Fine-Grained Micro-Tasks for MapReduce Skew-Handling

Josh Rosen, Bill Zhao
EECS, UC Berkeley
{*joshrosen, billz*}@*eecs.berkeley.edu*

Abstract

Recent work on MapReduce has considered the problems of *skew*, where a job's tasks exhibit large variance in size and processing cost, and *stragglers*, tasks that run slowly due to conditions on particular nodes.

In this paper, we discuss an extremely simple approach to mitigating skew and stragglers: break the workload into many small tasks that are dynamically scheduled at runtime. This approach is only effective in systems with high-throughput, low-latency task schedulers and efficient data materialization, so we propose techniques for scaling these components.

To demonstrate the efficacy of this technique, we compare micro-tasks to other skew handling techniques using the Spark cluster computing framework.

1 Introduction

MapReduce [3] is a popular programming model for distributed processing of large data sets. A large body of recent work has focused on improving MapReduce performance and extending its programming model.

A common MapReduce performance problem is "outlier" tasks that take significantly longer to complete than their peers. This significantly impacts job performance, since a MapReduce job completes only once its final task finshes, and wastes resources, since many nodes may remain idle while waiting for the slow tasks to complete. There are two main causes of outliers tasks: skew and stragglers.

With *skew*, a job's tasks have different processing requirements, due to processing uneven amounts of data or performing uneven amounts of work per record.

A similar problem is *straggler* tasks, which run slowly due to properties of individual machines. For example, a task might run slow due to interference from other processes or due to being scheduled on a slow or faulty node. This issues can be common in shared clusters composed of heterogeneous hardware.

Several techniques have been developed to address skew and stragglers. Many of these techniques were developed for Hadoop [8] MapReduce and are influenced by its characteristics, including its its high task scheduling overhead. Simple techniques, like running many small tasks, may provide much of the benefit of more-sophisticated skew handling techniques at a fraction of their complexity. This is supported by our experience in developing Shark [26], a SQL data warehousing system that runs on top of Spark [28], a MapReduce-like cluster computing framework.

In the remainder of this paper, we will explore the challenges of skew and stragglers, survey existing techniques to mitigate them, and explore how fine-grained micro-tasks can effectively mitigate skew.

2 Skew Handling

Skew causes a job's tasks to exhibit significant variance in their processing requirements.

We will consider skew during the reduce phase, since the reduce phase is prone to more types of skew than the map phase and many of the same skew-mitigation techniques that we will consider can be applied to map-phase skew. Reduce phase skew comes in three main forms: record size skew, partitioning skew, and computational skew [22].

2.1 Partitioning Skew

Partitioning skew is caused by uneven map output partition sizes / record counts, which may be caused by poor choices of partitioning functions. For example, constructing an invertex index by hash partitioning a set of words based upon their first letter can lead to skew if the first-letter frequencies are skewed [18]. The problem is that keys are unevenly distributed among reducers; we consider uneven value distributions in the next section.

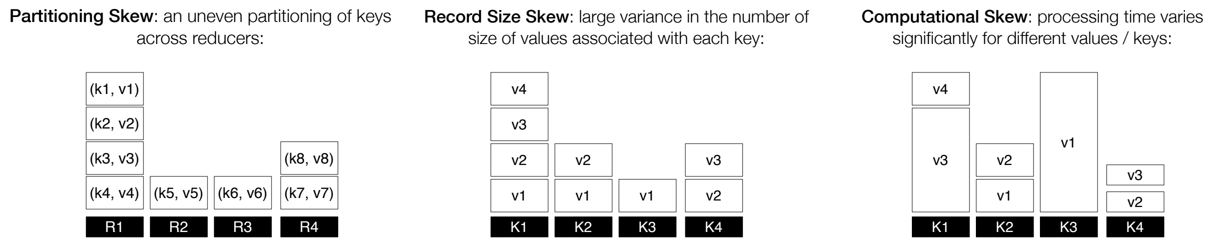
Partitioning Skew: an uneven partitioning of keys across reducers:
(k1, v1)
(k2, v2)
(k3, v3)
(k4, v4)
(k5, v5)
(k6, v6)
(k8, v8)
(k7, v7)
R1
R2
R3
R4
Record Size Skew: large variance in the number of size of values associated with each key:
v4
v3
v2
v1
v2
v1
v1
v3
v2
K1
K2
K3
K4
Computational Skew: processing time varies significantly for different values / keys:
v4
v3
v2
v1
v1
v3
v2
K1
K2
K3
K4

Figure 1: Three types of skew that can occur during the reduce phase.

