A Comparison of Hadoop Tools for Analyzing Tabular Data

Ivan Tomašić, Aleksandra Rashkovska, Matjaž Depolli and Roman Trobec
Jožef Stefan Institute, Slovenia
E-mail: ivan.tomasic@ijs.si, aleksandra.rashkovska@ijs.si, matjaz.depolli@ijs.si, roman.trobec@ijs.si

Keywords: Hadoop, MapReduce, Pig, Hive, BigData

Received: December 24, 2012

The paper describes the application of Hadoop modules: MapReduce, Pig and Hive, for processing and analyzing large amounts of tabular data acquired from a computer simulation of heat transfer in bio tissues. The Apache Hadoop is an open source environment for storing and analyzing BigData. It was installed on a cluster of six computing nodes, each with four cores. The implemented MapReduce job pipeline is described and the essential Java code segments are presented. The Java implementation employing MapReduce is compared to the Pig and Hive implementations regarding execution time and programming overhead. The experimental measurements of execution times of the employed parallel MapReduce tasks on 24 processor cores result in a speedup of 20, relative to the sequential execution, which indicates that a high level of parallelism is achieved. Furthermore, our test cases confirm that the direct employment of MapReduce in Java outperforms Pig and Hive by more than two times, while Hive being 20% faster than Pig. Still, Pig and Hive remain suitable and convenient alternatives for efficient operations on large data sets.

Povzetek: Prispevek opisuje uporabo Hadoop programskih modulov: MapReduce, Pig in Hive za procesiranje in analizo tabelaričnih podatkov o prenosu toplote v tkivih.

1 Introduction

Since 2004, when the famous publication “MapReduce: Simplified Data Processing on Large Clusters” [1] was published from the Google’s team, the MapReduce paradigm has become one of the most popular tools for processing large datasets, mostly because it allows users to build complex distributed programs using a very simple model.

Apache Hadoop [2] is a highly popular set of open source modules for distributed computing, developed initially to support distribution for the Nutch search engine project. One of the key Hadoop components is the MapReduce on which the other, higher-level Hadoop-related components rely, e.g., Pig and Hive.

With the increasing popularity of the MapReduce and other non-relational data processing approaches, it became apparent that they can be used to construct efficient computing infrastructures. Furthermore, the Hadoop has proved its ability to store and analyze huge datasets often referred to as the BigData [3]. It is used by Yahoo and Facebook for their batch processing needs. Hadoop and Hive are among cornerstones of the storage and analytics infrastructure at Facebook [4]. Facebook Message, in particular, is the first ever user-facing application built on the Apache Hadoop platform [5].

The MapReduce can be seen as a complement to the parallel Relational Database Management System (RDBMS). It is a common opinion these days that the MapReduce is more suitable for batch processing analyzes of whole datasets and for applications where data is written once and read many times, whereas the RDBMS is better for databases that are continuously updated.

In this paper, we investigate the differences in approaches, performances, and usability, between MapReduce, Pig, and Hive Hadoop tools. Their performances were compared in the analysis of tabular simulation data of heat transfer in a biomedical application, in particular, cooling of a human knee after surgery [6]. Similar data sources can be found also in other scientific areas related to multi-parametric simulations [7], environmental data analysis [8], high energy physics [9], bioinformatics [10] etc., whereas some special problems may benefit from specific interfaces to the Hadoop [11].

2 Description of utilized Hadoop modules

Hadoop is composed of four modules:

- Common: support for other Hadoop modules,
- Hadoop Distributed file System (HDFS),
- YARN: a framework for job scheduling and cluster resource management, and
- MapReduce.

There is a number of Hadoop-related projects, but the ones most relevant to our data analyses are Pig and Hive.

2.1 Map/Reduce paradigm

MapReduce is a programming model and an associated implementation for processing and generating large data sets [1]. Some problems that can be simply solved by MapReduce are: distributed grep, count of URL access frequency, various representations of the graph structure of web documents, term-vector per host, inverted index, etc.

A MapReduce program execution consists of the four basic steps: (1) splitting the input, (2) iterating over each split and computing (key, value) pairs (parallel for each split), (3) grouping intermediate values by keys, (4) iterating over values associated with each unique key (in parallel for different keys), computing (usually reducing values for a given key) and outputting final (key, value) pairs.

The first step is done by the MapReduce framework, whereas for the second step a user provides a Map function, which is applied by the framework, commonly on each line of every split. Each Map function invocation outputs a list of (key, value) pairs. Note that each split is generally processed on different processor cores and machines in parallel.

As a simple example, let's consider the task of counting the number of occurrences for each word in a document. The Map function will count the number of occurrences of each word in a line and output a list of (key, value) pairs, for each line:

$$\{(\mathrm{word\_1}, \mathrm{num\_1_i}), (\mathrm{word\_2}, \mathrm{num\_2_i}), \ldots\},$$

where i is the line index.

