A Study of Skew in MapReduce Applications

YongChul Kwon, Magdalena Balazinska, Bill Howe
University of Washington, USA
Email:{yongchul,magda,billhowe}@cs.washington.edu

Jerome Rolia
HP Labs
Email: jerry.rolia@hp.com

***Abstract*—This paper presents a study of skew — highly variable task runtimes — in MapReduce applications. We describe various causes and manifestations of skew as observed in real world Hadoop applications. Runtime task distributions from these applications demonstrate the presence and negative impact of skew on performance behavior. We discuss best practices recommended for avoiding such behavior and their limitations.**

I. Introduction

MapReduce [1] has proven itself as a powerful and cost-effective approach for massively parallel analytics [2]. A MapReduce job runs in two main phases: map phase and reduce phase. In each phase, a subset of the input data is processed by distributed tasks in a cluster of computers. When a map task completes, the reduce tasks are notified to pull newly available data. This transfer process is referred to as a shuffle. All map tasks must complete before the shuffle part of the reduce phase can complete, allowing the reduce phase to begin. We consider the case where computational load is imbalanced among map tasks or among reduce tasks. We refer to such an imbalanced situation as *map-skew* and *reduce-skew* respectively. Skew can lead to significantly longer job execution times and significantly lower cluster throughput. Figure 1 illustrates the problem. Each line in the figure represents one task. Time increases from left to right. This job exhibits map-skew: a few map tasks take 5 to 10 times as long to complete as the average, causing the job to take twice as long as an execution without outliers.

In this paper, we consider skew originating from the characteristics of the algorithm and dataset. For these sources of skew, speculative execution (a popular strategy in MapReduce-like systems [1], [3], [4] to mitigate skew stemming from a non-uniform performance of physical nodes) is ineffective because the speculative tasks would take a similar amount of time as the original tasks.

In Section II, we describe sources of skew and give one example of each source's effect on a representative application. In Section III, we describe best practices for avoiding skew, their limitations, and present directions for future work.

II. Types of Skew in a Map Reduce Application

We present five types of skew that can arise in a MapReduce application. For each type, we relate an application where we encountered this source of skew in practice.

A. Sources of Map-side Skew

We identify three causes of skew in the map phase.

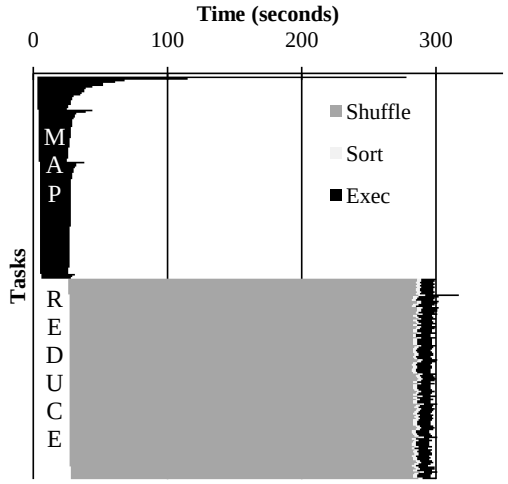
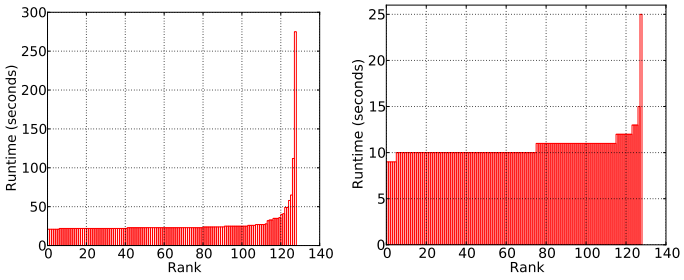


Fig. 1. A timing chart of a MapReduce job running the PageRank algorithm from Cloud 9 [5]. Exec represents the actual map and reduce operations. The slowest map task (first one from the top) takes more than twice as long to complete as the second slowest map task, which is still five times slower than the average. If all tasks took approximately the same amount of time, the job would have completed in less than half the time.

1) Expensive Record: Map tasks typically process a collection of records in the form of key-value pairs, one-by-one. Ideally, the processing time does not vary significantly from record to record. However, depending on the application, some records may require more CPU and memory to process than others. These expensive records may simply be larger than other records, or the map algorithm's runtime may depend on the record's value.

