Load Balancing in MapReduce Based on Scalable Cardinality Estimates

Benjamin Gufler [†1], Nikolaus Augsten [#2], Angelika Reiser [†3], Alfons Kemper [†4]

[†]*Technische Universität München*
Boltzmannstraße 3, 85748 Garching bei München, Germany
[1] benjamin.gufler@in.tum.de
[3] angelika.reiser@in.tum.de
[4] alfons.kemper@in.tum.de

[#]*Free University of Bozen-Bolzano*
Dominikanerplatz 3, 39100 Bozen, Italy
[2] augsten@inf.unibz.it

***Abstract*—MapReduce has emerged as a popular tool for distributed and scalable processing of massive data sets and is increasingly being used in e-science applications. Unfortunately, the performance of MapReduce systems strongly depends on an even data distribution, while scientific data sets are often highly skewed. The resulting load imbalance, which raises the processing time, is even amplified by the high runtime complexities of the reducer tasks. An adaptive load balancing strategy is required for appropriate skew handling.**

In this paper, we address the problem of estimating the cost of the tasks that are distributed to the reducers based on a given cost model. A realistic cost estimation is the basis for adaptive load balancing algorithms and requires to gather statistics from the mappers. This is challenging: (a) Since the statistics from all mappers must be integrated, the mapper statistics must be small. (b) Although each mapper sees only a small fraction of the data, the integrated statistics must capture the global data distribution. (c) The mappers terminate after sending the statistics to the controller, and no second round is possible. Our solution to these challenges consists of two components. First, a monitoring component executed on every mapper captures the local data distribution and identifies its most relevant subset for cost estimation. Second, an integration component aggregates these subsets and approximates the global data distribution.

I. Introduction

MapReduce [1] has emerged as a popular tool for processing batch-style jobs on massive data sets in cloud computing environments. Both industry and science are porting analytical workflows to this programming pattern in order to exploit the elasticity of the cloud. However, the simplistic assumptions of current systems about the data distribution limit the scaling of complex applications. As also identified by prior work [2], [3], data skew, i. e., clearly non-uniform distributions of single attribute values, and complex, non-linear reducer algorithms are the key problems that limit the scalability of MapReduce for analytic applications, for example, in e-science.

The mappers in a MapReduce system produce (key,value) pairs. The pairs are grouped by their keys and an individual group (called a cluster) is guaranteed to be processed by a single reducer. Each reducer is loaded with a set of clusters that it must process. The overall runtime of a map-reduce cycle is determined by the longest running reducer. The next cycle can only start when all reducers are done. Unbalanced reducer work loads lead to high runtime differences, parallelism is poorly exploited, and the overall runtime increases. In e-science applications we experienced runtime differences of hours between the reducers.

Current MapReduce frameworks like Apache Hadoop[1] use a simplistic approach to distribute the work load and assign the same number of clusters to each reducer. This approach fails if the key distribution is skewed since the individual clusters have different sizes. Reducers with many large clusters have a high work load since they must process more tuples. If in addition the reducer task is non-linear, it is not enough to load each reducer with the same number of tuples. For example, a reducer with runtime complexity n^3 that processes two clusters with a total of 6 tuples requires $3^3 + 3^3 = 54$ operations if both clusters are of size 3, but $1^3 + 5^3 = 126$ operations, i.e., twice as many, if the clusters' sizes are 1 and 5.

Data skew is a well-studied phenomenon in distributed databases and solutions for operations like joins and grouping/aggregation have been proposed [4], [5], [6]. However, these solutions are not generally applicable to MapReduce systems. The key finding of the database solutions is that tuples sharing the same value of the partitioning attribute need not necessarily be processed as a single group. MapReduce systems, on the other hand, provide the user with the guarantee that all tuples sharing the same partitioning key, i. e., all tuples of a cluster, are processed on the same reducer.

In prior work [2] we presented two load balancing algorithms, fine partitioning and dynamic fragmentation, that seamlessly integrate with current MapReduce systems. The clusters are grouped into partitions that are distributed to the reducers depending on their cost. The cost of a partition is the sum of the costs of its clusters. In order to give each reducer a similar amount of work, the cost of each partition must be computed. The quality of the load balancing is determined by the quality of the cost estimation.

Computing the partition costs is challenging. Each mapper

[1] http://hadoop.apache.org/mapreduce

sees only a small fraction of the data and has only partial information about the cluster sizes. In particular, the mapper can not know which fraction of a specific cluster it sees. Sending all partial cluster sizes to the controller and summing the costs for all clusters centrally is not feasible since the number of clusters can be in the order of the data size. The controller must base its cost estimation on small summaries. In addition, not all mappers do necessarily run at the same time. Thus the controller cannot incrementally retrieve information as is done, for example, in distributed top-k scenarios, where the distributed rankings are incrementally consumed until the central ranking is accurate enough.

In this paper we present TopCluster, a sophisticated distributed monitoring approach for MapReduce systems. TopCluster requires a single parameter, the cluster threshold τ, which controls the size of the local statistics that are sent from each mapper to the controller. The result is a global histogram of (key,cardinality) pairs, which approximates the cardinalities of the clusters with the most frequent keys. TopCluster provides the following guarantees:

- *Completeness:* All clusters with cardinalities above the cluster threshold τ are in the global histogram.
- *Error Bound:* The approximation error of the cluster cardinalities is bound by $\tau/2$.

The global histogram is used to estimate the partition cost. Since the global histogram contains the largest clusters, the data skew is considered in the cost estimation. For the remaining clusters, i. e., the clusters that are not present in the global histogram, TopCluster assumes uniform distribution and their total cost is efficiently computed in constant time. Capturing the largest clusters with high precision is important for an accurate cost estimation.

We further provide a strategy to automatically choose a good threshold τ based on the data skew and discuss an extension of TopCluster which approximates the local histograms that must be computed on the mappers.

This paper is structured as follows. We revisit the partition cost model for MapReduce and discuss a base line technique for monitoring cluster cardinalities in Section II. Our distributed monitoring approach, TopCluster, is presented in Section III. Formal error bounds for TopCluster are derived in Section IV. We discuss optional extensions to TopCluster for environments with limited memory resources, and for higher-dimensional monitoring in Section V. An experimental evaluation of TopCluster is given in Section VI. We discuss related work in Section VII, and finally conclude and point to future work in Section VIII.

II. Exact Global Histograms

In this section we briefly introduce the architecture of MapReduce systems, revisit the cost model underlying our load balancing, and introduce the exact global ranking, which is approximated by our TopCluster algorithm.

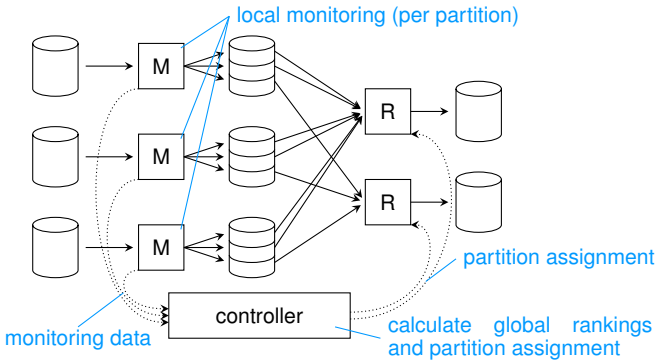


Fig. 1: MapReduce Architecture

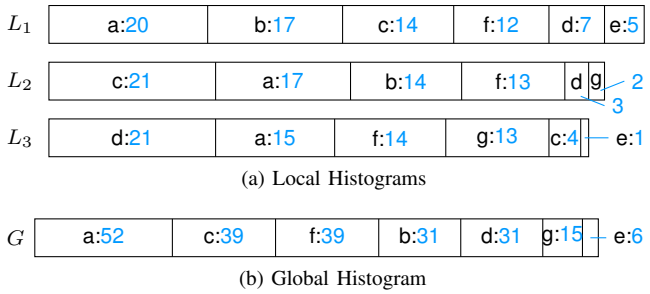
A. The MapReduce Architecture

Figure 1 illustrates the architecture of MapReduce systems. The input is split into blocks of constant size and the tuples of each block are processed independently by a *mapper* (M). Since the block size per mapper is constant, the number of mappers depends on the data size, and the mappers do not necessarily run concurrently. A mapper transforms its input tuples into (key,value) pairs. The set of all (key,value) pairs is called *intermediate data*. The intermediate data are hash-partitioned by their keys. Each partition is written to a separate file on disk. Since all mappers employ the same hash function for the partitioning, all tuples sharing the same key, called a *cluster*, are assigned to the same partition. The *controller* assigns the partitions to *reducers* (R), which process the partitions cluster by cluster. This processing scheme is guaranteed by the MapReduce paradigm and cannot be changed by the load balancing algorithm.