A recent analysis of Hadoop traces from three production clusters shows that most jobs use Hadoop's hash partitioner, which evenly distributes keys for more than 92% of jobs [23]. In contrast, many jobs performed poorly due to unbalanced data sizes or stragglers. As another example, [17] examined skew when running the CloudBurst [24] gene alignment algorithm and found that the number of keys assigned to each reducer was approximately equal, whereas the number of records assigned to each reducer displayed more significant skew. The same study describes one case of partitioning skew due to using few reducers to process few keys [17]; in Section 5, we will show how this case can be addressed through fine-grained partitioning and intelligent task scheduling. Thus, skew purely due to bad key partitioning is unlikely to arise in practice, unlike the other forms of skew that we consider next.

2.2 Record Size Skew

Record size skew is caused by large numbers of records that must belong to the same partition (*e.g.,* records with the same reduce key). This can occur if a data set contains a handful of "popular" reduce keys and a "long-tail" of less-popular keys. For example, many real-world graphs obey a power-law degree distribution, where a handful of vertices have many neighbors compared to the average vertex (*e.g.,* in a web graph, there are domain such as Yahoo.com that have tremendous numbers of incoming links).

In general, record size skew must be handled in an application-specific manner [19].

If the reduce function is commutative and associative, partitions with popular keys can be distributed across multiple nodes, processed separately, and recombined to form the final reduce output. PowerGraph uses this approach to execute graph algorithms on real-world graphs with power-law degree distributions by distributing high-degree vertices across multiple nodes [5]. In MapReduce, a map-side combining phase can be performed before the reduce phase to locally reduce the values for a key at each node. This can drastically reduce the amount of bandwidth required by the reduce phase, while using greater parallelism to process popular keys. In practice, map-side combining likely avoids record size skew for most jobs.

For arbitrary reduce functions that are not commutative and associative, we have few options. Using domain knowledge, a programmer may be able to devise a domain-specific way of splitting the key's processing. For example, large inverted indices may be split for processing across multiple nodes [19].

A related problem is a skewed distribution of a single key's values across mappers. For example, several systems have considered key partitioning strategies that minimize network traffic during the reduce phase by assigning keys to reducers that hold the majority of those keys' values (or by scheduling reduce tasks based on data locality at the granularity of an entire reduce task) [14, 10, 9, 25]. These techniques only offer benefit for specific types of record size skew, and it is unclear how often such cases arise in practice. These existing systems have only described artificial examples of such skew, such as running a word count program with map-side combining disabled [14].

2.3 Computational Skew

Computational skew is caused by large imbalances in the amount of processing performed on individual records. For example, a complex reduce function might take orders of magnitude more time to process some inputs. This problem is superficially similar to the stragglers problem, but existing straggler mitigation techniques will not work if the partition is disproportionately slow no matter where it is run.

In general, techniques for mitigating computational skew aim to achieve an even balance of work across reducers. This requires the ability to estimate individual keys' processing costs. In some cases, processing costs will be correlated with data size, so output size can be used as a cost function. More generally, the processing costs might depend on an arbitrary cost function. Due to the halting problem, we cannot develop a general-purpose analytical technique to predict the processing costs for different inputs, short of actually performing the processing. As a result, some skew-mitigation systems require the user to supply a custom cost estimation function. Other systems employ black-box sampling techniques to perform cost estimation [22].

Given a suitable cost function, there are several different techniques for balancing work across reducers. Some systems overpartition the map output and use a bin-packing technique to statically assign map output partitions to reducers [6, 15]. Others dynamically split large partitions at runtime [18].