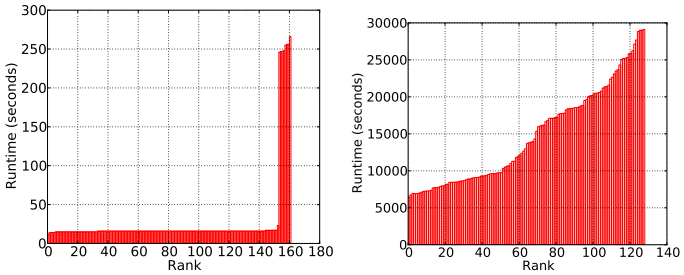
PageRank [6] is an application that can experience this type of skew. PageRank is a link analysis algorithm that assigns weights (ranks) to each vertex in a graph by iteratively aggregating the weights of its inbound neighbors. This application can thus exhibit skew if the graph includes nodes with a large degree of incoming edges. We took the PageRank implementation from Cloud 9 [5] and applied it to the freebase dataset [7]. We patched Cloud 9 so that it would properly handle graph nodes with large numbers of edges without running out of memory. The freebase graph is 2 GB in size, and contains 37M nodes and 342M edges. We stored the graph in a Hadoop sequence file, hash-partitioned on node id.[1]

[1]Cloud 9 provides multiple ways to partition data and multiple implementations of the PageRank algorithm. We chose the most straight forward schemes: hash partition and best practice implementation with combiner.



(a) Page Rank - Map (b) Page Rank - Reduce

Fig. 2. The distribution of task runtimes for PageRank with 128 map and 128 reduce tasks. A graph node with a large number of edges is much more expensive to process than many graph nodes with few edges. Skew arises in both the map and reduce phases, but the overall job is dominated by the map phase.



(a) CloudBurst - Map (b) CloudBurst - Reduce

Fig. 3. Distribution of task runtime for CloudBurst. Total 162 map tasks, and 128 reduce tasks. The map phase exhibits a bimodal distribution. Each mode corresponds to map tasks processing a different input dataset. The reduce is computationally expensive and has a smooth runtime distribution, but there is a factor of five difference in runtime between the fastest and the slowest reduce tasks.

Cloud 9 expresses each iteration of PageRank as a sequence of two MapReduce jobs. We observed skew in the first job. Figure 2(a) shows the distribution of map task runtimes in that job, during the first iteration of the algorithm (subsequent iterations show similar trends). The total runtime of this job is approximately 5 minutes. In Figure 2(a), the longest map task takes more than four minutes while most map tasks complete in 30 seconds. After investigation, we found that the slow map tasks were processing graph nodes with a large number of outgoing edges. These graph nodes were significantly slower to process, leading to the skew shown in the figure.

2) Heterogeneous Maps: MapReduce is a unary operator, but can be used to emulate an n-ary operation by logically concatenating multiple datasets as a single input. Each dataset may require different processing, leading to a multi-modal distribution of task runtimes.

For example, SkewedJoin is one of the join implementations in the Pig system [8]. Each map task in SkewedJoin distributes frequent join keys from one of the input datasets in a round-robin fashion to reduce tasks, but broadcasts joining records from the other dataset to all reduce tasks. These two algorithms exhibit different runtimes because the map tasks that perform the broadcasts do more I/O than the other map tasks.

CloudBurst [9] is a MapReduce implementation of the RMAP algorithm for short-read gene alignment[2]. CloudBurst aligns a set of genome sequence reads with a reference sequence. CloudBurst distributes the approximate-alignment computations across reduce tasks by partitioning the reads and references on their n-grams. The references and reads bearing frequent n-grams are handled similarly to frequent join keys in SkewedJoin: frequent n-grams from a reference sequence are replicated, and frequent n-grams from a read are distributed in round-robin.

Figure 3(a) shows the runtime distribution of map tasks in the CloudBurst application.[3] The total runtime for the job is over 8 hours. Unlike the PageRank application, the runtime distribution during the map phase exhibits a bimodal distribution and there is little variance within each mode. 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.

