Managing Skew in Hadoop

YongChul Kwon[1], Kai Ren[2], Magdalena Balazinska[1], and Bill Howe[1]

[1] University of Washington, [2] Carnegie Mellon University

{yongchul,magda,billhowe}@cs.washington.edu,kair@cs.cmu.edu

Abstract

*Challenges in Big Data analytics stem not only from volume, but also variety: extreme diversity in both data types (*e.g.*, text, images, and graphs) and in operations beyond relational algebra (*e.g.*, machine learning, natural language processing, image processing, and graph analysis). As a result, any competitive Big Data system must support some form of parallel user-defined operations (UDOs) that can capture complex data processing tasks over complex data types without changing the core of the parallel data processing engine. Hadoop and other popular systems have been shown to provide a convenient programming model for implementing parallel UDOs, but the "black-box" nature of UDOs complicates the automatic load balancing required to achieve parallel scalability. In this paper, we present an overview of some of our recent work that tackles the problem of load imbalance (*a.k.a.* skew) in parallel UDO evaluation. We first discuss the prevalence of skew in today's applications and clusters. We then discuss our experience with static and dynamic methods for mitigating it.*

1 Introduction

Users today are increasingly demanding support for complex Big Data analytics that go beyond traditional relational algebra operations: they need to process unusual data types (*e.g.*, text, arrays, and graphs), apply sophisticated statistical models, and adapt specialized domain-specific techniques. These requirements translate into an increased demand for parallel user-defined operations (UDOs). The MapReduce abstraction [4] and its implementation in Hadoop [9] are well-known for their effective support of UDOs in the form of pairs of map and reduce operations. As an example, the Mahout library [27] provides a set of popular machine learning algorithms in Hadoop — tasks that are difficult to implement in conventional database systems.

Although Hadoop simplifies the expression of UDOs, adequate performance is by no means guaranteed (*e.g.*, [18, 24]). UDOs complicate algebraic reasoning and other simplifying assumptions relied on by the database community to optimize query execution. Instead Hadoop developers commonly rely on non-generalizable "tricks" to achieve high performance: ordering properties of intermediate results, custom partitioning functions, and assumptions about the number of partitions. For example, the Hadoop-based sort algorithm that won the terasort benchmark in 2008 required a custom partition function to prescribe a global order on the data, and relied on assumptions about key distribution that were part of the published benchmark [21].

One of the key challenges in the parallel execution of UDOs is, of course, load balancing across UDO partitions. While load balancing has extensively been studied for relational operators [5, 26], the problem has long

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering
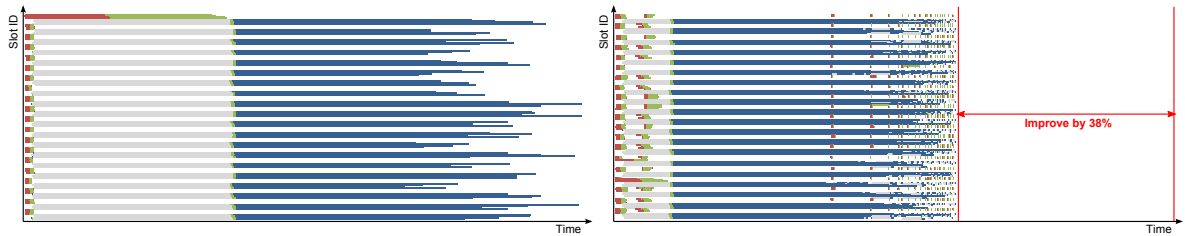
Figure 1: Execution details of the CloudBurst application (a) Original execution (b) Execution with load rebalancing is 38% faster. Red bars correspond to map tasks and blue ones to reduce tasks.

been under-studied for UDOs. In this paper, we summarize some of our recent work [14, 15, 16, 17, 24] that tackles the challenge of effective UDO parallelization in a Hadoop cluster in face of load imbalances: We first show a concrete example of load imbalance in an existing application and its impact on the execution time. We then summarize results from a measurement study in three Hadoop clusters showing, more generally, the prevalence of load imbalance problems in Hadoop clusters. Third, we discuss two techniques that we have developed for mitigating load imbalances in Hadoop applications either statically before execution or dynamically at runtime. We discuss the lessons learned from these two approaches and also point readers to other recent work that falls under each umbrella.

