MapReduce by examples

The code is available on:
https://github.com/andreaiacono/MapReduce

Take a look at my blog:
https://andreaiacono.blogspot.com/

What is MapReduce?

MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster

[src: *http://en.wikipedia.org/wiki/MapReduce*]

Originally published in 2004 from Google engineers Jeffrey Dean and Sanjay Ghemawat

Hadoop is the open source implementation of the model by Apache Software foundation

The main project is composed by:

- HDFS
- YARN
- MapReduce

Its ecosystem is composed by:

- Pig
- Hbase
- Hive
- Impala
- Mahout
- a lot of other tools

Hadoop 2.x

- YARN: the resource manager, now called YARN, is now detached from mapreduce framework
- java packages are under org.apache.hadoop.mapreduce.*

MapReduce inspiration

The name MapReduce comes from functional programming:

- **map** is the name of a higher-order function that applies a given function to each element of a list. Sample in Scala:

```
val numbers = List(1,2,3,4,5)
numbers.map(x => x * x) == List(1,4,9,16,25)
```

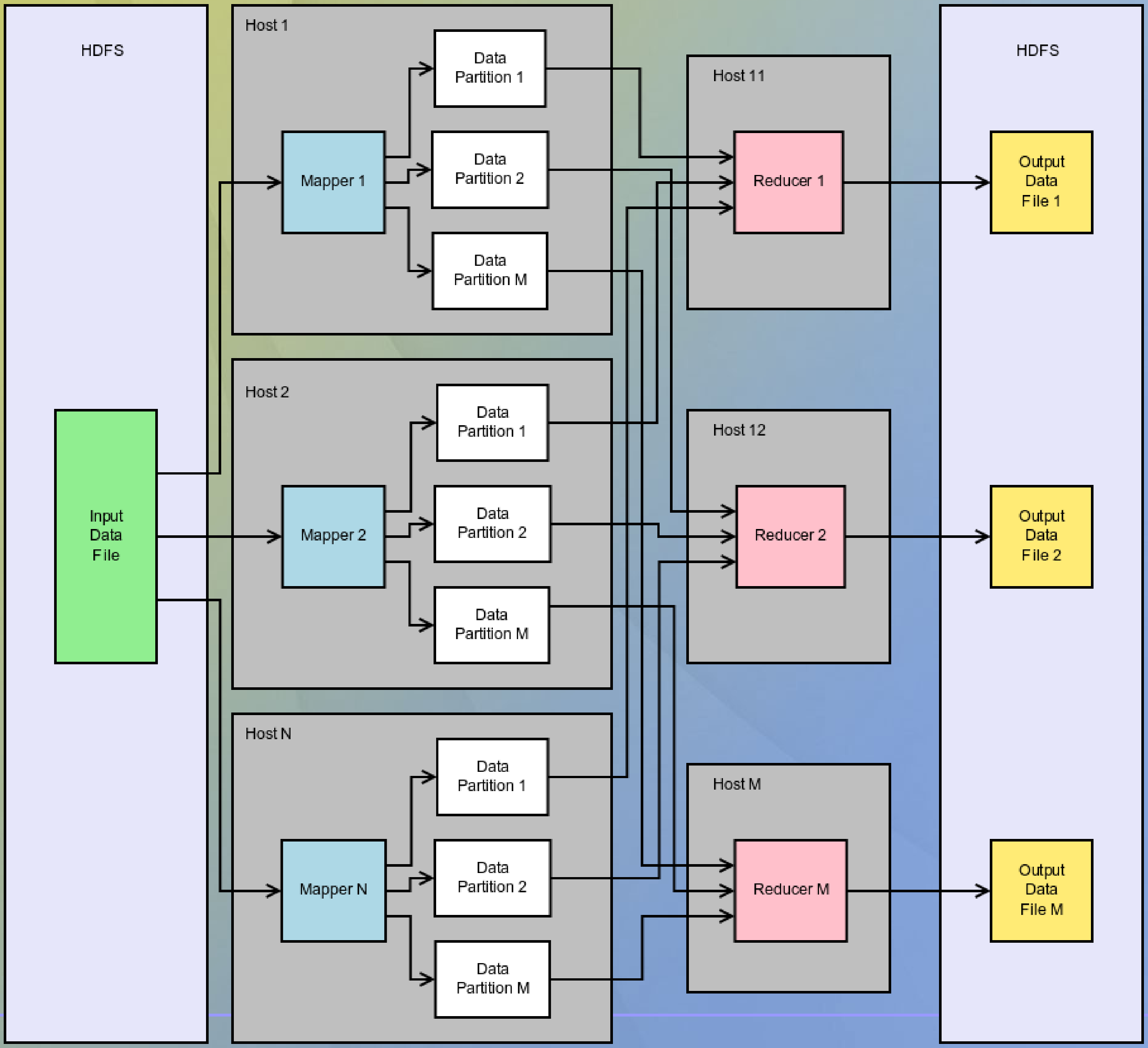
- **reduce** is the name of a higher-order function that analyze a recursive data structure and recombine through use of a given combining operation the results of recursively processing its constituent parts, building up a return value. Sample in Scala:

```
val numbers = List(1,2,3,4,5)
numbers.reduce(_ + _) == 15
```

MapReduce takes an input, splits it into smaller parts, execute the code of the mapper on every part, then gives all the results to one or more reducers that merge all the results into one.

src: http://en.wikipedia.org/wiki/Map_(higher-order_function)
http://en.wikipedia.org/wiki/Fold_(higher-order_function)

Overall view



How does Hadoop work?

