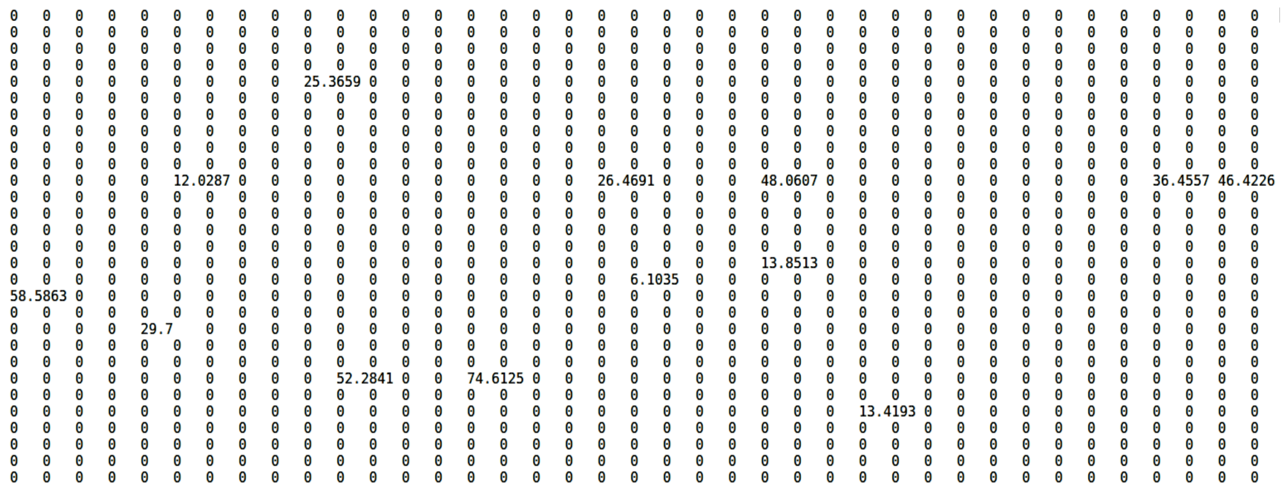

```
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   25.3659 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   12.0287 0   0   0   0   0   0   0   0   0   0   0   26.4691 0   0   0   48.0607 0   0   0   0   0   0   0   0   0   0   36.4557 46.4226
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   13.8513 0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   6.1035  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
58.5863 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   29.7    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   52.2841 0   0   74.6125 0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   13.4193 0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Map Reduce Vector Normalization

Matrix p-norm

Course Enabling Distributed Platforms
Final project

Author Valeriya Slovikovskaya
Matricola 503602

I. Map Reduce Vector Normalization

The purpose of this student project is to get familiar with Hadoop MapReduce framework, open source implementation of the MapReduce programming model, widely used by leading IT companies (today they are Amazon, Adobe, Ebay, Facebook, Google, LinkedIn, Spotify, Yahoo!, and many others) since it's first release in 2006.

We 1.) implement vectors normalization as MapReduce application, 2.) generate random test data implementing random matrix generator as MapReduce application, 3.) create docker based test environment to run our application on.

Hadoop MapReduce framework is open source implementation of the MapReduce programming model which is written in Java and provides a «streaming» interface to interact with user code over Unix pipes. Lin and Dyer [1] introduced design patterns that can be used to simplify and improve the performance of MapReduce algorithms.

1. Vector normalization MapReduce application

In our first toy application we use *order inversion* pattern (referenced and explained in [2, 3]) for vector normalization task. This task can be faced, for example, as a video processing sub problem, when a whole video sequence have to be normalized. The image sequence is viewed as a matrix with rows (or vectors) going through all frames, one per pixel position within each frame, with frames considered to be a single vector dimension (or matrix column), and, in this case, the number of dimensions is much larger than number of vectors (matrix rows) equal to number of pixels in a frame.