The MapReduce framework groups together all intermediate values associated with the same intermediate key (step 3). The resulting (key, values) pairs are one by one sent to the user-specified Reduce function which aggregates or merges together the values to form a new, possibly smaller, set of values (step 4). In our example the Reduce function will accept each unique word, as a key, and the numbers of their occurrences in each line, as values, sum the numbers of occurrences and output one (key, value) pair per word:

$$(word\_N, \mathrm{sum}\{num\_N_1, num\_N_2, \ldots, num\_N_i, \ldots\}).$$

The executions of the Map and Reduce functions are referred to as Map and Reduce tasks. A set of tasks executed for one application are referred to as a MapReduce job.

The main limitation of the MapReduce paradigm is that each Map and Reduce task must not depend on any data generated in other Map or Reduce tasks of the current job, as user cannot control the order in which the tasks execute. Consequently, the MapReduce is not directly applicable to recursive computations, and algorithms that depend on shared global state, like online learning and Monte Carlo simulations [12].

The MapReduce, as a paradigm, has different implementations. In the presented work, we have used MapReduce implemented in Apache Hadoop distributed in Cloudera [13]. A convenient comparison between MapReduce implementations is presented in [14].

2.2 Apache Hadoop MapReduce implementation

The splitting is introduced because it enables data processing scalability, which shortens the time needed to process the entire input data. The parallel processing can be better load-balanced if the splits are small. However, if the splits are too small, then the time needed to manage the splits and the time for the Map task creation may begin to dominate the total job execution time.

Hadoop splits are fixed-size, whereas a separate Map task is created for each split (Figure 1). The default Hadoop MapReduce split size is the same as the default size of an HDFS block, which is 64 MB. Hadoop performs data locality optimization by running the Map task on the node where the input data resides in the HDFS. With the default HDFS replication factor of three, files are concurrently stored on three nodes; hence, splits of the same file can be concurrently processed on three nodes without the need for being copied before.

In the Hadoop implementation, the Map tasks write their outputs to their local disks, not to the HDFS and are therefore not replicated. If an error happens on a node running a Map task before its output has been consumed by a Reduce task, then the Hadoop resolves the error by re-running the corrupted Map task on another node.

The Map tasks partition their outputs, creating one partition for each Reduce task (Figure 1 – each Map creates r output partitions). Each partition may contain

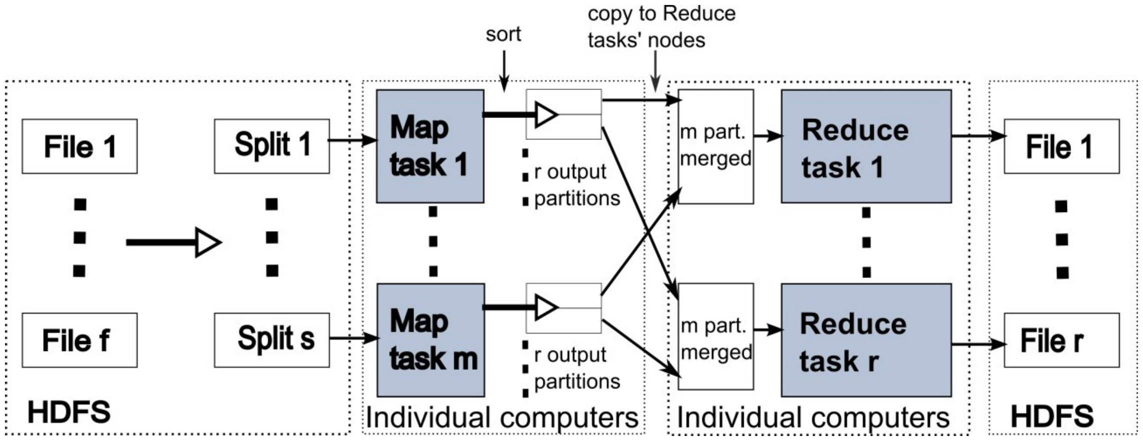

Figure 1: Schematic representation of Hadoop's MapReduce data flow (without using Combiners).

various keys and associated values. All records sharing the same key are processed by the same Reduce task. This is achieved by using the so-called Partitioner function. The default Hadoop MapReduce Partitioner employs a hash function on the keys from the Maps' outputs. Modulo function by the number of reducers is subsequently applied to the hash values resulting in the Reduce task indexes for each key.

The data flow between Map and Reduce tasks is colloquially known as "shuffle". The inputs for each Reduce task are pulled from the machines where the Map tasks ran. The input to a single Reduce task is generally formed from outputs of multiple Map tasks (Figure 1 – each reduce task receives *m* partitions, where *m* is the number of Map tasks); therefore Reduce tasks cannot convey on data locality. On nodes running Reduce tasks, the sorted map outputs are merged before being passed to a Reduce task. The number of Reduce tasks is specified independently for a given job. Each Reduce task outputs a single file, which is usually stored in the HDFS (Figure 1).