Init

- Hadoop divides the input file stored on HDFS into splits (tipically of the size of an HDFS block) and assigns every split to a different mapper, trying to assign every split to the mapper where the split physically resides

Mapper

- locally, Hadoop reads the split of the mapper line by line
- locally, Hadoop calls the method map() of the mapper for every line passing it as the key/value parameters
- the mapper computes its application logic and *emits* other key/value pairs

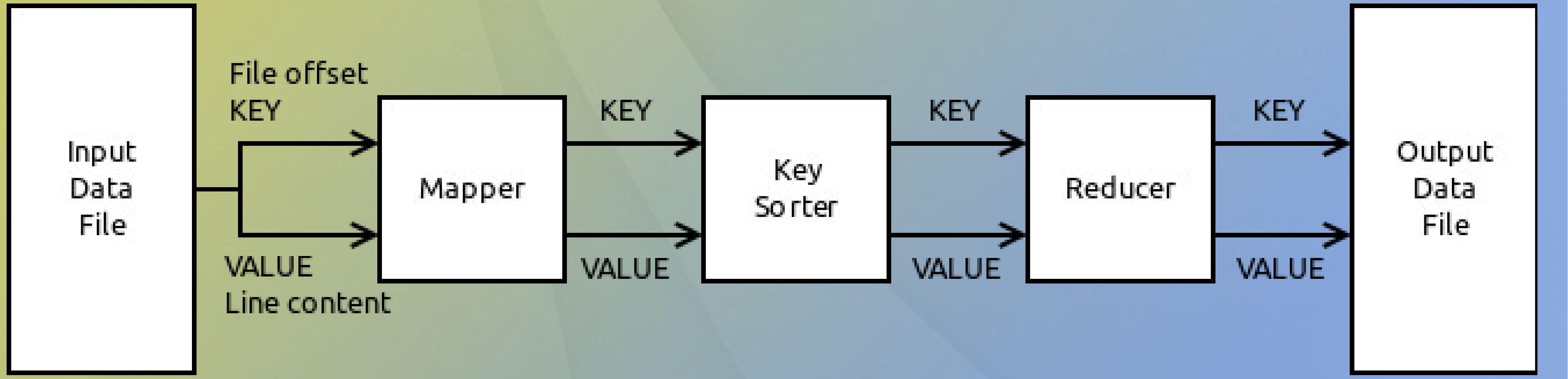
Shuffle and sort

- locally, Hadoop's partitioner divides the emitted output of the mapper into partitions, each of those is sent to a different reducer
- locally, Hadoop collects all the different partitions received from the mappers and sort them by key

Reducer

- locally, Hadoop reads the aggregated partitions line by line
- locally, Hadoop calls the reduce() method on the reducer for every line of the input
- the reducer computes its application logic and *emits* other key/value pairs
- locally, Hadoop writes the emitted pairs output (the emitted pairs) to HDFS

Simplied flow (for developers)



Serializable vs Writable

- Serializable stores the class name and the object representation to the stream; other instances of the class are referred to by an handle to the class name: this approach is not usable with random access

- For the same reason, the sorting needed for the shuffle and sort phase can not be used with Serializable

- The deserialization process creates a new instance of the object, while Hadoop needs to reuse objects to minimize computation

- Hadoop introduces the two interfaces Writable and WritableComparable that solve these problem

Writable wrappers

Java primitive	**Writable implementation**
boolean	BooleanWritable
byte	ByteWritable
short	ShortWritable
int	IntWritable VIntWritable
float	FloatWritable
long	LongWritable VLongWritable
double	DoubleWritable

Java class	**Writable implementation**
String	Text
byte[]	BytesWritable
Object	ObjectWritable
null	NullWritable

Java collection	**Writable implementation**
array	ArrayWritable ArrayPrimitiveWritable TwoDArrayWritable
Map	MapWritable
SortedMap	SortedMapWritable
enum	EnumSetWritable

Implementing Writable: the SumCount class

```
public class SumCount implements WritableComparable<SumCount> {

    DoubleWritable sum;
    IntWritable count;

    public SumCount() {
        set(new DoubleWritable(0), new IntWritable(0));
    }

    public SumCount(Double sum, Integer count) {
        set(new DoubleWritable(sum), new IntWritable(count));
    }

    @Override
    public void write(DataOutput dataOutput) throws IOException {

        sum.write(dataOutput);
        count.write(dataOutput);
    }

    @Override
    public void readFields(DataInput dataInput) throws IOException {

        sum.readFields(dataInput);
        count.readFields(dataInput);
    }
    // getters, setters and Comparable overridden methods are omitted
}
```

Glossary

Term	Meaning
Job	The whole process to execute: the input data, the mapper and reducers execution and the output data
Task	Every job is divided among the several mappers and reducers; a task is the job portion that goes to every single mapper and reducer
Split	The input file is split into several splits (the suggested size is the HDFS block size, 64Mb)
Record	The split is read from mapper by default a line at the time: each line is a record. Using a class extending `FileInputFormat`, the record can be composed by more than one line
Partition	The set of all the key-value pairs that will be sent to a single reducer. The default partitioner uses an hash function on the key to determine to which reducer send the data

Let's start coding!

WordCount

(the Hello World! for MapReduce, available in Hadoop sources)

We want to count the occurrences of every word of a text file

Input Data:

The text of the book ”Flatland”
By Edwin Abbott.

Source:
http://www.gutenberg.org/cache/epub/201/pg201.txt

WordCount mapper

```
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {

        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        @Override
        public void map(Object key, Text value, Context context)
                                throws IOException, InterruptedException {

            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken().trim());
                context.write(word, one);
            }
       }
 }
```

WordCount reducer

```
public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable>{

        private IntWritable result = new IntWritable();

        @Override
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
                            throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
       }
}
```

WordCount

Results:

```
a             936
ab            6
abbot         3
abbott        2
abbreviated   1
abide         1
ability       1
able          9
ablest        2
abolished     1
abolition     1
about         40
above         22
abroad        1
abrogation    1
abrupt        1
abruptly      1
absence       4
absent        1
absolute      2
...
```

MapReduce testing and debugging

- MRUnit is a testing framework based on Junit for unit testing mappers, reducers, combiners (we'll see later what they are) and the combination of the three

- Mocking frameworks can be used to mock Context or other Hadoop objects

- LocalJobRunner is a class included in Hadoop that let us run a complete Hadoop environment locally, in a single JVM, that can be attached to a debugger. LocalJobRunner can run at most one reducer

- Hadoop allows the creation of in-process mini clusters programmatically thanks to MiniDFSCluster and MiniMRCluster testing classes; debugging is more difficult than LocalJobRunner because is multi-threaded and spread over different VMs. Mini Clusters are used for testing Hadoop sources.

