Volume 4, Issue 1, January 2014 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

An Enhanced Framework for Performance Optimization of Apache Hadoop

Pallavi R. Gulve

Computer Department, BSIOTR,

Pune, India

Abstract— Hadoop is a well-known implementation of the MapReduce framework for running data-transformation jobs on clusters of commodity servers. In hadoop data transformations jobs are executing in parallel using multiple map and reduce tasks. The main objective of the proposed system is to propose a design improvement in shuffling mechanism used in reduce tasks, so that it will improve the performance of map-reduce jobs significantly in hadoop cluster. There is delay in job completion to the combining of the shuffle phase and reduce tasks, because of this the parallelism between multiple waves of map and reduce unused, fails to address data distribution skew among reduce tasks, and makes task scheduling inefficient. In this work, I propose to differentiate shuffle from reduce tasks and convert it into a platform service provided by Hadoop. The scope of this document is to focuses on the improve performance of shuffle service, which submits map output to respective nodes via a novel shuffle on write operation and schedule tasks considering workload balance. This paper describes benchmarks reviews performance considerations and with a Hadoop analytics cluster

Keywords— Hadoop, Big Data ,Map reduce, Sort, Performance

I. INTRODUCTION

Hadoop is well-known open-source implementation of the MapReduce programming model for processing big data in parallel [7]. Hadoop jobs mainly divided into two tasks, map task and reduce task. Both tasks distributed onto multiple nodes for parallel execution. Hadoop uses HDFS file system to store big data. Every map task runs on nodes where input data available and there is no communication between map tasks. There is lots of research going on to improve the performance of map tasks. Because data vicinity is acute to map performance, work has been done to preserve vicinity via map scheduling [2] or input replication[4]. Others designed interference [5] and topology [4] aware scheduling algorithms for map tasks. While there is broad work developing the parallelism and improving the efficiency in map tasks, only a few studies have been dedicated to advance reduce tasks. The end-to-end input data fetching phase in a reduce task, known as shuffle, involves thorough communications between nodes and can considerably delay job completion. Because the shuffle phase typically needs to copy midway output generated by almost all map tasks, techniques developed for improving map data vicinity are not appropriate to reduce tasks [1]. Hadoop attempts to hide the latency gained by the shuffle phase by starting reduce tasks as soon as map output files are available. There is existing work that tries to overlap shuffle with map by intelligently sending map output [6] or fetching map output in a globally sorted order [1].

Unfortunately, the combination of shuffle and reduce phases in a reduce task presents difficulties to attaining high performance in Hadoop clusters and makes existing methods [6] less effective in production systems. First, in production systems with limited number of reduce slots, a job often executes multiple reduce tasks. Because the shuffle phase starts when the equivalent reduce task is scheduled to run, only the first wave of reduce can be overlapped with map, leaving the possible parallelism unexploited. Second, tasks scheduling in Hadoop is unaware of the data distribution skew among reduce tasks [1], machines running shuffle-heavy reduce tasks become holdups. Finally, in a multi-user environment, one user's long running shuffle may conquer the reduce slots that would otherwise be used more proficiently by other users, sinking the utilization and throughput of the cluster. In this paper, we recommend to decouple the shuffle phase from reduce tasks and convert it into a platform service provided by Hadoop.

II. OBJECTIVE AND SCOPE

Hadoop is a popular implementation of the MapReduce framework for running data-intensive jobs on clusters of commodity servers. Although Hadoop automatically parallelizes job execution with concurrent map and reduce task. The objective of this paper is to propose a design improvement in shuffling mechanism of reduced task, which could significantly improve the performance of map-reduce jobs in hadoop cluster. I attribute the delay in job completion to the coupling of the shuffle phase and reduce tasks, which leaves the potential parallelism between multiple waves of map and reduce untapped, fails to address data distribution skew among reduce tasks, and makes task scheduling inefficient. In this work, we propose to decouple shuffle from reduce tasks and convert it into a platform service provided by Hadoop.

The scope of this document is to focuses on the performance improvement of shuffle service, which proactively push map output to respective nodes via a novel shuffle on write operation and flexible schedule tasks considering workload balance this seminar reviews performance considerations and describes relevant benchmarks with a Hadoop analytics cluster.