3) Non-Homomorphic Map: One of the key features of the MapReduce framework is that users can run arbitrary code as long as it conforms to the MapReduce interface: map() or reduce(), and typically initialization and cleanup. Such flexibility enables users to push, when necessary, the boundaries of what map and reduce phases have been designed to do: each map output can depend on a group of input records — *i.e.*, the map task is non-homomorphic. For example, although the conventional join algorithm in MapReduce requires both map and reduce phases, if the data are sorted on the join attribute, the join can be implemented directly in the map phase using a sort-merge algorithm. Similarly, a clustering algorithm can directly run during the map phase if the data are already partitioned by a prior MapReduce job [11]. In these scenarios, a map task may run what is normally reduce logic such as aggregation or join, consuming a group of records as a unit rather than a single record as in a typical MapReduce application. Thus, the map tasks may experience reduce-side skews discussed in Section II-B.

Users can also employ the MapReduce framework to implement a distributed analysis application by only leveraging the distributed execution and fault-tolerance features of the MapReduce engine (*e.g.*, [12]). In such scenarios, the map phase often runs arbitrary computation which is potentially non-homomorphic[4].

In the above scenarios, map tasks may run a CPU-intensive algorithm over many input records. If the runtime of the algorithm varies depending on the distribution of input data or the relationships between input data, then a job may incur significant map-skew.

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

[3]We ran the CloudBurst job on a biology dataset [10]. 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 alignments. We use 64 reduce tasks to process low-complexity fragments. The reduce phase processes 128 sequences at a time (first loading data from reference dataset in memory, then processing 128 sequences from the query dataset in a batch).

[4]The new interface since Hadoop 0.20 makes writing this kind of applications easier and cleaner.

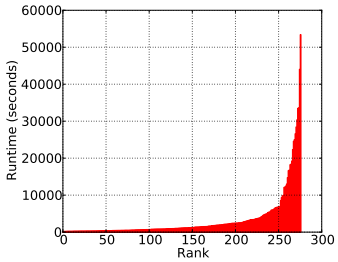
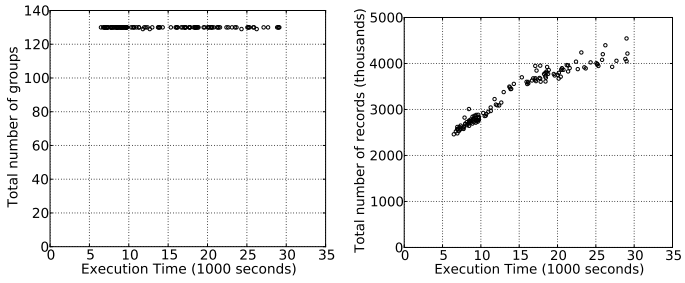


Fig. 4. Runtime distribution of the local clustering phase of the Friends-of-Friends algorithm [11], [12]. Total 276 map tasks. Even though all map tasks received the same amount of data, the slowest map takes more than 50000 seconds while the fastest one completes in 400 seconds due to different input data value distributions.



(a) Number of key groups (b) Number of input records

Fig. 5. Distribution of the number of key groups and input records per reduce task with respect to runtime for CloudBurst. The number of key groups assigned to each reducer is almost equal. However, the number of input records assigned to each reduce task varies by a factor of two. The runtime increases as input size increases, but the variance is larger than that of the input record distribution.

An example of such an application is a data clustering algorithm called Friends of Friends (FoF) [13] that we have implemented in multiple MapReduce-type systems, including Hadoop, in prior work [11], [12]. FoF is used by astronomers to analyze the structure of the universe within a snapshot of a simulation of the universe evolution. For each point in the dataset, the FoF algorithm uses a spatial index to recursively look up neighboring points to find connected clusters. The performance of a range query over a spatial index varies depending on the data distribution. In a dense region, every lookup returns a large number of neighbors, but in a sparse region the lookup returns few records. Thus, processing times depend on the distribution of input data to map tasks.

Figure 4 shows the runtime distribution of the FoF "local clustering phase" [11], [12], which runs in the map tasks. The data is space-partitioned by a prior MapReduce job. There are 276 map tasks. Each task is assigned a region of space such that all tasks have the same amount of data in bytes and in number of records (they differ by less than 2%). Even with this condition enforced, the runtime varies between 6 minutes and 13 hours.

B. Reduce-side skew

We identify two types of reduce-skew. The first one, *partitioning skew*, is unique to reduce. The other type of skew is analogous to the map-skew problems above.

1) Partitioning skew: In MapReduce, the outputs of map tasks are distributed among reduce tasks via hash-partitioning (by default) or some user-defined partitioning logic. The default hash-partitioning is usually adequate to evenly distribute the data. However, reduce-skew can still arise in practice. Consider the following two examples.