3 Stragglers

Stragglers are another common performance issue in MapReduce-like systems. We define a *straggler* as a task is disproportionately slow when run on a particular node, as opposed to a task that is slow because it must perform more computation than its peers. Stragglers may be caused by hardware failures, transient software issues, or clusters composed of a heterogeneous mix of hardware. Stragglers can significantly impact job performance because a job's completion time is determined by the completion time of its final task.

Several techniques have been developed to address stragglers. Recent work has attempted avoid stragglers by predicting which machines may run tasks slowly and using this information to choose better task assignments [2]. Stragglers may be caused by unpredictable, transient failures, so dynamic approaches must be used to mitigate stragglers at runtime. The original MapReduce paper recognized this problem and addressed it by launching additional "speculative" copies of the in-progress tasks as jobs neared completion [3]. Heuristics have been developed to identify stragglers and to decide when and where to launch speculative tasks [27, 1]. It is important to distinguish between computationally-skewed tasks and stragglers; Mantri accomplishes this by fitting tasks against a linear regression model [1].

Most published traces do not make distinctions between stragglers and tasks that are slow due to computational skew. A recent analysis of traces from three real clusters [23] found that between 40% and 75% of jobs running longer than 5 minutes had at least one straggler in either the map or reduce phase, where a straggler is defined as a task that runs 50% longer than the median task in its phase. In these traces, between 3% and 21% of speculative tasks successfully mitigated skew (*i.e.,* completed before the straggling task); this suggests that a large proportion of "stragglers" are actually computationally-skewed tasks, or that speculative execution is frequently performed too late in the job to offer benefit. Nevertheless, this shows that true stragglers occur in practice.

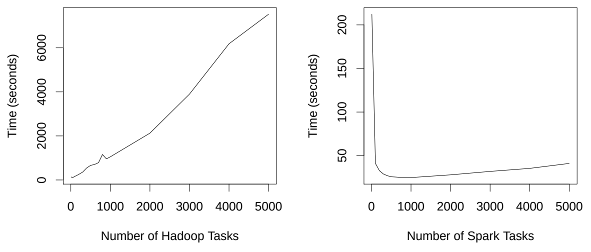

Figure 2: Query time vs number of reduce tasks. Spark's low task scheduling overhead makes it feasible to run large numbers of small tasks.

4 Skew Handing in Shark

Our exploration of micro-tasks is motivated by our experience in developing Shark [26], a Hive-compatible data warehouse built on top of the Spark cluster computing framework. Spark is a generalization of the MapReduce programming model that expresses programs as transformations of *resilient distributed datasets* (RDDs), partitioned datasets that can be operated on in parallel and automatically reconstructed based on lineage graphs.

4.1 Choosing the Number of Reducers

Many workloads benefit from additional parallelism during the reduce phase, so the number of reduce machines is typically chosen based on cost and performance tradeoffs that are determined by the rate of diminishing returns that come with using more machines. The number of reduce tasks, which is greater than or equal to the number of reduce machines, is usually determined by the hardware parallelism within each machine (*e.g.,* the number of cores) and limitations on the maximum size of a reduce task (reduce tasks that are too large may have working sets that do not fit in memory, leading to poor performance as data is spilled to disk).

Like Hadoop, we observed that job performance could be significantly affected by the number of reduce tasks used for a job. In early prototypes of Shark, the number of reduce tasks was specified by hand. Figure 2 shows performance vs. number of reduce tasks for a SQL query, using a recent version of Shark. Both curves are convex, but Hadoop's curve is extremely steep, so small differences in the number of reduce tasks could have a large performance impact. Recent improvements to Spark and Shark have flattened their curves, producing a large region in which running more tasks incurs relatively little overhead compared to running too few tasks.

4.2 Pre-Shuffle Partition Balancing

We sought to develop an optimizer to automatically select an optimal number of reduce tasks. This is challenging, because the prevalence of user-defined functions means that we often do not know the size of the data being shuffled until we begin to compute it.