III. METHODOLOGY

MapReduce is a programming model designed for treating big volumes of data in parallel by isolating the work into a set of sovereign tasks. MapReduce programs are written in a particular style inclined by functional programming concepts, specifically idioms for processing lists of data. This section explains the nature of this programming model and how it can be used to write programs which run in the Hadoop environment.

Reduce tasks are created and assigned a task ID by Hadoop during the initialization of a job. The task ID is then used to recognize the associated partition in each map output file. For example, shuffle fetches the partition that matches the reduce ID from all map tasks. When there are reduce slots available, reduce tasks are scheduled in the ascending order of their task IDs. Although such a design simplifies task management, it may lead to long job completion time and low cluster throughput. Due to the strict scheduling order, it is difficult to prioritize reduce tasks that are predicted to run longer than others. Further, partitions required by a reduce task may not be generated at the time it is scheduled, occupying the reduce slot and wasting cluster cycles which would otherwise be used by another reduce with all partitions ready.

The output of a map task is a pool of intermediate keys and their associated value lists. Hadoop categorizes each output file into partitions, one per reduce task and each containing a different subset of the in-between key space. By default, Hadoop determines which partition a key/value pair will go to by computing a hash value. Since the intermediate output of the same key are always allocated to the same partition, skew in the input data set will result in inequality in the partition sizes. Such a partitioning skew is witnessed in many applications the output of a map task is a pool of intermediate keys and their associated value lists. Hadoop categorizes each output file into partitions, one per reduce task and each containing a different subset of the in-between key space. By default, Hadoop determines which partition a key/value pair will go to by computing a hash value. Since the intermediate output of the same key are always allocated to the same partition, skew in the input data set will result in inequality in the partition sizes. Such a partitioning skew is witnessed in many applications.

As part of a reduce task, shuffle cannot start until the matching reduce is scheduled. Besides the inadequacy of job execution, the coupling of shuffle and reduce also leaves the possible parallelism between idle jobs. In a production environment, a MapReduce cluster is shared by many users and multiple jobs [7]. Each job only gets a share of the execution slots and often requires several execution waves, each of which involves of one round of map or reduce tasks. Because of the coupling, data shuffling in future reduce waves cannot be overlapped with map waves. De-coupling shuffle from reduce offers a number of paybacks. It enables skew-aware placement of shuffled data, elastic scheduling of reduce tasks, and complete overlapping the shuffle phase with map tasks.

IV. DESIGN PROCESS

In proposed design, a job-independent shuffle service that pushes the map output to its designated reduce node. It decouples shuffle and reduce, and allows shuffle to be performed independently from reduce. It calculates the map output partition sizes and automatically balances the placement of map output partitions across nodes. This process binds reduce IDs with partition IDs lazily at the time reduce tasks are planned, allowing flexible scheduling of reduce tasks.

A major requirement of this design is the compatibility to existing Hadoop jobs. To this end, I have to design shufflers and the shuffle manager as job-independent components, which are responsible for collecting and distributing map output data. This design allows the cluster administrator to enable or disable this approach through the options in the configuration files. Any user job can use this service without modifications. The shuffler implements a shuffle-on-write process that proactively pushes the map output data to different nodes for future reduce tasks every time such data is written to local disks. The shuffling of all map output data can be performed before the execution of reduces tasks. The shuffle manager maintains a global view of partition sizes across all slave nodes. An automatic partition placement algorithm is used to determine the destination for each map output partition. The objective is to balance the global data distribution and moderate the non-uniformity reduce execution time. The task scheduler in this design assigns a partition of a reduce task only when the task is dispatched to a node with available slots. To minimize reduce execution time, it always associates partitions that are already local on the reduce node to the scheduled reduce.

This decouples shuffle from a reduce task and implements data shuffling as a platform service. This allows the shuffle phase to be performed independently from map and reduce tasks. The introduction of this design to the Hadoop environment presents two challenges: user transparency and fault tolerance. Besides user-defined map and reduce functions, Hadoop allows customized practitioner and combiner. To ensure that this process is user-transparent and does not require any change to the existing MapReduce jobs, I design the Shuffler as an independent component in the TaskTracker.

