Dynamic Reduce Task Adjustment for Hadoop Workloads

Vaggelis Antypas
Department of Informatics
Athens University of Economics and Business
Greece
antypas@aueb.gr

Nikos Zacheilas
Department of Informatics
Athens University of Economics and Business
Greece
zacheilas@aueb.gr

Vana Kalogeraki
Department of Informatics
Athens University of Economics and Business
Greece
vana@aueb.gr

ABSTRACT

In recent years, we observe an increasing demand for systems that are capable of efficiently managing and processing huge amounts of data. Apache's Hadoop, an open-source implementation of Google's MapReduce programming model, has emerged as one of the most popular systems for *Big Data* processing and is supported by major companies like Facebook, Yahoo! and Amazon. One of the most challenging aspects of executing a Hadoop job, is to configure appropriately the number of reduce tasks. The problem is exacerbated when multiple jobs are executing concurrently competing for the available system resources. Our approach consists of the following components: (i) an algorithm for computing the appropriate number of reduce tasks per job, (ii) the usage of profiler-jobs for gathering information necessary for the reduce task computation and (iii) two different policies for fragmenting the reduce tasks to the available system resources when multiple jobs execute concurrently in the cluster. Our detailed experimental evaluation using traffic monitoring Hadoop jobs on our local cluster, illustrates that our approach is practical and exhibits solid performance.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]

1. INTRODUCTION

Nowadays we observe a huge increase in the amount of information that is generated in a daily basis. A plethora of different data-intensive applications (e.g. data mining and web indexing) run daily by major IT companies such as Facebook [1] and Yahoo! [2], that access ever-expanding data sets ranging from a few Gigabytes to several Terabytes or even Petabytes of data. For example, Facebook collects 15 Terabytes of data each day and applies business-intelligence and ad-hoc analysis [3]. This huge amount of data needs to be processed in a fast and scalable way. Consequently, traditional approaches (e.g. Database Management Systems - DBMS) are no longer sufficient. Arguably one of the most popular systems that was developed to support all these features is Google's MapReduce [4]. MapReduce enables the easy development of scalable parallel applications to process vast amounts of data on large clusters of commodity machines. The success of the MapReduce programming model led to the development of Hadoop [5], an open-source implementation of MapReduce. Several companies (including Facebook and Yahoo! among others) use Hadoop on a regular basis for processing large amounts of data for various internal purposes [6].

One of the most demanding aspects of executing different processing jobs in Hadoop is to appropriately configure the jobs in order to achieve the best possible performance in terms of completion time. For this matter, Hadoop enables its users to tune a plethora of configuration parameters (a set of which can be seen in Table 1) that affect the performance of jobs. One of the most important parameters that can affect the observed job's execution time is the number of reduce tasks. This parameter controls the parallelism in the reduce phase and can significantly decrease its execution time. However, detecting the appropriate number of reduce tasks can be a challenging endeavor as it depends on the processing performed by the job and the amount of input data [7]. The problem is exacerbated when multiple jobs execute concurrently and compete for the available system resources. There have been some recent proposals that try to address this problem [8] and [9]. However, both approaches consider only cases where the jobs execute serially, so they do not examine the implications of the problem when multiple jobs are submitted concurrently. Finally, works like [10], and [11] optimize several configuration parameters (including the number of reduce tasks) for a single-job execution, so they do not focus on how resources should be distributed between concurrently running jobs.