Hadoop allows a user to specify an additional so-called Combiner function, which can be executed on each node that runs Map tasks. It receives all the data emitted by the Map tasks on the same node as an input and forms the output that is further processed in the same way as the direct output from a Map task would be. The Combiner function may achieve data reduction on a node level, consequently minimizing data transfer over the network between the machines executing Map and Reduce tasks. The use of Combiner functions reduces the impact of the limited communication bandwidth on the performances of a MapReduce job. The Combiner function code is usually the same as the Reduce function.

2.3 Pig

The development cycle of a MapReduce program may be quite long. Furthermore, it requires an experienced programmer that knows how to describe a given data processing task as a set of MapReduce jobs.

Pig is a sequential language, called Pig Latin, which expresses operation on data, together with execution environment that runs Pig Latin programs [15]. A Pig Latin program comprises a series of high level data operations, translated to the MapReduce jobs that can be executed on a Hadoop cluster. Pig is designed to reduce programming time by providing a higher level procedural utilization of the MapReduce infrastructure. It allows a programmer to concentrate on the data rather than on the details of execution.

Pig runs as a client-side application and has an interactive shell named Grunt used for running Pig Latin programs.

2.4 Hive

A programmer familiar with SQL language may prefer to describe data operations with SQL language, even if the data is not stored in a RDBMS. Hive is Hadoop's data warehouse system that provides mechanism to project structure onto data stored in HDFS or a compatible file system [16]. It provides a SQL-like language called HiveQL. It does not support the full SQL-92 specification, but provides some extensions that are consequences of the MapReduce infrastructure supporting each Hive query. The primary way of interacting with Hive is the Hive shell used to insert and execute HiveQL instructions.

Like RDBMS, Hive stores data in tables. When the tables are loaded with data, Hive stores them in its warehouse directory [17]. Before execution, usually when the select statement is called, Hive, like Pig, transforms the instructions to a set of MapReduce jobs executed on a Hadoop cluster.

The most significant difference between Hive and Pig is that Pig Latin is a procedural programming language, whereas HiveQL is a declarative programming language. A Pig Latin program is a sequential list of operations on an input relation, in which each step is a single transformation. On the other hand, HiveQL is a language based on constraints that, when taken together, define a data operation.

3 Analyzing simulation data

3.1 Description of the Hadoop cluster

The Apache Hadoop open source Cloudera distribution was installed on a cluster built of six computing nodes. The nodes are connected with Gigabit Ethernet. Each node has a quad-core Intel Xeon 5520 processor, 6 GB of RAM and 500 GB hard disk. All nodes run 64-bit Ubuntu Server 12.04 operating system.
One of the nodes is designated as the namenode while others are the datanodes. The namenode also hosts the jobtracker. All machines in the cluster run an instance of a datanode and a tasktracker. For a description of the HDFS and MapReduce nodes please refer to [18, 19].

3.2 Input data

The computer simulation of two hours cooling of a human knee after surgery is performed for 10 different knee sizes, 10 different initial temperature states before cooling, and 10 different temperatures of the cooling pad. This results in 1000 simulation cases. The results of those simulation cases are gathered in 100 files, each for one knee size and one initial state, and for all cooling

CASE	*Parameters*
1	T1
2	T1-T5
3	T1,T6,T11,T16,T21
4	T1-T21
5	T1,T6,T11,T16,T21,T46,T51,T56,T61
6	T1-T21,T46-T61
7	T1,T6,T11,T16,T21,T26,T31,T36,T41,T46,T51, T56,T61,T66,T71,T76,T81
8	T1-T85

Table 1: List of test cases.

