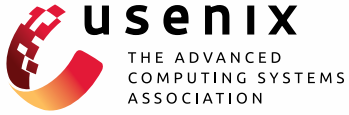


Real-Time Scheduling of Skewed MapReduce Jobs in Heterogeneous Environments

Nikos Zacheilas and Vana Kalogeraki, ***Athens University of Economics and Business***

https://www.usenix.org/conference/icac14/technical-sessions/presentation/zaheilas

This paper is included in the Proceedings of the 11th International Conference on Autonomic Computing (ICAC '14).

June 18–20, 2014 • Philadelphia, PA

ISBN 978-1-931971-11-9

Open access to the Proceedings of the 11th International Conference on Autonomic Computing (ICAC '14) is sponsored by USENIX.

Real-time Scheduling of Skewed MapReduce Jobs in Heterogeneous Environments

Nikos Zacheilas and Vana Kalogeraki
Athens University of Economics and Business
Athens, Greece
zaheilas@aueb.gr, vana@aueb.gr

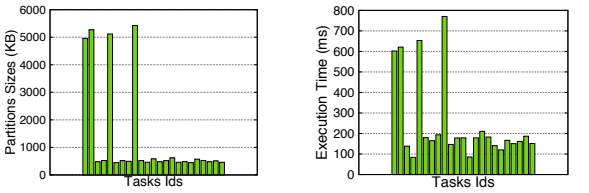
Abstract

Supporting real-time jobs on MapReduce systems is particularly challenging due to the heterogeneity of the environment, the load imbalance caused by skewed data blocks, as well as real-time response demands imposed by the applications. In this paper we describe our approach for scheduling real-time, skewed MapReduce jobs in heterogeneous systems. Our approach comprises the following components: (i) a distributed scheduling algorithm for scheduling real-time MapReduce jobs end-to-end, and (ii) techniques for handling the data skewness that frequently arises in MapReduce environments and can lead to significant load imbalances. Our detailed experimental results using real datasets on a truly heterogeneous environment, Planetlab, illustrate that our approach is practical, exhibits good performance and consistently outperforms its competitors.

1 Introduction

Today, we are experiencing increased demand for processing large amounts of data-intensive tasks. Systems such as IBM's InfoSphere BigInsights [20], Amazon's DynamoDB [2] and Google's MapReduce [10], have rapidly become de facto big data processing frameworks. These systems need to be fast, scalable and highly available. In particular, Google's MapReduce [10] framework has been proposed as a powerful and cost-effective approach for massive-scale processing. It has been utilized by some of the major computing companies, including Amazon, eBay, Facebook, IBM, LinkedIn, Twitter and Yahoo!, via its open-source implementation Hadoop [17] in a wide variety of application domains including real-time analysis of sensor data streams, real-time stock market data analysis and financial trading applications.

The MapReduce model breaks intense processing jobs into smaller tasks that run in parallel on multiple machines. Jobs are split into two stages of processing, *map* and *reduce*: input data are processed by tasks comprising the *map* phase, generating intermediate *(key, value)* pairs. Different *values* corresponding to the same *key* are then aggregated, and each *key* along with its associated *values* are transmitted to the *reduce* phase for further data processing. The partitioning of the intermediate $(key, value)$ pairs to the *reduce* tasks is based on a partitioning function, which in most cases is a simple hash function. However, *partitions* (*i.e.,* the set of intermediate $(key, value)$ pairs that will be processed by the same *reduce* task) can be of varying size, leading to significant data skewness challenges and critical delays on the execution times of the corresponding *reduce* tasks.



(a) Partition Sizes Processed Per Reduce Task

(b) Execution Time Per Reduce Task

Figure 1: Impact of skewed partitions on reduce tasks' execution times

We use the following example to illustrate the load imbalances that can occur due to skewness (*i.e., partitions* of varying sizes) as well as the effect on the execution times of the tasks, when processing skewed data-intensive MapReduce tasks. Figure 1 illustrates the distribution of the *partitions*' sizes and the corresponding execution times of the *reduce* tasks for a MapReduce job that processes a Youtube social graph (detailed information about the experiment can be found in the experimental evaluation section). The experiment run on Planetlab, using 82 processing cores and applying the hash function $(hashcode(key) mod R$, where R is the number of reduce tasks) proposed in the original MapReduce framework [10]. The figure clearly depicts that the exhibited skewness of the *partition* sizes affects the execution time of *reduce* tasks due to the uneven distribution of the data.

The problem is further exacerbated by the fact that jobs often have real-time response requirements, in the form of *deadlines*. As was pointed out by a recent study on Facebook's and Yahoo!'s production workload traces, 95% percentage of their production jobs are short running with an average real-time response requirement of 30 seconds [6], [30]. Timely execution of the tasks is a challenging problem due to the large heterogeneity and resource sharing in the nodes; in a shared processing environment, the execution times of the jobs are greatly affected by multiple tasks invoked concurrently and asynchronously by many jobs that are executed on the same computing resources. Conditions such as slow or misbehaving tasks (*i.e.,* due to hardware failures or misconfiguration), can severely affect the performance of the entire jobs [10]. Nevertheless, to make the deployment of these systems practical, the jobs must be able to operate autonomously in highly dynamic environments, and meet real-time demands, even under load spikes and unpredictable initiation of new tasks.

Current scheduling techniques in the Hadoop MapReduce framework (the most widely used MapReduce implementation) are not adequate, as they either adopt Fair scheduling [19] or Capacity scheduling [18]. These strive to balance the load across the resources rather than meeting real-time demands of the jobs. Recent approaches for scheduling MapReduce jobs include the LATE [42] and EDF [39] schedulers, however both approaches focus on scheduling and do not examine the impact of skewed data on the execution time of jobs. With respect to the skewed *partitions* problem, work has been mainly done by [14] and [27]. The first aims at distributing similar sized *partitions* among the available *reduce* tasks, but in heterogeneous environments this approach can lead to the assignment of large-size *partitions* to slow nodes. Skewtune [27] on the other side, proposes the repartitioning of heavily skewed *partitions*, however the overhead of such schemes can degrade the performance of *short* running jobs. It is clear that none of the existing works offers a unified solution for the two problems but rather examines them separately without taking into account their interaction.

In this work, we present DynamicShare, our system for supporting the execution of real-time, skewed MapReduce jobs in heterogeneous environments. Our goal is to address the joint problem of: (a) scheduling MapReduce jobs dynamically to maximize the probability of meeting their real-time response requirements, and (b) effectively handle the issue of data skewness. We focus on the execution of short running jobs, similar to Facebook Corona [37], (typical queries are: *identify common friends in Facebook*), which execute in the order of seconds. To our knowledge this is the first proposal towards a unified solution to the stated problem. Our approach makes the following contributions:

1. We present a distributed scheduling algorithm for scheduling jobs end-to-end. DynamicShare uses measurements of *laxity* values of the tasks, projected latencies and measurements of resource loads to adjust their scheduling order to compensate for queuing delays, estimates the execution times of the tasks using a non-parametric regression technique and identifies overloaded nodes early on through a Local Outlier Factor algorithm.

