Module 4

MapReduce and Apache Hadoop

4.1 Motivations for using MapReduce Paradigm

Recall words of Clive Humby "Data is the new oil" [1]. Oil resources are different in size, quality and localization. Drilling for it requires all spectrum of continuously evolving techniques. The same is with data – with increasing size and complexity of data more advanced techniques and resources have to be used. Storing data has to shift from CSV files to databases on a single machine or distributed system. To process data in distributed system many software and hardware issues have to be addressed.

4.2 MapReduce Overview

In the "single-person scale" one has a laptop with a file system, some database and a single- or multi-thread way to process data. The "Google scale" is different, but has similar building bricks, which are Google File System (distributed file system) [5], Bigtable (distributed database) [2] and MapReduce (distributed computation system) [3].

The MapReduce Paradigm [3] comes as the consequence of the observation that majority of calculations in Google around year 2004 can be described by two steps - Map and Reduce. In the former one a group of mappers receive $\langle key, value \rangle$ pairs, transform them and eventually yield zero or more new $\langle key, value \rangle$ pairs. Subsequently, all pairs generated by mappers are grouped by a key and passed to a group of reducers. Each reducer receive pairs, where key is associated with a collection of values, which come from mappers. Each reducer can create zero or more results.

Good illustration of data processing on map and reduce side is "Word-Count" Example, where one wants to calculate the total number of all words in all lines in all provided files. A mapper gets pairs, where a value contains words of a given line, tokenizes the line and for each token a new pair, say $\langle word, "1" \rangle$ is emitted. A reducer receives as a key and a collection of values associated with the key. Then it sums values from the received collection and produces a single $\langle word, sum\_of\_occurrences \rangle$ pair. Consult pseudo-code in Figure 4.1 and general view on work of mappers and reducers in Figure 4.2.

map(k_1,v_1) → **list**(k_2,v_2)

```
map(String key; String value):
// key: document name
// value: document text
for each word w in value:
  EmitIntermediate(w, "1");
```

.

reduce(k_2,**list**(v_2)) → **list**(k_3,v_3)

```
reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
  result += ParseInt(v);
Emit(key, AsString(result));
```

Figure 4.1: Mapper and Reducer pseudo-code for the "Word-count" example.

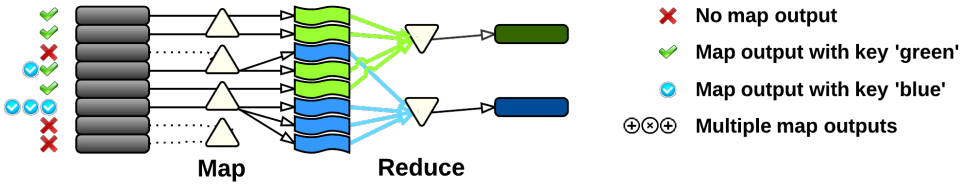

Figure 4.2: The general schema of mappers and reducers work.

Dean and Ghemawat described a few examples of MapReduce usage, which are Distributed Grep, Count of URL Access Frequency, Creation of Reverse Web-Link Graph and a few more.

Many algorithms can be expressed in MapReduce Paradigm very well. Some of them due to a network traffic are suitable to implement, some should be expressed in a modified way. Algorithms such as Logistic Regression, Naïve Bayes and Random Forest for classification are suitable examples already implemented in Apache Mahout [10]. Other algorithms suitable for MapReduce paradigm are Singular Value Decomposition, Latent Dirichlet Allocation and Latent Semantic Analysis also available in Apache Mahout. Interestingly, in April 2014 Apache Mahout developers team announced that the library will be switched to support Apache Spark and abandon any new implementations of MapReduce algorithms.

Algorithms which are less suitable for MapReduce Paradigm are graphic-al or generally algorithms which require many iterations, especially when the number of iterations is large or undetermined. Such algorithms, like PageRank can be implemented more efficient using other approaches which employ some elements of ecosystems based on MapReduce. The example of such are Google Pregel [8] and Apache Giraph [6].

Jimmy Lin in his article from 2012 [7] highlighted that MapReduce Paradigm, comprised to a "hammer", is not suitable for many algorithms, which are not "nails", however in the absence of the perfect tool he encouraged to throw away everything, which is not a nail.

Let's investigate implementations of a MapReduce "hammer" and see what other options are in a toolbox.