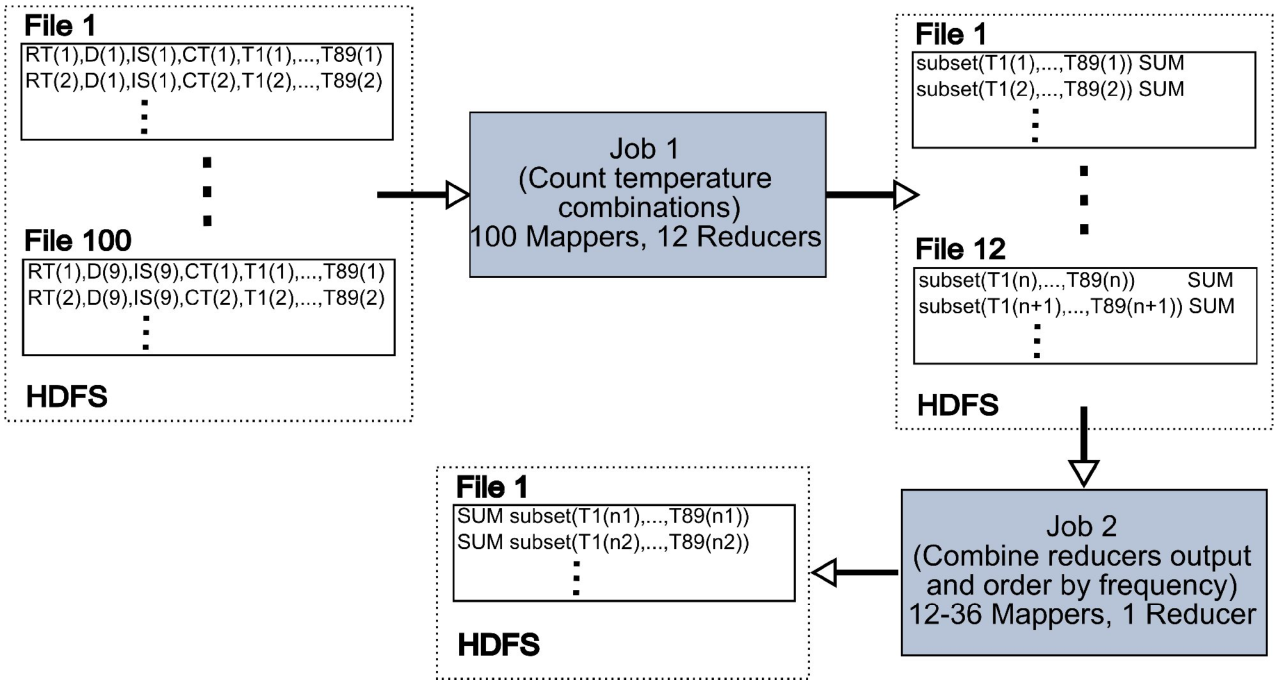
File 1
RT(1),D(1),IS(1),CT(1),T1(1),...,T89(1)
RT(2),D(1),IS(1),CT(2),T1(2),...,T89(2)
File 100
RT(1),D(9),IS(9),CT(1),T1(1),...,T89(1)
RT(2),D(9),IS(9),CT(2),T1(2),...,T89(2)
HDFS
Job 1
(Count temperature combinations)
100 Mappers, 12 Reducers
File 1
subset(T1(1),...,T89(1)) SUM
subset(T1(2),...,T89(2)) SUM
File 12
subset(T1(n),...,T89(n)) SUM
subset(T1(n+1),...,T89(n+1)) SUM
HDFS
Job 2
(Combine reducers output and order by frequency)
12-36 Mappers, 1 Reducer
File 1
SUM subset(T1(n1),...,T89(n1))
SUM subset(T1(n2),...,T89(n2))
HDFS

Figure 2: MapReduce jobs pipeline.

temperatures. Each file contains 71970 rows or approximately 44 MB of data. Each data row is composed of the following parameters, i.e., columns: RT, D, IS, CT, T1, T2, … , T85, where are: RT - relative time in a simulation case, D - knee size, IS –initial state, CT – cooling temperature, T1-T85 – inner and outer knee temperatures, i.e., temperatures at a particular location in the knee center, 8 locations on the knee skin and 8 respective locations under the cooling pad, all taken in the current and in previous time steps. In order to assess the periodicities in the knee simulation results, we demand from the MapReduce to count the occurrences of the same value arrays for a subset of knee temperatures T, more precisely, to count the occurrences of identical rows after having projected only columns of T that are of interest. For the SQL code of this operation please refer to the code in Figure 5. We will refer to, in the rest of the paper, the number of occurrences of identical rows as temperature frequencies.

We defined and examined 8 cases with different sets of T. The cases are given in Table 1. Cases with odd numbering take only the current values for the temperatures: Case 1 – the knee center; Case 3 – the knee center and 4 locations on the knee skin; Case 5 – the knee center, 4 locations on the knee skin, and 4 respective locations under the cooling pad; Case 7 – all current temperatures. The cases with even numbering incorporate denoted temperatures T and their value in 4 previous time steps, e.g., in Case 2, T1-T5 represents five temperature values at time steps t_i, t_{i-1}, t_{i-2}, t_{i-3}, t_{i-4}, for each of T from T1-T5, etc.

3.3 MapReduce

The MapReduce jobs pipeline, used for solving our test cases, is illustrated in Figure 2. The sizes of the input files are smaller than the HDFS block size (in our case: 64 MB). Hence, the number of input Map tasks in Job 1 is equal to the number of input files [20] (in our case: 100), i.e., each input file is processed by a different Map task and no additional splitting is performed. Because the number of Reduce tasks is not explicitly set for Job 1, it becomes, by default, equal to the number of task tracker nodes (in our case: 6), multiplied by the value of the *mapred.tasktracker.reduce.tasks.maximum* configuration property [20] (in our case: 2). The output of Job 1 consists therefore of 12 files. Each file contains a unique combination of temperatures and the number of their occurrences. Job 2 combines Reduce tasks' outputs from Job 1 into a single file (in Job 2, the number of Reduce tasks is explicitly set to 1). It also sorts the input columns in the output file by temperature frequencies. The number of Map tasks in Job 2 depends on the test case (Table 1) and varies between 12 for Case 1 and 36 for Case 8 as the amount of data emitted by Job 1 increases with the case number. The details of the jobs implementations are given in Figure 3 and the following text.