Vector normalization task involves using an aggregate statistic, namely, - determining maximum and minimum vector values, as intermediate calculations. Another well known example of this kind of problem is a background subtraction task in computer vision[1]: in the first processing pass values of mean and variance are calculated for each pixel location in each block of frames in a sequence, then, in the second pass, this values are used to calculate z-scores and classify every pixel as background or foreground one. In both cases reducers needs to read the same input values twice (or, in general, multiple times) to perform the computation.

When mapper emits only once for each vector dimension with the key being the dimension and the value being a tuple of the vector's id and the value for the dimension, reducer reads all input data to find the minimum and maximum values and then it gets unable to proceed with producing the desired output because only a forward iterator is provided to the key/value pairs.

In their paper White et al. indicate the possible solutions, such as to buffer the data in memory and pass over it again (this will not scale and eventually the system memory will be exhausted), or, alternatively, to compute the min/max values in the first pass and have another job that computes the final output, loading the min/max values as side data from a shared location. The later scales but the order inversion design pattern, when applied to this particular case, brings further improvements.

Map Input (vecid, <dim_0,dim_1>>)	Map Output (<dim, flag>, <vecid, val>)	Reduce Input (<dim,flag>,[<$vecid_0$,val_0>,...])	Reduce Output (vecid,<dim,val>)
(0, <9,6>)	(<0,0>,<0,9>) (<0,1>,<0,9>) (<1,0>,<0,6>) (<1,1>,<0,6>)	(<0,0>,[<0,9>,<1,0>,<2,1>,<3,3>]) (<0,1>,[<0,9>,<1,0>,<2,1>,<3,3>])	(0,<0,1.>) (1,<0,0.>) (2,<0,0.1111>) (3,<0,0.3333>)
(1,<0,1>)	(<0,0>,<1,0>) (<0,1>,<1,0>) (<1,0>,<1,1>) (<1,1>,<1,1>)	(<1,0>,[<0,6>,<1,1>,<2,0>,<3,6>]) (<1,1>,[<0,6>,<1,1>,<2,0>,<3,6>])	(0,<1,1.>) (1,<1,0.1667>) (2,<1,0.>) (3,<1,1.>)
(2,<1,0>)	(<0,0>,<2,1>) (<0,1>,<2,1>) (<1,0>,<2,0>) (<1,1>,<2,0>)		
(3,<3,6>)	(<0,0>,<3,3>) (<0,1>,<3,3>) (<1,0>,<3,6>) (<1,1>,<3,6>)		

Figure 1. Example input and output when normalizing a set of vectors' values using the *order inversion* design pattern [2].

1. Background subtraction is a successful method of segmenting objects of interest in a surveillance setting.

```
class Mapper
    method Map(vecid i, vector V)
        for all <dim d, val v> Є vector V do
            $t \leftarrow$ <vecid i, val v>
            Emit(tuple <dim d, flag 0>, tuple t)
            Emit(tuple <dim d, flag 1>, tuple t)
class Reducer
    method Configure()
        $m \leftarrow M \leftarrow p \leftarrow 0$
    method Reduce(tuple <dim d, flag f>, tuples)
        if $p \neq d$ then
            # Reset extrema for new dimension
            $m \leftarrow \infty$
            $M \leftarrow -\infty$
            $p \leftarrow d$
        if $f = 0$ then
            for all tuple <vecid i, val v> Є tuples do
                UpdateExtrema(val m, val M, val v)
            else
                for all tuple <vecid i, val v) Є tuples do
                    $v \leftarrow (v - m)/(M - m)$
                    Emit(vecid i, tuple <dim d, val v>)
```

Algorithm 1. MapReduce algorithm for normalizing vectors using the *order inversion* design pattern [2].

A *single job* is enough if **mapper** is modified to emit the same value twice with a new key, that includes not only vector dimension (or matrix column index) but also the flag (0 or 1) indicating the «stage» of processing the same value (first or second). The sort is performed on the dimension first and the flag second.