MRUnit test for WordCount

```
@Test
public void testMapper() throws Exception {
    new MapDriver<Object, Text, Text, IntWritable>()
            .withMapper(new WordCount.TokenizerMapper())
            .withInput(NullWritable.get(), new Text("foo bar foo"))
            .withOutput(new Text("foo"), new IntWritable(1))
            .withOutput(new Text("bar"), new IntWritable(1))
            .withOutput(new Text("foo"), new IntWritable(1))
            .runTest();
}

@Test
public void testReducer() throws Exception {
    List<IntWritable> fooValues = new ArrayList<>();
    fooValues.add(new IntWritable(1));
    fooValues.add(new IntWritable(1));

    List<IntWritable> barValue = new ArrayList<>();
    barValue.add(new IntWritable(1));

    new ReduceDriver<Text, IntWritable, Text, IntWritable>()
            .withReducer(new WordCount.IntSumReducer())
            .withInput(new Text("foo"), fooValues)
            .withInput(new Text("bar"), barValue)
            .withOutput(new Text("foo"), new IntWritable(2))
            .withOutput(new Text("bar"), new IntWritable(1))
            .runTest();
}
```

MRUnit test for WordCount

```java
@Test
public void testMapReduce() throws Exception {

    new MapReduceDriver<Object, Text, Text, IntWritable, Text, IntWritable>()
            .withMapper(new WordCount.TokenizerMapper())
            .withInput(NullWritable.get(), new Text("foo bar foo"))
            .withReducer(new WordCount.IntSumReducer())
            .withOutput(new Text("bar"), new IntWritable(1))
            .withOutput(new Text("foo"), new IntWritable(2))
            .runTest();
}
```

TopN

We want to find the top-n used words of a text file

Input Data:

The text of the book ”Flatland”
By E. Abbott.

Source:
http://www.gutenberg.org/cache/epub/201/pg201.txt

TopN mapper

```
public static class TopNMapper extends Mapper<Object, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    private String tokens = "[_|$#<>\\^=\\[\\]\\*/\\\\,;,.\\-:()?!\"']";

    @Override
    public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {

        String cleanLine = value.toString().toLowerCase().replaceAll(tokens, " ");
        StringTokenizer itr = new StringTokenizer(cleanLine);
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken().trim());
            context.write(word, one);
        }
    }
}
```

TopN reducer

```
public static class TopNReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

        private Map<Text, IntWritable> countMap = new HashMap<>();

        @Override
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
                                                throws IOException, InterruptedException {

            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }

            countMap.put(new Text(key), new IntWritable(sum));
        }

        @Override
        protected void cleanup(Context context) throws IOException, InterruptedException {

            Map<Text, IntWritable> sortedMap = sortByValues(countMap);

            int counter = 0;
            for (Text key: sortedMap.keySet()) {
                if (counter ++ == 20) {
                    break;
                }
                context.write(key, sortedMap.get(key));
            }
        }
    }
```

TopN

Results:

```
the     2286
of      1634
and     1098
to      1088
a       936
i       735
in      713
that    499
is      429
you     419
my      334
it      330
as      322
by      317
not     317
or      299
but     279
with    273
for     267
be      252
...
```

TopN

In the ***shuffle and sort*** phase, the partioner will send every single word (the key) with the value "1" to the reducers.

All these network transmissions can be minimized if we reduce locally the data that the mapper will emit.

This is obtained by a *Combiner*.

TopN combiner

```
public static class Combiner extends Reducer<Text, IntWritable, Text, IntWritable> {

        @Override
        public void reduce(Text key, Iterable<IntWritable> values, Context context)
                           throws IOException, InterruptedException {

            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }
```

TopN

Hadoop output

```
   Without combiner                          With combiner

Map input records=4239                  Map input records=4239
Map output records=37817                Map output records=37817
Map output bytes=359621                 Map output bytes=359621
Input split bytes=118                   Input split bytes=116
Combine input records=0                 Combine input records=37817
Combine output records=0                Combine output records=20
Reduce input groups=4987                Reduce input groups=20
Reduce shuffle bytes=435261             Reduce shuffle bytes=194
Reduce input records=37817              Reduce input records=20
Reduce output records=20                Reduce output records=20
```

Combiners

If the function computed is

- **commutative** [a + b = b + a]
- **associative** [a + (b + c) = (a + b) + c]

we can reuse the reducer as a combiner!

Max function works:
max (max(a,b), max(c,d,e)) = max (a,b,c,d,e)

Mean function does not work:
mean(mean(a,b), mean(c,d,e)) != mean(a,b,c,d,e)

Combiners

Advantages of using combiners

- Network transmissions are minimized

Disadvantages of using combiners

- Hadoop does not guarantee the execution of a combiner: it can be executed 0, 1 or multiple times on the same input

- Key-value pairs emitted from mapper are stored in local filesystem, and the execution of the combiner could cause expensive IO operations

TopN in-mapper combiner

```
private Map<String, Integer> countMap = new HashMap<>();
private String tokens = "[_|$#<>\\^=\\[\\]\\*/\\\\,;,.\\-:()?!\"']";

@Override
public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {

    String cleanLine = value.toString().toLowerCase().replaceAll(tokens, " ")
    StringTokenizer itr = new StringTokenizer(cleanLine);
    while (itr.hasMoreTokens()) {

        String word = itr.nextToken().trim();
        if (countMap.containsKey(word)) {
            countMap.put(word, countMap.get(word)+1);
        }
        else {
            countMap.put(word, 1);
        }
    }
}

@Override
protected void cleanup(Context context) throws InterruptedException {

    for (String key: countMap.keySet()) {
        context.write(new Text(key), new IntWritable(countMap.get(key)));
    }
}
```

TopN in-mapper reducer

```java
private Map<Text, IntWritable> countMap = new HashMap<>();

@Override
public void reduce(Text key, Iterable<IntWritable> values, Context context)
    throws IOException, InterruptedException {

    int sum = 0;
    for (IntWritable val : values) {
        sum += val.get();
    }

    countMap.put(new Text(key), new IntWritable(sum));
}

@Override
protected void cleanup(Context context) throws InterruptedException {

    Map<Text, IntWritable> sortedMap = sortByValues(countMap);

    int counter = 0;
    for (Text key: sortedMap.keySet()) {
        if (counter ++ == 20) {
            break;
        }
        context.write(key, sortedMap.get(key));
    }
}
```

Combiners - output

Without combiner

```
Map input records=4239
Map output records=37817
Map output bytes=359621
Input split bytes=118
Combine input records=0
Combine output records=0
Reduce input groups=4987
Reduce shuffle bytes=435261
Reduce input records=37817
Reduce output records=20
```

With combiner

```
Map input records=4239
Map output records=37817
Map output bytes=359621
Input split bytes=116
Combine input records=37817
Combine output records=20
Reduce input groups=20
Reduce shuffle bytes=194
Reduce input records=20
Reduce output records=20
```

With in-mapper

```
Map input records=4239
Map output records=4987
Map output bytes=61522
Input split bytes=118
Combine input records=0
Combine output records=0
Reduce input groups=4987
Reduce shuffle bytes=71502
Reduce input records=4987
Reduce output records=20
```