In the Map function of Job 1, from each input row, only the relevant columns (see Table 1) are extracted.

For example, in Case 2, only the columns belonging to T1-T5 will be extracted in the *SearchString* variable. Reduce functions sum, i.e., count the number of occurrences of each combination of temperatures (the key) and outputs it as the new value for the current key. Because all the values for the same key are processed by a single Reduce task, it is evident that the output from Job 1 consists of unique combinations of temperatures and the number of their occurrences.

In Job 2, the Map function inverts its (key, value) pairs, making temperature occurrences the keys, and emits them to the Reduce function that outputs the received pairs. The sorting by occurrence is done by the framework as explained in Section 2.2.

```
      //Job 1
public void map(LongWritable key,Text value,
OutputCollector<Text,IntWritable> output, Reporter reporter)
throws IOException{
        String line = value.toString();
        String[] lineElements  = line.split(",");
        String SearchString = null
        //depending on a case (Table I) concatenate different
        lineElements in //SearchString
        …
        word.set(SearchString);
        output.collect(word, new IntWritable(1));
}
public void reduce(Text key, Iterator<IntWritable> values,
OutputCollector<Text, IntWritable> output, Reporter reporter)
throws IOException{
        int sum = 0;
        while (values.hasNext()){
                sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
}
```

```
      //Job 2
public void map(LongWritable key, Text value,
OutputCollector<IntWritable,Text> output, Reporter reporter)
throws IOException{
        String line = value.toString();
        //\t is the default delimiter used by a reducer
        String[] lineElements  = line.split("\t");
        output.collect(new
        IntWritable(Integer.parseInt(lineElements[1])),
                                new Text(lineElements[0]));
}
public void reduce(IntWritable key, Iterator<Text> values,
OutputCollector<IntWritable, Text> output, Reporter reporter)
throws IOException{
        //there is only one value
        output.collect(key, values.next());
}
```

Figure 3: Java code segments of Map and Reduce tasks for Job 1 and Job 2.

3.4 Pig

The Pig program that has the same functionality as the MapReduce code described before must be tailored for each specific case. The Pig code for Case 2 is shown in Figure 4.

After having loaded the data files, we group the records by columns with ordinal numbers 4 to 8 corresponding to temperatures T1 to T5 (note that column indexes are zero based). For other cases, the ordinal numbers of columns are as defined in Table 1. Then we count the number of temperatures in each group and afterwards we order the grouped records by the temperature occurrence. At the end, the results are stored in an output file.

```
records = LOAD '/user/path_to_data_files/*'
                              USING PigStorage(',');
grouped_records = GROUP records BY ($4, $5, $6, $7, $8);
count_in_group = FOREACH grouped_records
                     GENERATE group,
                     COUNT(records) AS count_temp;
count_in_group_ordered = ORDER count_in_group
                              BY count_temp DESC
                              PARALLEL 1;
STORE count_in_group_ordered
                     INTO 'path_to_destination folder';
```

Figure 4: The Pig program.

For the execution of the presented Pig program, we use the default settings with an exception: we use the keyword PARALLEL with the ORDER statements to specify that we want only one Reducer task to be executed for the ORDER statement. Hence, a single file is produced as a final result, as in the MapReduce approach. For the three given instructions: GROUP, FOREACH and ORDER, Pig generates three sequential MapReduce jobs named “GROUP BY”, “SAMPLER” and “ORDER BY”. We use the same names to refer to those generate jobs.

3.5 Hive

The Hive code that has the same functionality as the MapReduce and Pig programs described before is also tailored for each specific case. The Hive code for Case 2 is given in Figure 5.

First, we create the table Temp_Simul and load the simulation data in it. LOAD instruction is just a file system operation in which Hive copies the input files into Hive’s warehouse directory. The resulting table *Results_Case_2* is generated for the results of the SELECT statement that evaluates temperature frequencies. The SELECT statement is customized for columns determined by Case 2. For other cases, the columns should be named as defined in Table 1.

When executing the SELECT statement, Hive generates and executes only two MapReduce jobs, in contrast to the Pig that executes three MapReduce jobs. Hive allows a specification of a maximum or a constant number of reducers. We have not specified them; therefore we gave Hive freedom in specifying the

```
CREATE TABLE `Temp_Simul` (`col_0` INT ,
                           `col_1` INT ,
                           `col_2` INT ,
                           `col_3` FLOAT ,
                           ...
                           col_88` FLOAT )
    COMMENT "Results from simulations"
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
LOAD DATA INPATH 'path_to_data_files/*'
    INTO TABLE Temp_Simul;
CREATE TABLE Results_Case_2 AS
    SELECT col_4, col_5, col_6, col_7, col_8,
    COUNT(1) AS NumOfOccurences
        FROM Temp_Simul
        GROUP BY col_4, col_5, col_6, col_7, col_8
        ORDER BY NumOfOccurences DESC;
```

Figure 5: The Hive program.