In this work we aim to provide a tool that will enable the automatic configuration of the number of reduce tasks per job. By tuning this parameter we can improve the parallelism in the second phase of the MapReduce computation; reducing the job's completion time. We examine the problem in cases where multiple jobs execute concurrently (i.e. Hadoop workload) [12] in the cluster and they must share the available resources (i.e. reduce slots). Our scheme has the following steps: Initially, we compute the appropriate number of reduce tasks for each job using historical information regarding the input dataset, as well as information about the nature of the processing that each job aims to perform. In the case that we do not have historical data about the submitted job we invoke a small profiler-job for gathering the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PCI 2015, October 01-03, 2015, Athens, Greece

© 2015 ACM. ISBN 978-1-4503-3551-5/15/10...$15.00

DOI: http://dx.doi.org/10.1145/2801948.2801953

Configuration Parameter	Description	Default Value
mapred.reduce.tasks	Number of reduce tasks	1
dfs.block.size	Size in bytes of each data block in HDFS	$128m$
mapred.compress.map.output	Whether or not to compress the output of map tasks	$false$
io.sort.mb	Map buffer size in MB	100
io.sort.factor	The number of streams to merge at once in the sort phase	10

Table 1: Basic Configuration Parameters in Hadoop

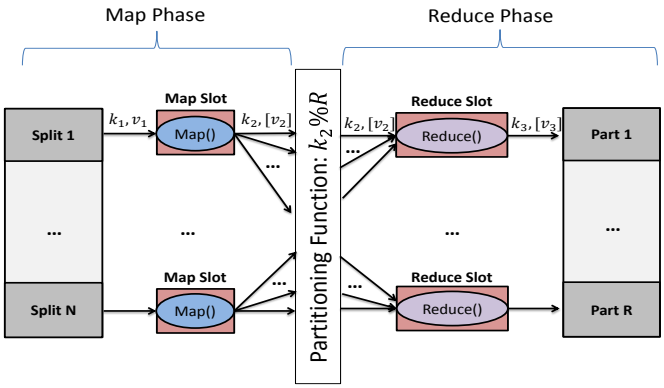

Figure 1: MapReduce Model

necessary information for the reduce tasks computation. Finally, in the case that the sum of the reduce tasks computed for each job is greater than the available reduce slots, we examine different techniques for distributing the reduce tasks to the available slots. In summary, the key contributions of this paper are as follows:

- We design an algorithm for computing the appropriate number of reduce tasks per job utilizing statistics from past execution runs.
- We invoke small profiler-jobs on a sample of the input datasets when we do not have historical data for the submitted jobs. These profiler-jobs perform the same computations on their map phase as their corresponding actual jobs, but execute on a small sample of the input dataset.
- We implement two different algorithms for fragmenting the computed reduce tasks to the available cluster resources (i.e. reduce slots).
- We conduct thorough experimental evaluation with actual traffic monitoring jobs that execute in the city of Dublin in our local cluster indicating the benefits of our approach.

2. MAPREDUCE BACKGROUND

MapReduce [4] is a programming model for processing large data sets. Users define two functions, the *map* function and the *reduce* function with the the following specifications: $map(k_1, v_1) \Rightarrow [k_2, v_2]$ and $reduce(k_2, [v_2]) \Rightarrow [k_3, v_3]$. The map function takes as input *key/value* pairs, processes them and generates intermediate *key/value* pairs. The system merges all the intermediate values associated with the same intermediate key and forwards them to the reduce function. The reduce function receives as input each intermediate key along with its list of values and produces the final output, which also comes in the form of *key/value* pairs. A visual representation of the above steps can be seen in Figure 1. Users can define the number of tasks that will invoke these functions. Map/Reduce tasks execute in the available map/reduce slots. Slots are the system resources (i.e. CPU cores) reserved by a job, and provide a limit on the amount of tasks that can run concurrently in the cluster.

Hadoop [5] is the most commonly used open-source implementation of the MapReduce programming model. Hadoop consists of two main components: The Hadoop MapReduce module which enables the parallel processing of very large data sets using the MapReduce paradigm, and the Hadoop Distributed File System (HDFS) which is responsible for the reliable storage and management of data. We focus on Hadoop's MapReduce module as it is the component we enhanced in order to support dynamic reduce task adjustment. Hadoop's MapReduce module is based on a master-slave architecture. It comprises one "master" node called *JobTracker* and a number of "slave" nodes called *TaskTrackers*. The JobTracker essentially serves as the link between the users and the system. More specifically, it handles the MapReduce jobs that the users submit, inserting them in a queue of pending jobs which are executed (by default) on a FIFO basis. Each job consists of multiple map and reduce tasks that execute the user-defined functions. The JobTracker is responsible for the assignment of map and reduce tasks from each job to the TaskTrackers. The latter execute these tasks in the available map and reduce slots and also manage data movement between the map and reduce phases. Map/Reduce slots depict how many concurrent map/reduce tasks can execute in the corresponding slave node.