With in-mapper and combiner

```
Map input records=4239
Map output records=4987
Map output bytes=61522
Input split bytes=116
Combine input records=4987
Combine output records=20
Reduce input groups=20
Reduce shuffle bytes=194
Reduce input records=20
Reduce output records=20
```

Mean

We want to find the mean max temperature for every month

Input Data:

Temperature in Milan

(DDMMYYY, MIN, MAX)

```
01012000, -4.0, 5.0
02012000, -5.0, 5.1
03012000, -5.0, 7.7
...
29122013,  3.0, 9.0
30122013,  0.0, 9.8
31122013,  0.0, 9.0
```

Data source:
http://archivio-meteo.distile.it/tabelle-dati-archivio-meteo/

Mean mapper

```
private Map<String, List<Double>> maxMap = new HashMap<>();

@Override
public void map(Object key, Text value, Context context)
                                throws IOException, InterruptedException {

    String[] values = value.toString().split((","));
    if (values.length != 3) return;

    String date = values[DATE];
    Text month = new Text(date.substring(2));
    Double max = Double.parseDouble(values[MAX]);

    if (!maxMap.containsKey(month)) {
        maxMap.put(month, new ArrayList<Double>());
    }
    maxMap.get(month).add(max);
}

@Override
protected void cleanup(Mapper.Context context) throws InterruptedException {
    for (Text month: maxMap.keySet()) {

        List<Double> temperatures = maxMap.get(month);
        Double sum = 0d;
        for (Double max: temperatures) {
            sum += max;
        }
        context.write(month, new DoubleWritable(sum));
    }
}
```

Mean mapper

```
private Map<String, List<Double>> maxMap = new HashMap<>();

@Override
public void map(Object key, Text value, Context context)
                                throws IOException, InterruptedException {

    String[] values = value.toString().split((","));
    if (values.length != 3) return;

    String date = values[DATE];
    Text month = new Text(date.substring(2));
    Double max = Double.parseDouble(values[MAX]);

    if (!maxMap.containsKey(month)) {
        maxMap.put(month, new ArrayList<Double>());
    }
    maxMap.get(month).add(max);
}

@Override
protected void cleanup(Mapper.Context context) throws InterruptedException {
    for (Text month: maxMap.keySet()) {

        List<Double> temperatures = maxMap.get(month);
        Double sum = 0d;
        for (Double max: temperatures) {
            sum += max;
        }
        context.write(month, new DoubleWritable(sum));
    }
}
```

Is this correct?

Mean

Sample input data:

```
01012000, 0.0, 10.0
02012000, 0.0, 20.0
03012000, 0.0, 2.0
04012000, 0.0, 4.0
05012000, 0.0, 3.0
```

Mapper #1: lines 1, 2
Mapper #2: lines 3, 4, 5

Mapper#1: mean = (10.0 + 20.0) / 2 = 15.0
Mapper#2: mean = (2.0 + 4.0 + 3.0) / 3 = 3.0

Reducer mean = (15.0 + 3.0) / 2 = 9.0

But the correct mean is:
(10.0 + 20.0 + 2.0 + 4.0 + 3.0) / 5 = 7.8

Not correct!

jug
Milano

Mean mapper

```
private Map<Text, List<Double>> maxMap = new HashMap<>();

@Override
public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {
    String[] values = value.toString().split((","));
    if (values.length != 3) return;

    String date = values[DATE];
    Text month = new Text(date.substring(2));
    Double max = Double.parseDouble(values[MAX]);

    if (!maxMap.containsKey(month)) {
        maxMap.put(month, new ArrayList<Double>());
    }
    maxMap.get(month).add(max);
}

@Override
protected void cleanup(Context context) throws InterruptedException {

    for (Text month: maxMap.keySet()) {

        List<Double> temperatures = maxMap.get(month);
        Double sum = 0d;
        for (Double max: temperatures) sum += max;
        context.write(month, new SumCount(sum, temperatures.size()));
    }
}
```

This is correct!

Mean reducer

```
private Map<Text, SumCount> sumCountMap = new HashMap<>();

@Override
public void reduce(Text key, Iterable<SumCount> values, Context context)
                 throws IOException, InterruptedException {

    SumCount totalSumCount = new SumCount();
    for (SumCount sumCount : values) {

        totalSumCount.addSumCount(sumCount);
    }

    sumCountMap.put(new Text(key), totalSumCount);
}

@Override
protected void cleanup(Context context) throws InterruptedException {

    for (Text month: sumCountMap.keySet()) {

        double sum = sumCountMap.get(month).getSum().get();
        int count = sumCountMap.get(month).getCount().get();

        context.write(month, new DoubleWritable(sum/count));
    }
}
```