B. The Partition Cost Model

We briefly revisit the *partition cost model* [2] for load balancing in MapReduce, which underlies our TopCluster algorithm. The partition cost model takes into account both skewed data distributions and complex reducer side algorithms. The cost for each partition is computed and the partitions are distributed to the reducers such that the work load per reducer is balanced. Since the clusters within a partition are processed sequentially and independently, the partition cost is computed as the cost sum of all clusters in the partition. The cluster cost, in turn, is a function of the cluster cardinality and the complexity of the reducer side algorithm. While the reducer complexity is a parameter specified by the user, the cluster cardinalities must be monitored by the framework.

Monitoring the cluster cardinalities is challenging. The computation of the partition cost is treated as a black box in the partition cost model. The only approach presented in previous work [2] assumes uniform distribution of cluster cardinalities within each partition. This simplistic assumption limits the effectiveness of the load balancing algorithm when the data are skewed. Our TopCluster algorithm is a sophisticated approach to monitor the cluster cardinalities. TopCluster takes into



(a) Local Histograms

(b) Global Histogram

Fig. 2: Local and Exact Global Histograms.

account data skew and allows us to compute good partition cost estimations also when the data are highly skewed.

C. Global Histogram

We introduce the *exact global histogram*, which stores all information required to compute the exact cost for all partitions. Unfortunately, computing the exact global histogram is not feasible for large data sets. Our TopCluster algorithm is an efficient and effective approximation of the exact global histogram. We use the exact global histogram as a baseline to assess the quality of our approximation.

Definition 1 (Local Histogram): Let I_i be the bag of all intermediate (key,value) pairs produced by mapper i. The *local histogram* L_i is defined as a set of pairs (k, v), where $k \in \{x \mid \exists y((x, y) \in I_i)\}$ is a key in I_i and v is the number of tuples in I_i with key k.

Definition 2 (Global Histogram): Given m local histograms L_i, $1 \leq i \leq m$, of pairs (k, v), where k is the key and v is the associated cardinality, and the presence indicator $p_i(k)$, which is true for k if and only if k exists in L_i,

$$p_i(k) = \begin{cases} \text{true} & \text{if } \exists v((k, v) \in L_i), \\ \text{false} & \text{otherwise,} \end{cases}$$

the *global histogram* G is the set $\{(k, v)\}$ with

(i) $\exists v((k, v) \in G) \Leftrightarrow \exists i(p_i(k) = \text{true})$

(ii) $\forall (k, v) \in G : v = \sum_{\substack{1 \leq i \leq m \\ (k,v') \in L_i}} v'$

The global histogram maps all keys in the intermediate data I to the cardinality of the respective cluster. It is a sum aggregate over all local histograms. Since a global cluster can consist of 1 to m local clusters, the cardinality of the exact global histogram is bounded by

$$\max_{1 \leq i \leq m} |L_i| \leq |G| \leq \sum_{1 \leq i \leq m} |L_i|.$$

Example 1: We compute the global histogram from the local histograms in a scenario with $m = 3$ mappers:

$$L_1 = \{(a, 20), (b, 17), (c, 14), (f, 12), (d, 7), (e, 5)\}$$
$$L_2 = \{(c, 21), (a, 17), (b, 14), (f, 13), (d, 3), (g, 2)\}$$
$$L_3 = \{(d, 21), (a, 15), (f, 14), (g, 13), (c, 4), (e, 1)\}$$

The exact global histogram (see Figure 2) is

$$G = \{(a, 52), (c, 39), (f, 39), (b, 31), (d, 31), (g, 15), (e, 6)\}.$$

Lemma 1: Consider a scenario with m mappers, which each produces $O(|I|/m)$ tuples for the intermediate result I. The computation of the local histogram requires $O(|I|/m \log(|I|/m))$ time and $O(|I|/m)$ space; the computation of the exact global histogram requires $O(|I| \log |I|)$ time and $O(|I|)$ space.

Proof: In the worst case, all keys are unique such that the number of clusters is equal to the number of tuples. The local histogram for $O(|I|/m)$ clusters requires $O(|I|/m)$ counters. We store and update the counters in a binary search tree, which requires $O(|I|/m \log(|I|/m))$ time and $O(|I|/m)$ space. Since the keys of all mappers can be different, the size of the global histogram can be of size $O(|I|)$. The global histogram is aggregated from the local histograms using an m-way merge in $O(|I| \log |I|)$ time. ■

Lemma 1 assumes that all mappers generate a similar number of tuples. This is a reasonable assumption since each mapper gets the same share of the input data. The number of mappers, m, increases with the size of the overall input such that the input per mapper is constant. Thus the space complexity $O(|I|/m)$ is independent of the overall size of the input data. For linear mappers processing a few hundred megabytes of data (the typical case), the local histogram is efficiently computed on each mapper. For mappers that produce very large intermediate results, such that the exact local histogram computation is too expensive, we present an approximate solution in Section V-B.

The complexity of the global histogram computation is a problem. Recall that $|I|$ is the data volume that is processed in a distributed manner in the reduce phase. This data volume is of the same scale as the input data to the MapReduce job or even larger [3]. Storing and processing monitoring data of this size on a central controller is infeasible and approximate solutions must be applied.

D. Problem Statement

Our goal is to develop an algorithm that approximates the global histogram. The approximation algorithm must be efficient, and the complexity of the algorithm should be independent of the data volume, $|I|$. The approximation error should be small, also in the presence of heavy data skew.

We measure the error of an approximation to the exact global histogram as the percentage of tuples that the approximated histogram assigns to a different cluster than the exact histogram. For the error computation, we do not identify the clusters by their key, but we order the clusters by their size and compare clusters with the same ordinal number in sort order, i. e., we compare the largest clusters from the exact and approximated histograms, then the second-largest clusters, etc. This is reasonable since the processing cost of a cluster in the partition cost model is independent of its key.

Example 2: Consider the exact histogram $G = \{(a, 20), (b, 16), (c, 14)\}$ and the approximated histogram $\tilde{G} = \{(a, 20), (c, 17), (b, 13)\}$. The difference between the largest clusters in both these histograms is 0. For the second largest, b in G and c in $\tilde{G}$, we note a difference of 1, the

same as for the smallest clusters. So, in total we have a difference of $0 + 1 + 1 = 2$ tuples. Both the histograms contain information on 50 tuples. As every tuple assigned to a wrong cluster is counted twice (once for the cluster it is missing in, and once for the cluster it is assigned to), we obtain an approximation error of $1/50 = 2\%$.

III. TopCluster

The exact global histogram discussed in the preceding section is infeasible for large scale data of, e. g., e-science applications. In this section we propose TopCluster, an approximation algorithm for the global histogram, which scales to massive data sets. The quality guarantees of TopCluster are discussed in Section IV.

A. Overview

Given the cluster threshold τ, the TopCluster algorithm computes an approximation of the global histogram for each partition in three steps.

1) Local histogram (mapper): For each partition a local histogram L_i is maintained on mapper i.
2) Communication: When mapper i is done, for each partition it sends the following information to the controller: (a) the presence indicator for all local clusters and (b) the histogram for the largest local clusters (histogram head).
3) Global histogram (controller): The controller approximates the global histogram by (a) sum-aggregating the heads of the local histograms from all m mappers and (b) by computing a cardinality estimation for all clusters that are not in the local histograms.

To simplify the discussion, we consider the computation of the global histogram for a single partition. The same procedure is repeated for each partition.

B. The Head of the Local Histograms

The local histograms L_i are too large to be processed by the controller. Each mapper i, $1 \leq i \leq m$, sends only the head of the local histogram, which contains only the largest clusters.