Case: 1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
11159	8933	391	387	298	294	298	294
11097	8860	323	319	228	224	228	224
10945	8778	298	294	227	217	215	211
10924	8351	271	267	221	216	199	181
10729	7807	264	232	220	211	194	168

Table 2: Top 5 temperature frequencies for each case.

number of reducers for each job. Still, the number of Reduce tasks for Job 2 was always equal to one.

4 Results and Discussion

As expected, the three presented approaches gave identical quantitative result. The five highest numbers of temperature frequencies, for each test case from Table 1, are given in Table 2. We have presented only temperature frequencies since the temperature values that are associated with these frequencies are specific to the knee simulation and are not in the scope of this paper. We see that the lowest numbers appear in Case 8, which was expected because in Case 8 the largest number of parameters (T) is projected from the source data.

	Job1									***Job2***								
Case:	1	2	3	4	5	6	7	8	**Total**	1	2	3	4	5	6	7	8	**Total**
No. of Map tasks	100	100	100	100	100	100	100	100		12	12	12	18	16	20	26	36	
No. of Reduce tasks	12	12	12	12	12	12	12	12		1	1	1	1	1	1	1	1	
Tot. time maps (s)	1122	1080	1119	1187	1121	1287	1162	1826	**9903**	32	31	51	78	59	184	64	443	**941**
Tot. time red. (s)	100	80	91	148	108	207	118	413	**1264**	4	4	10	16	12	31	12	50	**139**
CPU time spent (s)	588	618	667	790	686	933	719	1,494	6494	7	9	55	95	70	185	73	330	823
Total duration (s)	40	37	38	43	49	51	40	79	**377**	13	14	22	28	26	48	24	73	**248**

Table 3: MapReduce approach: MapReduce tasks execution times.

	Job1 (GROUP BY)								
Case:	1	2	3	4	5	6	7	8	**Total**
No. of Map tasks	34	34	34	34	34	34	34	34	
No. of Reduce tasks	5	5	5	5	5	5	5	5	
Total time spent by all maps in (s)	517	543	527	807	540	874	723	1077	**5608**
Total time spent by all reduces (s)	31	24	45	180	58	297	109	597	**1342**
CPU time spent (s)	371	394	446	695	502	1098	582	1734	5822
Total duration (s)	31	34	38	76	40	100	59	330	**708**

	Job2 (SAMPLER)									***Job3 (ORDER BY)***								
Case:	1	2	3	4	5	6	7	8	**Total**	1	2	3	4	5	6	7	8	**Total**
No. of Map tasks	1	1	1	10	3	18	5	35		1	1	1	10	3	18	5	35	
No. of Reduce tasks	1	1	1	1	1	1	1	1		1	1	1	1	1	1	1	1	
Tot. time maps (s)	7	6	9	52	17	105	26	315	**537**	7	7	18	108	35	215	56	536	**983**
Tot. time red. (s)	4	4	4	4	4	6	4	4	**32**	4	4	15	64	25	124	42	202	**481**
CPU time spent (s)	2	2	5	35	11	70	17	139	282	2	5	31	176	62	346	100	669	1391
Total duration (s)	13	15	17	17	18	19	18	23	**140**	13	15	36	82	43	145	62	226	**622**

Table 4: Pig approach: MapReduce tasks execution times.

	Job1									***Job2***								
Case:	1	2	3	4	5	6	7	8	**Total**	1	2	3	4	5	6	7	8	**Total**
No. of Map tasks	17	17	17	17	17	17	17	17		2	2	2	3	2	4	3	10	
No. of Reduce tasks	5	5	5	5	5	5	5	5		1	1	1	1	1	1	1	1	
Tot. time maps (s)	119	153	166	254	193	364	224	652	**2125**	8	9	24	66	32	103	44	186	**473**
Tot. time red. (s)	17	22	31	82	42	136	52	810	**1192**	3	3	15	77	26	146	41	265	**576**
CPU time spent (s)	113	138	202	389	237	580	291	991	2941	4	7	39	156	63	272	92	501	1134
Total duration (s)	19	23	27	39	31	58	35	215	**447**	12	14	32	112	49	182	68	293	**762**

Table 5: Hive approach: MapReduce tasks execution times.

Table 3 shows the MapReduce execution times for Job 1 and Job 2, for each test case. It also shows the total CPU time and associated total duration of the analysis for each case. Table 4 and Table 5 show corresponding execution times of the MapReduce tasks generated by Pig and Hive, respectively.

4.1 Execution time

Although the interpretation of the temperature values and their occurrence in a specified combination are not important for this paper, each execution case (Table 1) draws different amounts of data to the Map and Reduce functions in Job 1 and Job 2, which influences their execution times, as evident from Table 3.