3. METHODOLOGY

The objective in our work is as follows: Given a Hadoop workload consisting of multiple jobs that execute concurrently, our goal is to automatically adjust the reduce tasks used per job in order to minimize the workload's end-to-end execution time (i.e. makespan). The problem is not trivial as we have to detect the appropriate number of reduce tasks per job and then fragment these quantities to the available reduce slots. The last step is necessary in order to enable the parallel execution of the jobs' reduce tasks in the available reduce slots.

3.1 Reduce Task Adjustment

The first step for solving our problem is to detect the appropriate number of reduce tasks per job. We propose an algorithm that adjusts the reduce tasks based on the intermediate data size and the reduce task's buffer size. The algorithm aims to keep the intermediate data in the reduce tasks' in-memory buffers in order to minimize the amount of disk spills and thus decrease the reduce phase completion time. In order to invoke the algorithm we need prior knowledge of the intermediate data size. We get this information

either by prior execution runs of the jobs or by invoking short-term running profiler-jobs.

Dynamic Reduce Task Adjustment Algorithm. We propose an algorithm that runs prior to the submission of the jobs in the Hadoop cluster and computes the appropriate number of reduce tasks. It accomplishes this by having a prior knowledge of the size of the *key/value* pairs that each job generates in its map phase. So in order to run this algorithm it is necessary to have past execution runs of the jobs that are assigned for execution.

First we iterate through all the input files that comprise our jobs' input. So for each input file, we compute the number of lines and multiply that number with the approximate size of a typical map record (*key/value* pair) generated by the jobs. That computation gives us the file's map output size. If we assume repetitive jobs in our cluster similar to [13], we can compute the typical map record size from past runs. So let fs_f, $f \in \{1, ..., n_{files}\}$ be the map output size of the f-th input file, while $lines_f$ be the number of lines of file f, mrs_j be the approximate size of each map record, $TMOS_j$, $j \in Jobs$ be each job's total map output size and sel_j the selectivity of the job j which depicts the ratio of map output records to map input records. Then we can calculate each file's map output size fs_f as follows:

$$fs_f = lines_f \times mrs_j \times sel_j, \forall f \in \{1, ..., n_{files}\} \quad (1)$$

and we can computate each job's total map output size as follows:

$$TMOS_j = \sum_{f=1}^{n_{files}} fs_f, \forall j \in Jobs \quad (2)$$

Once the $TMOS_j$ value has been calculated based on the entire input, we can compute the number of reduce tasks for each job. We do this is by dividing the total map output size ($TMOS_j$) with the size of the reduce task's buffer. Applying this computation enables us to use as many reduce tasks as possible, so that the amount of data that each task will have to process is smaller than (or at the most equal to) the reduce task's buffer size. In the case that this is not possible, the reduce tasks have to spill data to the disk and retrieve them again when they have adequate space in their buffer. This is a time consuming procedure that naturally has a substantial negative effect on the job's performance (i.e. completion time). So let $rtasks_j$, $j \in Jobs$ be the appropriate number of reduce tasks for each of our Hadoop jobs. Also let rbs be the reduce task's buffer size. Then we have the following Equation for the computation of the appropriate number of reduce tasks:

$$rtasks_j = \lceil \frac{TMOS_j}{rbs} \rceil, \forall j \in Jobs \quad (3)$$

All the steps that were described in this Section comprise our algorithm for the calculation of the appropriate number of reduce tasks, which can be seen in Algorithm 1.