Definition 3 (Local Histogram Head): Given a local histogram L_i and a local threshold τ_i, the head of the histogram, $L_i^{\tau_i}$, is defined as a subset of L_i such that the cardinalities of all clusters is at least τ_i; if there is no cluster of size τ_i or larger, the next smallest cluster(s) is (are) also in the head $L_i^{\tau_i}$:

$$L_i^{\tau_i} = \{(k,v) \in L_i \mid v \geq \tau_i \vee (v < \tau_i \wedge \nexists (k', v') \in L_i (v < v'))\}$$

The local threshold τ_i can be different for all local histograms; the sum of all local thresholds is the global cluster threshold τ, which is the input parameter for TopCluster. In the basic TopCluster algorithm we choose the local threshold to be $\tau_i = \tau/m$. In Section V we discuss an extension of TopCluster that adapts τ_i for each histogram based on the skew of the data. The following discussion holds independently of the choice of the local threshold τ_i.

C. Approximating the Global Histogram

The approximated global histogram has a *named* and an *anonymous* part. The named part is a histogram which stores cluster cardinalities for specific key values. The goal is to have the largest clusters in the named part of the global histogram. The cardinalities of all other keys are covered by the anonymous part. The clusters in the anonymous part have no name and we do not store values for each cluster; we only know the number of clusters and their average size. This is enough for the cost estimation. Since the largest clusters are in the named part, the error introduced by the assumption of a uniform distribution on the anonymous part is small.

a) Lower and Upper Bound Histograms: We define the upper and lower bound histograms, which are used to define the global histogram approximation. As we will show in Section IV, the cardinalities of all clusters in the lower/upper bound histograms are lower/upper bounds of the exact cardinality values of the respective clusters.

Definition 4 (Upper and Lower Bound Histogram): Given the head of m local histograms $L_i^{\tau_i}$, $1 \leq i \leq m$. The *lower bound histogram* G_l is defined as follows:

(i) $\exists v((k,v) \in G_l) \Leftrightarrow \exists i, v'((k,v') \in L_i^{\tau_i})$
(ii) $\forall (k,v) \in G_l : v = \sum_{\substack{1 \leq i \leq m \\ (k,v') \in L_i^{\tau_i}}} v'$

Let p_i be the presence indicator for L_i, let v_i be the smallest value in $L_i^{\tau_i}$, i. e.

$$v_i = \min_v \{(k,v) \in L_i^{\tau_i}\} \qquad 1 \leq i \leq m$$

The *upper bound histogram* G_u is defined as

(i) $\exists v((k,v) \in G_u) \Leftrightarrow \exists i, v'((k,v') \in L_i^{\tau_i})$
(ii) $\forall (k,v) \in G_u : v = \sum_{1 \leq i \leq m} \mathrm{val}(k,i)$ with

$$\mathrm{val}(k,i) = \begin{cases} v' & \text{if} (k,v') \in L_i^{\tau_i} \\ v_i & \text{if } p_i(k) \wedge \nexists v'((k,v') \in L_i^{\tau_i}) \\ 0 & \text{otherwise} \end{cases}$$

Note that both G_l and G_u contain values for the clusters that appear in at least one of the local histogram heads, so $|G_l| = |G_u|$, and the set of keys of both histograms is identical. The number of items in G_l and G_u is bounded by the largest local histogram head on the lower end (if this largest local histogram head contains all keys which appear in any of the other local histogram heads), and the sum over all local histogram heads (if all keys in the local histogram heads are distinct):

$$\max_{1 \leq i \leq m} |L_i^t| \leq |G_l| = |G_u| \leq \sum_{1 \leq i \leq m} |L_i^t|$$

Example 3: The head of the local histograms $L_i^{\tau_i}$ extracted from the local histograms introduced in Example 1 for $\tau_i = 14$ are shown in Figure 3. Key a is contained in all three local histogram heads. Therefore, its exact value is known, and the upper and lower bounds coincide: $20+17+15 = 52$. Key c is not contained in L_3^{14}. The corresponding lower bound is thus $14+21+0 = 35$. From $p_3(c) = \mathrm{true}$, we know c occurred in L_3. As $v_3 = 14$, the upper bound for b is $14+21+14 = 49$. Key b is not contained in L_3^{14} as well. The lower bound is

L_1^{14}	a:20	b:17	c:14	...(24)
L_2^{14}	c:21	a:17	b:14	...(18)
L_3^{14}	d:21	a:15	f:14	...(18)

Fig. 3: Head of Local Histograms for $\tau_i = 14$.

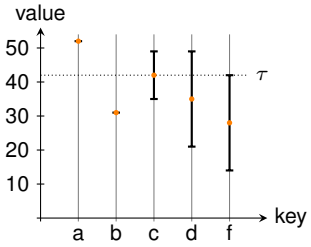


Fig. 4: Global Bounds

$17 + 14 + 0 = 31$. As $p_3(f) = \text{false}$, we know f was not contained in L_3 and we can use 0 in the calculation of the upper bound of f for local histogram 3: $17 + 14 + 0 = 31$. The bounds for the remaining keys are calculated analogously. We obtain the following bounds, which are also visualised in Figure 4 (the orange dots are explained in Example 4).

$$G_l = \{(a, 52), (c, 35), (b, 31), (d, 21), (f, 14)\}$$
$$G_u = \{(a, 52), (c, 49), (d, 49), (f, 42), (b, 31)\}$$

b) Named Histogram Part: We define two approximations for the global histogram. The *complete* histogram stores a cardinality value for all keys that exist in at least one local histogram head, the *restrictive* histogram is the subset of the complete histogram in which all cardinalities are at least τ.

Definition 5 (Global Histogram Approximation): Let G_l and G_u be the pair of lower and upper bound histograms in a given setting. The *complete global histogram* is defined as

$$\widetilde{G} = \left\{ \left(k, \frac{v_u + v_l}{2} \right) : (k, v_u) \in G_u \wedge (k, v_l) \in G_l \right\},$$

the *restrictive global histogram* is defined as

$$\widetilde{G_r} = \left\{ (k, v) \in \widetilde{G} \wedge v \geq \tau \right\}.$$

Example 4: We approximate the global histogram with the upper and lower bounds computed in Example 3. For the complete global histogram approximation, we obtain $\widetilde{G} = \{(a, 52), (c, 42), (d, 35), (b, 31), (f, 28)\}$. These values are symbolised by the orange dots in Figure 4. The restrictive global histogram approximation ($\tau = 42$) is $\widetilde{G_r} = \{(a, 52), (c, 42)\}$.

Although the restrictive histogram approximation has a longer anonymous part in which uniform distribution is assumed, it outperforms the complete histogram, in particular when the data skew is small. This is explained as follows.

An approximation error is only introduced if a cluster exists in a local histogram, but is not in the head. This is a frequent situation for almost uniform distributions. For a key k that is not in the head of a local histogram, $L_i^{\tau_i}$, but $p_i(k) = \text{true}$, we add v_i (the smallest value in $L_i^{\tau_i}$) to the upper bound, and 0 to the lower bound. The cardinality on mapper i for that cluster is estimated as the arithmetic mean $\frac{v_i}{2}$, even though, with an almost uniform data distribution, the real cardinality is likely to be close to v_i. The estimated global cardinality of such a cluster is typically slightly larger than $\frac{\tau}{2}$, but smaller than τ and is thus not contained in $\widetilde{G_r}$.

Example 5: Consider the cardinality estimation for the cluster with key f (see Figures 3 and 4). Although key f exists in all three local histograms, it is not in the heads of L_1 and L_2. With $v_1 = v_2 = 14$, we approximate the cardinality of cluster f with $14/2 = 7$ for both local histograms, leading to a value 28 in the global histogram approximation. The correct value is 39. Cluster f is not included in the restrictive histogram since its estimated cardinality is below $\tau = 42$.

c) Anonymous Histogram Part: The named part of the global histogram approximation contains cardinality values only for the largest clusters. In order to compute the partition cost, however, we need to consider *all* clusters in the partition. We assume uniform distribution on the small clusters that are not in the named histogram part. To compute the average cluster size, we determine the global sum of cluster cardinalities and the number of different clusters. The sum of the cluster cardinalities is easy to obtain by summing up all local tuple counts monitored on the mappers. The number of clusters, however, cannot be computed by counting all local clusters since two local clusters with the same key form a single global cluster and should not be counted twice. We discuss the computation of the cluster count in Section III-D.