First, consider an application that needs to process many small files. In Hadoop, processing a small number of large files is more efficient than processing a large number of small files. As a result, users often write MapReduce jobs that combine small files into larger *sequence files*. One of our science collaborators wrote such a MapReduce job. The map derives the target sequence file name from the content of the small files, then the output is distributed among reduce tasks by hash-partitioning on the target sequence file name with a default hash function. The user wanted to assign one reduce task for each sequence file but unfortunately the hash partitioning scheme did not evenly distribute the key groups (*i.e.*, file names) across the available reduce tasks. As a result, some reduce tasks ended up writing multiple sequence files, each in the order of a TB, while others completed almost immediately.

As a second example, even when the partitioning function perfectly distributes keys across reducers, some reducers may still be assigned more data simply because the key groups they are assigned contain significantly more values. Figure 3(b) shows the runtime distribution with respect to the number of input key groups and the number of input records per reduce task for the CloudBurst application above. While the keys are distributed evenly across the reduce tasks (Figure 5(a)), there is a factor of two difference between the smallest and the largest key-group in the number of input records (Figure 5(b)). As a result, the runtime of the reducers exhibits skew.

In general, balanced data allocation is a difficult problem if the partitioning logic relies upon values computed during the execution of the map algorithm because the values are not known beforehand.

2) Expensive Input: In MapReduce, reduce tasks process a sequence of (key, set of values) pairs. As in the case of expensive records processed by map, expensive (key, set of values) pairs can skew the runtime of reduce tasks. Since reduce operates on key groups instead of individual records, the expensive input problem can be more pronounced, especially when the reduce is a holistic operation.

A *holistic reduce* requires memory proportional to the size of the input data [14]. For example, both SkewedJoin and CloudBurst require that data from one relation (or reference sequences) should be buffered in memory, thus, require memory proportional to the size of buffered data. In this scenario, the processing time for reduce depend not only on the number of records in the key group and the number of key groups but also the distribution (*e.g.*, for a join key, the proportion of data from the two datasets in SkewedJoin) or relationships between input data values (*e.g.*, alignments in CloudBurst).

Figure 5(b) illustrates the impact of a holistic reduce in CloudBurst. There is factor of two difference in the number

of input records between the fastest and the slowest reduce tasks, but there is a factor of five difference in runtime between tasks. The difference might have been greater depending on the data and the parameters of the algorithms.

III. Best Practices

We present a survey of best practices to mitigate skew in a MapReduce job. We present them in order of our estimate of their implementation complexity. We discuss their benefits, their limitations, and opportunities for future work.

Best Practice 1. *Use domain knowledge when choosing the map output partitioning scheme if the reduce operation is expensive: Range partition or some other form of explicit partition may be better than the default hash-partition.*

The default hash-partitioning scheme on key is a well-known technique in the parallel database literature for ensuring an even data distribution [15]. However, when the reduce operation is expensive and susceptible to skew, this simple technique often fails. Frequently, load must be balanced not at the granularity of keys but at the granularity of values as shown in Figure 5. In case of *Holistic Reduce* operations, the partitioning strategy must be application-dependent, which puts a significant burden on the developer.

To choose or implement better domain-specific partitioning strategies, the user must already be familiar with the properties of the application and the data. We found that the next best practice is useful in achieving this goal.

Best Practice 2. *Try different partitioning schemes on sample workloads or collect the data distribution at the reduce input if a MapReduce job is expected to run several times.*

Future Work. It remains an area of future work to develop techniques that will automatically partition the reduce input in the case of holistic and expensive reduce functions. One possible approach is to leverage debug runs on data samples to learn properties of the data and reduce function and partition subsequent runs accordingly. Another approach is to dynamically detect when skew arises and repartition data on the fly. A third approach is to study the source-code of reduce functions and extract properties of their behavior. We are currently studying these methods.

Best Practice 3. *Implement a combiner to reduce the amount of data going into the reduce-phase and, as such, significantly dampen the effects of any type of reduce-skew.*

In MapReduce (and in Hadoop), at the output of the map phase and before the reduce phase, one can optionally implement a *combiner* that pre-aggregates the map output. Combiners are in general beneficial when the expected reduction ratio for data to be shuffled is significant.

The combiner optimization, however, may hurt performance if the CPU and disk I/O cost of the combiner is greater than the diminished network I/O cost [16]. As shown by Lin and Schatz [17], manually combining the output within a map is desirable, if possible, because it avoids extra serialization overheads to prepare the input for the combiner. This is true for the current Hadoop implementation. Other MapReduce and future Hadoop implementations may not have this issue.