It takes input from the combiner, the last user-defined component in map tasks, performs data shuffling and provides input data for reduce tasks. The shuffler performs data shuffling every time the output data is written to local disks by map tasks, thus we name the operation shuffle-on-write. The shuffler contains multiple DataSpillHandler, one per map task, to collect map output that has been written to local disks. Map tasks write the stored partitions to the local file system when a spill of the in-memory buffer follows. We interrupt the writer class IFile.Writer in the combiner and add a DataSpillHandler class to it. While the default writer writing a spill to local disk, the DataSpillHandler copies the spill to

a round buffer, DataSpillQueue, from where data is shuffled/ dispatched to different nodes in Hadoop. During output collection, the DataSizePredictor monitors input data sizes and resulted partition sizes, and reports these statistics to the shuffle manager.

The shuffler proactively drives data partitions to nodes where reduce tasks will be launched. Specifically, a DataDispatcher reads a partition from the DataSpillQueue and queries the shuffle manager for its destination. Based on the assignment decision, a partition could be dispatched to the shuffler on a different node or to the local union in the same shuffler.

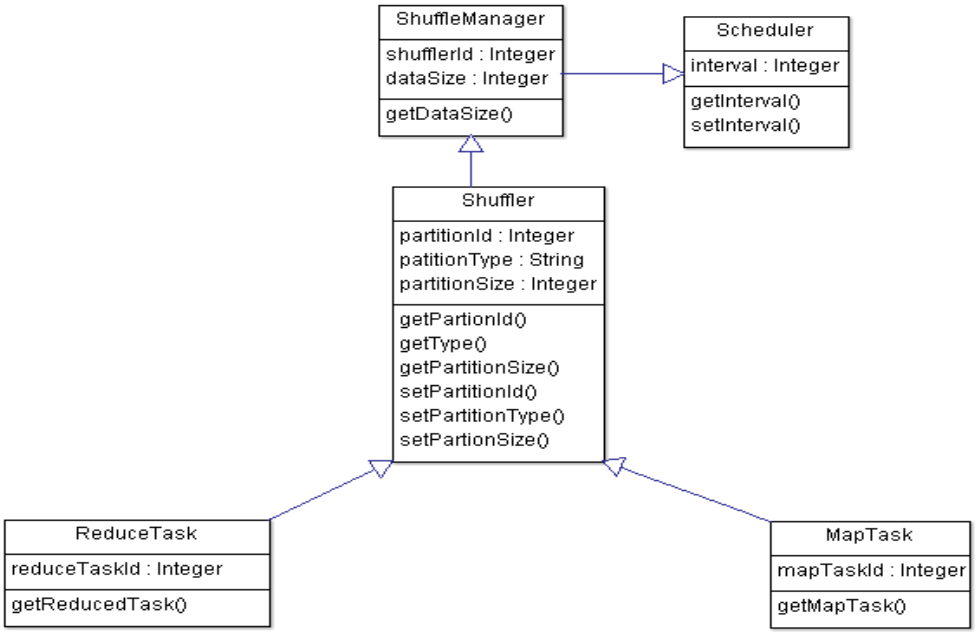

Figure 1 Class Diagram of the Design

The map output data shuffled at different times requirements to be merged to a single reduce input file and sorted by key before a reduce task can use it. The local merger accepts remotely and locally shuffled data and merges the partitions belonging to the same reduce task into one reduce input. To safeguard correctness, the merger only merges partitions from successfully finished map tasks.