Example 6: The sum of cluster cardinalities for the three local histograms in Example 1 is $75 + 70 + 68 = 213$. The global cluster count is 7. The cardinality sum of the restrictive approximation, $\widetilde{G_r}$, in Example 4 is $52 + 42 = 94$. Thus, for each of the four anonymous clusters we estimate a value of $\frac{213-94}{7-2} = 23.8$ tuples. We compute the approximation error as introduced in Section II-D. The absolute difference between the exact and the approximated clusters is 59.2, thus $59.2/2 = 29.6$ tuples were assigned to the wrong cluster. Even though the approximate global histogram provides no information on 5 out of 7 clusters (more than 70%), less than 14% of the tuples were assigned to the wrong cluster. For a reducer with n^2 complexity, we obtain an estimated cost of 7300.2, as opposed to the exact cost of 7929; an error of less than 8%.

D. Approximating the Presence Indicator p_i

So far, we assumed the presence indicators p_i to provide exact information. This may not be feasible since each single cluster must be monitored and the number of clusters may be $O(|I|)$. We replace the exact presence indicator by an approximation, $\widetilde{p_i}$, which is implemented as a bit vector of fixed length. We use this bit vector like a Bloom filter [7] on the controller in order to check for the presence of clusters whose keys were reported by other mappers. $\widetilde{p_i}$ may introduce false positives, but cannot introduce false negatives.

False positives impact the quality of the approximation of the global histogram. Recall from the preceding section that we calculate the approximated value of an item as the arithmetic mean of its upper and lower bound. The lower bound, G_l, is not affected by our approximation of p_i, as we do not employ p_i in the respective calculation. The upper bound, G_u, however, may change. Consider the definition of $\mathrm{val}(k,i)$ in Definition 4. For a key k that is not contained in the local histogram L_i, we must add 0. In case of a false positive on $p_i(k)$, however, we add v_i (the smallest value in the local histogram head $L_i^{\tau_i}$) to the upper bound, thereby overestimating the upper bound. As false negatives are impossible, we will never underestimate the bound. Hence, the upper bound remains in place, but it may become looser in case of false positives.

Regarding the presented variants of the approximated global histogram, the influence of approximating p_i is as follows. If we overestimate the upper bound of a cluster, the estimated cluster cardinality raises as well. Therefore, the actual values in the approximated global histogram may change as a consequence of approximating p_i. As the complete approximation contains all items occurring in any local histogram head, approximating p_i has no influence on which items are included in the complete approximation. The restrictive approximation chooses items based on their average values. If we overestimate the upper bound of an item, its average value rises as well. Hence, items may be included in the restrictive approximation of the global histogram which would not have been included with exact p_i.

Example 7: We approximate p_i with a bit vector of length 3, based on a hash function h mapping character keys a to 0, b to 1 etc. (mod 3). For the histogram heads with $\tau_i = 14$ from Example 3, we will have a false positive for key b on local histogram L_3, as $p_3(b) = \text{false}$, but $\widetilde{p}_3(b) = \text{true}$ ($h(b) = h(e)$), and key e is contained in local histogram L_3. We therefore calculate the upper bound for b as $17+14+14 = 45$ instead of $17+14+0 = 31$. In consequence, the estimated value for b in $\widetilde{G}$ increases from 31 to 38.

For the anonymous part of the histogram we need the number of clusters in the global histogram. We estimate the cluster count re-using the bit vectors introduced for approximating p_i. We calculate the disjunction of all bit vectors for a partition. Linear Counting [8] then allows us to estimate the number of clusters based on the bit vector length and the ratio of reset bits, taking into account also the probability of hash collisions.

IV. Approximation Guarantees

A. Upper and Lower Bounds

Since only the head of each local histogram is sent to the controller, it is not possible to compute the exact global histogram, G, at the controller. The approximate global histogram presented in the previous section takes values between the lower and upper bound histograms, G_l and G_u. We show that G_l and G_u are also respectively lower and upper bounds for the exact global histogram G.

Theorem 1: G_l is a lower bound on G:

$$\forall (k,v) \in G_l : \exists (k,v') \in G \wedge v \leq v'$$

Proof: As $(k,v) \in G_l \Leftrightarrow \exists i, v'((k,v') \in L_i^{\tau_i})$, and $L_i^{\tau_i} \subseteq L_i$, it is clear that $\forall (k,v) \in G_l : \exists (k,v') \in G$. Choose $(k,v) \in G_l$ and $(k,v') \in G$. Let $K' = \{i \in \{1,\ldots,m\} : \exists (k,v'') \in L_i^{\tau_i}\}$ and $K = \{i \in \{1,\ldots,m\} : \exists (k,v'') \in L_i\}$. From $L_i^{\tau_i} \subseteq L_i$ follows $K' \subseteq K$. With $v'' > 0$, we obtain $v \leq v'$. Moreover, $v = v' \Leftrightarrow K' = K$, i. e., in that case the bound is tight. ∎

Note that $L_i^{\tau_i}$ does not need to contain the *largest* elements of L_i for G_l to be a lower bound on G. The theorem is valid for any subsets $S_i \subseteq L_i$. For the following theorem to hold, however, $L_i^{\tau_i}$ must consist of the largest elements of L_i.

Theorem 2: G_u is an upper bound on G:

$$\forall (k,v) \in G_u : \exists (k,v') \in G \wedge v \geq v'$$

Proof: Analogously to the proof of Theorem 1, $\forall (k,v) \in G_u : \exists (k,v') \in G$. Choose $(k,v) \in G_u$ and $(k,v') \in G$. Let $K' = \{i \in \{1,\ldots,m\} : \exists (k,v'') \in L_i^{\tau_i}\}$ and $K = \{i \in \{1,\ldots,m\} : \exists (k,v'') \in L_i\}$. Then, $K' \subseteq K$. For local histograms $i \in K'$, the same, exact value is added to both v and v'. For local histograms $i \in \{1,\ldots,m\} \backslash K$, $p_i(k) = \text{false}$, and we add 0 to both v and v'. Finally, for local histograms $i \in K \setminus K'$, we add v_i to the value of the upper bound, v, but v_e, where $(k,v_e) \in L_i$, to the value of the exact global histograms, v. As $i \notin K'$, $(k,v_e) \notin L_i^{\tau_i}$. $L_i^{\tau_i}$ contains the t elements from L_i with the largest values, so $v_i \geq v_e$ follows, and in summary also $v \geq v'$. Moreover, $v = v' \Leftrightarrow K' = K$, i. e., in that case the bound is tight. ∎

B. Approximation Error

As an error estimation, we can derive an upper bound on the cardinality of the clusters that we might have missed in the approximated global histogram.

Theorem 3: Let L_i be local histograms, $1 \leq i \leq m$, G the corresponding exact global histogram, and τ a cluster threshold. Then the complete histogram approximation $\widetilde{G}$ has the following properties:

- *Completeness:* All clusters of the exact histogram G with cardinality at least τ are in the approximated histogram: $\forall k(\exists v((k,v) \in G \wedge v \geq \tau) \Rightarrow \exists v'((k,v') \in \widetilde{G}))$.
- *Error Bound:* The error for the cluster cardinalities in the approximated histogram is at most $\tau/2$: $\forall k((k,v) \in G \wedge (k,v') \in \widetilde{G} \Rightarrow |v - v'| < \tau/2$.

The error bound also holds for the restrictive histogram approximation $\widetilde{G_r}$, but completeness does not.

Proof: Choose $(k,v) \in \widetilde{G}$ and $(k,v') \in G$.
Completeness: $v \geq \tau = \tau_1 + \ldots + \tau_m$. There must be at least one local histogram L_i with $(k,v'') \in L_i$ and $v'' \geq \tau_i$. Then, $(k,v'') \in L_i^{\tau_i}$ and the cluster with key k is contained also in $\widetilde{G}$. As we may underestimate the cardinality of cluster k for local histograms i which do not contain k in their head, the cluster might not be contained in $\widetilde{G_r}$.