Profiler-jobs. The previously described algorithm requires prior knowledge about the average intermediate data size of the submitted jobs and the selectivity metric. So in case that this information is not available we propose the usage of profiler-jobs for gathering the necessary data. Profiling is a commonly used technique [9], [14] for estimating configuration parameters' impact on the performance of Hadoop jobs. These profiler-jobs perform the exact same computations at their map phase as their corresponding Hadoop jobs, but work on a small sample of the jobs' input.

Algorithm 1 Dynamic Reduce (DR) Task Adjustment Algorithm

```
Input: Jobs: the list of the assigned jobs, rbs: the reduce task's buffer size.
Output: rtasks[]: The reduce tasks to use per job.
job_counter ← 0
for j ∈ Jobs do
    mrs_j ← estimateMapRecordSize(j)
    sel_j ← estimateSelectivity(j)
    TMOS_j ← 0
    for all f ∈ job.input_files do
        lines_f ← countFileLines(f)
        fs_f ← lines_f × mrs_j × sel_j
        TMOS_j ← TMOS_j + fs_f
    rtasks[job_counter] ← ⌈TMOS_j / rbs⌉
    job_counter ← job_counter + 1
return rtasks[]
```

We run our profiler-jobs on a sample input and after their map tasks have finished their processing and have produced all their *key/value* pairs (i.e. intermediate data), we consider them finished. Reduce tasks do not process their input data since all we need is the size of the reduce tasks' input. In other words if we had their reduce tasks perform some type of processing, it would only increase the completion time to no avail.

We then retrieve the size of their intermediate data, the amount of intermediate records and the amount of input records through Hadoop's *Counters* class. We get the amount of intermediate records via the $MAP\_OUTPUT\_RECORDS$ Counter, we use MAP_OUTPUT_BYTES Counter for getting the intermediate data size and $MAP\_INPUT\_RECORDS$ Counter for retrieving the amount of input records. We compute the average intermediate data size (i.e. mrs_j in Equation 1) and the selectivity (i.e. sel_j in Equation 1) via the following Formulas:

$$mrs_j = \frac{MAP\_OUTPUT\_BYTES}{MAP\_OUTPUT\_RECORDS} \quad (4)$$

$$sel_j = \frac{MAP\_OUTPUT\_RECORDS}{MAP\_INPUT\_RECORDS} \quad (5)$$

Then we compute the number of reduce tasks per job by applying Algorithm 1. The only difference is that when profiler-jobs are applied, the *estimateMapRecordSize* and *estimateSelectivity* functions return the record's size and the selectivity as they are computed via Equations 4, 5.

3.2 Reduce Slots Allocation

Because the available reduce slots may be significantly smaller than the computed number of reduce tasks, we enhance our previous approach by adjusting the reduce tasks further in such cases. The goal is to limit the reduce tasks used per job so that their sum is equal to the available reduce slots and thus they can execute in parallel [12]. We considered two well-known allocation algorithms for achieving our goal [15], [16]. The first allocation algorithm fragments the reduce tasks proportionally to the jobs' requirements in regards to the appropriate number of reduce tasks (as computed in Section 3.1) while the second algorithm applies a

Algorithm 2 Proportionally Fair Reduce Task Allocation

1: Input: $rtasks[]$, $number\_of\_jobs$, $RSlots$
2: Output: $rtasks[]$
3: $tasks\_sum \leftarrow 0$
4: **for** ($i \leftarrow 0$ to $number\_of\_jobs - 1$) **do**
5: $\quad tasks\_sum \leftarrow tasks\_sum + rtasks[i]$
6: **for** ($i \leftarrow 0$ to $number\_of\_jobs - 1$) **do**
7: $\quad rtasks[i] \leftarrow \frac{rtasks[i]}{tasks\_sum} \times RSlots$
8: **return** $rtasks[]$

fair policy so that each job will use approximately the same number of reduce slots.