Combiners are effective at handling *Partitioning Skew* and *Expensive Input* at the reduce side when the skew observed during the reduce phase is mainly due to the volume of data transferred during the shuffle phase because a proper combiner can significantly reduce the transferred data size and mitigate the problems. To be more effective, we recommend using both a combiner and domain-specific partitioning strategy as described in the previous practice.

Future Work: An interesting area for future work is to automatically decide when to run a combiner, how much memory to give to the combiner, and more generally how to configure the combiner for optimal performance.

Best Practice 4. *Use a pre-processing MapReduce job that extracts properties of the input data in the case of a long-runing, skew-prone map phase. Appropriately partitioning the data before the real application runs can significantly reduce skew problems in the map phase.*

For MapReduce jobs that may experience *Expensive Input* on the map side and possibly a *Non-homomorphic Map*, skew can be eliminated by changing the allocation of input data to map tasks. If the behavior of the algorithm is known, then the user can run a separate MapReduce job that checks whether map-skew will occur. For example, a PageRank MapReduce preprocessing job can check whether there are graph nodes with large numbers of edges, and adjust the data partition accordingly, before executing the real PageRank job.

Future Work: In our previous work on the SkewReduce partition optimization framework [12], we developed an approach that semi-automates this process for feature extraction applications on top of multi-dimensional datasets. It remains, however, an area of future work to fully automate this process in the general case.

Best Practice 5. *Design algorithms whose runtime depends only on the amount of input data and not the data distribution.*

For MapReduce jobs with either *Holistic Reduce* or *Non-homomorphic Map* problems, the best approach to avoid skew is to re-design the map or reduce algorithms such that their runtime performance depends only on the size of the input data rather than the data value distribution. However, such redesign often requires extra expertise. For example, in the friends-of-friends clustering algorithm, we successfully eliminated skew by reimplementing the in-memory spatial index structure [11].

Future Work: An interesting area of future work is to design automated algorithms for avoiding or reacting to skew without forcing users to redesign their algorithms.

IV. Related Work

In the parallel database literature, there has been extensive research on handling data skew in the join operator [18]–[23] and aggregation algorithms [24]. While MapReduce shares many challenges and solutions, the fixed execution phases

(map, shuffle, reduce) and user-defined functions differentiate the practices for MapReduce applications from the skew-resistant relational algorithms in parallel databases.

MapReduce handles machine skew using speculative execution [1], [3], [4], which re-launches slow tasks on different machines and takes the result of the first replica to complete. Speculative execution is effective in heterogeneous computing environments or when machines fail. However, it is not effective against the types of skew described in this paper because rerunning the skewed data partition even on a faster machine can still yield a high response time. Lin analyzed the impact of *Partitioning Skew* in MapReduce jobs when the reduce keys follow a Zipfian distribution [25]. Ananthanarayanan *et al.* [26] proposed the Mantri system that can mitigate skew problems at runtime by improving scheduling decisions of MapReduce jobs. Mantri can handle certain types of skew well such as *Partitioning Skew* and *Expensive Input* when the source of the problem is directly related to the amount of input data, but does not offer a complete solution to data skew discussed in this paper. Ibrahim *et al.* [27] proposed a solution to *Partitioning Skew* on the reduce side with an explicit planning phase after the map phase in Hadoop. Finally, there are several skew-aware join implementations in MapReudce [8], [9], [28]. As we showed, however, the skewed join does not protect against all sources of skew. Each of these existing approaches thus addresses only some subset of possible skew problems.

In prior work, we developed SkewReduce, a system that derives a data partition plan and a task schedule given an input data sample and a user-defined cost models [12]. SkewReduce has the potential to address all types of skews described in this paper but it requires an appropriate cost model and a good input data sample. Additionally, it is applicable only to a specific class of applications.

V. Conclusion

MapReduce and its open source implementation Hadoop have made large scale data analysis widely accessible. Such runtime systems free users from problems associated with distributed coordination, fault-tolerance, and scalability. However, users may still suffer from performance problems related to skew if they are not careful regarding their map and reduce implementations and how data is partitioned across tasks. In this paper, we surveyed common sources of skew in MapReduce applications and demonstrated skew problems using real workloads. We presented best practices to address these problems and opportunities for future work.