2   Prevalence of Skew

We first consider a concrete example of a user-defined operation that exhibits a significant load imbalance, also referred to as *skew*.

CloudBurst [25] is a MapReduce implementation of the RMAP algorithm for short-read gene alignment.[1] CloudBurst aligns a set of genome sequence reads with a reference sequence. This application is thus somewhat similar to a similarity join: it takes two datasets as input (the reads and the reference sequence) and identifies similar pairs drawing one item from each dataset. In the map phase, it extracts and partitions the reads and reference on their *n*-grams. Pairs of substrings that share an *n*-gram are tested for approximate alignment in the reducers. Frequent *n*-grams are handled similarly to frequent join keys in SkewedJoin [5]: frequent *n*-grams from a reference sequence are replicated, and frequent *n*-grams from a read are distributed in a round-robin fashion. Figure 1(a) shows the timing chart of an execution of the CloudBurst application.[2] The total runtime for the job is over 8 hours. The runtimes of the map tasks exhibit a bimodal distribution. We verified that the two modes correspond to the two input datasets. Although there is no significant skew within each mode, the MapReduce job is experiencing skew because the two modes coexist in a single job. The reduce phase also exhibits skew. Even though the partition function distributes keys evenly across reducers (not shown in the figure), some reducers are still assigned more data simply because the assigned key groups contain significantly more values. As a result, the runtime of the reducers is also highly uneven. Figure 1(b) shows how much these types of load imbalances affect the overall runtime of a job. In this application, re-balancing load to eliminate skew, cuts the runtime by nearly 40%. The figure shows the runtime when using the SkewTune system [17] that we discuss in Section 5.

The example illustrates how skew can arise in a UDO and how significantly it can affect the runtime. Our case study paper [15] shows additional such examples. But how prevalent is this problem?

---

[1] http://rulai.cshl.edu/rmap/

[2] We ran the CloudBurst job on a biology dataset [11]. For each alignment, we allowed up to 4 mismatches including insertion and deletion. We used 160 map tasks and 128 reduce tasks for the entire alignment. We used 64 reduce tasks to process low-complexity fragments. The reduce phase processed 128 sequences at a time (first loading data from the reference dataset in memory, then processing 128 sequences from the query dataset in a batch).

Cluster	Duration	Start Date	End Date	Successful/Failed/Killed Jobs	Users
OPENCLOUD	20 months	2010 May	2011 Dec	51975/4614/1762	78
M45	9 months	2009 Jan	2009 Oct	42825/462/138	30
WEB MINING	5 months	2011 Nov	2012 Apr	822/196/211	5

Table 2: Summary of analyzed workloads

Stragglers	Map	Reduce	Map $\wedge$ Reduce	Map $\vee$ Reduce
OPENCLOUD	2967 (58%)	1940 (38%)	1109 (22%)	3798 (74%)
M45	3643 (55%)	2659 (40%)	1310 (20%)	4992 (75%)
WEB MINING	140 (38%)	35 (9%)	18 (5%)	157 (43%)

Table 3: Number and fraction of jobs that have stragglers in map, in reduce, and both phases among jobs longer than five minutes. More than 40% of jobs running longer than five minutes have at least one straggler.

To answer this question, we analyzed execution logs from three Hadoop MapReduce clusters used for research: OPENCLOUD, M45, and WEB MINING. The three clusters have different hardware and software configurations and range in size from nine nodes (WEB MINING), to 64 nodes (OPENCLOUD), to 400 nodes (M45). The clusters are used primarily for scientific research in a wide variety of domains: computational astrophysics, computational biology, computational neurolinguistics, information retrieval, text and web mining, image and video analysis, security malware analysis, graphs and social networking analysis, cloud computing systems development, and others. Table 2 summarizes the duration of each collected log and the number of Hadoop jobs that it covers. The data in each log comes from the Hadoop job configuration files and job history files. More details about the collected data (and its analysis) can be found in our technical report [24].

To assess the prevalence of the skew problem, we count the number of jobs with stragglers (*i.e.*, long-running tasks) considering only jobs with runtimes of at least 5 min (the penalty from skew is insignificant in a short job) and at least two tasks (a straggler task is only defined relative to some other task).