Proportionally Fair. With our first algorithm, we fragment the jobs' reduce tasks in a way that is fair, based on the requirements of each Hadoop job. In our case, the jobs' requirements are the number of reduce tasks calculated initially through the use of Algorithm 1. More formally we adjust the reduce tasks as follows:

$$rtasks_j^{'} = \frac{rtasks_j}{\sum_{j' \in Jobs} rtasks_{j'}} \times RSlots, \forall j \in Jobs \quad (6)$$

The number of reduce tasks depends on the initially computed number and is proportional to the available reduce slots (i.e. *RSlots* metric).

Example. Let us assume we have three jobs that we want to execute in parallel and the number of reduce tasks calculated for them initially is, 5 for the first job, 3 for the second and 4 for the third. That sums up to 12 reduce tasks in total. Furthermore, assume that our cluster consists of only 9 reduce slots. First we compute the portion of that 12 total reduce tasks that corresponds to each job. So for instance, for the first job we would have $\frac{5}{12} \approx 0.41$. After we have that portion for each job, we multiply it with the number of reduce slots in the cluster (9 in our case). The result of that computation rounded to the nearest integer, is the number of reduce slots that each job is going to use for its execution. Our proportionally-fair algorithm that we described above, can be seen in its entirety in Algorithm 2.

Round-robin. Our second algorithm fragments the jobs' reduce tasks into the available reduce slots in a Round-Robin (RR) manner. More specifically, we initially line up our jobs in the order they are going to be submitted for execution in our cluster and start giving them one reduce slot each. Once we traversed all the jobs (and still have some remaining reduce slots), we start again from the beginning and give each job one more reduce slot. We apply this procedure until we run out of reduce slots in the cluster. By fragmenting the reduce tasks via this approach, we make sure that all jobs will reserve approximately the same number of reduce slots (i.e. $rtasks_j = \frac{RSlots}{|Jobs|}, \forall j \in Jobs$). Finally, our round-robin algorithm can be seen materialized in Algorithm 3.

Job's Name	Job's Description
LineStats	Computes the mean reported delay as well as the standard deviation, observed by buses passing from the same lines with the same direction and at the same hour of the day.
PointStats	Computes the mean reported delay and standard deviation of buses that traverse points of interest in Dublin city (i.e. these points were detected by applying the DENCLUE [17] clustering algorithm) at the same hour of the day.
UniqueBuses	Detects for each line the number of bus vehicles that traverse it.

Table 2: Jobs' Description

Algorithm 3 Round-robin Reduce Task Allocation

1: Input: $rtasks[]$, $number\_of\_jobs$, $RSlots$
2: Output: $rtasks[]$
3: {Initialize all the elements of the $rtasks[]$ array to zero}
4: $threshold \leftarrow RSlots$
5: **while** ($threshold > 0$) **do**
6: $\quad$ **for** ($i \leftarrow 0$ to $number\_of\_jobs - 1$) **do**
7: $\quad\quad$ **if** ($threshold > 0$) **then**
8: $\quad\quad\quad rtasks[i] \leftarrow rtasks[i] + 1$
9: $\quad\quad\quad threshold \leftarrow threshold - 1$
10: $\quad\quad$ **else**
11: $\quad\quad\quad break$
12: **return** $rtasks[]$

4. EVALUATION

Settings. In order to assess the benefits of our proposal, we performed a series of experiments on our local Hadoop cluster which consists of 7 Virtual Machines (VMs). Each VM is equipped with two CPU processors and 3,096 MB RAM. We used Hadoop 1.2.1 and each VM represents a node in the Hadoop cluster, with one of them assuming the role of the "master" node. Moreover all VMs act as *TaskTrackers*, which means that are responsible for executing map and reduce tasks. Each node is configured to run at most two map and two reduce tasks in parallel. As a result we have a total of 14 available map slots and more importantly 14 reduce slots. Finally, the reduce tasks buffer size (i.e. *rbs* metric in Section 3.1) is set at 200MB (i.e. the default value).