We can calculate from Table 3 that the total time spent for Map and Reduce in Job 1 and Job 2, for all test cases on all executing nodes, is: $t_s = 9903 + 1264 + 941 + 139 = 12247$ s, while the total duration of the complete MapReduce analysis is: $t_m = 377 + 248 = 625$ s. The ratio t_s/t_m, which can assess the level of parallelism achieved, is 19.6. Consequently, we can conclude that the above analysis is about 20 times faster, if implemented by the MapReduce paradigm on 24 computing cores, relatively to the MapReduce execution time on a single core.

Table 4 and Table 5 show execution times of the MapReduce tasks generated by the Pig and Hive. The job durations, for all test cases and for all three approaches, are shown in Figure 6. The last triple presents the total execution times across all test cases. It is evident that the MapReduce tasks, written and executed directly, take, in average, approximately two times less time than those generated by Pig or Hive. Furthermore, Hive outperforms Pig for approximately 20%.

One can also notice that the Hive approach was faster than the direct MapReduce approach in the first three cases. By comparing Tables 3 and 5, it is evident that Hive gained the advantage in Job 1. This possibly happened because the number of Map and Reduce tasks, i.e., 17 and 5, applied by Hive, were more appropriate for the smaller amount of data. The superiority of the direct MapReduce approach is however more and more evident as the case number, therefore also the amount of data, increases.

5 Conclusion

In this paper, we have applied Hadoop tools for the analyses of tabular data coming from a complex computer simulation. Three approaches were applied; the first modeled the data operations directly with the MapReduce jobs, while the other two described the data operations using higher level languages Pig and Hive.

All three approaches gave the same quantitative result, but the execution times were different. From the presented time measurements it is evident that the directly programmed MapReduce tasks are in average two times faster than Pig or Hive. For our test cases, it is also evident that Hive outperforms Pig for 20%, probably because Hive generates one MapReduce job less than Pig. Hive outperformed the direct MapReduce approach in the cases with smaller amounts of data, probably because the number of Map and Reduced tasks employed by Hive was more optimal for smaller data sets.

As Pig and Hive use MapReduce in the background, it is expected that using the low-level approach will be faster. However, the high-level approaches could have advantages over the direct MapReduce approach if design efforts are also considered. Writing the mappers and reducers, compiling, debugging, packaging the code, submitting the jobs, and retrieving the results using the direct MapReduce approach takes developer's time. On

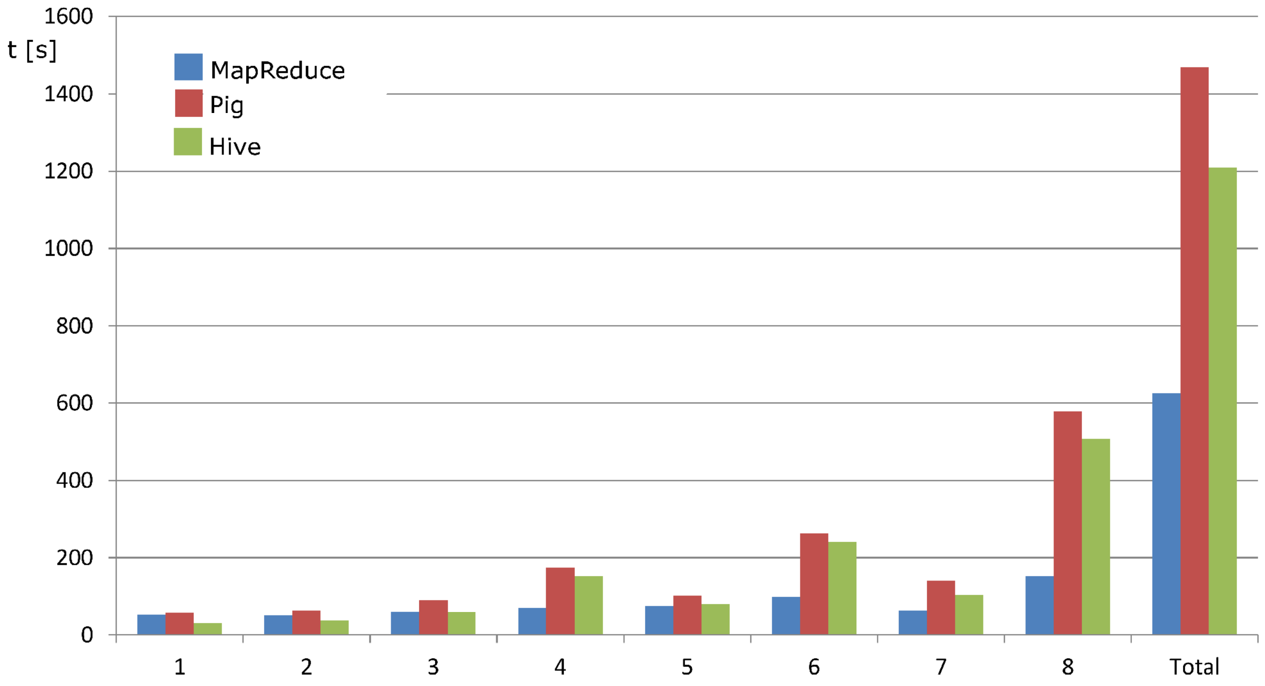

Figure 6: Sum of jobs execution times for each test case. The last series represents the total sum

the other hand, it is much easier to describe data operations with Pig or Hive for an user less familiar with the Java programing language. Users familiar with SQL language may prefer to use Hive, while users familiar with procedural languages would probably prefer to use Pig to describe the same data operations.