4.3 Popular implementations of MapReduce

4.3.1 Google MapReduce (2004)

In Google MapReduce a user has to implement program with map and reduce functions. The general view on the MapReduce computation model is presented in Figure 4.1. After starting calculations on a cluster a program is forked multiple times and one of its forks is elected for the

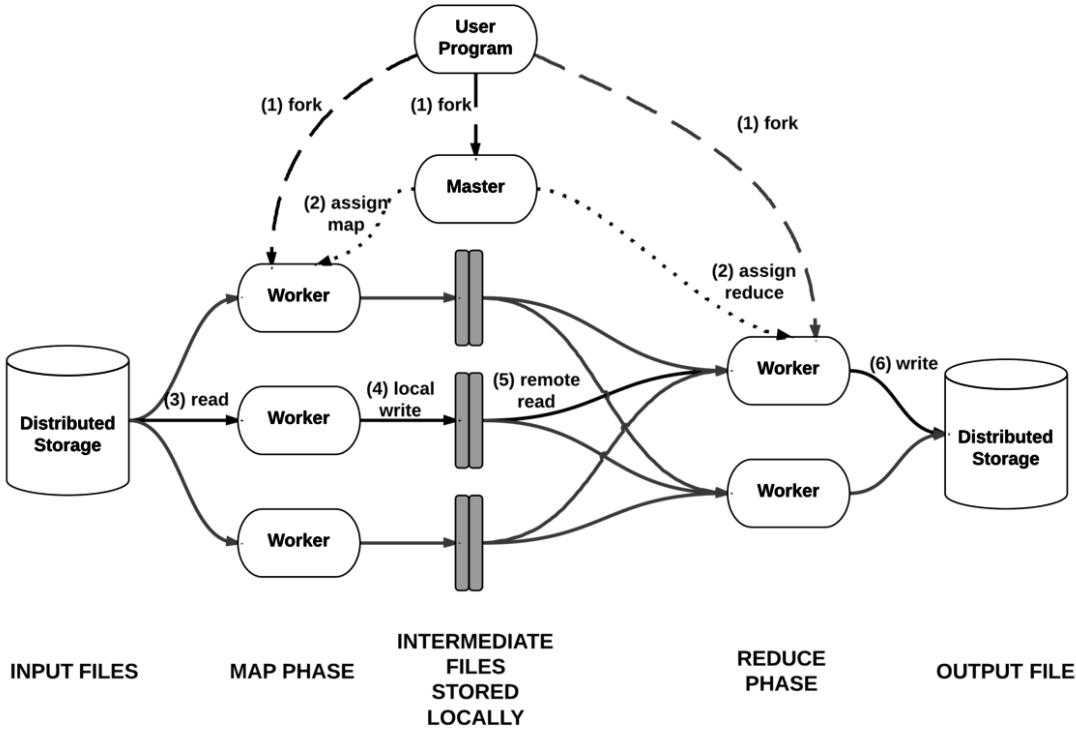

Figure 4.3: General overview on computation model in Google MapReduce. The figure is taken from [5]

Master role. The role of the Master is to orchestrate work of other processes and as the first task it divides processes into mappers and reducers. Map and reduce functions (or just a map function, with no reduction part) are a basic description of a single Job.

Mappers read data record by record, transform them as described earlier and store results locally. Master knows what is localization of records with the same key and passes this information to reducers, which fetch data from mappers. After processing data reducers store output in a distributed storage. The master monitors health of programs during execution of a job. In case of a mapper failure its calculations are reassigned to another mapper, so as in case of a reducer failure. This implicitly imply that results of mappers are available until reducers successfully finish their work, to ensure unproblematic takeover of reducer calculations.

Many enhancements have been added to the process, like combining output from mappers on the local machine (mini-reducer), which is needed for reducing network traffic. For example in the Word Count one can sum all occurrences of a word in a line.

As the result a developer has only to deliver a few functions and all issues with parametrization are resolved by the system.

4.3.2 Apache Hadoop

Apache Hadoop [11] is the open-source Java implementation of Google MapReduce (2004). As of now there is Hadoop MapReduce version 1 and 2. For simplicity in this course only Hadoop MapReduce v1 is considered.