Job's Description. For the evaluation of our approach we examined three applications that are applied in the EU-funded Insight project [18] for computing helpful traffic information from sensors placed on buses in the city of Dublin, Ireland [19]. These statistics enable us to detect unexpected traffic conditions in the city [20]. Applications were assigned concurrently in the Hadoop cluster for execution (e.g. the applications can be seen as a Hadoop workload [21]). You can see the details about the three applications in Table 2. Our experiments were performed with the same input dataset [19] varying its size in order to test the scalability of our approach. So, first we used a "small-sized" input dataset of approximately 300 MB which contained the bus traces of our original dataset that were recorded between 8:00 and 9:00 in the morning on all weekdays. Next we used a "medium-sized" input dataset of approximately

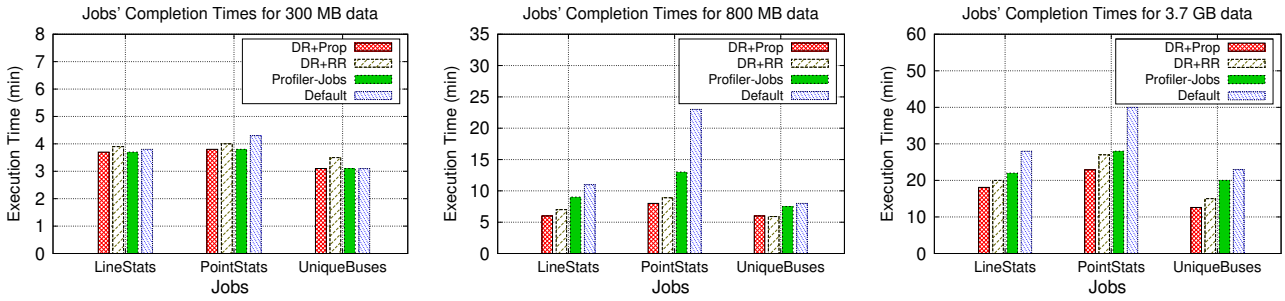

Figure 2: Per Job Execution Time Comparison of the Different Reduce Task Adjustment Methods

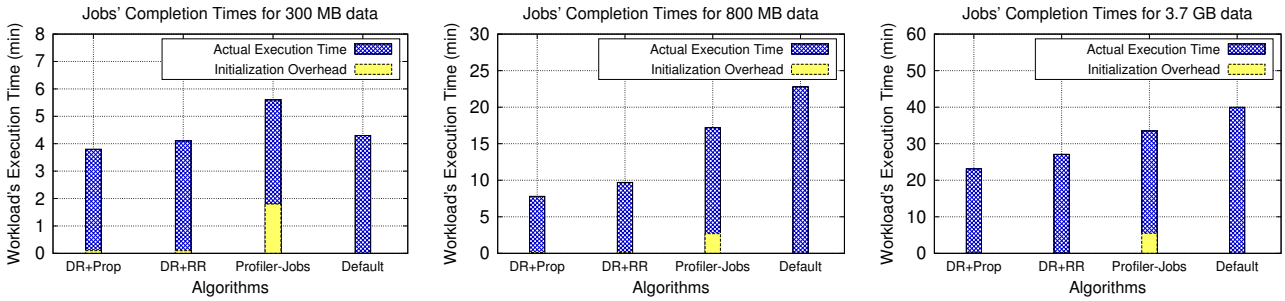

Figure 3: End-to-End Execution Time Comparison of the Different Reduce Task Adjustment Methods

800 MB which contained all the bus traces of our original dataset that were recorded on a weekend. Finally, we used a "large-sized" input dataset of approximately 3.7 GB which contained all the bus traces of our original dataset.