This change in mapper leads to change in a **partitioner** that has to only partition based on the dimension, ignoring the flag, so that all data for a specific dimension, ordered by the flag, is sent to the same reducer. Thus, for **reducer** the flag = 0 indicates that it needs to find the min/max values, and flag = 1, that immediately follows, signals to normalize the vector values. Reducer's instance variables are used to hold the state between flag values, as the grouping is performed on the entire key. Figure 1 shows a step-by-step example of matrix normalization, and map/reduce algorithm reported in pseudo code as Algorithm 1.

Our **driver** code reads input and output paths from the command line arguments. It creates a MapReduce job and assigns it our custom Mapper, Partitioner and Reducer.

We define **custom key/value output format classes** to be output formats for our Mapper. MapperKey is a WritableComparable object with matrix index, column index and flag as properties to identify and sort input key/value pair by. MapperValue is the Writable with matrix index, row index and column value provided as the object properties.

Mapper's map method has as an input value a string representing a serialized matrix, read from the files in input path. Mapper deserializes it, assigns to it the ordinal number as an unique id, and then emits two key/value pairs (<matrix id, column id, flag>, <matrix id, row id, column value>) differing only by flag.

Partitioner (ColumnIndexPartitioner) does not consider the flag, so that both emitted key/value pairs with the same matrix id and column id lend on the same reducer.

Reducer 's (MatrixNormReducer) reduce method takes as an input an array of one-column values in the form <matrix id, row id, column value> grouped by key <matrix id, column id, flag>. It determines minimum and maximum values for input pair with flag = 0, keeps this values in the instance variables, uses them to normalize column values for the following input with flag = 1.

Than it does some “in reducer” **combiner** work: it keeps all the calculated values for currently processed matrix in a TreeMap till the whole matrix being processed (note, that reducer does not know how many columns the matrix contains), once it is processed (next arrived key/value contains different matrix id) reducer transforms the data stored in TreeMap (where column values arrays are indexed by column id) into two dimensional array of doubles, serializes it, and emits this string as an output value. The last matrix stored in TreeMap is emitted by reducer's cleanup method.

At the end of the application run we have the output text files with each string representing the matrix with values in interval from 0 to 1.

2. Generating test data

To test the vector normalization application it is necessary to provide the stream of input data. For this purpose we use Hadoop/Mapreduce to run a Map job that outputs the set of randomly generated matrices. Each matrix is serialized as a string of numbers separated with tabs, where first two position of the string indicate matrix dimensions, and the others are the matrix values, read from left to right, from top to bottom.

We implement the tricky *data generator* pattern as described in O’Reilly MapReduce design patterns book [3], creating a FakeInputSplit, a custom MatrixGenInputFormat and letting a custom MatrixGenRecordReader generate the random data. We use Scalacheck v 1.13.3 (www.Scalacheck.org) Scala library to provide the stream of double numbers in the defined range [4].

Hadoop allows to modify the way data is loaded on disk in two major ways: configuring how contiguous chunks of input are generated from blocks in Hdfs, and configuring how records appear in the map phase. The two classes to be playing with are RecordReader and InputFormat.

Hadoop relies on the input format of the job to do three things: 1.) validate the input configuration for the job (i.e., checking that the data is there), 2.) split the input blocks and files into logical chunks of type InputSplit, each of which is assigned to a map task for processing, 3.) create the RecordReader implementation to be used to create key/value pairs from the raw InputSplit , these pairs are sent one by one to their mapper. The most common input formats are subclasses of FileInputFormat, with the Hadoop default being TextInputFormat .

The RecordReader abstract class creates key/value pairs from a given InputSplit . While the InputSplit represents the byte-oriented view of the split, the RecordReader makes sense out of it for processing by a mapper. It is in the RecordReader that the schema is defined, based solely on the record reader implementation, which changes based on what the expected input is for the job. Bytes are read from the input source and turned into a Writable Comparable key and a Writable value. Custom data types are very common when creating custom input formats, as they are a nice object-oriented way to present information to a mapper.

The implementation of random data generator isn't straightforward in Hadoop because one of the fundamental pieces of the framework is assigning one map task to an input split and assigning one map function call to one record. In this case, there are no input splits and there are no records, so it is necessary to fool the framework to think there are (as our FakeInputSplit does).