2. To handle the data skewness that arises in the *reduce* phase, we design two algorithms, a simple but efficient *Simple Partitions*' assignment algorithm that considers the sizes of the *partitions* and the variable processing capabilities of the nodes to make an appropriate placement, and a *Count-Min sketches* algorithm that enables an even better distribution of the *partitions* but at the expense of additional execution time for the *partitions*' assignment procedure.

3. We have implemented and evaluated DynamicShare on Planetlab, a truly heterogeneous environment, using 82 processing cores in total. Our experimental results utilizing two different datasets, from Youtube and Twitter networks, illustrate that our approach is practical, meets jobs' real-time response demands, effectively addresses the issue of highly skewed data, and outperforms its competitors.

2 System Model and Architecture

2.1 System Model

A MapReduce job is modelled as a sequence of invocations of *M map* and *R reduce* tasks (shown in Figure 2). Tasks are modelled as follows: $map(k_1, v_1) \Rightarrow [k_2, v_2]$ and $reduce(k_2, [v_2]) \Rightarrow [k_3, v_3]$. *Map* tasks take as input (k_1, v_1) pairs and return a list of *(key,value)* pairs of possibly different types, k_2 and v_2. The values associated with the same key k_2 are grouped together into a list and passed as input to the appropriate *reduce* task, which emits arbitrary *(key,value)* pairs of a final type, k_3 and v_3. All $(k_2, [v_2])$ pairs processed by the same *reduce* task on a cluster's node, are considered a *partition*. Recent works, such as [14], suggest the usage of a larger number of *partitions* compared to the number of *reduce* tasks to minimize the skewness of the intermediate data. Our framework follows this approach.

We consider soft real-time MapReduce jobs that are aperiodic and, thus, their arrival times are not known a priori. We focus on applications with intensive reduce phase and limited network transfers. Each job j is associated with a number of parameters: $Deadline_j$ is the

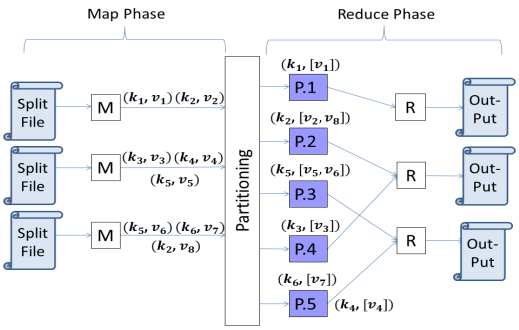


Figure 2: MapReduce Computation Model

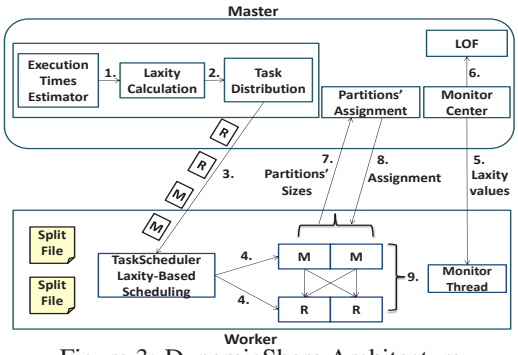


Figure 3: DynamicShare Architecture

time interval, starting at job initiation, within which job j must complete (deadline values are typically assigned by a system administrator based on the real-time demands of the jobs, determining the appropriate deadline for the job is outside the scope of the paper). $Proj\_exec\_time_j$ is the estimated amount of time required for the job to complete, this includes communication and queueing times at the system resources. Every job j should execute within its $Deadline_j$, that is, the sum of the computation times and the corresponding communication times of all tasks invoked by the job (denoted as projected end-to-end execution time) should be smaller than the $Deadline_j$. MapReduce jobs are data-intensive jobs, thus, the end-to-end execution time is mainly attributed to the execution times of the tasks and the communication times are negligible. $Laxity_j$ is defined as the difference between the $Deadline_j$ and $Proj\_exec\_time_j$, and is considered as a measure of urgency for the j job. The *laxity* value for each job is updated dynamically during execution; and determines the order with which the job's tasks will be scheduled at the system resources. Finally, $split\_size_j$ is the user-defined size of a split input file for job j.

Each task t of job j is described with the following metrics: $cpu_{i,t}$ and $memory_{i,t}$ represent the average percentage of CPU and memory required for task t to execute on Worker i. $m_{i,t}$ is the estimated mean execution time of a *map* task on Worker i. This includes the required time to read the total amount of input data, execute the *map* method and transmit the intermediate pairs to the *reduce* tasks. Similarly, $r_{i,t}$ depicts the estimated execution time of a *reduce* task, this corresponds to the time required for grouping $(key, value)$ pairs with the same *key* into a single (*key*,*list_of_values*) pair, plus the required time for executing the *reduce* method. Finally $partitions\_size_{i,t}$ holds the total size of the *partitions* that are assigned on Worker i for a *reduce* task t.

2.2 DynamicShare Architecture

The DynamicShare architecture (shown in Figure 3) comprises a single Master node and multiple Worker nodes. The Master receives MapReduce jobs, along with their corresponding deadline requirements. It is responsible for the assignment of the *map* and *reduce* tasks to Workers and the monitoring of the currently executing jobs. For each submitted job, the Master estimates the execution times of the *map* and *reduce* tasks in order to compute the execution time of the whole job and its corresponding *laxity* value. This value will be used by the TaskScheduler component at each Worker when scheduling the job's tasks (Section III). The Master is responsible to keep track of current resource usage statistics (*i.e.,* CPU, memory) along with their respective *laxity* values, as reported by the Monitor components at the individual Worker nodes during task execution. This information will also be used by the anomaly detection algorithm to identify overloaded nodes. The Master invokes the *partitions*' assignment algorithm to decide how to distribute the generated *partitions* to the available *reduce* tasks (Section IV). Finally, the input data file for each job is uploaded to the cluster and distributed as equal sized split files to the Workers for processing by the *map* tasks; the size of the split file is typically user-defined. Techniques such as the one proposed in [22] can further enhance our framework to distribute the split files based on the processing capabilities of the nodes in the cluster.

3 Dynamic Real-Time Scheduling of MapReduce Jobs

We develop a distributed scheduling approach that dynamically adjusts the execution of jobs on the system resources and measures the impact of overloaded nodes on meeting their end-to-end real-time demands. It consists of the following components: (a) A model for estimating the execution times of tasks based on a commonly used *non-parametric* regression technique, *k-Nearest Neighbor (k-NN) smoothing*. (b) A distributed least laxity first scheduling algorithm for scheduling jobs end-to-end, that uses measurements of the *laxity* values of the tasks to adjust their scheduling order to compensate for queuing delays at Workers. (c) Early detection of overloaded nodes via a Local Outlier Factor algorithm.