Future work is in implementing MapReduce paradigm with the MPI library [21] that could support more complex communication functions, which could result in more efficient execution of the computationally intensive services, on complex data sets in cloud environments [22].

Acknowledgement

The research was funded in part by the European Union, European Social Fund, Operational Programme for Human Resources, Development for the Period 2007-2013.

References

[1] J. Dean, and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in OSDI'04, 2004, pp. 137-149.

[2] "Welcome to Apache™ Hadoop®!," Oct., 2012; http://hadoop.apache.org/.

[3] B. Franks, "What is big data and why does it matter?," *Taming the big data tidal wave: finding opportunities in huge data streams with advanced analytics*, pp. 3-29, Hoboken, New Jersey: John Wiley & Sons, Inc., 2010.

[4] A. Thusoo, Z. Shao, S. Anthony *et al.*, “Data warehousing and analytics infrastructure at facebook,” in SIGMOD 2010, International conference on Management of data, pp. 1013-1020.

[5] D. Borthakur, J. Gray, J. S. Sarma *et al.*, “Apache hadoop goes realtime at Facebook,” in ACM SIGMOD International Conference on Management of Data, 2011, pp. 1071-1080.

[6] R. Trobec, M. Šterk, S. Almawed *et al.*, “Computer simulation of topical knee cooling,” *Comput. biol. med,* vol. 38, pp. 1076-1083, 2008.

[7] G. Kosec, Šarler, Božidar, “Solution of a low Prandtl number natural convection benchmark by a local meshless method.,” *International journal of numerical methods for heat & fluid flow,* vol. 23, no. 1, pp. 189-204, 2013.

[8] U. Stepišnik, and G. Kosec, “Modelling of slope processes on karst,” *Acta Carsologica,* vol. 40, no. 2, pp. 267-273, 2011.

[9] L. Wang, J. Tao, R. Ranjan *et al.*, “G-Hadoop: MapReduce across distributed data centers for data-intensive computing,” *Future Generation Computer Systems,* vol. 29, no. 3, pp. 739-750, 2013.

[10] R. C. Taylor, “An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics,” *BMC Bioinformatics,* vol. 11, no. SUPPL. 12, 2010.

[11] M. Niemenmaa, A. Kallio, A. Schumacher *et al.*, “Hadoop-BAM: Directly manipulating next generation sequencing data in the cloud,” *Bioinformatics,* vol. 28, no. 6, pp. 876-877, 2012.

[12] J. Lin, and C. Dyer, "Limitations of MapReduce," *Data-Intensive Text Processing with MapReduce*, Synthesis Lectures on Human Language Technologies, pp. 143-145: Morgan & Claypool Publishers, 2010.

[13] I. Cloudera. "CDH Proven, enterprise-ready Hadoop distribution – 100% open source," Oct, 2012; http://www.cloudera.com/hadoop/.

[14] Z. Fadika, E. Dede, M. Govindaraju *et al.*, "Benchmarking MapReduce Implementations for Application Usage Scenarios." 12th IEEE/ACM International Conference on Grid Computing (GRID). pp. 90-97, 2011.

[15] "Welcome to Apache Pig!," December, 2012; http://pig.apache.org/.

[16] "Welcome to Hive!," December, 2012; http://hive.apache.org/.

[17] T. White, "Hive," *Hadoop: The Definitive Guide*, pp. 365-409, Gravenstein Highway North, Sebastopol: O’Reilly Media, Inc., 2010.

[18] T. White, "The Hadoop Distributed Filesystem," *Hadoop: The Definitive Guide*, pp. 41-73, Gravenstein Highway North, Sebastopol: O’Reilly Media, Inc., 2010.

[19] T. White, "MapReduce," *Hadoop: The Definitive Guide*, pp. 15-40, Gravenstein Highway North, Sebastopol: O’Reilly Media, Inc., 2010.

[20] T. White, "MapReduce Types and Formats," *Hadoop: The Definitive Guide*, pp. 189-224, Gravenstein Highway North, Sebastopol: O’Reilly Media, Inc., 2010.

[21] T. Hoefler, A. Lumsdaine, and J. Dongarra, "Towards efficient mapreduce using MPI," 16th European Parallel Virtual Machine and Message Passing Interface Users' Group Meeting, EuroPVM/MPI, 2009, pp. 240-249.

[22] H. Mohamed, and S. Marchand-Maillet, "Distributed media indexing based on MPI and MapReduce," *2012 10th International Workshop on Content-Based Multimedia Indexing, CBMI 2012.* pp. 236-241.