Ananthanarayanan *et al.* [2] categorize a task as a straggler if the task takes at least 50% longer than the median task in the same phase of the same job. Following this definition, Table 3 shows the number of jobs that have stragglers in the map phase, the reduce phase, or both. Overall, more than 40% (and up to 75%) of jobs that run longer than five minutes have at least one straggler. We also find that even the map phase, which is usually regularized by assigning a fixed amount of bytes to each task, frequently experiences the straggler problem. This finding confirms that the input data size alone is not a good indicator of task runtime in these three clusters. The same finding was reported in the analysis of enterprise clusters [3].

Figure 2 shows the distribution of relative runtimes of straggler tasks with respect to the median task runtime of the same phase in the job. In all three clusters, 75% of reduce-side stragglers complete within 2.5x of the median runtime. Map stragglers have larger variations across clusters: 55% and 65% of map straggler tasks complete within 2.5x of median runtime in M45 and OPENCLOUD respectively. As observed in Ananthanarayanan *et al.* [2], the distribution is heavy-tailed. We also show the values at the 99%, 99.9%, and 100% percentiles in the table. Some straggler tasks run orders of magnitudes longer than the median runtime.

The straggler problem is thus prevalent in all clusters, and slow tasks frequently run more than 2.5x slower and sometimes orders of magnitude slower than the median.

3 Causes of Skew

It is well-known that skew has many causes [2, 4, 30]. The original MapReduce paper [4] considered the problem of skew caused by hardware malfunction or resource contention. To address this type of skew, MapReduce and Hadoop include a mechanism where the last few tasks in a job are speculatively replicated on a different machine
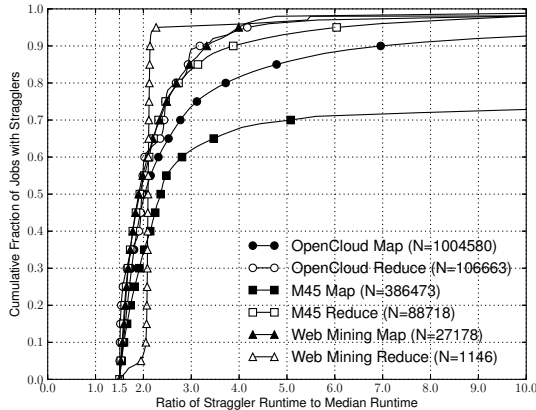
Percentile		99	99.9	100
OPENCLOUD	Map	70.0	1106.0	23068.5
	Reduce	15.3	64.9	54692.5
M45	Map	15.3	153.0	1537.5
	Reduce	11.4	143.8	1267.4
WEB MINING	Map	11.2	66.9	170.7
	Reduce	46.4	404.3	409.3

Figure 2: Cumulative fraction of the ratio of straggler runtime to median task runtime. N is the number of straggler tasks per cluster, per phase. The table shows straggler runtime ratios at specific percentiles.

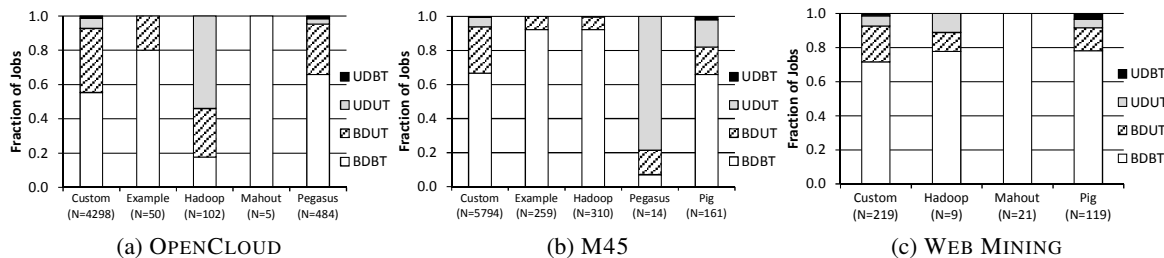(a) OPENCLOUD   (b) M45   (c) WEB MINING

Figure 3: Distribution of map task runtimes with respect to # of input records, grouped by application types. N is the number of successful jobs that run map functions in each category. The labels indicate the category: (U)BD(U)BT stands for (un)balanced input data, (un)balanced runtime.

and the job completes when the fastest replicas of these final tasks complete. More commonly, skew occurs when data is not evenly partitioned across tasks. This type of skew, called *data skew*, typically affects reducers since map tasks are normally assigned same-size chunks of input data. For reducers, the problem can occur either when a reducer processes a larger number of keys or a larger number of values than other reducers. In our work, we further find that a surprisingly large number of applications suffer from skew even when all UDO partitions are assigned the same amount of data [15]. The map phase of CloudBurst is one such example.