3.1 Estimating Execution Times of Tasks

To estimate the execution times of the entire jobs, the fundamental idea is to compute an approximation of the execution times of the *map* and *reduce* tasks and use this approximation in the computation of the execution time of the entire job. Techniques for estimating the execution time of MapReduce jobs, such as building job profiles based on previous execution times [38] or using debug runs before the actual execution [29] have been proposed for homogeneous environments, but without examining the implications of the problem in a heterogeneous setting, where the execution times of the tasks may vary.

We propose an estimation model that considers both the resource requirements of the newly submitted tasks and the previous task runs. There are two main approaches to address this: with *parametric* or *non-parametric* techniques [40]. For each *map* task we maintain a vector $\overrightarrow{x}$ of the task parameters as follows: $\overrightarrow{x} = (split\_size_j, cpu_{i,t}, memory_{i,t})$ Similarly, for *reduce* tasks we have: $\overrightarrow{x} = (partitions\_size_{i,t}, cpu_{i,t}, memory_{i,t})$
The $cpu_{i,t}, memory_{i,t}$ requirements of a newly submitted task are estimated via a histogram based approach similar to [25]. This approach is based on past runs. It distributes the processing requirements of previous tasks into histogram bins and utilizes the mean of the most populated bin for estimating the requirements of the newly issued task. If we model the execution time using *parametric* regression, the functional form of the $m(\overrightarrow{x})$ is assumed known and a technique like Least Squares can be applied for the calculation of the polynomial coefficients. However, in the case of a highly dynamic environment, like our setting, computing the execution time of tasks via a polynomial function is not efficient [21]. Thus, in such environments *non-parametric* techniques are more appropriate.

In *non-parametric* regression no assumption can be made about the functional form of $m(\overrightarrow{x})$, therefore the estimation is regarded as *data driven* because it depends only on previous task runs. All *non-parametric* regression techniques are modelled by the following equation:

$$\hat{m}(\overrightarrow{x}) = \frac{1}{n}\sum_{i=1}^{n} W_i(\overrightarrow{x}) y_i \tag{1}$$

where $W_i(\overrightarrow{x})$ is a weighting sequence and y_i the execution time of a previously issued task. Essentially, the $\hat{m}(\overrightarrow{x})$ can be considered as the weighted average of n previous task runs. Special care must be given to the number of previous runs that will be used. Too many past runs can lead to overly biased results, on the other hand few examples make the curve too "noisy". In our previous works [4], [23] we have shown that the number of runs to use depends on the job's characteristics.

The *non-parametric* regression technique we decided to implement for our estimation problem was *k-Nearest Neighbor (k-NN) smoothing*. In *k-NN smoothing*, the estimation of $\hat{m}(\overrightarrow{x})$ is based on the k past runs that are closest to the given vector $\overrightarrow{x}$. Utilizing a subset of the past runs, instead of all n past runs, is important because only those that have similar resource requirements with the currently examining task are considered in the estimation, and thus a better prediction is possible. We use the Euclidean distance of vectors to identify the closest past runs. In order to achieve better estimations, the impact of the previous runs on the estimation is weighted based on their distance from the examining vector, with the ones being closer receiving higher weights. A weighting function with several optimality properties is the Epanechnikov kernel $K(d)$, where: $K(d) = \frac{3}{4}(1-d^2)$, with $|d| < 1$. The d parameter, in our case is the calculated Euclidean distance of the vectors. The choice of the kernel function is not significant for the results of the approximation [33]. The Epanechnikov kernel function gives more weight to previous runs that are closer to the examining task's vector parameters. In order to be utilized by the estimator, the function must be scaled and normalized. We used the following weighting function:

$$W_i(\overrightarrow{x}) = \begin{cases} \frac{K_R(|\overrightarrow{x}-\overrightarrow{x_i}|)}{\hat{f}(\overrightarrow{x})} & \text{if } i \in N_{\overrightarrow{x}} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where $N_{\overrightarrow{x}} = \{i\text{: } \overrightarrow{x_i}$ is one of the k nearest neighbors of $\overrightarrow{x}\}$, $K_R(d)$ is the scaled Epanechnikov kernel function and is given by the following equation: $K_R(d) = \frac{1}{R}K(\frac{d}{R})$. The kernel is scaled by the factor R which is defined as: $R = max(|\overrightarrow{x} - \overrightarrow{x_i}|)$, $i \in N_{\overrightarrow{x}}$. Finally the $\hat{f}(\overrightarrow{x})$ factor in equation (2) is a normalized factor, and is given by the following formula: $\hat{f}(\overrightarrow{x}) = \frac{1}{k}\sum_{\overrightarrow{x_i} \in N_{\overrightarrow{x}}} K_R(|\overrightarrow{x} - \overrightarrow{x_i}|)$.

One important aspect of the algorithm is the choice of the value for k that will determine the number of past runs to be taken into account during the estimation. Too many examples can cause an increase on the bias $E\{\hat{m}(\overrightarrow{x}) - m(\overrightarrow{x})\}$, while few previous runs may lead to large variance $E\{\hat{m}^2(\overrightarrow{x})\}$. The value of k depends on the number n, so it must be adjusted accordingly. It has been proven [40] that by increasing k in proportion to $n^{\frac{4}{5}}$, the *k-NN* technique achieves a constant balance between the variance and the bias.

3.2 Least Laxity First Scheduling

We develop a dynamic, distributed least laxity first scheduling scheme that determines the order of execution of the tasks based on their urgencies and timing constraints. The Least Laxity First Scheduling (LLF) algorithm has been successfully employed in distributed and mobile real-time systems such as in [12], [24]. In LLF each job is associated with a *laxity* value, that represents a measure of urgency for the job and is used to order the execution of the tasks on the Workers. Given the dead-

line $Deadline_j$ and $Proj\_exec\_time_j$ for job j, we compute $Laxity_j$ as:

$$Laxity_j = Deadline_j - Proj\_exec\_time_j, \quad (3)$$

The projected execution time of j job is computed based on its estimates of the execution times of the *map* and *reduce* tasks, as follows: $Proj\_exec\_time_j = max\{m_{i,t}, ..., m_{k,t}\} + max\{r_{z,t}, ..., r_{l,t}\}$, where we consider the maximum execution times of the *map* and *reduce* tasks in the above computation since all tasks of the same phase run in parallel. The *laxity* value for each job is computed initially by the Master node and is used when scheduling the tasks of the job at each node. The closer the $Laxity_j$ value to zero, the more probable is for the job to miss its deadline; a negative value indicates that the deadline will be missed.