The **driver** (MatrixGenDriver) takes from the command line arguments 1.) number of map tasks, 2.) number of records per tasks, and 3.) output directory. It sets up map job assigning to it this parameters as well as custom input format class. All the output is written to the given output directory. The identity mapper is used for this job, and the reduce phase is disabled by setting the number of reduce tasks to zero.

The Fake**InputSplit** class simply extends InputSplit and implements Writable . There is no implementation for any of the overridden methods. This input split is used to trick the framework into assigning a task to generate the random data.

The **input format** has two main purposes: 1.) returning the list of input splits for the framework to generate map tasks from, and then 2.) creating the Random MatrixGenRecordReader for the map task. We override the getSplits method to return a configured number of FakeInputSplit splits. This number is pulled from the configuration. When the framework calls createRecordReader , a RandomStackOverflowRecordReader is instantiated, initialized, and returned.

The **record reader** is where the data is actually generated. The number of records to create is pulled from the job configuration. For each call to nextKeyValue method of our custom RecordReader, the next matrix in form if two dimensional array of doubles is received from our Scalacheck matrix generator and gets serialized. The counter is incremented to keep track of how many matrices have been generated. Once all the matrices are generated, the record reader returns false, signaling to the framework that there is no more input for the mapper (Figure 2).

The **map phase** remains completely unaware that tricky things are going on before it gets its input pairs.

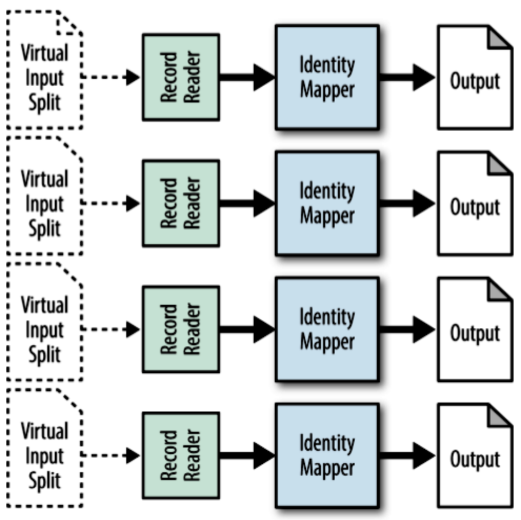

Figure 2. The structure of the generating data pattern [2].

II. Matrix p-norm

Our second MapReduce application is matrix p-norm calculated in parallel with Hadoop MapReduce.

In mathematics, a matrix norm is a natural extension of the notion of a vector norm to matrices. "Entry wise" matrix norms, that we consider in this paper, treats an *m x n* matrix as a vector of size *m x n*, and use one of the familiar vector norms. Generalizing the *p-norm* for vectors, $p \geq 1$, we get:

$$\|A\|_p = \|\mathbf{vec}(A)\|_p = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^p \right)^{1/p}$$

The special case $p = 2$ is the Frobenius norm, that can be also defined as:

$$\|A\|_{\mathrm{F}} = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2} = \sqrt{\mathbf{trace}(A^{\dagger} A)}$$

Here A^t denotes the conjugate transpose of *A*, the *n-by-m* matrix obtained from *A* by taking the transpose and then taking the complex conjugate of each entry. And the trace function returns the sum of diagonal entries of a square matrix.

Matrix norm is of particular interest because it is strictly related with the *condition number*, the measure that in many matrix problems provides information about the sensitivity of the solution to perturbations in the data. The most well-known example is the linear equation problem, for which various perturbation bounds involving condition number are available [5].

We limit ourselves to calculate *p-norm* (in very naive manner) for real valued matrices and for finite $p > 0$ and, separately, the Frobenius norm by calculating A^TA matrix values, this time in naive way as well as using oprimized *DIMSUM* algorithm.