To overcome this limitation, we implemented a technique that partitions the map output into many more hash buckets than reduce machines, examines the bucket sizes, and coalesces buckets to form the final set of reduce tasks. To mitigate partition size skew, the fine-grained buckets can be assigned to the final tasks using a greedy bin-packing heuristic. This is equivalent to the fine-partitioning technique proposed in [6]. This provides a mechanism to select the number of reduce tasks based on observed data statistics, but it is challenging to develop an appropriate policy to make optimal decisions.

Our implementation allows custom statistics to be gathered during the map phase at per-partition and global granularities. This same mechanism is used by other runtime query optimizations, such as selecting join algorithms based on the total data set size. To keep overheads low, statistics can be compressed. For example, we used a lossy logarithmic encoding to represent a partition's size in a single byte. For details on these additional optimizations, we refer the reader to [26].

4.3 Impact of Scheduling Overhead

We benchmarked an early prototype of Shark using this static bin-packing technique with a heuristic that attempted to maintain a minimum reduce task size (to avoid launching tasks that process too little data).

This demonstrated improvements over running the same number of reducers without skew-mitigation. After making several improvements to Shark's scheduler and materializing map output in memory instead disk (effectively flattening the curve in Figure 2), we discovered that this approach performed worse than running a large number of reduce tasks. In the next section, we explore the benefits and trade-offs of this surprisingly simple approach to mitigating skew.

5 Micro-Tasks

In this section, we introduce our approach for avoiding skew and stragglers during the reduce phase. The key technique is to run a large number of reduce tasks, splitting the map output into many more partitions than reduce machines in order to produce smaller tasks. These tasks are assigned to reduce machines in a "just-in-time" fashion as workers become idle, allowing the task scheduler to dynamically mitigate skew and stragglers.

5.1 Micro-Tasks and Stragglers

Running many small tasks lessens the impact of stragglers, since work that would have been scheduled on slow nodes when using coarser-grained tasks can now be performed by other idle workers. With large tasks, it can be more efficient to exclude slow nodes rather than assigning them any work. By assigning smaller units of work, jobs can derive benefit from slower nodes.

As an example, consider a cluster with 10 machines, one of which is half as fast as its peers. If we run 10 large reduce tasks, the slow machine could cause the job to run two times slower, making it more efficient to blacklist the slow machine instead of using it. If we run 100 reduce tasks on the same 10 machines and assume a sufficiently low degree of skew, then the slow node can process up to 5 tasks, while the other nodes process 95 tasks. Thus, the job can run up to 5% faster by using the slow node instead of blacklisting it.

5.2 Micro-Tasks and Skew

Micro-tasks can also help to mitigate skew. Increasing the number of hash partitions generally leads to smaller, more-even partitions because there is a lower probability of collision in the key's partitioning function. This does not provide the same partitioning quality guarantees as a system that manually partitions the most expensive keys, but it has a high probability of producing even partitions.

For inputs containing few distinct keys, fine-grained partitioning may result in many empty reduce tasks that receive no data. These empty reduce tasks are unproblematic, since they can be easily detected and ignored by the scheduler. Jobs with few distinct keys are the most sensitive to partitioning skew, since there may not be enough other work to mask the effects of a straggling task created by a key collision in the hash partitioning function. For jobs with large numbers of distinct keys, the impact of key collisions is small.

Micro-tasks do not specifically address record size or computational skew, but these types of skew must be handled in an application-specific manner. Micro-tasks help to achieve a more even assignment of indivisible units of work; techniques for reducing the units of work, such as map-side combining, are complimentary and orthogonal to this approach.

5.3 Just-in-Time Task Assignment

Prior work has considered storing map output in a small number of partitions, then statically assigning those partitions to reduce machines in order balance work across machines [6]. This approach mitigates skew but is susceptible to stragglers; because fine-grained partitions are statically assigned to reducers, these systems must rely on techniques like runtime task splitting [18] to handle stragglers and computational skew.

In contrast, scheduling many small reduce tasks allows work to be distributed to idle machines without having to interrupt executing tasks to steal work. [4] compared Dryad against Hadoop for a bioinformatics task and observed that Dryad displayed worse map performance as input data skew was increased. This performance gap was attributed to the use of many map tasks in the Hadoop implementation and the just-in-time work assignment performed by the scheduler (despite Hadoop's high task scheduling overhead). Run-time scheduling of many small tasks has successfully been used for straggler mitigation in distributed storage systems [21]. In the next section, we explore how to build efficient task schedulers to scale this technique.