Error Bound: Let $K' = \{i \in \{1,\ldots,m\} : \exists (k,v'') \in L_i^{\tau_i}\}$ and $K = \{i \in \{1,\ldots,m\} : p_i(k) = \text{true}\}$. Then, $K' \subset K$. We only make estimation errors for local histograms $i \in K \setminus K'$. Recall from Definition 4 that we defined v_i to be the smallest value contained in $L_i^{\tau_i}$. By using the arithmetic mean as the estimated cardinality, the largest possible error we make on each of these histograms is $v_i/2$. According to the definition of $L_i^{\tau_i}$, we know $v_i \leq \tau_i$. As $(k,v) \in \widetilde{G}$, $K' \neq \emptyset$ and there is at least one local histogram for which we know the exact cluster cardinality. Hence, the global error is at most $\sum_{i \in K \setminus K'} \frac{v_i}{2} \leq \sum_{i \in K \setminus K'} \frac{\tau_i}{2} < \sum_{i=1}^{m} \frac{\tau_i}{2} = \frac{\tau}{2}$. ■

V. Extensions to TopCluster

In this section, we discuss three possible extensions to TopCluster. First, we show how the parameter τ can be determined automatically in a distributed manner. Next, we reconsider our assumption from Section II-C that exact monitoring is feasible on all mappers, and analyse the implications on the approximated global histogram in situations where this assumption no longer holds. Finally, we discuss cost functions that depend on other parameters in addition to cluster cardinality.

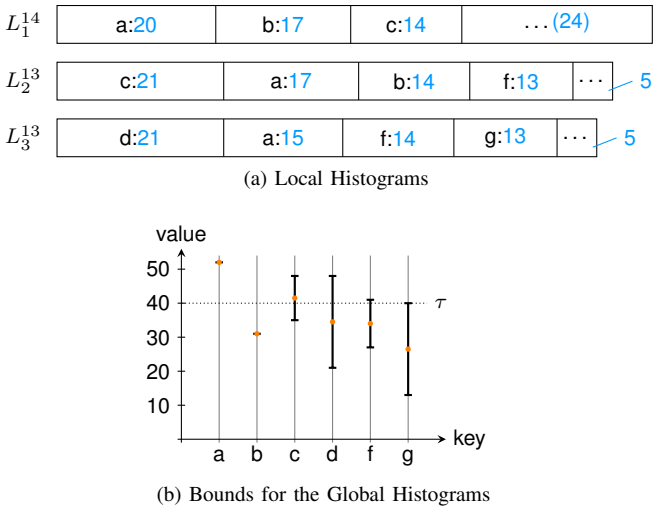
A. Adaptive Local Thresholds

So far, we assumed the parameter τ to be supplied by the user. Finding a suitable value for τ before starting a MapReduce job is challenging. Therefore, the system should be able to determine a suitable τ automatically. As explained in Section II, communication between all mappers is impossible. We devise a strategy in which every mapper determines the *relevant* items in its local histogram autonomously, and only sends those items to the controller. As we assume uniform distribution on the items not captured in the named part of the global histogram approximation, the clusters that depart most prominently from uniform distribution should be transmitted.

We base the decision on which items to transmit on the local data distribution, and only send the items with values exceeding the local mean value on mapper i, μ_i, by a factor of ε, where ε is a user-supplied error ratio. This allows us to keep the local error on every mapper within well-known bounds. The largest item that we possibly miss in the named global histogram is within $\tau = (1+\varepsilon)\sum_{i=1}^{m} \mu_i$.

With these settings, our approach works well with both uniform and skewed data distributions. If the data is skewed, only a small number of items with strong impact on the partition cost will exceed the local error threshold of $(1+\varepsilon)\mu_i$. We are therefore able to capture the partition cost reasonably well while keeping the communication volume for monitoring very low. If the data is distributed evenly, on the other hand, our assumption of uniform distribution of the items that are not communicated to the controller is accurate, and we obtain good cost estimates as well.

Example 8: Continuing our running example, from the monitored tuple and cluster counts, we calculate $\mu_1 = \frac{77}{7} = 11$, $\mu_2 = \frac{70}{7} = 10$, and $\mu_3 = \frac{68}{6} = 11\frac{1}{3}$, each on the corresponding mapper. We allow an error of $\varepsilon = 10\%$. The thresholds for the local item counts are thus 12.1, 11, and 12.47, respectively. The resulting local histograms are shown in Figure 5a).



(a) Local Histograms

(b) Bounds for the Global Histograms

Fig. 5: Histogram Aggregation for $\varepsilon = 10\%$

The restrictive global approximation based on this input is

$$\widetilde{G_r} = \{(a, 52), (c, 41.5)\}$$

and thus very close to the approximation we obtained in Example 4 using the user-supplied value $\tau = 42$ (resulting in $\tau_i = 14$).

B. Approximate Local Histograms

Current MapReduce systems choose the number of mappers for each MapReduce job based on a compromise. High parallelism and fast recovery in the case of node failures favour a high number of mappers processing a small amount of data each. On the other hand, the startup and management costs for each mapper plea for a low number of mappers and thus a higher data volume per mapper. Collecting monitoring data on every mapper introduces an additional argument for processing more data per mapper. We can interpret the local histogram that we create on every mapper as an approximation to the global histogram, based on the share of intermediate data generated by that mapper as a sample of the entire intermediate data. According to the law of large numbers, statistical measurements gain precision when the sample size is increased. Therefore, processing a larger share of input data on a single mapper is likely to provide us with better local histograms. We thus propose to increase the amount of data processed by every mapper. As this reduces the total number of mapper instances, we have the additional benefit of reducing the burden on the controller, as fewer local histograms must be aggregated.

In a worst case scenario, the number of clusters generated on a mapper could grow linearly with the amount of data processed. As we locally monitor every cluster, the monitoring data volume would grow at the same rate as well, occupying

resources which should rather be used for the actual processing. We therefore need to provision a way of limiting the amount of monitoring data that each mapper must maintain. In case the exact monitoring data would exceed this imposed limit, we can switch to approximate ranking algorithms, e. g. Space Saving [9], which allow to limit the amount of memory used. Space Saving was originally designed for finding the top-k items over data streams. At any point in time, it keeps the most frequent items in a cache of fixed size. A new item not yet contained in the cache replaces the least frequent item in the cache. Thereby, Space Saving guarantees that no item whose actual frequency is higher than the reported frequencies is missing in the obtained ranking.

Obviously, using approximate instead of exact local monitoring affects the cluster bounds computed on the controller.

Theorem 4: For local histograms approximated using the Space Saving algorithm [9], both the global lower and upper bound can be overestimated. Underestimation is only possible for the global lower bound.

Proof: Let $\widetilde{L}_i^{\tau_i}$ be the approximate local top-t histogram i of the exact local histogram L_i calculated using Space Saving. Let k be a key in $\widetilde{L}_i^{\tau_i}$ with estimated occurrence count $\widetilde{v}_k$ and actual occurrence count v_k, and let l be the least frequent item in $\widetilde{L}_i^{\tau_i}$ with estimated occurrence count $\widetilde{v}_l$ and actual occurrence count v_l. Then, $\widetilde{v}_k \geq v_k$ and $\widetilde{v}_l \geq v_l$ according to [9] (Lemma 3.4). *Overestimation:*

1) *Global Lower Bound:* We overestimate the global lower bound for item k if $\widetilde{v}_k > v_k$.
2) *Global Upper Bound:* Again, we overestimate the global upper bound of item k if $\widetilde{v}_k > v_k$. For an item k' not contained in $\widetilde{L}_i^{\tau_i}$, but appearing in L_i according to p_i, we overestimate the global upper bound if $\widetilde{v}_l > v_l$.

Underestimation: Critical situations for underestimation may arise for an item k' with actual occurrence count $v_{k'}$, which is not contained in $\widetilde{L}_i^{\tau_i}$, but would be contained in $L_i^{\tau_i}$.

1) *Global Lower Bound:* As k' is not contained in $\widetilde{L}_i^{\tau_i}$, we do not increase the lower bound, thus underestimating it.
2) *Global Upper Bound:* As k' is apparently contained in L_i, we will add $\widetilde{v}_l$ to the global upper bound. According to [9] (Theorem 3.5), $\widetilde{v}_l \geq v_{k'}$. Therefore, we do not underestimate the global upper bound.

■

From Theorem 4 follows that the upper bound calculated as described in Definition 4 remains valid if it is (completely, or partially) based on local histograms calculated using Space Saving. For the lower bound, this does not hold: due to the possible overestimation, it might no longer be in place. In consequence, we could overestimate the corresponding cluster's size. In order to keep the lower bound in place, we therefore decide to not increase it at all for mappers using Space Saving. A flag indicating the usage of Space Saving can be included in the communication between every mapper and the controller at the cost of one bit per mapper.