Tasks are ordered in the TaskScheduler at each node, based on the *laxity* values of the jobs that invoke them. The advantage of LLF compared to other scheduling approaches such as Earliest Deadline First (EDF) [39] and Hadoop's FIFO and FAIR scheduler, is, that, LLF is a dynamic scheduling algorithm that allows for compensating for queueing delays often experienced in distributed settings or that were mis-calculated at previous nodes. In LLF the task with the smallest laxity value has the higher priority. The *laxity* value of a job is adjusted as the tasks invoked by the job execute on the system resources. To avoid constantly updating the *laxity* values for a job, they are adjusted only when new tasks are inserted into the TaskScheduler's queue or when tasks finish execution or miss their deadlines. As the *laxity* values of the jobs are updated, the task belonging to the job with the smallest *laxity* value will execute next; if a job has negative *laxity* value this job has been estimated to miss its deadline and thus its tasks will be processed only when the TaskScheduler has pending tasks that all have missed their deadlines. That means a task with negative *laxity* will never preempt tasks with positive *laxity* values.

3.3 Identifying Overloaded Nodes

Because nodes' resource capacities are not directly correlated to the amount of data they are assigned for processing, it is possible that the data blocks are distributed unproportionally to the nodes. This may result in Workers becoming overloaded and not capable of completing their assigned tasks within the jobs' deadlines.

To identify overloaded Workers early on, we use the Local Outlier Factor (LOF) algorithm [5] on the *laxity* values of tasks of the same job that run on different Workers. LOF is a metric of anomaly detection that can be applied to a set and identify possible outliers. We consider as outliers, *laxity* values that significantly differ from the rest. Our goal is to proactively identify overloaded Workers before the *laxity* value of the corresponding job becomes negative.

The main idea of the LOF algorithm is to compare the local density of a point's neighbourhood with respect to the local density of its neighbours, seeking one or more points with significant difference from the rest. Thus, we compare the *laxity* values for the same job on the different Workers that execute the tasks of the job. More formally: Let $l\_distance(lax_A)$ be the distance of a lax_A to the l nearest neighbour, lax_A will be the *laxity* value of a job j that runs on a Worker that can possibly be overloaded. We denote the l nearest *laxity* values as $N_l(lax_A)$. This distance is used to define the *reachability distance* metric: $reach\_d_l(lax_A, lax_B) := max\{l\_distance(lax_B), d(lax_A, lax_B)\}$, where lax_B will be the *laxity* value of the same j job on a different Worker. The *reachability distance* of a value lax_A from lax_B depicts the true distance of the two values ($d(lax_A, lax_B)$), but also at least the $l\_distance$ of lax_B. The usage of this distance achieves more stable results as shown on [5]. The *local reachability density* of lax_A is defined by:

$$lrd_l(lax_A) := \frac{|N_l(lax_A)|}{\sum_{lax_B \in N_l(lax_A)} reach\_d_l(lax_A, lax_B)} \quad (4)$$

and intuitively is the inverse of the mean *reachability distance* of lax_A from its neighbouring *laxity* values. The local *reachability density* is then compared with those of the neighbours using the following equation:

$$LOF_l(lax_A) := \frac{\sum_{lax_B \in N_l(lax_A)} lrd_l(lax_B)}{|N_l(lax_A)| * lrd_l(lax_A)} \quad (5)$$

which is the average *reachability density* of the neighbours divided by lax_A's own local density. A *LOF* value below 1 indicates a denser region, which would dictate an inlier, while values significantly greater than 1 indicate outliers.

The Master utilizes formula (5) to compare the *laxity* values of the jobs running in the system, based on the *laxity* values reported by the Workers. If a value greater than 1 is detected, the corresponding node is marked as overloaded, giving the option to execute some of its tasks on a different Worker.

4 Skewed data

As described earlier, the MapReduce framework is susceptible to severe load imbalances caused due to the skewness exhibited in the *partitions*. Recall, that in the original MapReduce framework [10], a *partition* consists of all the intermediate pairs that give the same result when the partitioning function is applied to them. Each *partition* is then assigned to a different *reduce* task. The most commonly used partitioning function, utilized also

in Hadoop, is $hashcode(key)$ mod R. In our system two types of skewness frequently occur:

Skewed Key Frequencies. This occurs when some *keys* appear more frequently in the intermediate pairs, thus the *partition* they are part of becomes extremely large. This issue can be solved by putting more work on the *map* tasks, by aggregating the *values* of the same *key*, and creating intermediate pairs in the form of (*key, list_of_values*).

Skewed Tuple Sizes. This applies to *(key, value)* pairs with complex processing structures on the *value* field. For example in the Twitter social network there is a large assymetry in the types of users and their friendship lists, as a result, pairs have varying sizes, depending on the number of objects that occupy their lists. So this case applies to *partitions* for which the (*key, list_of_values*) pairs contain lists with large amount of data.

We focus on the *Skewed Tuple Sizes* problem, as it has the most significant impact on the execution time of *reduce* tasks. Recent works [14], [15], [26] have shown that solving this problem is not trivial; these primarily aim at partitioning the data in such a way so that all *reduce* tasks finish their processing in similar times.

We propose an approach that uses more *partitions* than the number of *reduce* tasks and takes into account the *partitions*' sizes and our estimates on the task execution times on Worker nodes, assigning the *partitions* in such a way that all *reduce* tasks contribute to the data processing, according to their processing capabilities. We propose an approach that puts more work on powerful nodes, assigning multiple *partitions* on them, while exploiting the slower nodes by assigning them light-sized *partitions*.

4.1 Partition Size Calculation

We exploit two approaches to calculate the size of the *partitions*, the first utilizes the amount of values corresponding to the keys of a *partition* (similar to [14]), while the second uses the Count-Min Sketches [9] data-structure which enables the usage of more hash functions in the calculations.

Simple Partitions. Assume we have p *partitions*, with $p \geq R$. Let $s_m(k)$ be the number of *values* in the *list_of_values* that corresponds to key k, on a *map* task m, $m \in \{0,...,M\}$. We define as $P_m(i), i \in \{0,...,p\}$ the set that contains the *keys* of the i-th *partition* on the m-th *map* task. Then the total size for this *partition* on *map* task m will be: $S_m(i) = \sum_{k \in P(i)} s_m(k)$.

Each *map* task calculates the sizes of all generated *partitions* and sends them to the Master node, who is responsible to aggregate these values for each *partition* in order to calculate its total size. So the Master computes for each *partition* the following value: $S(i) = \sum_{m \in \{1,...,M\}} S_m(i), i = \{0,...,p\}$. These values will be the *partitions*' sizes and will be utilized in the dynamic partitioning assignment algorithm (discussed in the next section).

Count-Min Sketches. Our second technique for estimating the *partitions*' sizes is based on the use of sketches. A sketch is a synopsis data-structure utilized extensively in query-optimization [11]. It provides the capability of capturing the basic features of a dataset by monitoring a significant subset. We use a special type of sketch, called Count-Min Sketch [9], which is mainly used for frequency counting in data streams [8].

Each *map* task creates a local sketch which can be seen as a two-dimensional array, $sketch_m[i,j]$, that stores information regarding the generated *key-value* pairs. Each row of the array corresponds to a different hash function that can be used for the distribution of the intermediate *key-value* pairs to the *partitions*, and each column corresponds to a different *partition*. So, suppose that we have d rows, $H = \{h_i,\ i = 1,...d\}$ be the set of the chosen hash functions, and p columns, the same number as the *partitions* that will be used. It is recommended in [9] that the chosen hash functions need to be pairwise-independent, so we generate d hash functions in the form of $f(x) = (a*x+b)\ mod\ pr$, where a,b are random integers and pr a prime number.