We analyze the prevalence of skew caused by uneven data allocation or uneven processing times in the three Hadoop cluster logs. For each phase of each job, we compute the ratio of the maximum task runtime in the phase to the average task runtime in that phase. We classify phases where this ratio is greater than 0.5 (meaning that at least one straggler took twice as long to process its data as the average) as *unbalanced in time (UT)*. Otherwise, the phase is said to be *balanced in time (BT)*. We compute the same ratio for the input data and classify phases as either balanced or unbalanced in their data (BD or UD).

Map Phase: Figure 3 shows the result for the map phases. We group the jobs by application type based on the mappers package names,[3] then break the results into four different types based on whether the input data and/or the runtime are balanced or not (*i.e.*, (U)BD(U)BT as defined in the previous paragraph).

As expected for the map phase, most jobs are balanced with respect to data allocated to tasks. However, a

---

[3]*Hadoop* corresponds to map functions that are part of the Hadoop MapReduce framework (*e.g.*, *IdentityMapper*). *Example* represents map functions defined in example applications in Hadoop (*e.g.*, *WordCount*). *Pegasus* is a graph mining system running on top of Hadoop [12]. If we do not recognize the package name, we assume that the users wrote the functions, and label them as *Custom*.
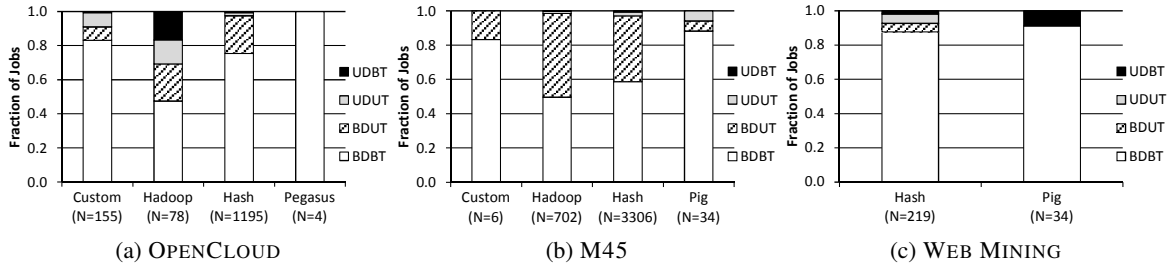
Figure 4: Distribution of reduce task runtimes with respect to # of reduce keys grouped by partition functions. The labels indicate the category: (U)BD(U)BT stands for (un)balanced input data, (un)balanced runtime.

significant fraction of jobs, more than 20% for all but Mahout in the OPENCLOUD cluster, remain unbalanced in runtime and are categorized as BDUT. These results agree with the result of the previous study in enterprise clusters [3]. Mahout is effective at reducing skew in the map phase, clearly demonstrating the advantage of specialized implementations. Interestingly, Pig produces unbalanced map phases similar to users' custom implementations. Overall, allocating data to compute nodes simply based on data size alone is insufficient to eliminate skew.