Acknowledgments

This work is supported in part by NSF CAREER award IIS-0845397, NSF CluE Award IIS-0844572, an HP Labs Innovation Research Award, gifts from Microsoft Research, and Balazinska's Microsoft Research Faculty Fellowship. The dataset for the FoF algorithm was graciously supplied by T. Quinn and F. Governato of the University of Washington Astronomy Department. It was produced using allocations of advanced NSF-supported computing resources operated by the Pittsburgh Supercomputing Center, NCSA, and the TeraGrid.

References

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

[2] Apache Hadoop Project, "Powered By Hadoop," http://wiki.apache.org/hadoop/PoweredBy/, 2011.

[3] "Hadoop," http://hadoop.apache.org/.

[4] M. Isard *et al.*, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. of the EuroSys Conf.*, 2007.

[5] Jimmy Lin, "Cloud 9: A MapReduce library for Hadoop," http://www.umiacs.umd.edu/~jimmylin/Cloud9/docs/index.html, 2010.

[6] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proc. of the 7th WWW Conf.*, 1998, pp. 107–117.

[7] Google, "Freebase Data Dumps," http://download.freebase.com/datadumps/, 2010.

[8] A. F. Gates *et al.*, "Building a high-level dataflow system on top of map-reduce: the pig experience," *Proc. VLDB Endow.*, vol. 2, pp. 1414–1425, August 2009.

[9] M. C. Schatz, "Cloudburst," *Bioinformatics*, vol. 25, pp. 1363–1369, June 2009.

[10] M. Kalyuzhnaya *et al.*, "Functional metagenomics of methylotrophs," *Methods in Enzymology (to appear)*, 2011.

[11] Y. Kwon *et al.*, "Scalable clustering algorithm for N-body simulations in a shared-nothing cluster," in *Proc. of the 22nd Scientific and Statistical Database Management Conference (SSDBM)*, 2010.

[12] ——, "Skew-resistant parallel processing of feature-extracting scientific user-defined functions," in *Proc. of the 1st ACM Symp. on Cloud computing (SOCC)*, 2010.

[13] M. Davis *et al.*, "The evolution of large-scale structure in a universe dominated by cold dark matter," *Astroph. J.*, vol. 292, pp. 371–394, May 1985.

[14] J. Gray *et al.*, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Min. Knowl. Discov.*, vol. 1, pp. 29–53, January 1997.

[15] D. J. Dewitt *et al.*, "The Gamma database machine project," *IEEE TKDE*, vol. 2, no. 1, pp. 44–62, 1990.

[16] H. Herodotou *et al.*, "Starfish: A self-tuning system for big data analytics," in *Proc. of the Fifth CIDR Conf.*, 2011.

[17] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in mapreduce," in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, 2010, pp. 78–85.

[18] D. J. DeWitt *et al.*, "Practical Skew Handling in Parallel Joins," in *Proc. of the 18th VLDB Conf.*, 1992.

[19] K. A. Hua and C. Lee, "Handling data skew in multiprocessor database computers using partition tuning," in *Proc. of the 17th VLDB Conf.*, 1991.

[20] C. B. Walton *et al.*, "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins," in *Proc. of the 17th VLDB Conf.*, 1991.

[21] R. S. Wei Li, D Gao, "Skew handling techniques in sort-merge join," in *Proc. of the SIGMOD Conf.*, 2002.

[22] Y. Xu *et al.*, "Handling data skew in parallel joins in shared-nothing systems," in *Proc. of the SIGMOD Conf.*, 2008.

[23] Y. Xu and P. Kostamaa, "Efficient outer join data skew handling in parallel DBMS," in *VLDB*, 2009.

[24] A. Shatdal and J. Naughton, "Adaptive parallel aggregation algorithms," in *Proc. of the SIGMOD Conf.*, 1995.

[25] J. Lin, "The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce," in *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, 2009.

[26] G. Ananthanarayanan *et al.*, "Reining in the outliers in map-reduce clusters using mantri," in *Proc. of the 9th OSDI Symp.*, 2010.

[27] S. Ibrahim *et al.*, "Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud," in *Cloud Computing Technology and Science, IEEE International Conference on*, 2010, pp. 17–24.

[28] R. Vernica *et al.*, "Efficient parallel set-similarity joins using mapreduce," in *Proc. of the SIGMOD Conf.*, 2010, pp. 495–506.