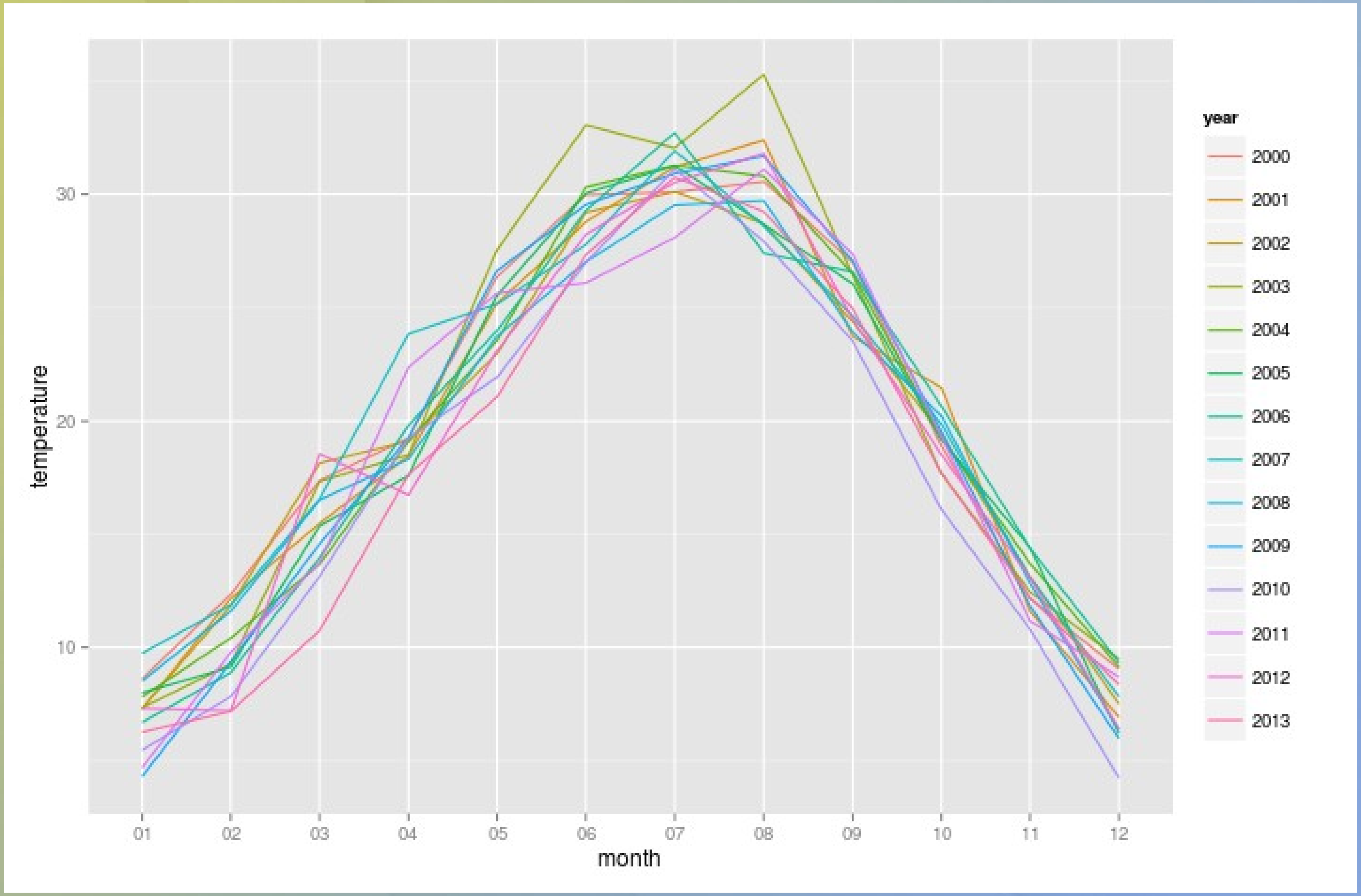
Mean

Results:

```
022012    7.230769230769231
022013    7.2
022010    7.851851851851852
022011    9.785714285714286
032013    10.741935483870968
032010    13.133333333333333
032012    18.548387096774192
032011    13.741935483870968
022003    9.278571428571428
022004    10.41034482758621
022005    9.146428571428572
022006    8.903571428571428
022000    12.344444444444441
022001    12.164285714285715
022002    11.839285714285717
...
```

Mean

Result:



R code to plot data:

```
temp <- read.csv(file="results.txt", sep="\t", header=0)
names(temp) <- c("date","temperature")
ym <- as.yearmon(temp$date, format = "%m-%Y");
year <- format(ym, "%Y")
month <- format(ym, "%m")
ggplot(temp, aes(x=month, y=temperature, group=year)) + geom_line(aes(colour = year))
```

Join

We want to combine information from the users file with Information from the posts file (a join)

Input Data - Users file:

"user_ptr_id" "reputation" "gold" "silver" "bronze"
"100006402" "18" "0" "0" "0"
"100022094" "6354" "4" "12" "50"
"100018705" "76" "0" "3" "4"
…

Input Data - Posts file:

"id" "title" "tagnames" "author_id" "body" "node_type" "parent_id" "abs_parent_id" "added_at" "score" …
"5339" "Whether pdf of Unit and Homework is available?" "cs101 pdf" "100000458" "" "question" "\N" "\N"
"2012-02-25 08:09:06.787181+00" "1"
"2312" "Feedback on Audio Quality" "cs101 production audio" "100005361" "<p>We are looking for feedback on the audio in our videos. Tell us what you think and try to be as <em>specific</em> as possible.</p>" "question" "\N" "\N" "2012-02-23 00:28:02.321344+00" "2"
"2741" "where is the sample page for homework?" "cs101 missing_info homework" "100001178" "<p>I am sorry if I am being a nob ... but I do not seem to find any information regarding the sample page reffered to on the 1 question of homework 1." "question" "\N" "\N" "2012-02-23 09:15:02.270861+00" "0"
...

Data source:
http://content.udacity-data.com/course/hadoop/forum_data.tar.gz

Join mapper

```
@Override
public void map(Object key, Text value, Context context)
                throws IOException, InterruptedException {

    FileSplit fileSplit = (FileSplit) context.getInputSplit();
    String filename = fileSplit.getPath().getName();
    String[] fields = value.toString().split(("\t"));

    if (filename.equals("forum_nodes_no_lf.tsv")) {
        if (fields.length > 5) {
            String authorId = fields[3].substring(1, fields[3].length() - 1);
            String type = fields[5].substring(1, fields[5].length() - 1);
            if (type.equals("question")) {
                context.write(new Text(authorId), one);
            }
        }
    }
    else {
         String authorId = fields[0].substring(1, fields[0].length() - 1);
         String reputation = fields[1].substring(1, fields[1].length() - 1);
         try {
             int reputationValue = Integer.parseInt(reputation) + 2;
             context.write(new Text(authorId),new IntWritable(reputationValue));
         }
         catch (NumberFormatException nfe) {
             // just skips this record
         }
    }
}
```

Join reducer

```
@Override
public void reduce(Text key, Iterable<IntWritable> values, Context context)
                    throws IOException, InterruptedException {

    int postsNumber = 0;
    int reputation = 0;
    String authorId = key.toString();

    for (IntWritable value : values) {

        int intValue = value.get();
        if (intValue == 1) {
            postsNumber ++;
        }
        else {
            reputation = intValue -2;
        }
    }
    context.write(new Text(authorId), new Text(reputation + "\t" + postsNumber));
}
```

Join

Results:

USER_ID	REPUTATION	SCORE
00081537	1019	3
100011949	12	1
100105405	36	1
100000628	60	2
100011948	231	1
100000629	2090	1
100000623	1	2
100011945	457	4
100000624	167	1
100011944	114	3
100000625	1	1
100000626	93	1
100011942	11	1
100000620	1	1
100011940	35	1
100000621	2	1
100080016	11	2
100080017	53	1
100081549	1	1