Reduce Phase: We perform a similar analysis for the reduce phase by using the number of reduce keys as the input-size measure. Instead of application types, we group the jobs by the partition function employed. The partition function is responsible for redistributing the reduce keys among the reduce tasks. We label the partition functions by inspecting their package names.[4] If the package name is not well-known (*e.g.*, `foo.bar`), we assume that it is customized by the user (labeled *Custom*). Figure 4 shows the results.[5] Overall, hash partitioning effectively redistributes reduce keys among reduce tasks for more than 92% of jobs in all clusters (*i.e.*, BDBT+BDUT). Interestingly, we observe that as many as 5% of all jobs in all three clusters experienced the *empty reduce* problem, where a job has reduce tasks that processed *no* data at all due to either a suboptimal partition function or because there were more reduce tasks than reduce keys. For the jobs with a balanced data distribution, the runtime is still unbalanced (*i.e.*, BDUT jobs) for 22% and 38% of jobs in the OPENCLOUD and M45 clusters, respectively. In both these clusters, custom data partitioning is more effective than the default scheme in terms of balancing both the keys and the computation. Other partitioning schemes that come with the Hadoop distribution do not outperform hash partitioning in terms of balancing data and runtime. A noticeable portion of UDBT jobs (in which data are not balanced but runtime is balanced) use the total order partitioner, which tries to balance the keys in terms of the number of values. Pig, which uses multiple partitioners, consistently performs well in both M45 and WEB MINING clusters.

Overall, UDOs in Hadoop can thus suffer from skew when the data is unevenly distributed to UDO partitions but also when the computation is uneven across the partitions.

4   Tackling Skew with Static Optimization

When the cause of skew is an uneven data distribution across UDO partitions or an uneven processing time across partitions, MapReduce's speculative execution mechanism is not helpful. This approach re-executes the same task on the same input data but on a different machine. It would lead to the same, slow execution time.

Instead, an alternate approach is to use finer-grained partitions so as to minimize the size of serial units of work. The problem with this approach is that finer-grained partitions increase overheads, especially in a system

---

[4]*Hadoop* represents all partition functions in the Hadoop distribution except the default hash partitioning (labeled *Hash*).

[5]In the M45 workload, the number of reduce keys was not recorded for the jobs that use the new MapReduce API due to a bug in the Hadoop API. The affected jobs are not included in the figure.

such as Hadoop, where the overhead of each new task is not negligible.

In SkewReduce [14], we developed instead a static optimizer that takes a UDO as input and outputs a data partition plan that strives to balance load across processing nodes. SkewReduce does not simply balance the amount of data assigned to each operator partition, but rather the expected processing time for the assigned data.

While a static optimizer could be applied to different types of UDOs, SkewReduce was designed for *feature extracting applications*. In these applications, data items (events, particles, pixels) are embedded in a metric space, and the task is to identify and extract emergent features (populations, galaxies) from the low-level data. These applications are captured by two UDOs: one that extracts features from sub-spaces and another that merges features that cross partition boundaries.

The key idea behind SkewReduce is to ask the user for extra information about their UDOs (the feature extraction and merge functions). The user creates additional *cost functions* that characterize the UDOs runtimes. A cost function takes as input a sample S of the input data, the sampling rate α that this sample corresponds to, and B, the bounding hypercube where the sample was taken from. The function returns a real number, which should represent an estimate of the processing time for the data. The sampling rate α allows the cost function to properly scale up the estimate to the overall dataset. The ideal cost function returns values proportional to the actual cost of processing the data partition delimited by the hypercube. The quality of the SkewReduce plans depends on the quality of the cost functions. However, we found that even inaccurate cost functions could help reduce runtimes compared to no optimization at all. For the domain experts such as scientists and statisticians, we posit that writing a cost model is easier than debugging and optimizing distributed programs. The cost models are specific to the implementation and can be reused across many data sets for the same UDOs.

Given the cost models, a sample of the input data, and the cluster configuration (*e.g.*, the number of nodes and the scheduling algorithm), SkewReduce searches a good partition plan for the input data by (a) applying finer grained data partitioning if significant data skew is expected for some part of the input data, (b) keeping coarse grained partitions when data skew is not anticipated, and (c) balancing the partition granularity in a way that minimizes expected job completion time under the given cluster configuration.

More specifically, the algorithm proceeds as follows. Starting from a single partition that corresponds to the entire hypercube bounding the input data I, and thus also the data sample, S, the optimizer greedily splits the most expensive leaf partition in the current partition plan. The optimizer stops splitting partitions when two conditions are met: (a) All partitions can be processed and merged without running out of memory; (b) No further leaf-node split improves the runtime: i.e., further splitting a node increases the expected runtime compared to the current plan. To verify the second condition, the SkewReduce optimizer first checks that the original execution time of the partition is greater than the expected execution time of the two sub-partitions running in parallel, followed by a newly added merge function to reconcile features found in individual sub-partitions. If the condition is satisfied, SkewReduce further verifies that the resulting tasks can also be scheduled in the cluster in a manner that reduces the runtime. It proceeds with the split only when both conditions are met.