Apache Hadoop differs from Google MapReduce (2004) in a few aspects. For example the number of map and reduce slots, read from a cluster properties file, are set during start of the system and can not be changed when system is working. Apache Hadoop Ecosystem comes has it's solutions corresponding to Google's, namely Hadoop Distributed File System [9] and

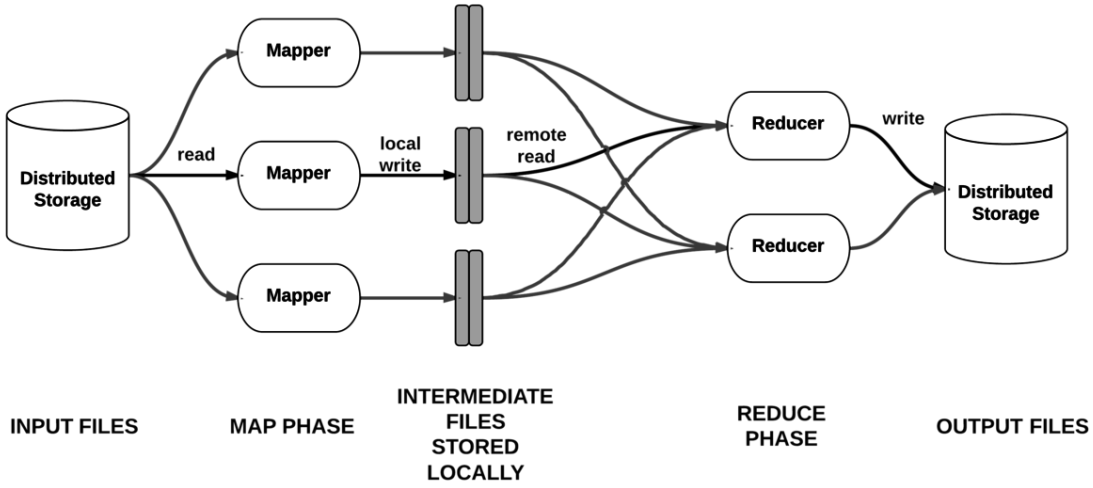

Figure 4.4: General overview on computation model in Apache Hadoop. Map and Reduce slots are fixed to their roles.

HBase [4], equivalents of respectively Google File System and Bigtable. The understanding of Google MapReduce implementation makes easier to understand Hadoop MapReduce as the way of calculating results are optimizations are similar.

In case of Hadoop, Master is called JobTracker, which is fixed to its role as a dedicated single service. Workers from Google MapReduce are called tasks. The number of task per phase is set in a configuration. Tasks are executed and monitored by TaskTracker - one per host. JobTracker communicate with TaskTrackers to monitor job execution. The simplified view on how Hadoop MapReduce works in illustrated in Figure 4.4.

4.3.3 Apache Spark – Beyond MapReduce

Apache Spark [12] goes beyond MapReduce Paradigm. The set of primitive, supported operations (map and reduce) from Apache Hadoop is extended by Group-by, Order, Join, etc. The main advantage of Spark comes from processing data in-memory, which is claimed to be ten to one hundred times faster. The sequence of operations is remembered, so in case of any node failure previous operations on data subset is recomputed. Code may be developed in Java, Scala or Python. Apache Spark Ecosystem is natural choice for iterative processing of big data as presented in Figure 4.5.

4.4 Libraries and Tools in Apache Hadoop Ecosystem

Concluding earlier sections – Apache Hadoop gives broad opportunity to process large amounts of data in simplified way, by only implementing two functions – map and reduce. There is also possibility to join data by providing appropriate mappers for data source A and data source B.

The only thing which is slowing down data analysis in Apache Hadoop is a tremendous number of lines of code, which have to be written in order to provide some basic operations. This is like replacing SQL queries with in Java code, which is much slower and less clear. The same thing goes with loading and storing data from and to HDFS and HBase - it is entirely possible in Java, but with longer code. Hadoop community and multiple companies interested in this ecosystem got the same feeling, that some onerous operations can be described with much less effort.

Thanks to this on top of three basic building bricks of Apache Hadoop (i.e. Hadoop MapReduce, HDFS, HBase) more elements have been added like covered in “Web-scale data mining and

