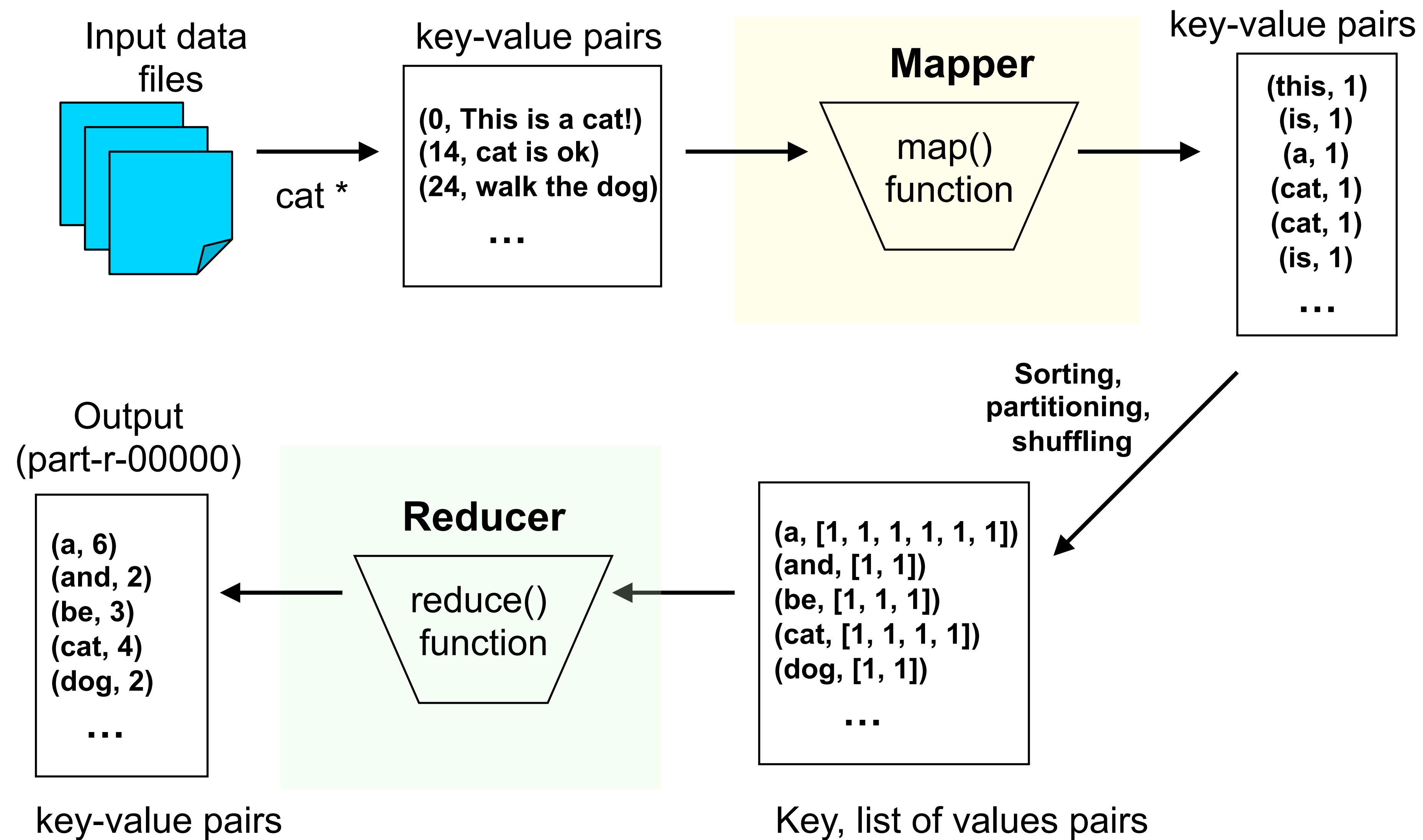
MapReduce

Arend Hintze

Distributed Word Count Example



Abstract Definition

Data

<string, string>,
<string, string>,
<string, string>,
<string, string>,
<string, string>,
<string, string>,
…

Mapper

iterates over list and returns a dictionary:
<key1, value>
<key2, value>
<key3, value>
…