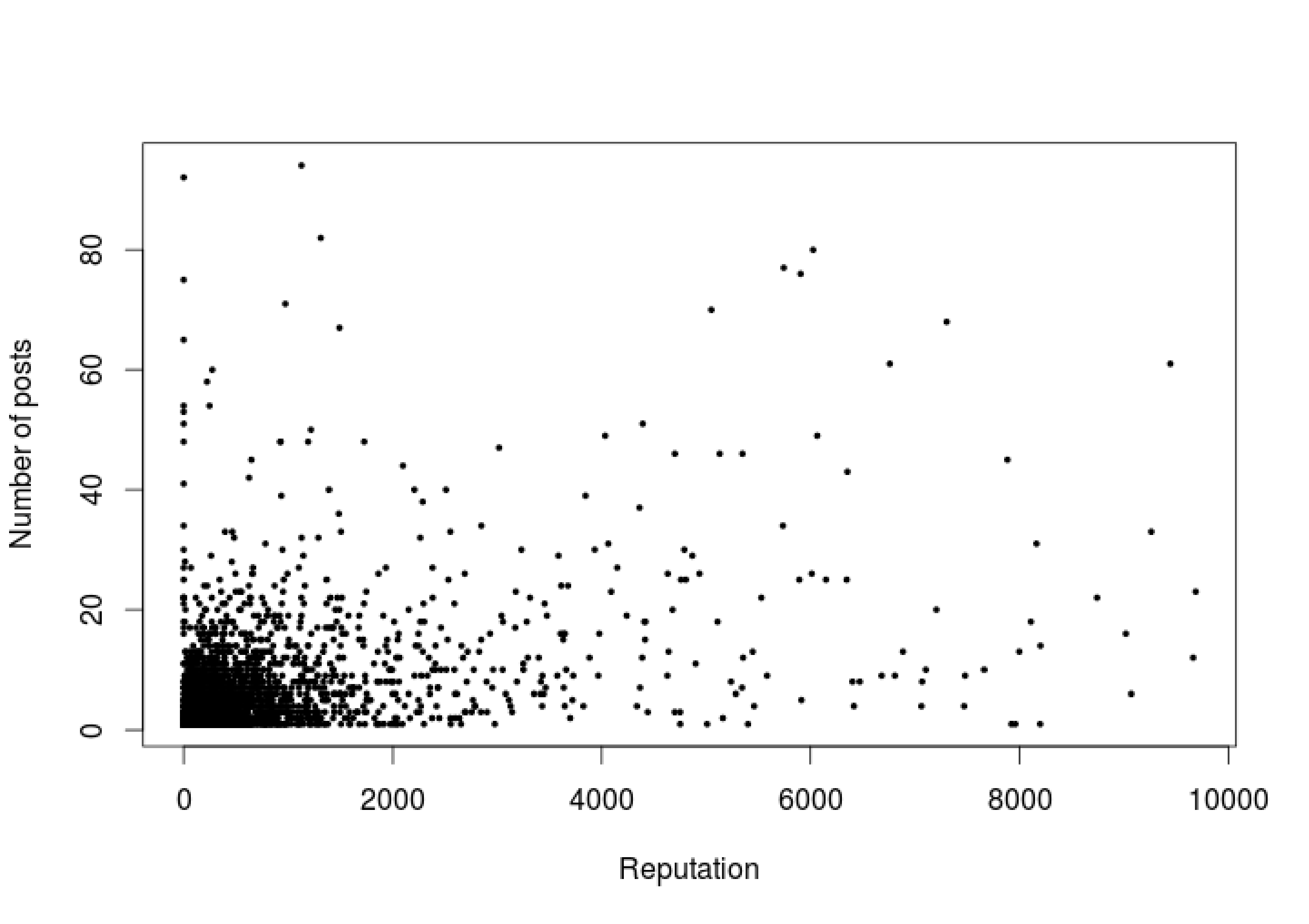
...

Join

R code to plot data:

```
users <- read.csv(file="part-r-00000",sep='\t', header=0)
users$V2[which(users$V2 > 10000,)] <- 0
plot(users$V2, users$V3, xlab="Reputation", ylab="Number of posts", pch=19, cex=0.4)
```

Result:



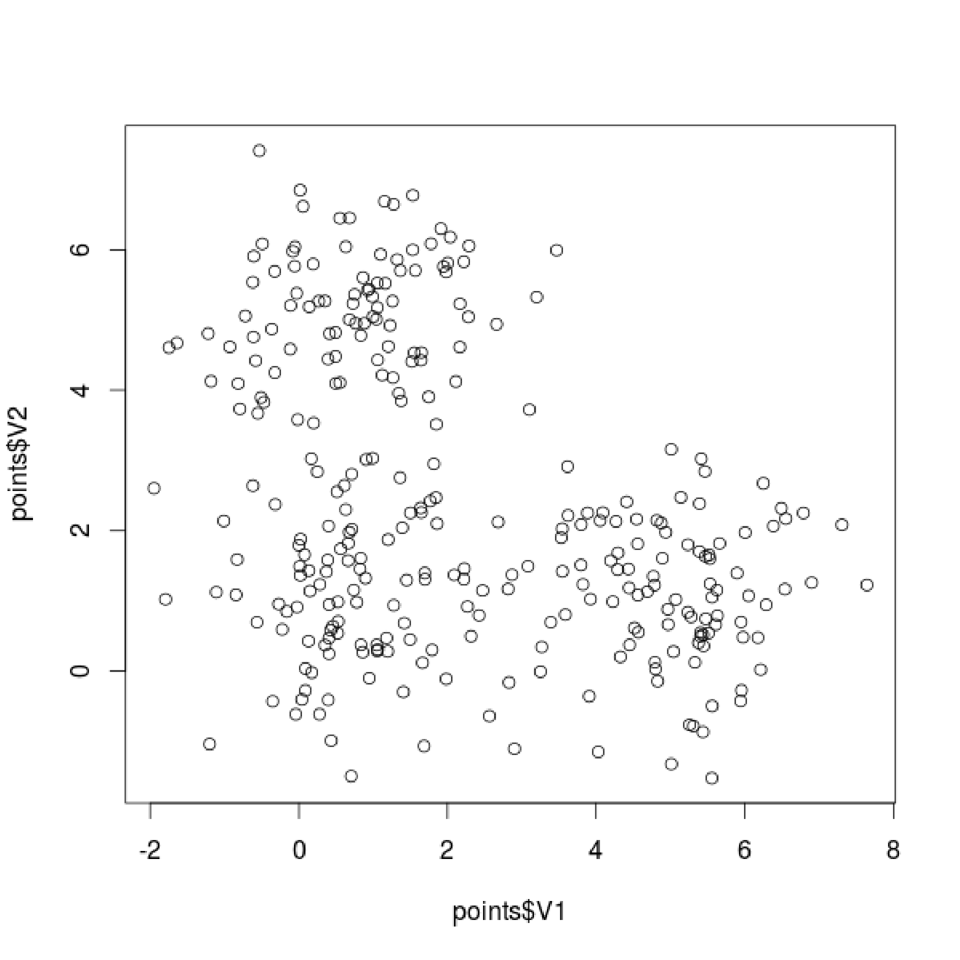
K-means

We want to aggregate 2D points in clusters using K-means algorithm

Input Data:

A random set of points

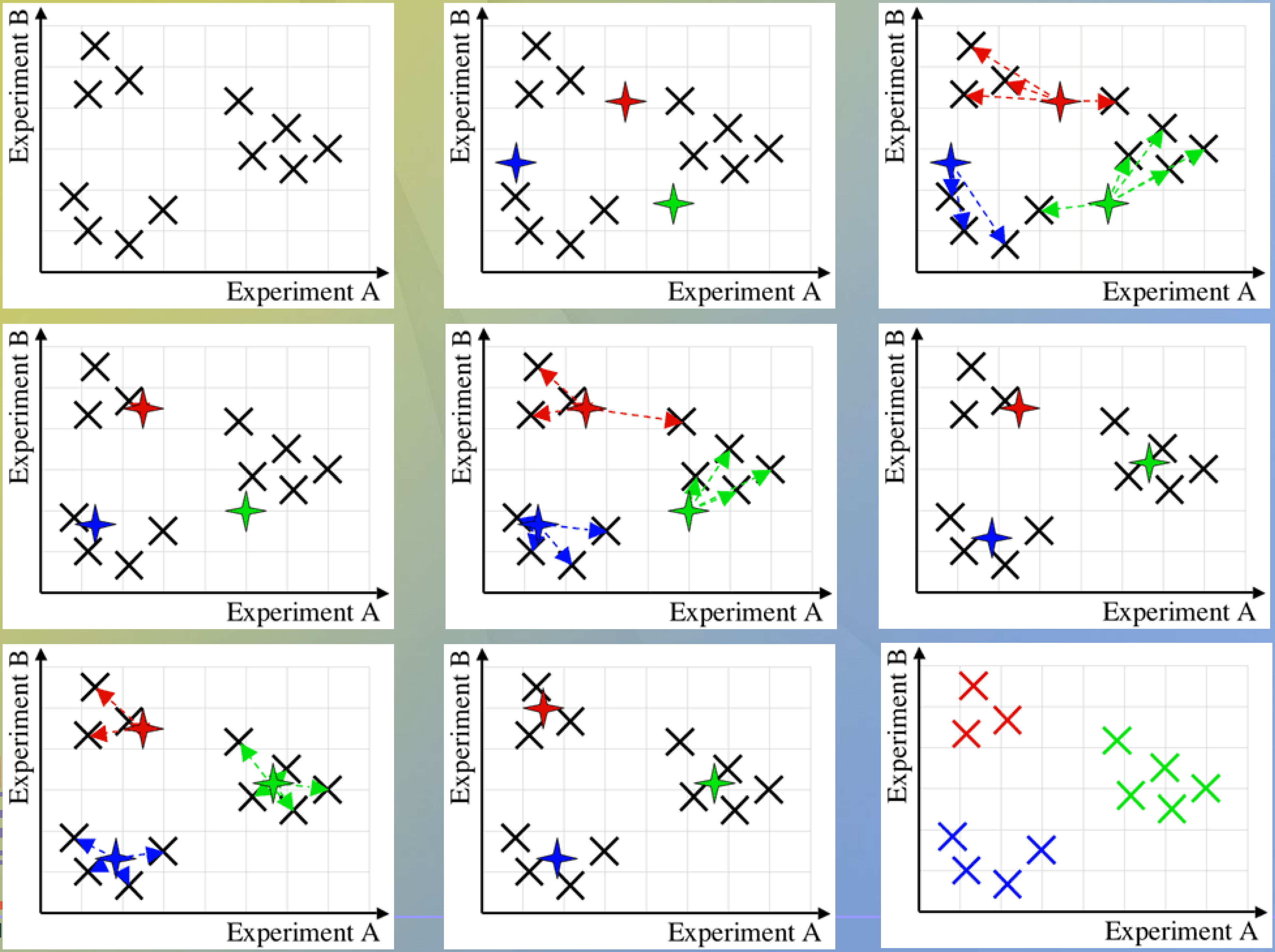
```
 2.2705   0.9178
 1.8600   2.1002
 2.0915   1.3679
-0.1612   0.8481
-1.2006  -1.0423
 1.0622   0.3034
 0.5138   2.5542
 ...
```



R code to generate dataset:

```
N <- 100
x <- rnorm(N)+1; y <- rnorm(N)+1; dat <- data.frame(x, y)
x <- rnorm(N)+5; y <- rnorm(N)+1; dat <- rbind(dat, data.frame(x, y))
x <- rnorm(N)+1; y <- rnorm(N)+5; dat <- rbind(dat, data.frame(x, y))
```

K-means algorithm



K-means mapper

```
@Override
protected void setup(Context context) throws IOException, InterruptedException {
    URI[] cacheFiles = context.getCacheFiles();
    centroids = Utils.readCentroids(cacheFiles[0].toString());
}

@Override
public void map(Object key, Text value, Context context)
                                    throws IOException, InterruptedException {

    String[] xy = value.toString().split(" ");
    double x = Double.parseDouble(xy[0]);
    double y = Double.parseDouble(xy[1]);
    int index = 0;
    double minDistance = Double.MAX_VALUE;
    for (int j = 0; j < centroids.size(); j++) {
        double cx = centroids.get(j)[0];
        double cy = centroids.get(j)[1];
        double distance = Utils.euclideanDistance(cx, cy, x, y);
        if (distance < minDistance) {
            index = j;
            minDistance = distance;
        }
    }

    context.write(new IntWritable(index), value);
}
```

K-means reducer

```
public class KMeansReducer extends Reducer<IntWritable, Text, Text, IntWritable> {

    @Override
    protected void reduce(IntWritable key, Iterable<Text> values, Context context)
                            throws IOException, InterruptedException {

        Double mx = 0d;
        Double my = 0d;
        int counter = 0;

        for (Text value: values) {
            String[] temp = value.toString().split(" ");
            mx += Double.parseDouble(temp[0]);
            my += Double.parseDouble(temp[1]);
            counter ++;
        }

        mx = mx / counter;
        my = my / counter;
        String centroid = mx + " " + my;

        context.write(new Text(centroid), key);
    }
}
```