Method's Description. We applied our reduce task adjustment proposal, with (Profiler-Jobs in Figures 2 and 3) and without (DR in Figures 2 and 3) profiler-jobs and compared it with the default policy (Default in Figures 2 and 3) applied by Hadoop that uses only one reduce task per job. Furthermore, we compared the two different allocation policies, Proportionally Fair (Prop in Figures 2 and 3) and Round-Robin (RR in Figures 2 and 3), as our reduce tasks were bound by the 14 available reduce slots. For the profiler-jobs we utilized 10% of the input data for the calculation and applied the proportionally-fair allocation for fragmenting the reduce tasks. For each unique execution combination, we performed three separate runs on our cluster and we extracted the average completion time of each job, as well as the total execution time (completion of all 3 jobs). The last metric depicts the execution time of the whole workload and depends on the execution time of the slowest running job.

Per Job Execution Time. In Figure 2, you can see the observed execution time per job for the different adjustment policies. As you can observe, our proposed schema is able to decrease the execution times of most jobs compared to the default approach. Furthermore, using profiler-jobs seems to be beneficial only if the dataset is large. In case of small datasets (as in the $300MB$ scenario) invoking this profiler-job can deteriorate the job's performance. As you can notice especially for the larger input dataset using the proportionally fair allocation can significantly enhance the performance as it reduces the execution time of the long running job.

Workload's Execution Time. In regards to the total execution time of the examined workload, in Figure 3 we illustrate the benefits of our proposal. Our algorithm that uses past execution runs for computing the reduce tasks, minimizes the required execution time adding small overhead in the execution time compared to profiler-jobs that may require even 2.5 minutes to complete. Furthermore, our proportionally fair approach outperforms the round-robin technique as it allocates more resources to *LineStats* and *PointStats* which are the ones that require the most execution time. The round-robin policy would be beneficial in cases where all the jobs in the workload have similar processing requirements.

5. RELATED WORK

Zhuoyao Zhang et al. [8] proposed a performance evaluation framework, called AutoTune, that automates the user's efforts of tuning the number of reduce tasks along a Hadoop workflow, in order to optimize the workflow's overall completion time and resource usage. Their approach focuses on jobs that execute sequentially and each job's output serves as the input of the next. In contrast our work considers jobs that execute in parallel. Authors in [9] examined the idea of using a pre-job called Flubber in order to calculate the ideal number of reduce tasks for a given Hadoop job. This approach is similar to our proposal which uses profiler-jobs for the calculation of the appropriate number of reduce tasks. The main difference with our approach is that Flubber pre-jobs are more complicate as they try to depict the impact of skewed data and thus require more processing time and add significant overhead to the initialization of the actual job.

Matei Zaharia et al. [22] proposed a novel Hadoop scheduler for the execution of Hadoop jobs in the available re-

sources. Their research indicated that Hadoop's default scheduler can cause severe performance degradation in heterogeneous environments and for that reason they designed a scheduling algorithm, called LATE, that is highly robust to heterogeneity. In [14], authors focused on the even distribution of data between map and reduce tasks, especially in the presence of skewed data. They propose and evaluate two approaches for balancing the load between reduce tasks. Both approaches utilize a pre-processing job to analyze the data distribution. Furthermore, authors in [23] provide a novel schema that enables the dynamic configuration of map and reduce slots that aims to minimize the makespan of a Hadoop workload.

Finally, there are many works that focus on estimating the performance (i.e. in terms of execution time) of Hadoop jobs. This way the users acquire useful historical data, which can utilize on future runs, thus improving their jobs' performance. Kristi Morton et al. [24], proposed a progress indicator system for MapReduce jobs, named Parallax. Authors in [25] and [13] also proposed frameworks for predicting the performance of Hadoop jobs. These works are orthogonal to ours and can further improve Hadoop's performance.

6. CONCLUSIONS

In this paper we studied the problem of automatically tuning the number of reduce tasks for concurrently running Hadoop jobs, trying to minimize their end-to-end execution time. More specifically, we provided an algorithm for computing the appropriate number of reduce tasks per job when we have past execution runs. We also examined the applicability of small profiler-jobs when no historical data are available. Furthermore, we compared two well-known techniques for fragmenting the reduce tasks to the available reduce slots. Finally, we evaluated the performance of our proposals in our cluster using commonly used traffic monitoring applications, indicating the benefits of our approach.