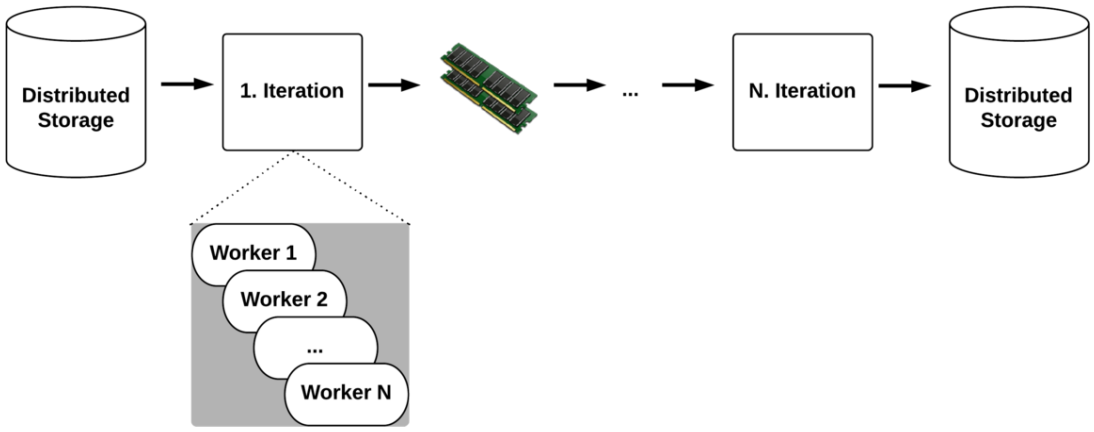

Figure 4.5: General overview on computation model in Apache Spark.

processing" like Apache Pig, Apache Hive, Apache Oozie, Apache Giraph and many more.

4.4.1 Hadoop Streaming

Code natively supported by Apache Hadoop has to be runnable in JVM. To bypass this limitation one can use Hadoop Streaming, which creates (proxy) tasks within JVM and runs indicated by a user program on a cluster machine. Each (proxy) task reads input as described in previous section, then sends it to a non-JVM program and receives its results, eventually further processed within JVM. Communication between Hadoop (proxy) task and a program is performed over standard input/output.

Thanks to Hadoop Streaming a developer may freely use programs written in Python, R, Bash, etc. on map, reduce or both sides.

A Hadoop proxy task reads $\langle key, value \rangle$ pair, concatenates key and value with a custom separator (by default it is a tabulator sign) and sends it to a program (one line per $\langle key, value \rangle$ pair). A program splits an input line by a separator (lack of a separator character means that the whole line is taken as the key), processes it and sends output, which is in the same format as the input. Finally results from the program are decoded into $\langle key, value \rangle$ pair. As expected Hadoop Streaming is slower then native Hadoop code and its usage should be as limited as possible.

The example of code on map (see Figure 4.6. – Python, Figures 4.7. – R) and reduce side (see Figure 4.8.) for counting number of occurrences of keys in input data is provided with the Linux console instruction to test code (see Figure 4.9.) and run it with Hadoop Streaming (see Figure 4.10.).

A user has to specify input and output directories, any files (including scripts to be run) which are not available across all Hadoop machines should be attached with parameter "-file" (see Figure 4.10.).

```
#!/usr/bin/python

import sys

for line in sys.stdin:
  txt = line.strip()
  key, value = txt.split('\t')
  print key + '\t' + '1'
```

Figure 4.6: Mapper code counting occurrences of keys written in Python for Hadoop Streaming

```
#! /usr/bin/env Rscript
options(warn=-1)
sink("/dev/null")
input <- file("stdin", "r")

while(length(line <- readLines(input, n=1, warn=FALSE)) > 0) {
  parts <- unlist(strsplit(line, "\t"))
  key <- parts[1]
  sink()
  cat(parts[1], 1, "\n", sep="\t")
  sink("/dev/null")
}
close(input)
```

Figure 4.7: Mapper code counting occurrences of keys written in **R** for Hadoop Streaming

```
#!/usr/bin/python

import sys

prevKey = None; count = 0

for line in sys.stdin:
  key, value = line.strip().split('\t')
  if prevKey != key:
    if prevKey:
      print prevKey + '\t' + str(count)
    prevKey = key
    count = 1
  else:
    count+=1

if prevKey:
  print prevKey + '\t' + str(count)
```

Figure 4.8: Reducer code counting occurrences of keys written in Python for Hadoop Streaming

```
$ cat input.txt | map.py | sort | reduce.py
```

Figure 4.9: The Linux command for testing in Bash Hadoop Streaming code

```
$ hadoop jar ${HADOOP_HOME}/hadoop-streaming*.jar \
  -input inDir \
  -output outDir \
  -mapper map.py \
  -reducer reduce.py \
  -file map.py \
  -file reduce.py \
  -numRecureTasks 2
```

Figure 4.10: The example of command using Hadoop Streaming

4.4.2 Apache Pig

Simple operations known from SQL or R should be easy to express shortly. Also if possible splitting logical operations in map and reduce phases should be hidden to simplify development. Yahoo! meets these requirements with Pig (see Figure 4.11.).

```
A = LOAD 'someData.txt' AS (id:long, val:chararray);
B = ORDER A BY id;
STORE B INTO 'someOutput';
```

Figure 4.11: The example Pig script

Apache Pig is the solution consisting of PigLatin (the scripting language), Grunt Shell (the interactive console) and Pig Server. Pig Server receives PigLatin code, checks syntactical correctness and transforms the script into one or more Hadoop MapReduce Jobs. It also monitors execution of jobs and reports information about it to a user. Pig can be also used in the local mode outside of a Hadoop cluster and with local file system (see Figure 4.12).