K-means driver - 1

```
public static void main(String[] args) throws Exception {

        Configuration configuration = new Configuration();
        String[] otherArgs = new GenericOptionsParser(configuration, args).getRemainingArgs();
        if (otherArgs.length != 3) {
            System.err.println("Usage: KMeans <in> <out> <clusters_number>");
            System.exit(2);
        }
        int centroidsNumber = Integer.parseInt(otherArgs[2]);
        configuration.setInt(Constants.CENTROID_NUMBER_ARG, centroidsNumber);
        configuration.set(Constants.INPUT_FILE, otherArgs[0]);

        List<Double[]> centroids = Utils.createRandomCentroids(centroidsNumber);
        String centroidsFile = Utils.getFormattedCentroids(centroids);
        Utils.writeCentroids(configuration, centroidsFile);

        boolean hasConverged = false;
        int iteration = 0;
        do {

            configuration.set(Constants.OUTPUT_FILE, otherArgs[1] + "-" + iteration);
            if (!launchJob(configuration)) {
                System.exit(1);
            }
            String newCentroids = Utils.readReducerOutput(configuration);
            if (centroidsFile.equals(newCentroids)) {
                hasConverged = true;
            }
            else {
                Utils.writeCentroids(configuration, newCentroids);
            }
            centroidsFile = newCentroids;
            iteration++;

        } while (!hasConverged);

        writeFinalData(configuration, Utils.getCentroids(centroidsFile));
    }
```

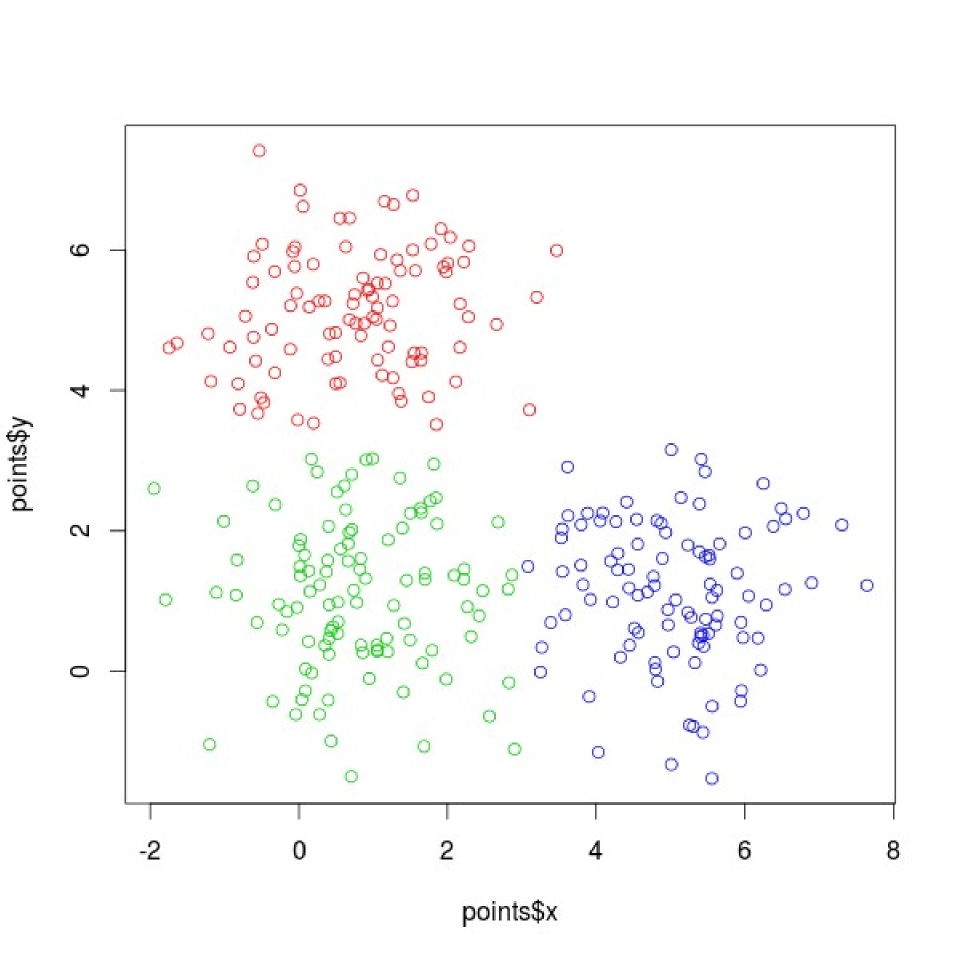
K-means driver - 2

```
private static boolean launchJob(Configuration config) {

    Job job = Job.getInstance(config);
    job.setJobName("KMeans");
    job.setJarByClass(KMeans.class);

    job.setMapperClass(KMeansMapper.class);
    job.setReducerClass(KMeansReducer.class);

    job.setMapOutputKeyClass(IntWritable.class);
    job.setMapOutputValueClass(Text.class);

    job.setNumReduceTasks(1);

    job.addCacheFile(new Path(Constants.CENTROIDS_FILE).toUri());

    FileInputFormat.addInputPath(job, new Path(config.get(Constants.INPUT_FILE)));
    FileOutputFormat.setOutputPath(job, new Path(config.get(Constants.OUTPUT_FILE)));

    return job.waitForCompletion(true);
}
```

K-means

Results:

```
 4.5700  0.5510   2
 4.5179  0.6120   2
 4.1978  1.5706   2
 5.2358  1.7982   2
 1.747   3.9052   0
 1.0445  5.0108   0
-0.6105  4.7576   0
 0.7108  2.8032   1
 1.3450  3.9558   0
 1.2272  4.9238   0
 ...
```



R code to plot data:

```
points <- read.csv(file="final-data", sep="\t", header=0)
colnames(points)[1] <- "x"
colnames(points)[2] <- "y"
plot(points$x, points$y, col= points$V3+2)
```

Hints

- Use MapReduce only if you have really big data: SQL or scripting are less expensive in terms of time needed to obtain the same results

- Use a lot of defensive checks: when we have a lot of data, we don't want the computation to be stopped by a trivial NPE :-)

- Testing can save a lot of time!

Thanks!

The code is available on:
https://github.com/andreaiacono/MapReduce

Take a look at my blog:
https://andreaiacono.blogspot.com/