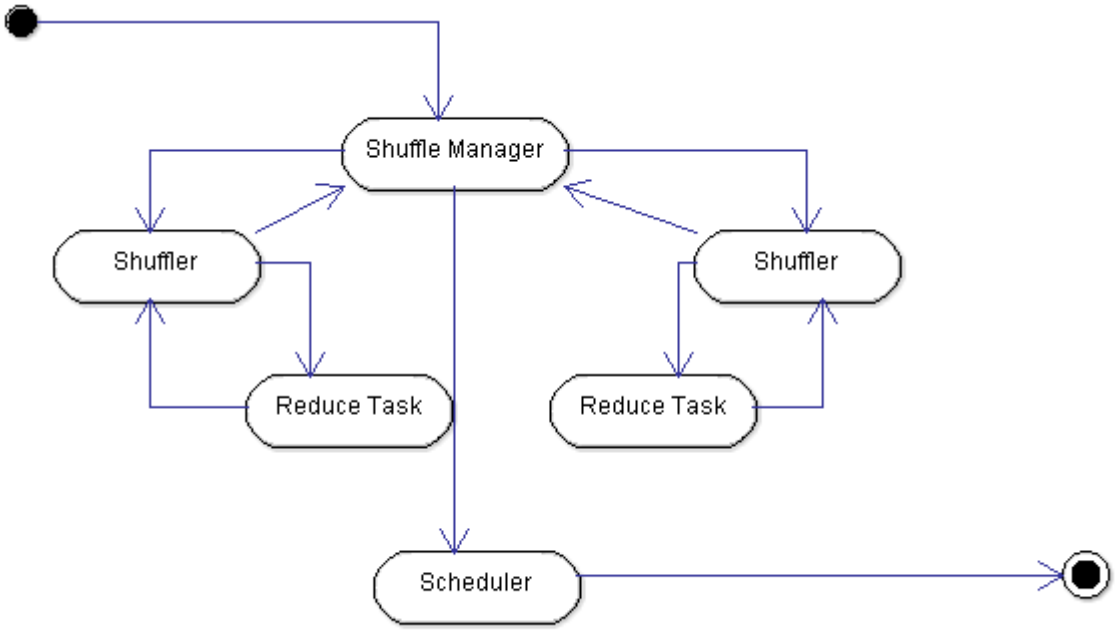

Figure 2 Activity Diagram of the design

The proposed design is robust to the failure of map and reduce tasks. Similar to hadoop mapreduce design this design also maintains an accounting of spill files from all map tasks. If a map task fails, its data spills in the DataSpillQueue and

merger will be rejected. The merger merges partitions only when the conforming map tasks commit their execution to the JobTracker. This prevents reduce tasks from using partial data. We also keep the merged reduce inputs in the merger until reduce tasks finish. In case of a failed reduce task, a new reduce can be started locally without fetching all the needed map output. The shuffle-on-write workflow relies on key information about the partition placement for each running job. The objective of partition placement is to balance the distribution of map output data across different nodes, so that the reduce workloads on different nodes are even.

The optimal partition placement can be determined when the sizes of all partitions are known. However, this requires that all map tasks are finished when making the placement decisions, which effectively impose a serialization between map tasks and the shuffle phase. This process guesses the final partition sizes based on the amount of processed input data and current partition size, and uses the estimation to guide partition settlement. The size of a map output partition hinges on the size of its input dataset, the map function, and the partitioner. We found that the ratio of map output size and input size, also known as map selectivity, is invariant given the same job configuration. As such, the partition size can be determined using the metric of map selectivity and input data size. The shuffle manager monitors the execution of individual map tasks and estimates the map selectivity of a job by constructing a mathematical model between input and output sizes. For a given job, the input dataset is divided into a number of logical splits, one per map task. Since individual map tasks run the same map function, each map task shares the same map selectivity with the overall job execution.

By perceiving the execution of map tasks, where a number of input/output size pairs are collected, shuffle manager builds a model estimating the map selectivity metric. Shuffle manager makes k observations of the size of each map output partition. As suggested in [8], it derives a linear model between partition size and input data size:

$$p_{i,j} = a_j + b_j \cdot D_i,$$

Where,

$p_{i,j}$ is the jth partition size in the ith observation

D_i is the corresponding input size.

We use linear deterioration to obtain the parameters for m partitions, one per reduce task. Since MapReduce jobs contain many more map tasks than reduce tasks, we are able to collect adequate samples for building the model. Once a model is obtained, the final size of a map output partition can be calculated by replacing D_i with the actual input size of the map task.

With predicted partition sizes, the shuffle manager determines the finest partition placement that balances reduce workload on different nodes. Because the execution time of a reduce task is linear to its input size, evenly placing the partitions leads to balanced workload.

Formally, the partition placement problem can be formulated as: given m map output partitions with sizes of *p1*, *p2*, . . . , *pm*, find the placement on n nodes, *S1*,*S2*, . . . ,*Sn*, that minimizes the placement difference:

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\mu - \sum_{j \in Si} Pj\right)}$$

where μ is the average data size on one node.

Partition placement problem can be viewed as the load balancing problem in multiprocessor systems. While the optimal solution can be excessively expensive to attain, I propose a heuristic based approach to approximate an optimal placement.

In Hadoop, reduce tasks are allocated map output partitions statically during job initialization. When there are reduce slots available on idle nodes, reduce tasks are dispatched according to the ascending order of their task IDs. This restriction on reduce scheduling leads to inefficient execution where reduces that are waiting for map tasks to finish occupy the slots for a long time. Because shuffle proactively thrusts output partitions to nodes, it requires that reduce tasks are launched on nodes that hold the corresponding shuffled partitions.

