points-txt/small`	10000	50	50
`/data/points-txt/medium`	100000	100	100
`/data/points-txt/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.