When a new *key-value* pair has been generated, then each of the d hash functions are applied to it, and for the j corresponding position in the array, a counter increases by one, because one more *value* will be added to this *partition*. Initially: $sketch_m[i,j] = 0, \forall i \in \{1,...,d\}, j \in \{1,...,p\}$. So when all the intermediate pairs have been generated, we have:

$$sketch_m[i,j] = \sum_{\forall k: h_i(k)=j} s_m(k), \forall i \in \{1,...,d\}, j \in \{1,...,p\} \quad (6)$$

When all *map* tasks have finished, the generated sketches are emitted to the Master for the creation of the global sketch array. The global sketch will be also a $d \times p$ array, and will be populated using the following equation:

$$sketch[i,j] = \sum_{m=1}^{M} sketch_m[i,j], \forall i \in \{1,...,d\}, j \in \{1,...,p\} \quad (7)$$

This global sketch holds all the information about the *partitions*' sizes, and will be utilized from the *partitions*' assignment algorithm.

4.2 Dynamic Partitioning Algorithm

Once the *partitions* sizes have been estimated, the goal of the dynamic partitioning algorithm is to decide the placement of the *partitions* to the corresponding *reduce* tasks in a way that minimizes the execution time of the *re-*

duce phase, thus increasing the possibility of jobs meeting their deadlines.

Simple Partitions. In the Simple Partitions scheme, we estimate the execution time of each *partition* assigned on a specific *reduce* task, via the *k-NN* estimator (discussed in Section 3.1). So, we sort the *partitions* with respect to their sizes in descending order and try to find the *reduce* task with the smaller execution time. As a *reduce* task might have some *partitions* already assigned to it, we estimate whether the new assignment will still allow the end-to-end execution times of the currently scheduled tasks to be within their deadline constraints, even if we added the new *partition*. The sorting of the *partitions* ensures that the most heavy-sized will be assigned to *reduce* tasks which run on Workers that exhibit the best performance. Finally the Master returns the corresponding assignment to the *map* tasks, in order to know where to emit the generated *partitions*.

Sketch-based approach. In the Sketch-based approach, the same procedure is applied only this time for each row of the global sketch array. The idea is to generate a *partitions*' assignment for each of the possible hash functions and then choose the function that achieves the least execution time. The function's assignment plan will be utilized for the actual distribution of the *partitions*. This approach is applicable because each row of the global sketch table can be seen as a Simple *Partitions* sizes model. The sketch-based approach adds an additional cost in the *partitions*' assignment procedure because the *partitions*' assignment algorithm must be applied for each row of the sketch array. On the other hand, it increases the possibility of achieving a better *partitions*' assignment, with respect to the execution time, because more assignment plans are considered.

5 Evaluation

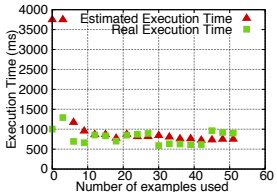
We have performed an extensive experimental study of our approach on Planetlab, a fully distributed heterogeneous environment, using 82 processing cores in total; one dedicated node was utilized as Master and the others were Worker nodes executing *map* and *reduce* tasks. We assume that the Master is failure-free.

Two different jobs were used for the evaluation of our DynamicShare framework. The first job was a Twitter friendship request query on 2GB of available data during the period of Jan 1, 2013 to April 30, 2013, extracted using the Streaming API 2 of Twitter [36], where the goal of each MapReduce job was to identify the unique friends of each user, by examining the tagged and mentioned parts of a tweet. We used a total of 5,900,000 tweets, distributed to fifty-nine available processing cores, each holding about 100,000 tweets. Thus the job consisted of 59 *map* and 23 *reduce* tasks. The job was issued with two different deadlines, the first with $15,000$ ms (strict deadline) and the second with $20,000$ ms (relaxed deadline). The *map* tasks read the available tweets and for each tweet a $(user\_id, list\_of\_friends)$ pair is emitted to the appropriate *reduce* tasks. The latter receive this input and create for each *user_id* the set that contains his unique friends.

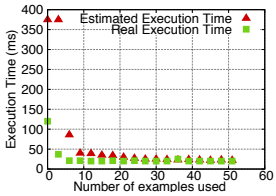
The second job was a friends counting application for a 39MB Youtube [41] social graph. The goal of the job was to calculate the number of unique friends per user, that is, the degree of each node in the social graph. The dataset contained 2,987,628 edges, which were distributed to the available *map* slots leading to approximately 45,000 edges per *map* task. We used the same number of *map* and *reduce* tasks as in the Twitter job, in order to have a fair comparison between the two applications. Youtube jobs were also issued with two different deadlines, to take into account two different type of urgencies strict and relaxed, specifically as strict deadline we used 2000 ms, while as relaxed 4000 ms. These jobs represent commonly issued jobs on Facebook and Yahoo! [6], [30] and thus demonstrate the applicability of our framework in a production workload.

Accuracy of Estimation Model. In the first set of experiments we evaluated the accuracy of our estimation model. In Figure 4 we illustrate the accuracy of our model by comparing the estimated and the actual execution time of a task running on a Worker, as a function of different numbers of previous runs used for the estimation. The results are from the execution runs of one Worker, but similar results were observed for all Workers. As the figure shows, initially when we do not have any previous run of the job's task, the estimated execution time is larger than the actual execution time of the task. However, as tasks execute and more observations become available, the estimates from the *k-NN* estimator are close to the real one.

Laxity Based Scheduling. In the second set of experiments we evaluate the benefit of the least laxity first scheduling (LLF) approach by measuring its ability to meet the deadlines of the jobs. We compared our approach with the following algorithms: (i) Earliest Deadline First (EDF) scheduling as was proposed in [39], where the scheduling criteria is the deadline value, tasks with smaller deadlines will run first. (ii) First In First Out (FIFO) scheduling, where the tasks are ordered based on their arrival order (this is the default scheduling approach used of Hadoop), and (iii) Fair Scheduling (FAIR) scheduling where all tasks are scheduled round-robin so that they get equal time on the available slots. Similarly to [30] we utilized Poisson job arrivals for simulating the assignment of jobs to the framework. A fixed 70% percentage of the assigned jobs had strict deadlines, while the rest had relaxed. This workload mix was used for the