6 Scalable Support for Micro-Tasks

Supporting thousands of small reduce tasks requires high-throughput, low-latency task scheduling and efficient map-ouput materialization. In this section, we propose techniques for scaling these components.

6.1 Scalable Task Scheduling

The scheduling problem is simplified by using hierarchical scheduling. A *job scheduler* obtains a set of machines to run map and reduce tasks. The job scheduler can be based on a cluster resource management framework, such as Mesos [13] or Hadoop YARN. On top of the job scheduler, a *task scheduler* is responsible for launching map and reduce tasks using the resources obtained by the job scheduler. This organization can help to reduce task launching overhead, since resources (such as JVMs) can be reused by multiple tasks.

Reduce-task scheduling is usually easier than map-task scheduling, because reduce tasks typically do not have locality preferences. In the absence of scheduling preferences/constraints, an efficient reduce task scheduler is simply a distributed queue that idle workers use to obtain tasks. The main scalability bottleneck is the number of control messages that must be processed by the master node.

This bottleneck can be removed by adding additional hierarchy to the task schedulers. A single master can delegate task scheduling for groups of machines to multiple second-level schedulers. For example, each rack in a datacenter might contain a task-scheduler responsible for launching tasks on that rack's machines. The underlying data flow still performs machine-to-machine communication across racks, but the scheduling control path becomes tree-structured. This limits the message fan-in at any particular scheduler and offers a way to proportionately scale the scheduling.

This design creates another problem: how do we divide work among the second-level schedulers? Work can be divided by initially assigning an even amount of work to each second-level scheduler and allowing idle second-level schedulers to communicate with the master scheduler to steal work from busy schedulers. This is more scalable than implementing work-stealing between individual workers.

6.2 Efficient Map Output Materialization

Typical MapReduce-like systems materialize map output before reduce phases, instead of pipelining output to reducers as it is produced. Hadoop materializes map outputs in the local filesystem, although some have explored materializing it in a distributed filesystem [20]. This approach allows techniques like map-side combiners to be used and affords flexibility in how reducers are scheduled.

Sometimes, data that is materialized to disk will remain in the operating system's buffer cache. However, performance can suffer if the cache is emptied prematurely. We have encountered multiple cases where the Linux buffer cache was unexpectedly emptied or where it performed poorly with large numbers of small files.

Because map output materialization does not offer useful fault-tolerance benefits, map output can be materialized in memory and, if necessary, spilled to disk. This approach may become increasingly feasible as the price of memory falls and cluster memories grow.

When dividing map output into many reduce partitions, individual map output files may be small. If each map output partition is naïvely materialized as its own file, these file writes may may be slow due to filesystem metadata overheads, and reads may lead to thrashing. This problem can be largely mitigated by storing map output in memory or by using SSDs, but it is sometimes necessary to store at least a portion of the output on conventional disks.

To overcome this issue, we propose to materialize multiple small output files into one on-disk file and extend the disk cache tracker to track files as `(filename, offset, length)` triples.

Increasing the number of map outputs creates the potential for reads to perform more random seeks, which could lead to heavy disk thrashing. To minimize the total number of seeks, partitions can be spilled in decreasing order of size. If we launch reduce tasks in an approximately fixed order, then the pattern of reads will be approximately linear when considered at a coarse granularity. We can exploit this by performing read-ahead.

In addition, this approach can be extended to support shuffle pre-fetching, where some map outputs (or portions of them) are sent to reducers during the map phase, by statically scheduling the first set of reducers [6].

6.3 Additional Optimizations

Several other implementation techniques can further decrease the scheduling overhead. Serialized user-defined functions and application code can be cached at worker machines, minimizing the amount of data that must be sent with each task. In applications with global, read-only state, such as the current model in a machine learning algorithm, techniques like *broadcast variables* [28] can be used to efficiently multicast these objects without sending them with each task. These optimizations are currently implemented in Spark.