Approximating the local histograms can also interfere with the choice on how many items to transmit to the controller. Recall from Section V-A that, with adaptive thresholds, we base the decision on which items to transmit on the average local cluster cardinality. In order to calculate this average cluster cardinality, we need to keep track of the sum of all cluster cardinalities, and the cluster count. When monitoring all clusters exactly, we obtain both these values as a side product. With approximate monitoring, this is no longer the case. The total cardinality is still trivial to monitor. For the cluster count, we re-use the bit vectors created for approximating p_i as described in Section III-D and apply Linear Counting [8] in order to obtain an estimation. Based on the approximate average cluster cardinality derived from this information, we can then pick the relevant items from our monitoring data to send to the controller. In an extreme case, we might not have monitored all clusters which should be transmitted, i. e., even the cardinality of the smallest monitored clusters is larger than the threshold. If that situation arises, we inform the user on the actual error margin that we are able to guarantee for the given memory limit. If better estimates are required, the memory must be increased.

Switching from exact local histograms to Space Saving is possible at runtime, if the monitoring data volume exceeds a predefined threshold. The bit vector p_i is not affected by switching to Space Saving. For the total cluster cardinality on that mapper, we can initialise the counter with the sum of all cluster cardinalities counted so far. Then, we can discard the monitoring data on the clusters with the lowest cardinalities observed so far, in order to reduce the consumed memory. The remaining cluster information is the initial state with which we can continue the monitoring process using Space Saving.

C. Going Beyond Tuple Count

Throughout this paper, we considered the cluster cardinality being the only parameter for the cost estimation. In some applications additional parameters might be desirable. For example, if serialized objects (which are a collection of items each) are passed as tuples, the data volume per cluster could be an appropriate additional parameter of the cost function. The TopCluster technique is not specific to monitoring cardinalities. The same technique is also applicable to other parameters like data volume. Correlations between the parameters can be important for an accurate cost estimation, i.e., we need to know both cardinality and data volume of a specific cluster. TopCluster reconstructs these correlations on the controller using the cluster keys.

VI. Experimental Evaluation

We experimentally evaluate TopCluster, our distributed monitoring technique for MapReduce systems, on both synthetic data with different distributions, and real-world e-science data.

All experiments are run on a simulator. The simulator generates or loads the input data and distributes it into partitions the same way standard MapReduce systems do. The simulator allows us to generate input data with controlled skew in the

key values. Further, the simulator emulates the runtime of the reducers, which provides us with the ground truth for our cost estimation.

We use the following parameters in our evaluation. The synthetic data sets follow Zipf distributions with varying z parameters. Many real world data sets, for example, word distributions in natural languages, follow a Zipf distribution. The skew is controlled with the parameter z; higher z values mean heavier skew. For the synthetic data sets we run 400 mappers that generate 1.3 million output tuples each. The total of 2,000 clusters is distributed to 40 partitions with a hash function; we found the number of 40 partitions being a typical setting for the MapReduce environments used in scientific processing. The real e-science data in our experiments is the merger tree data set from the Millennium simulation [10]. We distribute the data to the mappers the same way Hadoop does, resulting in 389 mappers that each process 1.3 million tuples. We partition the data by the mass attribute, obtain 32 000 clusters, and create 40 partitions. We repeat each experiment 10 times and report averages.

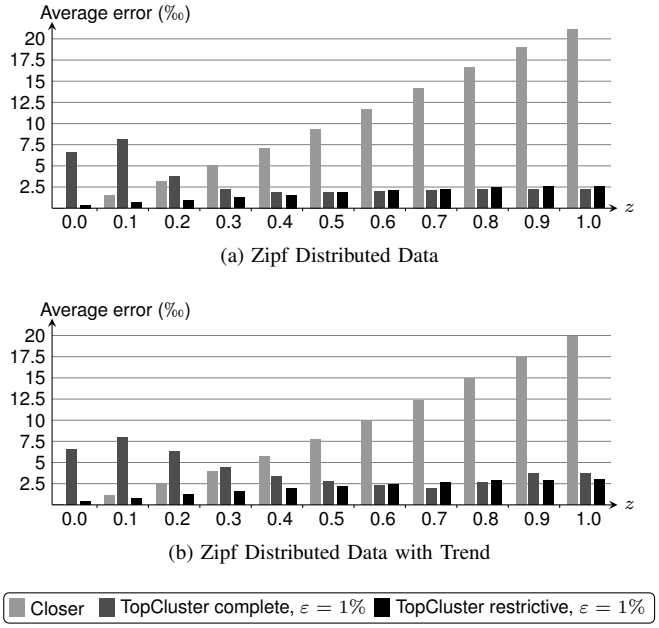
A. Histogram Approximation Error

We analyse the quality of the approximation obtained with both restrictive and complete TopCluster and compare it to the state-of-the-art approximation [2], which is referred to as *Closer*. *Closer* counts the number of tuples per partition; the size of the individual clusters, which is required for the cost estimation, is assumed to be the same for all clusters in a partition. We compute the approximation error as the percentage of tuples assigned to the wrong cluster (see Section II-D).

The results for Zipf distributions with varying z parameter are shown in Figure 6a ($\varepsilon = 1\%$). TopCluster-restrictive outperforms the other approximations in almost all settings, and the approximation error is very small (below 3‰). For heavily skewed data, TopCluster-complete achieves similar or even slightly better results than TopCluster-restrictive. The state-of-the-art solution *Closer* performs marginally better than TopCluster-restrictive if the data is perfectly balanced ($z = 0$). With increasing skew, this behaviour changes immediately, and TopCluster-restrictive widely outperforms *Closer*.

The difference between the restrictive and the complete variant of TopCluster is explained as follows. If the data is heavily skewed, the skew is visible on all mappers and a similar set of clusters is in the heads of all local histograms. Thus we get exact values for many clusters in the named part of the global histogram, leading to a small approximation error. There is little or no benefit in omitting clusters of size smaller than τ (restrictive variant). For moderate skew, on the other hand, the restrictive variant is beneficial since the clusters with higher approximation error are omitted. For $z = 0.1$, for instance, the approximation error is reduced by more than an order of magnitude with respect to the complete variant.

We repeat the experiments using a data distribution which simulates a trend over time (Figure 6b). Such trends may appear in scientific data sets, e. g., due to shifting research interests. In order to simulate a trend, we fix two Zipf



(a) Zipf Distributed Data

(b) Zipf Distributed Data with Trend

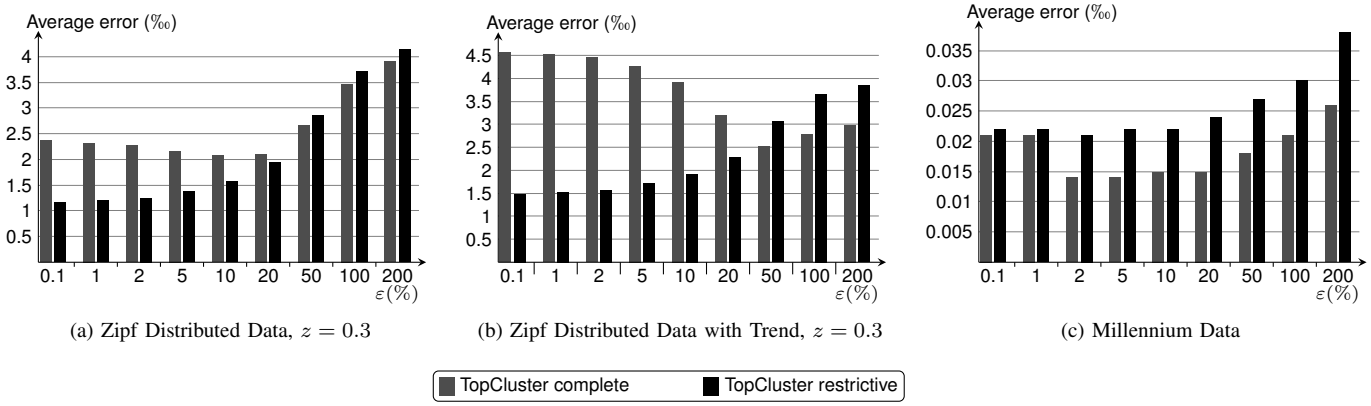
Fig. 6: Approximation Error for Varying Skew

distributions. For every value drawn by a mapper i, the mapper follows the first distribution with a probability of $\frac{i}{m}$, and the second distribution with a probability of $\frac{m-i}{m}$, where m is the total number of mappers. Also in this setting, restrictive TopCluster shows the best overall result and outperforms the complete version of TopCluster for almost all skew values. While both TopCluster variants perform well for high skew values, the performance of *Closer* decreases substantially with increasing skew.