In Pig primitives (integers, floats, chararrays aka. Strings) as well as complex types (tuples, maps, bags - unordered collections, see Figure 4.13.) are available. Having loaded data from HDFS, HBase or any other appropriate storage (see Figure 4.14.), data are put into tuples, which one can easily project (see Figure 4.16), filter, sample, group, join and many more, eventually storing results (see Figure 4.15.).

Brevity is the strong point of Apache Pig. The other is its extensibility, thanks to availability of User Defined Functions (UDFs). A UDF[1] code can be written in Java (see Figure 4.24.), compiled and registered as JAR file in Pig script (see Figure 4.25). One can also write UDFs in Jython (see Figures 4.23 and 4.22.), JavaScript, Ruby, etc. This gives the way to transform data in any chosen way. Of course multiple operations like calculating the absolute value, ceil or floor have been already implemented and are available in piggybank library shipped with Apache Pig [2]. UDFs for processing textual data e.g. for tokenization, trimming, splitting or using regular expressions are also present in piggybank.

In large portions of code there are always some similar fragments of code, which normally are wrapped into functions. With Pig one can use macros (compare Figures 4.19. and 4.19.) which is a named portion of code which takes input and returns output. When code is submitted macros are unfolded.

[1] http://pig.apache.org/docs/r0.12.1/udf.html
[2] http://pig.apache.org/docs/r0.12.1/func.html

Popular macros can be gathered in a separate file and linked to a main script with the IMPORT command. This mechanism has its limitations as imported scripts cannot contain some commands and keywords like DEFINE or DEFAULT.

DEFINE and DEFAULT (see Figure 4.20.) keywords can be used to indicate value of a variable. The first one is definite, value of a variable cannot be changed and any attempt to do so is rising an error. The second option allows overriding the value during a script submission.

Variables extend Pig flexibility allowing to reuse code with different input or output paths, or a choice of scripts to be imported. There is much more as Pig allows to set about of memory per task (see Figure 4.18.), choose number of reducers to be used (see Figure 4.17.), etc.

When developing Pig script one should frequently check if written code is behaving as expected (see Figure 4.21.) by storing results into file system (STORE) or printing it to the console (DUMP), simply checking if data schema is the same as assumed (DESCRIBE) or directly see sample input, intermediate and the final calculation result (ILLUSTRATE). One can also check how the code will be split into mappers and reducers (EXPLAIN).

Of course Pig allows also using of Hadoop Streaming by first registering a script written in Jython (see Figures 4.26.), JavaScript, etc. code and then using the key-phrase STREAM ... THROUGH (see Figure 4.27.). Again, the default separator is a tabulator sign.

Finally, much effort is put into running Pig scripts on Apache Spark (the project code name is Spork). The only difference is running a Pig script is in the execution command ("pig -x mapreduce" vs "pig -x spark").

```
# run Pig in interactive mode on Hadoop cluster; Use HDFS
$pig
# sam as above (explicit choice of the mode)
$pig -x mapreduce
# run Pig in interactive mode outside of Hadoop (local mode);
# use a local file system
$pig -x local
# run a Pig script in a local mode for batch processing
$pig -x local myScript.pig
```

Figure 4.12: Ways in which one can use Apache Pig, i.e. with interactive vs. batch mode and local vs. mapreduce mode.

```
tuple (1,a)
bag   {(1),(2),(3)}
map   ['a'#1234  5,'b'#678]
```

Figure 4.13: Complex Types.

```
/***** load TSV data from FS *****/

-- load everything without assigning names and types, default loader is PigStorage
A = LOAD 'mInput.txt';
-- the same as above
A = LOAD 'mInput.txt' using PigStorage();
-- the same as above, the separator is the comma character
A = LOAD 'mInput.txt' using PigStorage(',');
-- as previous, but fields are named
A = LOAD 'mInput.txt' using PigStorage(',') as (id, txt);
-- as previous, but fields are named and have determined data type
A = LOAD 'mInput.txt' using PigStorage(',') as (id:long, txt:chararray);

/***** load JSON Data from FS *****/
B = load 'input.json' using JsonLoader();

/***** load data from HBase *****/
C = LOAD 'hbase://TableB' USING
    org.apache.pig.backend.hadoop.hbase.HBaseStorage(
      'cf1:a cf1:b cf2:*', '-loadKey=true') AS
      (id:long, fieldAFromCf1:chararray,
      fieldBFromCf1:chararray, mapFromCf2:map[]
    );
```