7 Experiments

To measure the effectiveness of fine-grained micro-tasks, we conducted experiments using artificial datasets on Amazon EC2. All experiments were conducted on a 50-node cluster consisting of `m1.xlarge` nodes. Each node had 8 virtual cores, 15 GB of memory, and 1.6 TB of local storage.

Our dataset consists of five million key-value pairs per mapper, for a total of 250 million pairs. The values in our dataset are random 48-character strings, while the keys are integers in the range [1, 1000000]. We disable map-side combining, and thus treat the data as computationally skewed.

A skewed key distribution is created by sampling from Zipf distribution and adding a random offset:

```
key = zipf.nextInt * 10 + rand.nextInt % 10
```

The multiplier and random offset produce a skewed dataset where there are multiple keys with similar frequencies. The resulting skew is tractable if we make good assignments of keys to reducers. If we had sampled from a single Zipf distribution, the total runtime would be dominated by the cost of processing the most popular key, making the handling of the other keys irrelevant.

We use the Zipf parameterization from Hadoop's `RandomDistribution.Zipf` generator,

$$\frac{p(i)}{p(j)} = \left(\frac{j - \min + 1}{i - \min + 1}\right)^{\sigma},$$

where min is the smallest integer sampled, $\sigma > 1.0$ is a parameter controlling the degree of skew, and $p(x)$ is the probability of sampling integer x.

Table 1 lists the ranges for each parameter in our experiments.

σ (skew parameter)	1.1, 1.1, 1.3, 1.4
Number of reducers	50, 100, 200, 400, 800
Hash bucket to reducer ratio	1, 2, 4, 8, 10
Output materialization	Disk, Memory

Table 1: Ranges of parameters used in the benchmarks

A skewed dataset is generated and multiple test jobs are executed against it. The test shuffles the entire dataset and performs a count. Our test code launches a single Spark session and tests all combinations of these parameters, repeating each test 5 times. Between each run, shuffle output and other intermediate results are dropped from the cache and `System.gc()` is called. Our benchmark was implemented in a modified version of Spark 0.6.1, using its standalone cluster mode.

7.1 Results

Figure 3 compares micro-tasks and bin-packing on a dataset generated with $\sigma = 1.4$, using in-memory map output materialization and a 4:1 hash bucket to reducer ratio for bin-packing. The leftmost bar represents a naïve approach of running one reducer per machine. In this particular benchmark, the clear winner is running 100 reducers with bin-packing. This configuration may win because it mitigates most skew through bin-packing, while using slightly more tasks in order to handle stragglers.

Micro-tasks appear to perform slightly worse than bin-packing (or on par with it) for most configurations. This may be an artifact of our benchmark: each key's processing requirements are directly related to its actual data size and our cluster exhibited few stragglers, providing ideal conditions for bin-packing to perform its best. In the future, we plan to run additional experiments using artificial stragglers and more complex cost functions.

For comparison, Figure 4 shows the same experiment conducted by materializing map output on disk. In this case, it appears that the cost of materializing and fetching the map output is a dominant cost. Because bin-packing requires map output to be materialized in a large number of buckets, both bin-packing and micro-tasks pay similar IO costs. As a result, the curves for micro-tasks and bin-packing are very similar in this benchmark.

8 Discussion

Micro-tasks offer a simple approach for handling skew and stragglers. The changes required to efficiently support micro-tasks will offer performance improvements for all jobs.

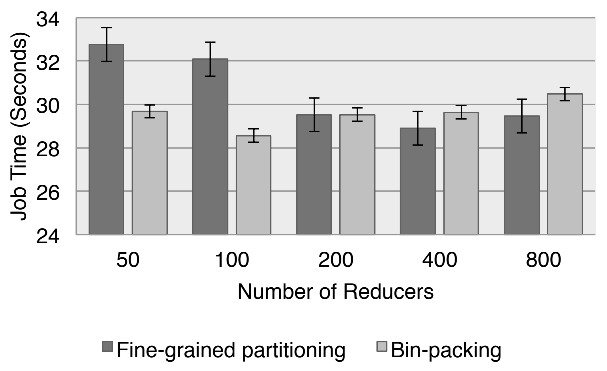

Figure 3: A comparison of micro-tasks and bin-packing, using $\sigma = 1.4$, in-memory map output materialization, and a 4:1 hash bucket to reducer ratio for bin-packing.