To this end, shuffle breaks the binding of reduce tasks and map output partitions and provides flexible reduce scheduling. A spontaneous approach for flexible reduce scheduling is to traverse the task queue and find a reduce that has shuffled data on the requesting node. However, this approach does not guarantee that there is always a "local" reduce available for dispatching. Shuffle employs a different approach that assigns partitions to reduce tasks at the time of dispatching. For single-user clusters, we improved Hadoop's FIFO scheduler to support the runtime task-partition binding. When a node with available reduce slots requests for new reduce tasks, the task scheduler first check with the shuffle manager to obtain the list of partitions that reside on this node.

The scheduler picks the first partition in the list and associates its ID with the first reduce task in the waiting queue. The selected reduce task is then launched on the node. As such, all reduce tasks are guaranteed to have local access to their input data. For multi-user clusters with heterogeneous workloads, we add the support for runtime task-partition association to the Hadoop Fair Scheduler (HFS). The minimum fair share allocated to individual users can negatively

affect the effectiveness of shuffle as reduce tasks may be launched on remote nodes to enforce fairness. We disable such fairness enforcement for reduce tasks to support more flexible scheduling. This allows some users to temporarily run more reduce tasks than others. We rely on the following designs to preserve fairness among users and avoid starvation.

First, the fair share of map tasks is still in effect, guaranteeing fair chances for users to make map output partitions. Second, while records are sorted by key within each partition after shuffling, partitions going to different users are randomly placed in the list, giving each user an equal opportunity to launch reduce tasks. Finally and most importantly, reduce tasks are started only when all their input data is available. This may provisionally violates fairness, but prevents wasted cluster cycles spent in waiting for incomplete maps and results in more efficient job execution.

V. OBSERVATION

We analyze the effectiveness of this design in dropping whole job completion time with more broad benchmarks. We will use the job completion time in typical Hadoop implementation as the standard and compare the standardized performance of shuffling and Hadoop-A. The results will show that for shuffle-heavy benchmarks such as self-join, terasort, and ranked-inverted-index, shuffle outdone the Hadoop by substantial performance. The proposed shuffle design will also beat Hadoop-A by approx. 22.7%, 21.9%, and 21.1% in these benchmarks. The result with k-means benchmark does not show substantial job execution time reduction between new shuffle and original Hadoop. This is because k-means only has 6 reduce tasks. With only one slot of reduce tasks, Hadoop was able to overlap the shuffle phase with map tasks and had analogous performance as new shuffling operation. However, due to the extra delay of remote disk access, Hadoop-A had lengthier reduces, thus lengthier total completion time.

Benchmarks like inverted-index, term-vector, and word count also fit in the shuffle-heavy group, but the shuffle volumes are lesser than other shuffle-heavy benchmarks. These benchmarks had fewer shuffle delay than other shuffle-heavy benchmarks just because there was fewer data to be copied during the shuffle phase. Therefore, the performance enhancement due to new shuffle was less. Proposed shuffle will reached approx. 20.3%, 19.7%, and 15.6% enhanced performance than Hadoop with these benchmarks, respectively. For these benchmarks, Hadoop-A still increased some performance improvement over stock Hadoop as the reduction on shuffle delay balanced the lengthy reduce phase. However, the performance improvement will be minimal with approx. 7.5%, 8.6%, and 5.5% improvement, respectively.

For the shuffle-light benchmarks, because the shuffle delay is insignificant. Both shuffle and Hadoop- A completes almost no performance gain. The performance degradation due to remote disk access in Hadoop-A is clearer in this scenario.

We also relate the shuffle delay between the new shuffle, and Hadoop-A.. We used the shuffle delay of new shuffle as the starting point. The results match with the observation we made in previous experiments. Improved Shuffle was able to reduce the shuffle delay considerably if the job had big volumes of shuffled data and multiple reduce slots. For benchmarks that have the largest shuffle-volume, the decreases in shuffle delay were more than 10x compared with Hadoop. For benchmarks with average shuffle volume, the improvement on shuffle delay was from 4.5x to 5.5x.