Figure 4.14: Examples of loading data in Apache Pig script.

```
-- load TSV from FS
A = LOAD 'mInput.txt' using PigStorage('\t') as (id:chararray, txt:chararray);

/***** store TSV data into FS *****/
-- store data separated with the TAB sign (by default), default storer is PigStorage
STORE A into 'mOutput_1.txt';
-- the same as above
STORE A into 'mOutput_2.txt' using PigStorage();
-- the same as above, the separator is the comma character
STORE A into 'mOutput_3.txt' using PigStorage(',');

/***** store JSON data into FS *****/
STORE A into 'mOutput_4.txt' using JsonStorage();

/***** store data into HBase *****/
B = foreach A generate (chararray)$0, (chararray)$1;
C = STORE B into 'hbase://TableB' USING
    org.apache.pig.backend.hadoop.hbase.HBaseStorage(
   'id cf1:txt');
```

Figure 4.15: Examples of storing data in PigLatin.

```
REGISTER '/user/lib/pig/piggybank.jar'
-- Load data with four fields: 3 scalars and 1 collection
A = LOAD 'myinput.txt'
  using PigStorage()
  as (a:int,b:long,c,coll:{(a:long)});
/*
   Multiple ways to project data.
   Only the last assignment is effective
*/
B = foreach A generate $0,$3; -- a,coll
B = foreach A generate $0..$2; -- a,b,c
B = foreach A generate a..c; -- a,b,c
B = foreach A generate b..; -- b,c,coll
B = foreach A generate ..c; -- a,b,c
B = foreach A generate *; -- a,b,c,coll

-- Transformations
B = foreach A generate (long)a, SUM(coll.a);
B = foreach A generate a as id:long, SUM(coll.a) as sum:long;
```

Figure 4.16: Examples of data projections in PigLatin.

```
-- set default number of reducers
SET default_parallel 16
-- read data separated by the tab sign
A = LOAD 'input.txt' as (id:int,b:long);
-- group data; choose customized level of parallelism
B = GROUP A BY id PARALLEL 32;
-- store results to FS
STORE B INTO 'output.txt';
```

Figure 4.17: The example of setting a configuration parameter in PigLatin. Using "default_parallel" one sets the default number of reducers, which might be later overridden.

```
SET mapred.child.java.opts '-Xmx4g'

A = load 'someData.txt' as (id:long, val:chararray);
B = order A by id;
```

Figure 4.18: From the Pig script level a developer may also configure amount of memory per task.

```
DEFINE COUNT_SUM_ELEMS(inTab, grFld, sumFld) RETURNS countedSummed{
  grpd = group $inTab by $grFld;
  $countedSummed = foreach grpd generate group as $grFld, COUNT($inTab), SUM($inTab.$sumFld);
};

%DEFAULT mInput '../myOuterInput*.txt'
DEFINE mOutput 'myOutput.txt'
-- load data separated by the tab sign
A = load '$mInput' as (a:int,b:long);
C = COUNT_SUM_ELEMS(A,a,b);
store C into '$mOutput';
```

Figure 4.19: The example use of a macro defined by a user. Compare it with code from Figure 4.20., i.e. a basic script without macros.

```
%DEFAULT mInput '../myOuterInput*.txt'
DEFINE mOutput 'myOutput.txt'
-- load data separated by the tab sign
A = load '$mInput' as (a:int,b:long);
B = GROUP A by a PARALLEL 32;
C = foreach B generate group as a,COUNT(A) as cnt,SUM(A.b) as sum;
store C into '$mOutput';
```

Figure 4.20: A PigLatin script without use of macro.

```
-- load data separated by the tab sign
A = LOAD 'myinput.txt' using PigStorage('\t') as (a:int,b:long,c);
B = foreach A generate $0,$3;
-- get schema of B; e.g. B:{(a:int,b:long,c:bytearray)}
describe B;
-- print sample of datasets leading to the creation of B
illustrate B;
-- print B to console; e.g.(1,100,example)
dump B;
-- store data separated by the comma sing
-- e.g. 1,100,example[NEWLINE]
store B into 'myresults.txt' using PigStorage(',');
```

Figure 4.21: Different ways of "printing" and storing results or inspecting a code.

```
@outputSchema{'retTuple:(casted:long)'}
# @outputSchema{'casted:long'}
def castToLong(inVal):
  try:
    return (long(inVal));
  except:
    return (None);
```