B. Approximation Quality vs. Head Size

The parameter ε controls the length of the local histogram heads, i.e., the number of cluster cardinalities that the mappers send to the controller. We evaluate the cluster quality and the size of the histogram heads depending on ε.

Approximation Quality. The results for a Zipf distribution and a distribution with trend, both with $z = 0.3$, i. e., rather moderate skew, are shown in Figure 7a and Figure 7b, respectively. The results for the heavily skewed Millennium data set are depicted in Figure 7c. For the complete approximation, we note an interesting effect: the error decreases for small ε values before growing again for larger values of ε. This is explained as follows. The error is minimal for ε values that allow the skewed clusters to be in the head, but the clusters with uniform distribution, which introduce the approximation error, are ignored. Very low values of ε allow too many non-skewed clusters, very high values of ε miss part of the skew. The restrictive TopCluster approximation automatically removes poorly approximated clusters and is robust to this effect; the approximation error grows with increasing ε (the shorter the histogram head, the higher the error). The overall error of both TopCluster variants is very small in all settings: it is below 5‰ for all scenarios with synthetic data, and even



(a) Zipf Distributed Data, $z = 0.3$ (b) Zipf Distributed Data with Trend, $z = 0.3$ (c) Millennium Data

Fig. 7: Approximation Error for Varying ε

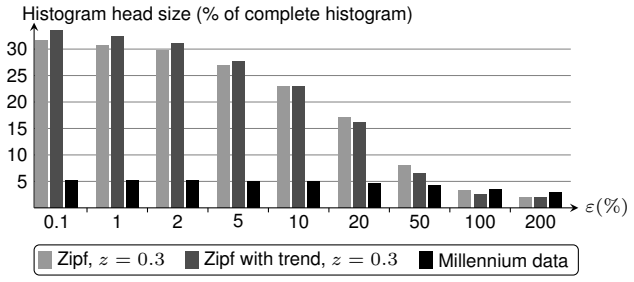


Fig. 8: Histogram Head Size for Varying ε

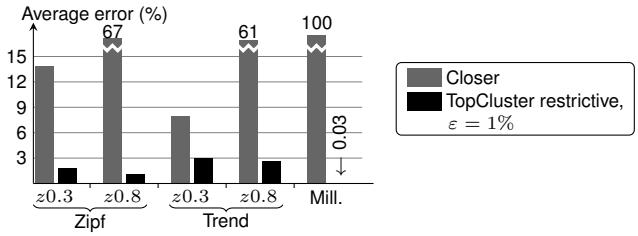


Fig. 9: Cost Estimation Error

smaller for the heavily skewed Millennium data.

Histogram Head Size. We measure the size of the local histogram heads with respect to the full local histogram. Only the heads of the local histograms are sent from the mappers to the controller; short histogram heads increase the efficiency. The experimental results are shown in Figure 8. For the synthetic data set with moderate skew ($z = 0.3$) the head size is decreased to $1/3$ of the full local histogram even for a very low error margin of $\varepsilon = 0.1\%$. By increasing the error margin to $\varepsilon = 200\%$ the head size is further decreased by an order of magnitude to 2%. Note that the approximation error is below 1% also for the highest error margin in our setting (see Figure 7). For the heavily skewed Millennium data the histogram head is only about 5% the size of the full local histogram also for small ε values.

Overall, the results are very encouraging. Due the high approximation quality of TopCluster, the error margin can be increased to get head sizes that are 20 to 50 times smaller than the full local histogram. Thus TopCluster scales to large data sets with good approximation quality.

C. Cost Estimation Error

We measure the quality of the cost estimation for reducers with quadratic runtime and compare our restrictive TopCluster approximation with *Closer*, the state-of-the-art approach. We use the histogram approximations to compute the expected partition cost and compare the result with the exact cost. Figure 9 shows the average error over all partitions. TopCluster clearly outperforms *Closer* in all settings. The advantage of TopCluster increases with the data skew since *Closer* assumes uniform distribution of the cluster sizes within each partition. For the heavily skewed Millennium data, TopCluster outperforms prior work by more than four orders of magnitude. Note how the approximation error of the histogram (see Figure 6) is amplified by the non-linear reducer task.

D. Influence on Execution Time

We evaluate the impact of load balancing to the job execution time. Following the setting in [2], we assigned the partitions to 10 reducers and compute the execution time per reducer for an algorithm with quadratic complexity. Assuming that all reducers run in parallel, the slowest reducer determines the job execution time. In Figure 10 we compare the standard load balancing of MapReduce to both TopCluster and *Closer*. The percentage in the figure is the execution time reduction over standard MapReduce; higher bars mean shorter runtimes. Both load balancing algorithms clearly outperform the standard load balancing of MapReduce. TopCluster is as good as *Closer* in the settings in which *Closer* is almost optimal, and better in the other setting.

For data with moderate skew ($z = 0.3$), the job execution time primarily depends on the number of clusters that a reducer must process. If the data distribution is similar on all mappers, both load balancing techniques obtain good results. For data sets that exhibit a trend over time, TopCluster outperforms *Closer*.

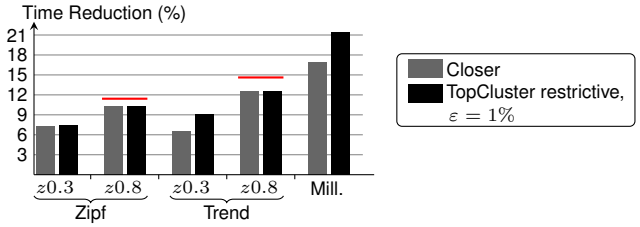


Fig. 10: Execution Time Reduction

For the data set with $z = 0.8$, the job execution is dominated by the time required to process the "heaviest" clusters. A good load balancing algorithm should assign less partitions to reducers with heavy clusters. We show the highest achievable reduction of execution time as red lines in Figure 10. This limit is given by the processing time required for the largest cluster in the data set. For $z = 0.8$, also the *Closer* approach is able to obtain near-optimal load balancing. The reason is that, for these configurations, it is sufficient to recognize partitions with expensive clusters, and a good cost approximations for the smaller clusters is less relevant.

For the heavily skewed Millennium data set distinguishing "more expensive" from "less expensive" partitions is not sufficient. Rather, the actual cost differences become important since partitions with very large clusters must be assigned to a dedicated reducer. Assuming uniform distribution within a partition that contains a very large cluster leads us to underestimating the partition cost. TopCluster, on the other hand, captures the largest clusters explicitly. This has not only a significant impact on the estimated partition cost, as shown in the preceding section. It also allows for a much better load balancing, as indicated by the execution time reduction for the Millennium data set in Figure 10.

VII. Related Work

Statistical information has been used as a basis for query optimisation in relational database systems for several decades [11]. Histograms [12] have been established as a compact yet reasonably accurate source of information for optimisers, and are still an active area of research [13]. They are used for selectivity estimations and join ordering decisions in both centralised and distributed database systems. The latter additionally need to decide upon operator placement [14], and consider load balancing issues [15]. Typically, histograms are built on attributes of the base relations. In order to reduce the impact of error propagation, however, approaches like extracting histograms from re-occurring intermediate [16] and integration of feedback from previously executed queries [17], [18] have been subject of research within the database community as well. Still, histograms are considered to be built and updated off-line, so the effort for histogram creation is not of primary interest in literature. In contrast, in MapReduce systems, histograms need to be created ad-hoc for two reasons. First, MapReduce systems lack a data loading phase, so the data sets to process are potentially unknown to the framework before the actual processing starts. Second, the processing routines are supplied by the user as a black box. Even if the data sets were known beforehand, estimating parameters of (intermediate) results automatically is not possible.