To show the effectiveness of SkewReduce, we built a prototype running on top of Hadoop. Figure 5 shows the relative runtimes when applying the Friends-of-Friends clustering algorithm, a popular feature-extraction application used in astronomy, to two datasets, one from the astronomy domain and the other from oceanography. As the figure shows, SkewReduce yields runtimes 6X to 8X lower than a default Hadoop configuration. It is at least 2X faster than the optimal uniform data partition, whose choice depends on the dataset being processed.

Lessons learned: The key lesson learned from our SkewReduce work is that static optimization of the input data partition can dramatically improve runtimes in the presence of significant skew, caused by uneven processing times. The optimization time was short compared to the actual query execution time (*i.e.*, minutes vs hours or seconds vs minutes). Interestingly, runtimes improved even when the user-provided cost functions were not perfectly accurate but were good enough to identify expensive regions in the data. The drawback of this approach is two-fold: (1) it puts an extra burden on the user who must provide cost functions and (2) it cannot handle any unexpected conditions at runtime. We refer readers to our full paper for details [14].
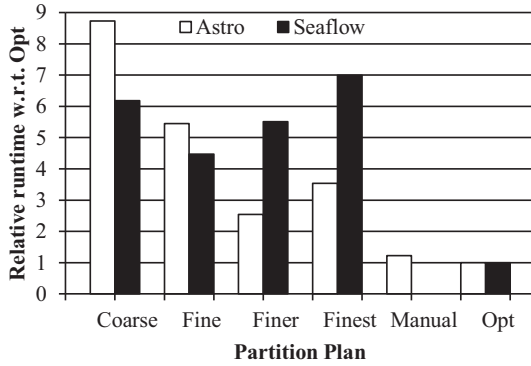
Completion time (hours for Astro, minutes for Seaflow)

Dataset	Coarse	Fine	Finer	Finest	Manual	Opt
Astro	14.1	8.8	4.1	5.7	2.0	1.6
Seaflow	87.2	63.1	77.7	98.7	-	14.1

Figure 5: Relative runtimes of different partitioning strategies compared with SkewReduces' optimized plan (Opt). The table shows the actual completion times for each strategy (units are hours for Astro and minutes for Seaflow). Manual plan is shown only for the Astro dataset.

5   Mitigating Skew Dynamically

To address the limitations of SkewReduce, we built SkewTune [17], a version of Hadoop extended with the ability to *dynamically* re-balance load across map or reduce tasks in a job. Because it re-balances load automatically, SkewTune must make an assumption about Hadoop programs: that separate invocations of the user-defined map and reduce functions are independent of each other. Load can thus be re-balanced at the granularity of records for map tasks and key-groups for reduce tasks.

SkewTune is designed for MapReduce-type systems, where each operator reads data from disk and writes data to disk and is thus decoupled from other operators in the data flow. SkewTune continuously monitors the execution of a UDO and detects the current bottleneck task that dominates and delays the job completion. Once such a task is identified, the task is stopped, and its unprocessed input data is repartitioned among idle nodes. The repartition only occurs when there is an idle resource (*i.e.*, the MapReduce scheduler runs out of other tasks to schedule) or when new nodes are dynamically added to accelerate the job (*e.g.*, using spot instances in Amazon EC2). Thus, if there are no tasks experiencing significant data skew, SkewTune imposes no overhead. If a node is idle and at least one task is expected to take a long time to complete, SkewTune stops the task with the longest expected time-remaining. It repartitions and thus parallelizes the remaining input data for that task taking into consideration to expected future availability of all nodes in the cluster. SkewTune continuously detects and removes bottleneck tasks until the job completes.

When repartitioning a task, SkewTune takes into account the predicted availability of all nodes in the cluster to minimize the job completion time. The availability is estimated from the progress of running tasks. The remaining unprocessed input data of a task is repartitioned in a way that fully utilizes nodes that are available immediately or are expected to become available in the near future. SkewTune also minimizes the side-effect of repartitioning so that the original output can be reconstructed simply by concatenating the output of split tasks.