We use the results of Zadeh and G. Carlsson [6] who extended MapReduce utility by proposing a MapReduce based method of A^TA calculation for very large matrices that cannot be stored or even streamed through a single machine. This is notably the case of social network-related matrices, in which case the number of individuals is large compared to the number of observed features.
As an example, they considered the case of the *m x n* sparse matrices (with at most *L* nonzeros per row, say $L=20$) with very large *m* (for example, $m = 10^{13}$) and relatively small *n* ($n = 10^4$), that should be segmented in rows to be kept in storage distributed over many machines.

NaiveMapper (r_i)	NaiveReducer $((c_i, c_j), <v_1, . . . ,v_R>)$

for all pairs (a_{ij}, a_{ik}) in r_i do
Emit $((c_j, c_k) \rightarrow a_{ij} a_{ik})$
end for
	output $c_i^T c_j \rightarrow \Sigma_{i=1} v_i$

Algorithm 2. A naive implementation of A^TA calculus in a MapReduce framework would be to simply form all dot products between columns.

```
DIMSUMMapper ($r_i$)

for all pairs ($a_{ij}$, $a_{ik}$) in $r_i$ do
    with probability $min(1, \gamma/(||c_j|| \, ||c_k||))$
    emit $((c_j, c_k) \to a_{ij} a_{ik})$
end for
```

```
DIMSUMReducer ($((c_i, c_j), <v_1, . . . ,v_R>)$)

if $\gamma/(||c_j|| \, ||c_k||) > 1$ then
    output $b_{ij} \to \Sigma_{i=1} v_i \; / \; (||c_j|| \; ||c_k||)$
else
    output $b_{ij} \to \Sigma_{i=1} v_i \; / \; \gamma$
end if
```

Algorithm 3. DIMSUM calculus of A^TA[2]. Here $\gamma = \Omega(n/\varepsilon^2)$ is the oversampling parameter, it should be set greater than 1 to guarantee the correctness of calculations, proposed guideline is $\gamma = 2 \log(n)$

Zadeh and Carlsson discuss two main complexity measures applicable to the map reduce problem: *shuffle size* and *reduce-key complexity*. These complexity measures together capture the bottlenecks when handling data on multiple machines: first we can't have too much communication between machines, and second we can't overload a single machine. The number of emissions in the map phase is called the *shuffle size*, since that data needs to be shuffled around the network to reach the correct reducer. The maximum number of items reduced to a single key is called the reduce-key complexity and measures how overloaded a single machine may become.

Then authors reason that naive approach for computing A^TA (Algorithm 2) will have $O(mL^2)$ emissions, which for $m = 10^{13}$, $n = 10^4$, $L = 20$ is infeasible. Furthermore, the maximum number of items reduced to a single key can be as large as m. Thus the *reduce-key complexity* for the naive scheme is m.

To solve the problem they propose a non-adaptive sampling technique that picks only that rows and columns of A whose similarity is above a certain threshold. This way the singular values of A are estimated within relative error ε with constant probability.

The complexity measures of this approach are crucially improved: the shuffle size $O(n^2/\varepsilon^2)$ and reduce-key complexity $O(n/\varepsilon^2)$. This bounds are independent of m, the larger matrix dimension (Algorithm 3).

We implement both methods. Our MapReduce application consists of 1.) matrix row generators that generated random matrix saved in file row by row; 2.) trivial Hadoop MapReduce matrix *p-norm* implementation with mapper applying power operation to non-zero matrix values, combiner locally summing values by column, and reducer summing them by row and performing final power operation; 3.) naive Frobenius norm calculus implementation with mapper and reducer the same as in Algorithm 1 with the only difference that reducer applies also trace and squared root operation on A^TA matrix to output the norm; 4.) Hadoop MapReduce[3] implementation of DIMSUM algorithm of Frobenius norm calculus.

The later is composed of two MapReduce jobs: first calculating the euclidean norm of each column $||c_j||$, second loading this data and calculating Frobenius norm according to Algorithm 3.