We will display that shuffle efficiently hides shuffle latency by covering map tasks and data shuffling. In this subsection, we study how the stable partition placement affects job performance. To separate the effect of partition placement, we first ran benchmarks under Hadoop and logged dispatching past of reduce tasks. Then, we arranged new shuffle to place partitions on nodes in a way that leads to the similar reduce execution sequence. As such, job implementation enjoys coincided shuffle provided by shuffle, but bears the same partitioning angle in Hadoop. We match the performance with stable partition placement and Hadoop. Performance enhancement due to balanced partition placement. The results will display that new shuffle reaches 8-12% performance improvement over Hadoop. We add the performance increase to the prediction-based partition placement that softens the partitioning angle. It stops idler tasks from delaying job execution time. The partition placement in new shuffle relies on perfect predictions of the specific partition sizes.

VI. CONCLUSIONS

Hadoop provides a simplified implementation of the MapReduce framework, but its design poses challenges to attain the best performance in job execution due to tightly coupled shuffle and reduce, partitioning skew, and inflexible scheduling. In this paper, I have proposed improved shuffle operations, a novel user-transparent shuffle service that provides optimized data shuffling to improve job performance. It decouples shuffle from reduce tasks and proactively pushes data to be shuffled to Hadoop node via a novel shuffle-on write operation in map tasks. This Shuffle further optimizes the scheduling of reduce tasks by automatic balancing workload on multiple nodes and runtime flexible reduce scheduling. We will implement new shuffle as a configurable plug-in in Hadoop and evaluated its effectiveness on a 4-node cluster with various workloads. Experimental results will show that shuffle is able to reduce job completion time by as much as 30.2%. Proposed shuffle design will also significantly improve job performance in a multi-user Hadoop cluster running heterogeneous workloads.

ACKNOWLEDGMENT

I am thankful to my guide Prof. Archna Lomte, Head of computer department Prof. Gayatri Bhandari for their kind inputs in preparing this research paper.

REFERENCES

[1] *https://github.com/cloudera/impala*.

[2] *http://aws.amazon.com/about-aws/whats-new/2010/10/20/amazonelastic-mapreduce-introduces-resizing-running-job-flows*

[3] G. Ananthanarayanan et al. Pacman: *Coordinated memory caching for parallel jobs*. In NSDI, 2012.
[4] Jinshuang Yan, Xiaoliang Yang, Rong Gu, Chunfeng Yuan, and Yihua Huang :*Performance Optimization for Short MapReduce Job Execution in Hadoop* ,Department of Computer Science and Technology National Key Laboratory for Novel Software Technology Nanjing University Nanjing 210093, China ,2013.
[5] J. Dean and S. Ghemawat. MapReduce: *Simplified data processing on large clusters*. In OSDI, 2004.
[6] Y. Low et al. Distributed graphlab: *a framework for machine learning and data mining in the cloud.* VLDB, 2012.
[7] Jinshuang Yan, Xiaoliang Yang, Rong Gu, Chunfeng Yuan, and Yihua Huang: *Performance Optimization for Short MapReduce Job Execution in Hadoop* .ICCGC,2012,23-30
[8] Beyer, Mark. "*Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data*". Gartner. Retrieved 13 July 2011, 13-20
[9] Verma, a., cherkasova, l., and campbell, r. H. Aria: *automatic resource inference and allocation for mapreduce environments*. In Proc. of the ACM Int'l Conference on Autonomic Computing (ICAC) (2011), 45-90
[10] Ananthanarayanan, g., agarwal, s., kandula, s., greenberg, a., stoica, i., harlan, d., and harris, e. Scarlett: *coping with skewed content popularity in mapreduce clusters.* In Proc. of the ACM European Conference on Computer Systems (EuroSys) (2011),20-37
[11] Chiang, r. C., and huang, h. H. Tracon: *interference ware scheduling for data-intensive applications in virtualized environments*. In Proc. of Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC) (2011), 34-78
[12] Dewitt, d., and gray, j. *Parallel database systems: the future of high performance database systems*. Communication of ACM 35, 6 (1992), 85–98.
[13] Kwon, y., balazinska, m., howe, b., and rolia, J. *Skew-resistant parallel processing of feature-extracting scientific user-defined functions*. In Proc. of the ACM Symposium on Cloud Computing (SOCC) (2010),35-89
[14] Wolf, j., rajan, d., hildrum, k., khandekar, r., kumar, v., parekh, s., wu, k.-l., and balmin, a. *Flex: a slot allocation scheduling optimizer for mapreduce workloads. In Proc.* of the ACM/IFIP/USENIX Int'l Conference on Middleware (2010), 40-63.