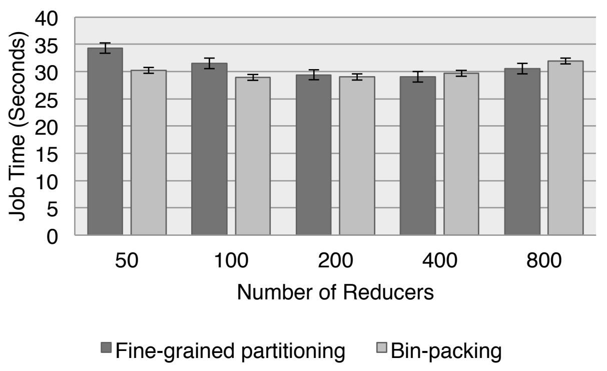

Figure 4: The same experiment in Figure 3, but with map output materialized on disk

There are some special cases where existing skew handling techniques should be preferred. Running many partitions to mitigate skew is not easy if the job requires that reduce output is range-partitioned (*e.g.,* to sort a data set). Unless the distribution of keys is known in advance, we cannot choose granular ranges that are likely to evenly partition the data. *SkewTune* [18] is well-suited for this case: it dynamically splits large partitions into smaller ones at runtime, while preserving ordering in order to support range partitioning. Similarly, techniques based on sampling can be used to estimate the key distribution in order to select an optimized range partitioning function.

For very small jobs, scheduling overhead accounts for a larger proportion of the total job time. In these cases, running fewer tasks may be preferable.

9 Related Work

Strategies for mitigating skew can be divided into two complimentary categories: techniques for eliminating the underlying skew (or reducing its magnitude) and techniques for load-balancing the skewed work. The former are often application-specific, although there are some techniques like map-side combining that apply to many real jobs.

Many load-balancing techniques rely on accurate cost-estimation. Some systems estimate processing costs by gathering data statistics (offline or at run-time). There are interesting trade-offs between the size of statistics / samples and their accuracy. *TopCluster* approximates a global histogram of the data distribution by computing approximate histograms at each mapper and aggregating them at a master controller [7]. *Themis* mitigates partitioning skew by sampling data sets to estimate their distributions in order to choose skew-avoiding partitioning functions [22]. Other systems, such as *SkewReduce*, require users to provide cost functions [16].

Micro-tasks can acheive good load-balancing without requiring cost functions. In multiprocessor scheduling, some systems have focused on producing good schedules when task durations are unknown and skewed [11, 12].

Many other systems have addressed skew in MapReduce. *SkewReduce* [16] addresses computational skew by using cost functions to statically choose better partitionings. *SkewTune* [18] mitigates partitioning skew at runtime by dynamically splitting large partitions. Runtime task splitting may be hard to do in general; there are many cases to consider, depending on the task's algorithm and whether the split occurs during or after data transfer. Supporting fault-tolerance for split tasks adds additional complexity. Micro-tasks may offer similar benefits while confining the extra complexity to the scheduler and storage layer.

Several systems have considered locality-aware reduce task scheduling, aiming to reach a trade-off between load-balancing and locality-awareness [14, 10, 9, 25]. These techniques are only beneficial for specific types of skew that seem unlikey to arise in practice and the published speed-ups are small. As datacenters move towards flatter network topologies, the benefits of these techniques may diminish. Thus, micro-tasks do not pay a large price for sacrificing locality.

10 Conclusion

We have proposed micro-tasks as a comprehensive solution for skew and straggler problems in common MapReduce applications. This approach offers skew-handling benefits on-par with more complicated techniques, while also addressing stragglers and requiring no cost estimation or advance knowledge of the data distribution.

While much attention has been given to how to choose good schedules, there is comparatively less work on improving the latency and throughput of task schedulers. Thus, we believe that we are far from limit of task-scheduling performance, offering hope that micro-tasks may be suitable for use in large clusters.