Figure 4.22: The example of User Defined Function (UDF) written in Jython.

```
REGISTER udf.py USING jython as pUdf;

A = load 'someData.txt' as (id:long, val:chararray);
B = foreach A generate id, FLATTEN(pUdf.catToLong(val)) as castedToVal:long;
```

Figure 4.23: The Pig script using Jython UDF presented in Figure 4.22.

```
package exampleUdfPackage;

import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
import org.apache.pig.data.TupleFactory;

public class MyUpperCase extends EvalFunc<Tuple>{
  public Tuple exec(Tuple input) throws IOException {
    if (input == null || input.size() == 0)
      return null;
    try{
      String str = (String)input.get(0);
      Tuple t = TupleFactory.getInstance().newTuple();
      t.append(str.toUpperCase());
      return t;
    }catch(Exception e){
      throw new IOException("Caught exception processing input row ", e);
    }
  }
}
```

Figure 4.24: The example of User Defined Function (UDF) written in Java.

```
register 'MyProject.jar';

A = load 'someData.txt' as (id:long, val:chararray);
B = foreach A generate id,
        FLATTEN(exampleUdfPackage.MyUpperCase(val))
        as castedToVal:long;
```

Figure 4.25: Use of Java UDF from Figure 4.24

```
#!/usr/bin/env python
import sys
import re
sys.path.append(".")

# input comes from STDIN (standard input)
for line in sys.stdin:
  # rm endline signs; separate keyTABvalue into key,value
  k,text=line.strip().split('\t')
  # remove white-spaces
  text = re.sub('\W+','',text)
  print k+'\t'+text
```

Figure 4.26: The example of a script written in Python for using with Apache Pig via STREAM..THROUGH key-phrase.

```
A = load 'docWithoutLang.txt' as (id:chararray, text:chararray);
B = foreach A generate id, REPLACE(text,'\t',' ') as text;
-- stream each tuple through easyStreamingExample.py;
-- multiple tuples may be passed to one process
C = STREAM C THROUGH
  `easyStreamingExample.py` as (id:chararray, text:chararray);
dump C;
```

Figure 4.27: The example of using STREAM..THROUGH key-phrase with a Python script presented in Figure 4.26.

4.4.3 Apache Oozie

Another useful item from Apache Hadoop Ecosystem is Apache Oozie. Apache Oozie is a pure workflow scheduler (in contrast to Apache Pig) binding jobs written in Hadoop MapReduce, Apache Pig, Hive with some auxiliary actions like sending emails with a custom information, as illustrated in Figure 4.28. Apache Oozie workflow can be run single time or repetitively. A workflow description being Direct Acyclic Graph is written in XML format (see Figure 4.29.). Workflows can be parametrized with configuration file (see Figure 4.30.), which is attached to workflow submission request sent from a commandline (see Figure 4.31.).

Workflow schedulers like Apache Oozie are especially useful in production environments where the same flows of actions are executed repeatedly.

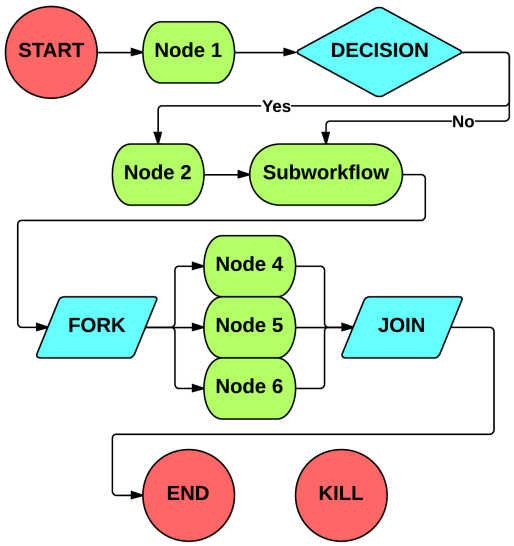

Figure 4.28: The graphical example of a Apache Oozie workflow.

```
<workflow-app xmlns="uri:oozie:workflow:0.2" name="workflowExample">
    <start to="nodeWithPigAction"/>
    <action name="nodeWithPigAction">
        <pig>
            <job-tracker>${jobTracker}</job-tracker>
            <name-node>${nameNode}</name-node>
            <configuration>
                <property>
                    <name>mapred.job.queue.name</name>
                    <value>${queueName}</value>
                </property>
            </configuration>
            <script>example.pig</script>
            <param>inputData=${inputPath}</param>
            <param>results=${outputPath}</param>
        </pig>
        <ok to="endNode"/>
        <error to="failNode"/>
    </action>
    <kill name="failNode">
        <message>Error occurred, [${wf:errorMessage(wf:lastErrorNode())}]</message>
    </kill>
    <end name="endNode"/>
</workflow-app>
```