7. ACKNOWLEDGMENTS

This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program Education and Lifelong Learning of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thalis-DISFER, Aristeia-MMD, Aristeia-INCEPTION Investing in knowledge society through the European

8. REFERENCES

[1] Facebook, http://www.facebook.com.

[2] Yahoo!, http://www.yahoo.com.

[3] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive - A Warehousing Solution Over a Map-Reduce Framework," *PVLDB, Pages 1626-1629*, 2009.

[4] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[5] Hadoop, http://hadoop.apache.org.

[6] Companies "powered by" Hadoop, https://wiki.apache.org/hadoop/PoweredBy.

[7] N. Zacheilas and V. Kalogeraki, "Real-Time Scheduling of Skewed MapReduce Jobs in Heterogeneous Environments," in *ICAC*, Philadelphia, PA, Jun. 2014, pp. 189–200.

[8] Z. Zhang, L. Cherkasova, and B. T. Loo, "AutoTune: Optimizing Execution Concurrency and Resource Usage in Mapreduce Workflows." in *ICAC*, 2013, pp. 175–181.

[9] R. Paravastu, R. Scarlat, and B. Chandrasekaran, "Adaptive Load Balancing in Mapreduce using Flubber," *Duke University Project Report*, 2012.

[10] H. Herodotou and S. Babu, "Profiling, What-if Analysis, and Costbased Optimization of MapReduce Programs," *PVLDB 4(11): 1111-1122*, 2011.

[11] J. Shi, J. Zou, J. Lu, Z. Cao, S. Li, and C. Wang, "MRTuner: A Toolkit to Enable Holistic Optimization for MapReduce Jobs," *PVLDB 7(13): 1319-1330*, 2014.

[12] J. Wang, Y. Yao, Y. Mao, B. Sheng, and N. Mi, "Fresh: Fair and efficient slot configuration and scheduling for hadoop clusters," in *CLOUD*, 2014, pp. 761–768.

[13] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *ICAC*, Karlsruhe, Germany, 2011, pp. 235–244.

[14] L. Kolb, A. Thor, and E. Rahm, "Load balancing for mapreduce-based entity resolution," in *ICDE*, 2012, pp. 618–629.

[15] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *SIGOPS*, 2009, pp. 261–276.

[16] T. Sandholm and K. Lai, "Mapreduce optimization using regulated dynamic prioritization," *SIGMETRICS*, vol. 37, no. 1, pp. 299–310, 2009.

[17] A. Hinneburg and D. A. Keim, "An Efficient Approach to Clustering in Large Multimedia Databases with Noise," in *KDD*, vol. 98, 1998, pp. 58–65.

[18] Insight Project, http://www.insight-ict.eu/.

[19] Dublin Bus Data, http://dublinked.com/datastore/datasets/dataset-304.php.

[20] N. Zygouras, N. Zacheilas, V. Kalogeraki, D. Kinane, and D. Gunopulos, "Insights on a Scalable and Dynamic Traffic Management System," *EDBT*, pp. 653–664, 2015.

[21] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating mapreduce performance using workload suites," in *MASCOTS*, 2011, pp. 390–399.

[22] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments." in *OSDI*, vol. 8, no. 4, 2008, p. 7.

[23] Y. Yao, J. Wang, B. Sheng, and N. Mi, "Using a Tunable Knob for Reducing Makespan of MapReduce Jobs in a Hadoop Cluster," in *CLOUD*. IEEE, 2013, pp. 1–8.

[24] K. Morton, A. Friesen, M. Balazinska, and D. Grossman, "Estimating the progress of mapreduce pipelines," in *ICDE*, 2010, pp. 681–684.

[25] G. Song, Z. Meng, F. Huet, F. Magoules, L. Yu, and X. Lin, "A Hadoop Mapreduce Performance Prediction Method," in *HPCC_EUC*, 2013, pp. 820–825.