We implemented the SkewTune strategy in the Hadoop MapReduce engine. Experimental results show that SkewTune can significantly improve completion time by up to factor of four without code changes. At the same time, it imposes, negligible overheads in the absence of skew.

Lessons learned: The most important lesson learned from the SkewTune approach is that dynamic load re-allocation is a workable solution for UDOs that follow a well-specified API, such as Hadoop, where the system knows how to rebalance load without breaking the application. The overhead of load re-allocations is small compared to the potential savings. At the same time, this approach is more flexible than static planning since it can handle skew independent of its cause: cluster conditions, hardware malfunction, mis-configuration, unlucky input data order, etc. We refer readers to our full paper for details on the SkewTune approach [17].

Both SkewTune and SkewReduce are freely available at `http://nuage.cs.washington.edu/`

6 Related Approaches

Skew-resilient operators: One approach to handling skew is through the implementation of skew-resilient operators. This approach has extensively been studied for the relational join operator (*e.g.*, [5, 30]). Pig, a declarative layer on top of Hadoop, implements an algorithm proposed in the parallel database literature [5] to handle data skew in a join algorithm [6]. Pig also includes mechanisms to mitigate skew in ORDER BY and GROUP BY operators [6]. Vernica *et al.* and Metwally *et al.* studied set-similarity joins in MapReduce [20, 29]. To handle skew, Vernica *et al.* first scans the input data to collect statistics on frequencies and load balances according to the statistics. Metwally *et al.* analyze set similarity measures, extracts a common structure in computing similarities, and design a series of MapReduce steps according to the structure. Metwally *et al.* also analyze skew that may occur in each step, and describe skew-handling strategies. There also exist work on handling skew in relational aggregate operators [26] including studies based on Hadoop [19].

    Studies of skew: Qiu *et al.* implemented three applications for bioinformatics using cloud technologies and reported their experience and measurement results [22]. Although they also found skew problems in two applications, they discussed a potential solution rather than tackling the problem. For data and computation imbalance problems, Ke *et al.* also found that these problems are prevalent in a variety of industrial applications [13]. Ananthanarayanan *et al.* studied the causes to outlier (*i.e.*, straggler) tasks and ways of mitigating outlier tasks caused by data imbalances or resource contention [2].

    Speculative execution enhancements: Several approaches have improved the basic MapReduce speculative execution. The LATE scheduler [31] speculatively runs the task with the longest estimated time remaining. It also schedules these tasks only on fast nodes and it limits resources used for speculation. The Mantri system [2] further only restarts a task when its runtime is inconsistent with its input data size and improves the network-aware task placement.

    Static optimizations: Related to SkewReduce, recent work also studied the possibility to leverage cost models to improve load balance [7, 8, 23]. That work was specific to balancing load in the reduce phase of a Hadoop job. Gufler *et al.* [7, 8] support non-linear cost models for reducers as functions of the number of bytes and the number of records a reducer needs to process. Their algorithm splits the reduce input data into a fixed number of small partitions, estimates their cost, and distributes the small partitions accordingly. A second algorithm further splits partitions that grow too large. The LEEN approach [10] further tries to balance load (*i.e.*, fairness) with data locality in the reduce phase of a job.

    Dynamic methods: Related to SkewTune, other techniques have recently explored dynamic load allocation in MapReduce [1, 28]. Vernica *et al.* propose an adaptive MapReduce system using situation-aware mappers that continuously monitor the execution of all mappers and adaptively resplit the map input data [28]. With an adaptive combiner and partitioner, the system further tries to balance the reduce input. However, the situation-aware mappers can not handle computational skew at the reducers, where some key-groups take longer to process than others.

7 Conclusion

The simple MapReduce API provides users great flexibility when implementing UDOs but it also makes users responsible for handling various performance problems. Skew is one challenge that users frequently face today. In this paper, we first reviewed the causes of skew in MapReduce applications and the prevalence of the problem in three real-world Hadoop clusters. We then presented an overview of two systems SkewReduce and SkewTune that mitigate skew statically and dynamically. With little or no user effort, the two systems can significantly improve job completion times compared to the default Hadoop setup. Interesting future work includes but is

not limited to further minimizing user efforts in skew mitigation through better programming interfaces and execution models and static and dynamic analysis or optimizations in multi-tenant environments.