Figure 4.29: The example of a simple, one node workflow.

```
masterNode=localhost
nameNode=hdfs://${masterNode}:8020
jobTracker=${masterNode}:8021
queueName=default

oozie.wf.application.path=${nameNode}/user/${user.name}/
exampleWorkflow/

inputPath=${nameNode}/user/${user.name}/workflowdata/in/
outputPath=${nameNode}/user/${user.name}/working_dir/out/
```

Figure 4.30: The example of a configuration file for the workflow from Figure 4.29.

```
# run a workflow
$ oozie job -oozie http://localhost:11000/oozie \
-config ${LOCAL_PATH}/configuration.properties \
-run
# get info about workflow execution
$ oozie job -oozie http://localhost:11000/oozie \
-info 0000001-131219141340399-oozie-oozi-W

# kill a working workflow
$ oozie job -oozie http://localhost:11000/oozie \
-kill 0000001-131219141340399-oozie-oozi-W
```

Figure 4.31: Examples of running an Oozie workflow, checking it status and aborting execution.

4.5 Exercises

4.5.1 Guidelines for doing Exercises

To work with Hadoop on your own machine use Cloudera Quickstart Virtual Machine (v.4.7) or newer with your favourite virtualization system. The user name and password is "cloudera". In this virtual machine all elements of the ecosystem are fit in appropriate versions.

4.5.2 Exercise 1: Hadoop Streaming/Apache Pig Streaming

Detect a language used in a text stored in the value. To do so, you may use stopwords corpora from Natural Language Toolkit library (NLTK) and choose the language which the largest number of associated stopwords.

Use Hadoop Streaming/Apache Pig with R, Python, Bash or Perl. See the following example of the input line: "45bc lorem ipsum dolor sit"
and the output line: "45bc lorem ipsum dolor sit latin"
Submit a file *exercise.4.1.zip* containing zipped folder with all scripts and resources used (like *stopwords.zip*) as well as *README.txt* file containing usage description.

4.5.3 Exercise 2: Apache Pig

Using data about parking violations from the service New York City Open Data citizens detected a few fire hydrants near which drivers parked cars against the law more frequent then in other locations. The top five hydrants earned to New York City more then $33,000 per year. As the result of passing this finding to the Department of Transportation road markings were altered.

This is the example of how fairly simple use of data open data may positively influence life of citizens. (This case has been described in the technology website Gizmodo)

Using derivative of NYC Open Data enclosed with this lecture find

1. The top 10 most earning hydrant addresses in general. Assume all violations were fined with a $115 bill.
2. For the above localizations find average number of tickets per year. Take into consideration years 2012-2014

A well-formed address consists of a street name and a purely numeric house number. House numbers like "1/3/5" or "BW" are not acceptable. The example of a correct result for Exercise 2.1 is "Forsyth St 152"

You are expected to **enclose** Apache Pig script(s) and auxiliary code (Java, Python, etc.) used by it **as well as** the final results.

4.5.4 Exercise 3: Apache Oozie

Pack code from Exercise 1 or Exercise 2 into a Apache Oozie workflow. Send appropriately written files workflow.xml, conf.properties. Add your scripts and java code.

Bibliography

[1] Charles Arthur. Tech giants may be huge, but nothing matches big data. The Guardian, 2013.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):1–26, June 2008.

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):1–13, 2004.

[4] Lars George. *HBase: The Definitive Guide*. O'Reilly Media, 1 edition, 2011.

[5] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29, 2003.

[6] Tomasz Kajdanowicz, Wojciech Indyk, Przemyslaw Kazienko, and Jakub Kukul. Comparison of the Efficiency of MapReduce and Bulk Synchronous Parallel Approaches to Large Network Processing. In *2012 IEEE 12th International Conference on Data Mining Workshops*, pages 218–225. IEEE, December 2012.

[7] Jimmy Lin. MapReduce is Good Enough? If All You Have is a Hammer, Throw Away Everything That's Not a Nail! September 2012.

[8] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, page 135, New York, New York, USA, 2010. ACM Press.

[9] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, May 2010.

[10] Technical Staff and Lucid Imagination. Introducing Apache Mahout. Technical report, 2009.

[11] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[12] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark : Cluster Computing with Working Sets. In *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, page 10, 2010.