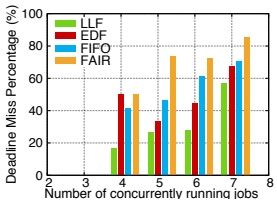


(a) Twitter dataset

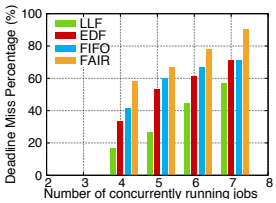


(b) Youtube dataset

Figure 4: Comparison of real and estimated execution time



(a) Twitter dataset



(b) Youtube dataset

Figure 5: Comparison of percentages of deadline misses

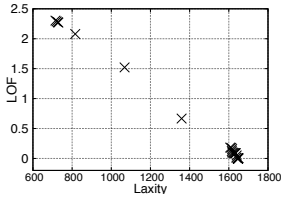


Figure 6: LOF Outlier Detection

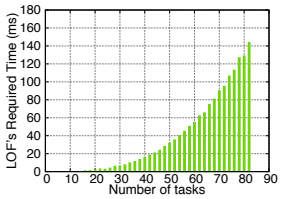
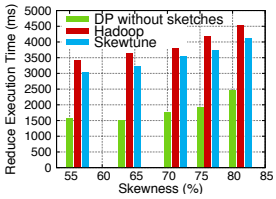
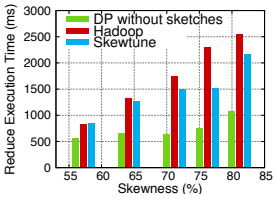


Figure 7: LOF Overhead



(a) Twitter dataset



(b) Youtube dataset

Figure 8: Comparison between Hadoop, DP and Skewtune under varying skewness

evaluation of all four algorithms.

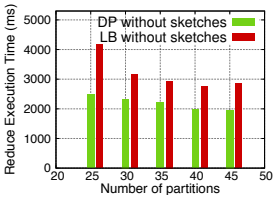
We evaluated the ability of the scheduling algorithms to meet the job deadlines, for varying number of concurrently running jobs. As shown in Figure 5, for a small number of concurrent jobs, all algorithms are able to meet their deadlines. The experiment shows that at all times LLF maintains the smaller percentage of deadline misses. LLF achieves good results even for higher number of jobs, for example when 6 jobs were issued, there was a significant increase on the deadline misses on all the other algorithms due to the increase of the required execution times on some Workers, however, LLF took into account this situation and scheduled the tasks appropriately, thus maintaining few deadline misses.

LOF Evaluation. We now illustrate the working of our anomaly detection algorithm. Due to lack of space we present only the results for the Youtube jobs (as was expected, Twitter jobs had similar results). In Figure 6 we display a snapshot of the LOF's execution. The algorithm identifies overloaded nodes by examining each task's estimated *laxity* value, under normal operation tasks of the same job in different Worker nodes would have similar *laxity* values. In Figure 6 you see that the majority of the tasks have *laxity* values close to 1650 ms. However, five tasks have significant lower *laxity* values resulting to an increase of their lof values and the characterization of the corresponding Workers as overloaded.

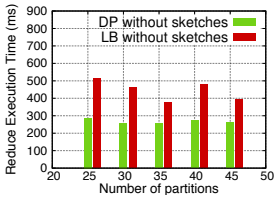
We also examine the overhead of the LOF algorithm in terms of its execution time. The execution time is mainly affected by the number of tasks that report their *laxity* values and need to be examined. So we run multiple *Youtube* jobs with varying number of tasks and report LOF's required execution times (shown in Figure 7). As expected, the execution time increases with the number of tasks. However, the figure shows that the algorithm takes only a few milliseconds to execute, even when it examines 84 tasks.

Dynamic Partitioning. In the last set of experiments we point out the benefits of our proposed *partitions*' assignment algorithm, *Dynamic Partitioning* (*DP*), with respect to handling skewed intermediate data on the *Twitter* and *Youtube* jobs. To avoid the key frequencies skewness, *map* tasks merge for each *key* all the values into a single $(key, list\_of\_values)$ intermediate pair. We calculate the skewness between the generated *partitions* with the following equation: $skewness = 100 - 100 * \frac{\sum_{i=1}^{p} S(i)}{p * max\{S(i)\}}$ The closer the ratio is to 1 the less skewed are considered the generated intermediate pairs. Let P_r be the set that contains the *partitions* processed by *reduce* task r, then the total size of data processed by r will be: $S(P_r) = \sum_{i \in P_r} S(i)$. The achieved balance in regards to the data processed by the *reduce* tasks is given by the following formula: $Balance = 100 * \frac{\sum_{r=1}^{R} S(P_r)}{R * max\{S(P_r)\}}$ The larger the ratio, the more balanced is considered the distribution of the *partitions*, because all *reduce* tasks will process approximately the same amount of data.

We compare the *DP* algorithm utilized by our DynamicShare framework to two schemes: (i) the default *Hadoop* approach where the number of *partitions* is fixed, and equals the number of *reduce* tasks, and (ii) *Skewtune* [27] (*Hadoop*'s most popular enhancement for the skewness issue) that uses the same setting as *Hadoop* but also monitors the execution of *reduce* tasks. When it

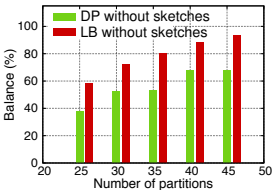


(a) Twitter dataset

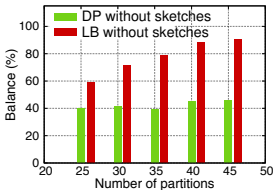


(b) Youtube dataset

Figure 9: Comparison between LB and DP in regards to execution time

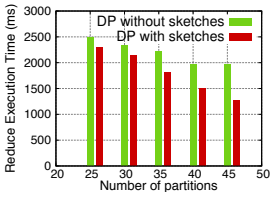


(a) Twitter dataset

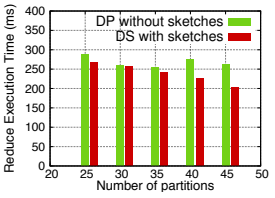


(b) Youtube dataset

Figure 10: Comparison between LB and DP in terms of balance

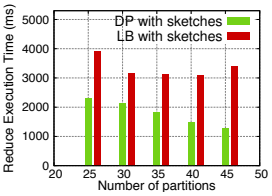


(a) Twitter dataset

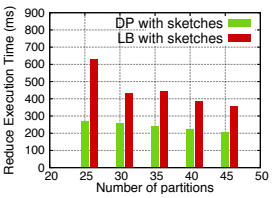


(b) Youtube dataset

Figure 11: Comparison of DP with and without sketches in regards to execution time



(a) Twitter dataset



(b) Youtube dataset

Figure 12: Comparison between LB and DP in regards to execution time when sketches are enabled

detects a task which significantly lacks behind the other normally running, it orders the slow task to repartition its assigned data to the faster *reduce* tasks.

Figure 8 shows the comparison between our algorithm (Simple Partitions schema) with the *Hadoop* MapReduce approach and *Skewtune*. *DP* maintains the execution time of the *reduce* phase low even when the *skewness* reaches 80%. *Skewtune* fails to meet the performance of our algorithm due to the extra overhead of the repartitioning. Although it detects the tasks that suffer from skewed data it requires the repartitioning of their assigned data to the other normally running tasks, a procedure that is beneficial in case of long running jobs as was pointed out in [27], but for short running jobs, such the ones we examine in our work, leads to performance degradation. We do not consider *Skewtune* in case of variable number of *partitions* because it works at *reduce* task level. Regardless of the chosen number of *partitions* to use, some of them will be assigned in a task running on a slow node. These *partitions* will have to be repartitioned thus the overhead will be similar with the presented case.