References

[1] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *Proc. of the Third SoCC Conf.*, 2012.

[2] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using Mantri. In *Proc. of the 9th OSDI Symp.*, 2010.

[3] Y. Chen, S. Alspaugh, and R. H. Katz. Design insights for MapReduce from diverse production workloads. Technical Report UCB/EECS-2012-17, EECS Department, University of California, Berkeley, 2012.

[4] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of the 6th OSDI Symp.*, 2004.

[5] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proc. of the 18th VLDB Conf.*, pages 27–40, 1992.

[6] A. F. Gates, J. Dai, and T. Nair. Apache Pigś optimizer. *IEEE Data Engineering Bulletin*, 36(1), 2013.

[7] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Handling data skew in MapReduce. In *The First International Conference on Cloud Computing and Services Science*, 2011.

[8] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in MapReduce based on scalable cardinality estimates. In *Proc. of the 28th ICDE Conf.*, 2012.

[9] Hadoop. `http://hadoop.apache.org/`.

[10] 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 *IEEE CloudCom 2010*, pages 17–24, 2010.

[11] M. Kalyuzhnaya, D. Beck, and L. Chistoserdova. Functional metagenomics of methylotrophs. *Methods in Enzymology*, 495, 2011.

[12] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: a peta-scale graph mining system implementation and observations. In *ICDM*, 2009.

[13] Q. Ke, V. Prabhakaran, Y. Xie, Y. Yu, J. Wu, and J. Yang. Optimizing data partitioning for data-parallel computing. In *HotOS*, 2011.

[14] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proc. of the First SoCC Conf.*, June 2010.

[15] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A study of skew in MapReduce applications. In *The 5th International Open Cirrus Summit*, 2011.

[16] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune in action (demonstration). *Proc. of the VLDB Endowment*, 5(12):1934–1937, 2012.

[17] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating skew in MapReduce applications. In *Proc. of the SIGMOD Conf.*, pages 25–36, 2012.

[18] Y. Kwon, D. Nunley, J. P. Gardner, M. Balazinska, B. Howe, and S. Loebman. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *Proc. of the 22nd SSDBM Conf.*, 2010.

[19] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A Platform for Scalable One-Pass Analytics using MapReduce. In *Proc. of the SIGMOD Conf.*, June 2011.

[20] A. Metwally and C. Faloutsos. V-smart-join: a scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proc. of the VLDB Endowment*, 5(8):704–715, 2012.

[21] O. O'Malley. Apache hadoop wins terabyte sort benchmark. `http://developer.yahoo.com/blogs/hadoop/posts/2008/07/ apache_hadoop_wins_terabyte_sort_benchmark/`.

[22] X. Qiu, J. Ekanayake, S. Beason, T. Gunarathne, G. Fox, R. Barga, and D. Gannon. Cloud technologies for bioinformatics applications. In *Proc of the MTAGS'09 Workshop*, pages 1–10, 2009.

[23] S. R. Ramakrishnan, G. Swart, and A. Urmanov. Balancing reducer skew in MapReduce workloads using progressive sampling. In *Proc. of the Third SoCC Conf.*, 2012.

[24] K. Ren, Y. Kwon, M. Balazinska, and B. Howe. Hadoop's adolescence: a comparative workload analysis from three research clusters. Technical Report UW-CSE-12-06-01, University of Washington, June 2012.

[25] M. C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, June 2009.

[26] A. Shatdal and J. Naughton. Adaptive parallel aggregation algorithms. In *Proc. of SIGMOD Conf.*, 1995.

[27] T. M. Team. Apache Mahout project. `http://mahout.apache.org/`.

[28] R. Vernica, A. Balmin, K. S. Beyer, and V. Ercegovac. Adaptive MapReduce using situation-aware mappers. In *Proc. of the EDBT Conf.*, 2012.

[29] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *Proc. of the SIGMOD Conf.*, pages 495–506, 2010.

[30] C. Walton, A. Dale, and R. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proc. of the 17th VLDB Conf.*, 1991.

[31] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. of the 8th OSDI Symp.*, 2008.