2 The magnitudes of each column is assumed to be loaded into memory and available to both the mappers and reducers. The magnitudes of each column are natural values to have computed already, or can be computed with a trivial mapreduce. It's also assumed the entries of A have been scaled to be in [−1, 1], which can be done with little communication by finding the largest magnitude element.

3 Nowadays a great variety of projects are leaning towards multi-paradigm programming languages such as Scala, or recent solutions such as Spark.

Series of unit tests is provided to assure the implementation correctness.

III. Test environment

All applications within the Hadoop ecosystem (like Hadoop, Hive, HBase, Spark, etc.) are made available as architecture specific deb/rpm packages by Apache Bigtop project (bigtop.apache.org)[4]. Along with comprehensive packaging, testing, and configuration of the components, Bigtop provides vagrant recipes, raw images, and docker recipes for deploying Hadoop from zero. It supports many operating systems, including Debian, Ubuntu, CentOS, RedHead, Fedora, and openSUSE.

Considering the fact that nowadays 1.) build environments is commonly shipped in docker images[5], 2.) docker containers are provisioned in fully automated way (with Puppet, Chef or Ansible provisioning systems), 3.) and applications are deployed in automated and unified process of continuous integration, targeting their development, test and production environments, we tried to get acquainted with Bigtop working examples (github.com/apache/bigtop) and to repeat the steps of creating a "puppetized" docker images, then docker images with building tools as Jdk, Apache Maven, gradle, wget, tar, git, etc provided, then running provision script that spun up several docker containers and applies puppet recipes for each of it, installing and configuring Hdfs, yarn and mapreduce application components.

We landed up with minimalist version of Bigtop docker images[6] and provision system including only Hdfs – Yarn – MapReduce (no Hive, Pig, Crunch, Mahut, ets, no Kerberos for major simplicity of experiment). We used johnrengelman.shadow plugin (version 1.2.3) to build a fat jar including Scala and Scalacheck libraries. With build/provisioning shell script we spun up the Hadoop cluster of three docker container and run, first, our matrix generator, than our vector normalization application. (See project Github repository github.com/vslovik/hadoop_mapreduce_vectors_normalization).

In the end we were able to run our code in the cluster, on home machine, using the same tools that can be used in production.

Along with application code and modified (simplified) Bigtop provisioning script, we wrote a series of automated tests for Mapper, Partitioner, Reducer and Record Reader to exercise "full-featured" application development work flow.

References

1. Lin, J., Dyer, C.: Data-intensive text processing with MapReduce. Morgan & Claypool Publishers (2010)

4 Bigtop packages the entire upstream Hadoop ecosystem. It does this, in general, by building the same jars which come from Hadoop distributions : without patching them - and converting them to rpm/deb packages. That means the Hadoop tarballs get split out into a Linux friendly package structure, i.e. (/usr/lib/Hadoop/, /etc/Hadoop/conf, and so on ...).

5 Three key techniques in Docker: 1.) use of Linux Kernel's name spaces to create isolated resources, 2.) use of Linux Kernel's cgroups to constrain resource usage, 3.) union file system that supports git-like image creation. As a result systems based on docker containers are lightweight and portable.

6 vslovik/slaves:centos-7 and vslovik/slaves:ubuntu-16.04 on DockerHub

2. White, B., Yeh, T., Lin, J., Davis, L.: Web-scale computer vision using MapReduce for multimedia data mining. In: Proceedings of the Tenth International Workshop on Multimedia Data Mining, pp. 1-10 (2010).

3. Miner, D., Shook, A.: MapReduce Design Patterns. O'Reilly (2013)

4. Scala exercises. Generators. https://www.Scala-exercises.org/Scalacheck/generators

5. Higham, Nicholas J. (1987) A Survey of Condition Number Estimation for Triangular Matrices. SIAM Review, 29 (4). pp. 575-596. ISSN 0036-1445

6. Zadeh, R.B., Carlsson, G. (2013) Dimension Independent Matrix Square Using MapReduce (DIMSUM). Symposium on Theory of Computing, 2013.