In the previous experiment we evaluated *DP* with fixed number of *partitions* equal to the *reduce* tasks, in order to have a fair comparison with *Skewtune* and *Hadoop*. To examine the impact of extra *partitions*, we compared our algorithm with the *Load Balance (LB)* algorithm proposed for MapReduce in [14] which also proposes the usage of more *partitions*. The *LB* algorithm strives to maximize the *Balance* metric via a fair distribution of the generated *partitions*. For a fair comparison with our *DP* algorithm when sketches are utilized, we enhanced *LB* with sketches for the estimation of the *partitions*' sizes. In the *partitions*' assignment, each hash function of the global sketch is examined and the one that achieves the best *Balance* is chosen.

The displayed results concern 80% *skewness* between the *partitions*' sizes. Having the same number of *partitions* as *Hadoop* on *LB* does not offer any gain because the *partitions*' assignment will be same as *Hadoop*'s, so we only consider larger values for the number of *partitions*. In Figure 9 you can see the execution time of the *reduce* phase when we use different number of *partitions*. *LB* achieves better results than the default *Hadoop* approach but the *reduce* phase requires higher execution time than our approach. This is mainly due to the fact that our approach is more opportunistic and less "fair" in the work that will be executed from the different tasks. Tasks on nodes with high processing capabilities will process more *heavy-sized partitions* than those that run on slower nodes.

This "unfairness" between the *partitions*' sizes that are processed per *reduce* task, is illustrated in Figure 10. For 40 *partitions*, *LB* achieves approximately 90% *Balance* in the *partitions*' sizes that are processed by the *reduce* tasks for the Twitter job, while our proposal reaches 64%. The difference in the execution times is significant though, as our approach requires approximately 1900 ms while *LB* 2800 ms. The results indicate that in heterogeneous environments, trying to achieve balance between the work assigned on the nodes of the cluster, may not be the right approach. An opportunistic algorithm, such as *DP*, achieves better results as it considers the heterogeneity during the *partitions*' assignment.

Finally we examined the benefit of using sketches in the *partitions*' assignment. Figure 11 shows that sketches mitigate the required execution time of the *reduce* phase as the number of *partitions* increases. This is the expected behaviour since the *key-value* pairs have

more available *partitions* to be spread. The extra hash functions used in the sketches can generate more balance sized *partitions*, thus better decisions are possible for the *DP* algorithm to minimize the *reduce* phase execution time. We examined the requirements of the two approaches in regards to the time they require for deciding the *partitions'* assignment plan. For 45 *partitions*, the no-sketches approach required approximately 80 ms, while when we applied sketches, the assignment required 200 *ms*, so for cases such as the Twitter jobs where the benefits of the sketches approach are larger than 500 ms the extra overhead is negligible. However for very short jobs such as Youtube, a no sketches approach is preferable because the benefits of the sketch-based assignment are overlapped by the extra overhead of the assignment algorithm. In Figure 12, we display results concerning the comparison between *DP* and *LB* when sketches are enabled for both algorithms, although *LB* reduces the required execution time, *DP* still achieves better results.

6 Related Work

Zaharia *et al* [42] were the first to study the problem of scheduling MapReduce jobs in heterogeneous environments. They proposed techniques for prioritizing and scheduling backup copies of slow tasks. Contrary to their work, our approach focuses on meeting real-time response requirements for the tasks and identifies straggling tasks via their *laxity* values. In [1], the authors propose task-stealing solutions in the *map* phase. In an heterogeneous environment like Planetlab, task-stealing could degrade the performance due to the communication overhead between the nodes, and thus could augment the problem. A utility-driven task placement strategy was proposed in [31] using extra processing slots per node when possible. [39] propose the usage of EDF scheduling of user submitted jobs. Our approach differs from them because it does not only consider the real-time demands of the jobs, but also effectively handles the issue of data skewness.

Much work has been done with respect to estimating the execution times of MapReduce tasks such as [29], which utilizes debug runs to calculate the processing speeds of the assigned job. [21] applied *non-parametric* regression for tasks executing in heterogeneous environments, but not on a MapReduce framework, and also using only the input data size as the vectors' value. We were inspired by recent works in automatic anomaly detection ([3], [13], [34], [35]) and adopted the usage of machine learning techniques for the estimation of the tasks' execution times.

Focus on the skewed data impact on MapReduce was mainly expressed by [14], [15], [27] and [28]. We compared our approach with these proposals and displayed results that indicate their inapplicability in our setting. Techniques like [14] and [15] aim to equally distribute the *partitions* to the available *reduce* tasks, however as we pointed out in the experiments such decision is not beneficial in a heterogeneous environment. [27] adds the overhead of repartitioning the assigned *partitions*, an overhead which deteriorates the performance of short jobs and can lead to deadline misses. [32] requires a pre-processing step for estimating the *partitions'* sizes, adding overhead in the calculations thus making the execution of short jobs impossible.

Authors in [16] propose a new abstraction on top of Hadoop, the *Shuffler* component which is responsible for keeping the received partitions in the node where the *reduce* tasks will run. The newly added component enables intermediate data transmission between the *map* and *reduce* phase, reducing the cost of the *shuffle* phase. However, as it was pointed out in [7] when the intermediate data to be emitted are rather small, as in our case, the benefits of online transmission are negligible.

7 Conclusion

In this paper we study the problem of scheduling real-time skewed MapReduce jobs in heterogeneous environments. We propose a holistic approach based on (a) a non-parametric regression technique for estimating the execution times of the tasks, (b) dynamic distributed least laxity first scheduling algorithm for scheduling jobs end-to-end, (c) techniques for identifying straggling nodes, and (d) dynamic partitioning algorithms to handle the impact of the data skewness on the execution times of the tasks. Our experimental results on Planetlab indicate a clear improvement in the system's performance. In our future work we aim at examining if it is possible to dynamically decide the number of *partitions* used per job. This decision will enable us to balance the trade-off between *reduce* phase execution time and the computation overhead of the *partitions'* assignment algorithm.

8 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 Social Fund, the FP7 INSIGHT project and the ERC IDEAS NGHCS project.

References

[1] AHMAD, F., CHAKRADHAR, S., RAGHUNATHAN, A., AND VIJAYKUMA, T. Tarazu: Optimizing MapReduce On Heterogeneous Clusters. *ASPLOS, London, UK* (2012).

[2] AMAZON'S DYNAMODB. `http://aws.amazon.com/dynamodb/`.

[3] BODÍK, P., GRIFFITH, R., SUTTON, C., FOX, A., JORDAN, M., AND PATTERSON, D. Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters. *HotCloud* (2009).