In the future, we would like to implement our proposed task scheduler architecture and perform additional experiments to measure performance using straggling or heterogeneous nodes. We also plan to investigate other benefits of micro-tasks, including the use of micro-tasks as an alternative to preemption when scheduling mixtures of batch and latency-sensitive jobs.

References

[1] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the outliers in map-reduce clusters using Mantri. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[2] Edward Bortnikov, Ari Frank, Eshcar Hillel, and Sriram Rao. Predicting execution bottlenecks in map-reduce clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing*, HotCloud'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[4] J. Ekanayake, T. Gunarathne, and J. Qiu. Cloud technologies for bioinformatics applications. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):998–1011, 2011.

[5] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. of the 10th USENIX conference on Operating systems design and implementation, OSDI*, volume 12.

[6] B. Guffler et al. Handling data skew in mapreduce. In *CLOSER*, 2011.

[7] Benjamin Gufler, Nikolaus Augsten, Angelika Reiser, and Alfons Kemper. Load balancing in mapreduce based on scalable cardinality estimates. *Data Engineering, International Conference on*, 0:522–533, 2012.

[8] Hadoop. http://hadoop.apache.org/.

[9] M. Hammoud, M.S. Rehman, and M.F. Sakr. Center-of-gravity reduce task scheduling to lower mapreduce network traffic. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 49–58. IEEE, 2012.

[10] M. Hammoud and M.F. Sakr. Locality-aware reduce task scheduling for mapreduce. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 570–576. IEEE, 2011.

[11] Mor Harchol-Balter. Task assignment with unknown duration. In *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, ICDCS '00, pages 214–, Washington, DC, USA, 2000. IEEE Computer Society.

[12] Mor Harchol-Balter, Mark E. Crovella, and Cristina D. Murta. On choosing a task assignment policy for a distributed server system. *J. Parallel Distrib. Comput.*, 59(2):204–228, November 1999.

[13] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A.D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 22–22. USENIX Association, 2011.

[14] 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 *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 17–24. IEEE, 2010.

[15] Y.C. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 75–86. ACM, 2010.

[16] YongChul Kwon et al. Skew-resistant parallel processing of feature-extracting scientific user-defind functions. In *SoCC*, 2010.

[17] YongChul Kwon et al. A study of skew in mapreduce application. In *The 5th Open Cirrus Summit*, 2011.

[18] YongChul Kwon et al. Skewtune: mitigating skew in mapreduce applications. In *SIGMOD '12*, 2012.

[19] J Lin. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce. *Proc. LSDS-IR*, pages 1613–0073, 2009.

[20] Diana Moise, Thi-Thu-Lan Trieu, Luc Bougé, and Gabriel Antoniu. Optimizing intermediate data management in mapreduce computations. In *Proceedings of the First International Workshop on Cloud Computing Platforms*, CloudCP '11, pages 5:1–5:7, New York, NY, USA, 2011. ACM.

[21] E. Nightingale, J. Elson, O. Hofmann, Y. Suzue, J. Fan, and J. Howell. Flat datacenter storage. In *Proceedings of the 10th USENIX conference on Operating systems design and implementation*, 2012.

[22] A. Rasmussen, M. Conley, G. Porter, R. Kapoor, A. Vahdat, et al. Themis: an i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 13. ACM, 2012.

[23] K. Ren, Y.C. Kwon, M. Balazinska, and B. Howe. Hadoops adolescence: A comparative workload analysis from three research clusters. 2012.

[24] M.C. Schatz. Cloudburst: highly sensitive read mapping with mapreduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[25] Yen-Liang Su, Po-Cheng Chen, Jyh-Biau Chang, and Ce-Kuen Shieh. Variable-sized map and locality-aware reduce on public-resource grids. *Future Gener. Comput. Syst.*, 27(6):843–849, June 2011.

[26] Reynold Shi Xin, Joshua Rosen, Matei Zaharia, Michael Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and rich analytics at scale. Technical Report UCB/EECS-2012-214, EECS Department, University of California, Berkeley, Nov 2012.

[27] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 29–42, 2008.

[28] Matei Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI, 2012.