If the map-reduce join follows the scheme of relational data processing, experienced users can apply the same techniques for avoiding skew as used by database systems [4], [5], [6]. Hadoop, e. g., supports the use of Eager Aggreation [6] by providing a corresponding interface. For more complex application scenarios, however, these techniques are no longer applicable (e. g., Eager Aggregation is only possible for algebraic aggregation functions), or they do not provide substantial benefit (e. g., if the aggregation is a simple string concatenation, which only reduces the tuple count, but not the data volume to transmit). We focus on a load balancing approach for the generic processing scenario of MapReduce. Our approach provides well-balanced distributed execution both when database load balancing techniques are not applicable, and when non-expert users do not consider data skew within their MapReduce applications.

Existing distributed top-k solutions ([19], [20], [21], [22]) are not applicable in our scenario for two reasons. First, their goal is to reconstruct a global ranking, while we are not interested in the global *order* of the items. Instead, we must estimate the actual *value* for the items, i. e., the overall cardinality of the clusters, since the cost estimation is based on these values. Second, they require multiple, coordinated communication rounds. However, both scalability, and fault tolerance of MapReduce systems heavily rely on the possibility to run the mapper instances of a single MapReduce job independently of each other. TopCluster is designed to support MapReduce's mapper processing scheme by not requiring multiple communication rounds between all mappers.

The impact of skewed data distributions on MapReduce style systems has only very recently gained attention of the research community. In previous work [2], we defined a cost model for partition-based load balancing in MapReduce considering both skewed data distributions, and complex reducer side algorithms. The monitoring approach presented there assumes uniform cluster cardinality within each partition, which limits the effectiveness of the load balancing algorithm, especially for heavily skewed data distributions. In this paper, we proposed a more detailed distributed monitoring technique which can seamlessly be integrated in the cost framework.

An alternative approach to load balancing in MapReduce, LEEN, was proposed by Ibrahim et al. [3]. Besides skewed data distributions, they also consider data locality in the mapper outputs in order to reduce the communication overhead of MapReduce's shuffle phase. In contrast to our work, they monitor and process each cluster individually, which we consider infeasible for large scale data processing. Next, LEEN balances the data volume on the reducers, which does not directly correlate with workload balancing. Our approach balances the workload and thus gains low response times of MapReduce jobs. Finally, the heuristics they propose for assigning k clusters to r reducers has complexity $O(kr)$, i. e.,

it depends both on the data set (number of clusters), and the processing environment (number of reducers). The complexity of the algorithm proposed in [2], which we use in this paper, is independent of both these parameters.

Data skew has been—and still is—subject to intensive research in the database community, especially in the field of distributed database systems. Both distributed database systems and MapReduce employ hash partitioning techniques to distribute data to the processing nodes. Interesting situations in databases arise when processing distributed joins and grouping/aggregation. For join handling, the involved data sets are partitioned according to their join attribute values. The Gamma project [4] showed that processing all tuples sharing the same join attribute value on the same host is not required in order to obtain the correct result. Grouping and aggregation go hand in hand in relational database systems. Every group is reduced to a single, flat tuple. Aggregation functions are used to reduce a bag of values of a non-grouping attribute within each group to a single value. In such a scenario, early aggregation [6] can be employed to reduce the data volume per group/cluster which needs to be transferred. MapReduce systems, in contrast, guarantee that all items belonging to the same cluster are processed on the same node. Grouping and aggregation are separate processing steps. The grouping operation, i. e., the transition from map to reduce, ensures that all items sharing the same distribution key end up on the same reducer, where they can be accessed using an iterator interface and processed arbitrarily. Therefore, both splitting clusters and early aggregation are not possible here without further knowledge of the application.

A probabilistic candidate pruning approach with fixed error bounds for top-k aggregation is introduced in [23]. We can integrate this approach into our framework as an additional selection strategy (see Section III-C). By calculating the probabilistic bounds only at the end of the aggregation phase, we can also overcome the problem of high calculation costs, which lead to multiple variants of the algorithm.

VIII. Conclusion

In this paper, we motivated the need for a load balancing component in MapReduce, which considers both skewed data distributions and complex reducer side algorithms. We presented TopCluster, a distributed monitoring system for capturing data skew in MapReduce systems. Inspired by distributed top-k algorithms, TopCluster is tailored to suit the characteristics of MapReduce frameworks, especially in its communication behaviour. As confirmed by our experimental results, TopCluster provides a good basis for partition cost estimation, which in turn is required for effective load balancing.

In future work, we plan to extend our load balancing component in order to support the processing of multiple data sets within one MapReduce job, e. g., for improved join processing.

Acknowledgments

This work was funded by the German Federal Ministry of Education and Research (BMBF, contract 05A08VHA) in the context of the GAVO-III project and by the Autonomous Province of Bolzano - South Tyrol, Italy, Promotion of Educational Policies, University and Research Department.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *CACM*, vol. 51, no. 1, 2008.
[2] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Handling Data Skew in MapReduce," in *Closer*, 2011.
[3] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud," in *CloudCom*, 2010.
[4] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical Skew Handling in Parallel Joins," in *VLDB*, 1992.
[5] J. W. Stamos and H. C. Young, "A Symmetric Fragment and Replicate Algorithm for Distributed Joins," *IEEE TPDS*, vol. 4, no. 12, 1993.
[6] W. P. Yan and P.-Å. Larson, "Eager Aggregation and Lazy Aggregation," in *VLDB*, 1995.
[7] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *CACM*, vol. 13, no. 7, 1970.
[8] K.-Y. Whang, B. T. V. Zanden, and H. M. Taylor, "A Linear-Time Probabilistic Counting Algorithm for Database Applications," *TODS*, vol. 15, no. 2, 1990.
[9] A. Metwally, D. Agrawal, and A. E. Abbadi, "An Integrated Efficient Solution for Computing Frequent and Top-*k* Elements in Data Streams," *TODS*, vol. 31, no. 3, 2006.
[10] V. Springel, S. D. M. White, A. Jenkins, C. S. Frenk, N. Yoshida, L. Gao, J. Navarro, R. Thacker, D. Croton, J. Helly, J. A. Peacock, S. Cole, P. Thomas, H. Couchman, A. Evrard, J. Colberg, and F. Pearce, "Simulating the Joint Evolution of Quasars, Galaxies and their Large-Scale Distribution," *Nature*, vol. 435, 2005.
[11] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," in *SIGMOD*, 1979.
[12] Y. E. Ioannidis, "The History of Histograms (abridged)," in *VLDB*, 2003.
[13] C.-C. Kanne and G. Moerkotte, "Histograms Reloaded: The Merits of Bucket Diversity," in *SIGMOD*, 2010.
[14] V. Poosala and Y. E. Ioannidis, "Estimation of Query-Result Distribution and its Application in Parallel-Join Load Balancing," in *VLDB*, 1996.
[15] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Jr., "Query Processing in a System for Distributed Databases (SDD-1)," *TODS*, vol. 6, no. 4, 1981.
[16] N. Bruno and S. Chaudhuri, "Exploiting Statistics on Query Expressions for Optimization," in *SIGMOD*, 2002.
[17] M. Stillger, G. Lohman, V. Markl, and M. Kandil, "LEO – DB2's LEarning Optimizer," in *VLDB*, 2001.
[18] A. C. König and G. Weikum, "Combining Histograms and Parametric Curve Fitting for Feedback-Driven Query Result-Size Estimation," in *VLDB*, 1999.
[19] P. Cao and Z. Wang, "Efficient Top-K Query Calculation in Distributed Networks," in *PODC*, 2004.
[20] H. Yu, H.-G. Li, P. Wu, D. Agrawal, and A. E. Abbadi, "Efficient Processing of Distributed Top-*k* Queries," in *DEXA*, 2005.
[21] S. Michel, P. Triantafillou, and G. Weikum, "KLEE: A Framework for Distributed Top-k Query Algorithms," in *VLDB*, 2005.
[22] T. Legler, W. Lehner, J. Schaffner, and J. Krüger, "Robust Distributed Top-N Frequent Pattern Mining Using the SAP BW Accelerator," *PVLDB*, vol. 2, no. 2, 2009.
[23] M. Theobald, G. Weikum, and R. Schenkel, "Top-k Query Evaluation with Probabilistic Guarantees," in *VLDB*, 2004.