COMBINER

MERGE

Reducer

receives a <key, <set>>
returns a <key,value>

1 the lazy fox
2 the foxy fox
…

M 1
the, 1
lazy, 1
fox 1

M 2
the, 1
foxy, 1
fox 1

the, 1,1
lazy, 1
fox 1,1
foxy 1

the, 2
lazy, 1
fox 2
foxy 1

requirements

- your code needs to start the job
- implement a mapper
- implement a reducer
- everything has to work within the HADOOP framework

template

```
import org.apache.hadoop.*;              // specify the Hadoop libraries used
import java.util.*;                      // specify the Java libraries used
public class ClassName {                 // name of the main class
        public static class MapperClass extends Mapper < Types  > {
                …
        }
        public static class ReducerClass extends Reducer < Types > {
                …
        }
        public static void main (String [] args) throws Exception {
                …
        }
}
```

the mapper class defines map

```
public class MapperClass extends Mapper < Types  > {
        …
        public void map (<Type> key, <Type> value, Context context) {
                …
        }
}
```

the reducer class defines reduce

```
public class ReducerClass extends Reducer < Types  > {
        …
        public void reduce (<Type> key, Iterable<Type> values, Context context) {
                …
        }
}
```

template main

```
public static void main (String args) throws Exception {
        Configuration conf = new Configuration();          // specifies job configuration

        job = new Job(conf, “program name”);               // create a job object
        job.setJarByClass(ClassName.class);                // name of class with main program
        job.setMapperClass(MapperClass.class);             // set name of mapper class
        job.setReducerClass(ReducerClass.class);           // set name of reducer class
        job.setMapOutputKeyClass(<ClassType>);             // type of output key of mapper
        job.setMapOutputValueClass(<ClassType>);           // type of output value of mapper
        job.setOutputKeyClass(<ClassType>);                // type of output key of reducer
        job.setOutputValueClass(<ClassType>);              // type of output value of reducer
        job.setNumReduceTasks(1);                          // set number of reducers
        FileInputFormat.addInputPath(job, new Path(args[0]));      // input data directory
        FileOutputFormat.setOutputPath(job, new Path(args[1]);     // output directory
        System.exit(job.waitForCompletion(true) ? 0 : 1);          // run the job
}
```

in case you don't know java… STREAM IT!

- instead of calling the java.mapper and then the java.reducer you can call a mapper script and a reducer script
- scripts don't have java.types -> everything is a string
- you deal with the strings
- the mapper receives a list of key,values instead of key,set

making a stream job

```
hadoop jar /usr/lib/hadoop-0.20-mapreduce/contrib/streaming/hadoop-
streaming-2.0.0-mr1-cdh4.4.0.jar
    -mapper "python mapper.py"
    -reducer "python reducer.py"
    -input /user/cloudera/stream/input
    -output /user/cloudera/stream/output
    -file simple/mapper.py
    -file simple/reducer.py
```

lots of text, still works “perfectly” fine
it is just slower (sometimes)

java vs. streaming

1 the lazy fox
2 the foxy fox
…

M 1
the, 1
lazy, 1
fox 1

M 2
the, 1
foxy, 1
fox 1

the, 1,1
lazy, 1
fox 1,1
foxy 1

the, 2
lazy, 1
fox 2
foxy 1

1 the lazy fox
2 the foxy fox
…

M 1
the, 1
lazy, 1
fox 1

M 2
the, 1
foxy, 1
fox 1

the 1
the 1
lazy 1
fox 1
fox 1
foxy 1

the, 2
lazy, 1
fox 2
foxy 1

mapper.py reducer.py

```
import sys

for line in sys.stdin:
    L=line.strip()
    for word in L.split():
        print(word+"\t1")
```

```
import sys
D=dict()
for line in sys.stdin:
    L=line.strip()
    w=L.split("\t")
    if(D.has_key(w[0])):
        D[w[0]]=D[w[0]]+int(w[1])
    else:
        D[w[0]]=int(w[1])
for k in D:
    print(str(k)+"\t"+str(D[k]))
```

chmod +x mapper.py and chmod +x reducer.py

```
cat text.txt | python mapper.py | sort | python reducer.py
```

do it!

Combiner

- Assume a very large job. Mappers might produce billions of key value pairs
- combiners help merging all the data
- combiners perform the same transformation as the reducer on subsets of the data!
- combiners might be applied never, once, multiple times
- not all reducers can be used as combiners!

max ([list1, list2]) = max([max(list1), max(list2)])
avg ([list1, list2]) avg([avg(list1), avg(list2)])

dataset example min