[4] BOUTSIS, I., AND KALOGERAKI, V. Resource Management using Pattern-based Prediction to Address Bursty Data Streams. *ISORC 2013, Paderborn, Germany* (2013).

[5] BREUNIG, M. M., KRIEGEL, H.-P., T.NG, R., AND SANDER, J. LOF: Indentifying Density-Based Local Outliers. *SIGMOD* (2000).

[6] CHEN, Y., GANAPATHI, A., GRIFFITH, R., AND KATZ, R. The Case for Evaluating MapReduce Performance Using Workload Suites. *MASCOTS* (2011).

[7] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEEGY, K., AND SEARS, R. MapReduce Online. *No. UCB/EECS-2009-136* (2009).

[8] CORMODE, G., AND HADJIELEFTHERIOU, M. Finding Frequent Items in Data Streams. *Proceedings of the VLDB Endowment Volume 1 Issue 2, August, Pages 1530-1541* (2008).

[9] CORMODE, G., AND MUTHUKRISHNAN, S. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms Volume 55, Issue 1, April 2005, Pages 5875* (2005).

[10] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *OSDI, San Francisco, CA* (2004).

[11] DOBRA, A., GAROFALAKIS, M., GEHRKE, J., AND RASTOGI, R. Processing Complex Aggregate Queries over Data Streams. *SIGMOD* (2002).

[12] DOU, A. J., KALOGERAKI, V., GUNOPULOS, D., MIELIKINEN, T., AND TUULOS, V. Scheduling for real-time mobile mapreduce systems. *DEBS 2011, New York, New York* (2011).

[13] FOX, A., KICIMAN, E., AND PATTERSON, D. Combining Statistical Monitoring and Predictable Recovery for SelfManagement. *WOSS* (2004).

[14] GUFLER, B., AUGSTEN, N., REISER, A., AND KEMPER, A. Handling Data Skew In MapReduce. *CLOSER* (2011).

[15] GUFLER, B., AUGSTEN, N., REISER, A., AND KEMPER, A. Load Balancing in MapReduce Based on Scalable Cardinality Estimates. *ICDE* (2012).

[16] GUO, Y., RAO, J., AND ZHOU, X. iShuffle: Improving Hadoop Performance with Shuffle-on-Write. *Presented as part of the 10th International Conference on Autonomic Computing* (2013).

[17] HADOOP. `http://lucene.apache.org/hadoop`.

[18] HADOOP CAPACITY SCHEDULER. `http://hadoop.apache.org/common/docs/r0.19.2/capacity_scheduler.html`.

[19] HADOOP FAIR SCHEDULER. `http://hadoop.apache.org/mapreduce/docs/r0.21.0/fair_scheduler.html`.

[20] IBM INFOSPHERE BIGINSIGHTS. `http://www-01.ibm.com/software/data/infosphere/biginsights/`.

[21] IVERSON, M. A., ÖZGÜNER, F., AND J.FOLLEN, G. Run-Time Statistical Estimation of Task Execution Times for Heterogeneous Distributed Computing. *HPDC* (1996).

[22] JIN, H., YANG, X., SUN, X.-H., AND RAICU, I. ADAPT: Availability-aware MapReduce Data Placement for Non-Dedicated Distributed Computing. *ICDCS, page 516-525. IEEE* (2012).

[23] KALOGERAKI, V. Resource management for real-time fault-tolerant distributed systems. *PhD Thesis, Univ. of California Santa Barbara* (2000).

[24] KALOGERAKI, V., MELLIAR-SMITH, P. M., AND MOSER, L. E. Dynamic scheduling of distributed method invocations. *RTSS* (2000).

[25] KARDOSA, M., AND CHANDRA, A. Resource Bundles: Using Aggregation for Statistical Wide-Area Resource Discovery and Allocation. *ICDCS* (2008).

[26] KOLB, L., THOR, A., AND RAHM, E. Load Balancing for MapReduce-based Entity Resolution. *ICDE* (2012).

[27] KWON, Y., BALAZINSKA, M., HOWE, B., AND ROLIA, J. SkewTune: Mitigating Skew in MapReduce Applications. *SIGMOD* (2012).

[28] LIU, Y., LI, M., ALHAM, N. K., HAMMOUD, S., AND PONRAJ, M. Load balancing in MapReduce environments for data intensive applications. *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on Shanghai 26-28 July* (2011).

[29] MORTON, K., FRIESEN, A., BALAZINSKA, M., AND GROSSMAN, D. Estimating the Progress of MapReduce Pipelines. *ICDE 2010* (2010).

[30] PALANISAMY, B., SINGH, A., LIU, L., AND LANGSTON, B. Cura: A Cost-optimized Model for MapReduce in a Cloud. *IPDPS* (2013).

[31] POLO, J., CASTILLO, C., CARRERA, D., BECERRA, Y., WHALLEY, I., STEINDER, M., TORRES, J., AND AYGUADÉ, E. Resource-Aware Adaptive Scheduling for MapReduce Clusters. *Middleware'11 Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware Pages 187-207* (2011).

[32] RAMAKRISHNAN, S. R., SWART, G., AND URMANOV, A. Balancing reducer skew in MapReduce workloads using progressive sampling. *SoCC '12 Proceedings of the Third ACM Symposium on Cloud Computing* (2012).

[33] SCOTT, D. Multivariate density estimation: Theory, practice and visualization. *Theory, Practice and Visualization. Wiley & Sons* (1992).

[34] TAN, Y., GU, X., AND WANG, H. Adaptive System Anomaly Prediction for Large-Scale Hosting Infrastructures. *PODC* (2010).

[35] TAN, Y., NGUYEN, H., SHEN, Z., GU, X., VENKATRAMANI, C., AND RAJAN, D. PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems. *ICDCS* (2012).

[36] TWITTER. `http://twitter.com`.

[37] UNDER THE HOOD: SCHEDULING MAPREDUCE JOBS MORE EFFICIENTLY WITH CORONA. https://www.facebook.com/notes/facebook-engineering/under-the-hood-scheduling-mapreduce-jobs-more-efficiently-with-corona/10151142560538920.

[38] VERMA, A., CHERKASOVA, L., AND CAMBELL, R. H. Resource Provisioning Framework for MapReduce Jobs with Performance Goals. *Proceedings of the 12th ACM/IFIP/USENIX International Middleware Conference (Middleware'2011), Lisboa, Portugal, December 12-16* (2011).

[39] VERMA, A., CHERKASOVA, L., KUMAR, V. S., AND CAMBELL, R. H. Deadline-based Workload Management for MapReduce Environments: Pieces of the Performance Puzzle. *Network Operations and Management Symposium (NOMS), 2012 IEEE* (2012).

[40] W.HÄRDLE. *Applied nonparametric regression*. Cambridge University Press, 1990.

[41] YOUTUBE. `http://www.youtube.com`.

[42] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R., AND STOICA, I. Improving MapReduce Performance in Heterogeneous Environments. *OSDI* (2008).