The Case for Evaluating MapReduce Performance Using Workload Suites

Yanpei Chen
Archana Ganapathi
Rean Griffith
Randy H. Katz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-21
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-21.html

March 30, 2011

The Case for Evaluating MapReduce Performance Using Workload Suites

Yanpei Chen, Archana Ganapathi, Rean Griffith, Randy Katz
EECS Dept., University of California, Berkeley
{ychen2, archanag, rean, randy}@eecs.berkeley.edu

***Abstract*—MapReduce systems face enormous challenges due to increasing growth, diversity, and consolidation of the data and computation involved. Provisioning, configuring, and managing large-scale MapReduce clusters require realistic, workload-specific performance insights that existing MapReduce benchmarks are ill-equipped to supply.**

In this paper, we build the case for going beyond benchmarks for MapReduce performance evaluations. We analyze and compare two production MapReduce traces to develop a vocabulary for describing MapReduce workloads. We show that existing benchmarks fail to capture rich workload characteristics observed in traces, and propose a framework to synthesize and execute representative workloads. We demonstrate that performance evaluations using realistic workloads gives cluster operator new ways to identify workload-specific resource bottlenecks, and workload-specific choice of MapReduce task schedulers.

We expect that once available, workload suites would allow cluster operators to accomplish previously challenging tasks beyond what we can now imagine, thus serving as a useful tool to help design and manage MapReduce systems.

I. Introduction

MapReduce is a popular paradigm for performing parallel computations on large data. Initially developed by large Internet enterprises, MapReduce has been adopted by diverse organizations for business critical analysis, such as click stream analysis, image processing, Monte-Carlo simulations, and others [1]. Open-source platforms such as Hadoop have accelerated MapReduce adoption.

While the computation paradigm is conceptually simple, the logistics of provisioning and managing a MapReduce cluster are complex. Overcoming the challenges involved requires understanding the intricacies of the anticipated workload. Better knowledge about the workload enables better cluster provisioning and management. For example, one must decide how many and what types of machines to provision for the cluster. This decision is the most difficult for a new deployment that lacks any knowledge about workload-cluster interactions, but needs to be revisited periodically as production workloads continue to evolve. Second, MapReduce configuration parameters must be fine-tuned to the specific deployment, and adjusted according to added or decommissioned resources from the cluster, as well as added or deprecated jobs in the workload. Third, one must implement an appropriate workload management mechanism, which includes but is not limited to job scheduling, admission control, and load throttling.

Workload can be defined by a variety of characteristics, including computation semantics (e.g., source code), data characteristics (e.g., computation input/output), and the real-time job arrival patterns. Existing MapReduce benchmarks, such as Gridmix [2], [3], Pigmix [4], and Hive Benchmark [5], test MapReduce clusters with a small set of "representative" computations, sized to stress the cluster with large datasets. While we agree this is the correct initial strategy for evaluating MapReduce performance, we believe recent technology trends warrant an advance beyond benchmarks in our understanding of workloads. We observe three such trends:

- *Job diversity:* MapReduce clusters handle an increasingly diverse mix of computations and data types [1]. The optimal workload management policy for one kind of computation and data type may conflict with that for another. No single set of "representative" jobs is actually representative of the full range of MapReduce use cases.
- *Cluster consolidation:* The economies of scale in constructing large clusters makes it desirable to consolidate many MapReduce workloads onto a single cluster [6], [7]. Cluster provisioning and management mechanisms must account for the non-linear superposition of different workloads. The benchmark approach of high-intensity, short duration measurements can no longer capture the variations in workload superposition over time.
- *Computation volume:* The computations and data size handled by MapReduce clusters increases exponentially [8], [9] due to new use cases and the desire to perpetually archive all data. This means that small misunderstanding of workload characteristics can lead to large penalties.

Given these trends, it is no longer sufficient to use benchmarks for cluster provisioning and management decisions. In this paper, we build the case for doing MapReduce performance evaluations using a collection of workloads, i.e., workload suites. To this effect, our contributions are as follows:

- Compare two production MapReduce traces to both highlight the diversity of MapReduce use cases and develop a way to describe MapReduce workloads.
- Examine several MapReduce benchmarks and identify their shortcomings in light of the observed trace behavior.
- Describe a methodology to synthesize representative workloads by sampling MapReduce cluster traces, and then execute the synthetic workloads with low performance overhead using existing MapReduce infrastructure.
- Demonstrate that using workload suites gives cluster operators new capabilities by executing a particular workload to identify workload-specific provisioning bottlenecks and

inform the choice of MapReduce schedulers.

We believe MapReduce cluster operators can use the workload suites to accomplish a variety of previously challenging tasks, beyond just the two new capabilities demonstrated here. For example, operators can anticipate the workload growth in different data or computational dimensions, provision the added resources just in time, instead of over-provisioning with wasteful extra capacity. Operators can also select highly-specific configurations optimized for different kinds of jobs within a workload, instead of having uniform configurations optimized for a "common case" that may not exist. Operators can also anticipate the impact of consolidating different workloads onto the same cluster. Using the workload description vocabulary we introduce, operators can systematically quantify the superposition of different workloads across many workload characteristics. In short, once workload suites become available, we expect cluster operators to use them to accomplish innovative tasks beyond what we can now imagine.

In the rest of the paper, we build the case for using workload suites by looking at production traces (Section II) and examining why benchmarks cannot reproduce the observed behavior (Section III). We detail our proposed workload synthesis and execution framework (Section IV), demonstrate that it executes representative workloads with low overhead, and gives cluster operators new capabilities (Section V). Lastly, we discuss opportunities and challenges for future work (Section VI).

A. MapReduce Overview

MapReduce is a straightforward divide and conquer algorithm. The input data consists of key-value pairs. It is stored on a distributed file system in mutually exclusive but jointly exhaustive partitions. A *map* function is applied on the input data to produce intermediate key-value pairs. The intermediate data is then *shuffled* to appropriate nodes, where the *reduce* function aggregates the intermediate data to generate the final output key-value pairs. For more details about MapReduce, we refer the reader to [10].

II. Lessons from Two Production Traces

There is a chronic shortage of production traces available for MapReduce researchers. Without examining these traces, it would be impossible to evaluate various proposed MapReduce benchmarks. We have access to two production Hadoop MapReduce traces, which we analyze and compare below.

One trace comes from a 600-machine cluster at Facebook (FB trace), spans 6 months from May 2009 to October 2009, and contains roughly 1 million jobs. The other trace comes from a cluster of approximately 2000 machines at Yahoo! (YH trace), covers three weeks in late February 2009 and early March 2009, and contains around 30,000 jobs. Both traces contain a list of job submission and completion times, data sizes for the input, shuffle and output stages, and the running time in task-seconds of map and reduce functions (e.g., 2 task running 10 seconds each will be 20 task-seconds). Thus, these traces offer a rare opportunity to compare two large-scale MapReduce deployments using the same trace format.

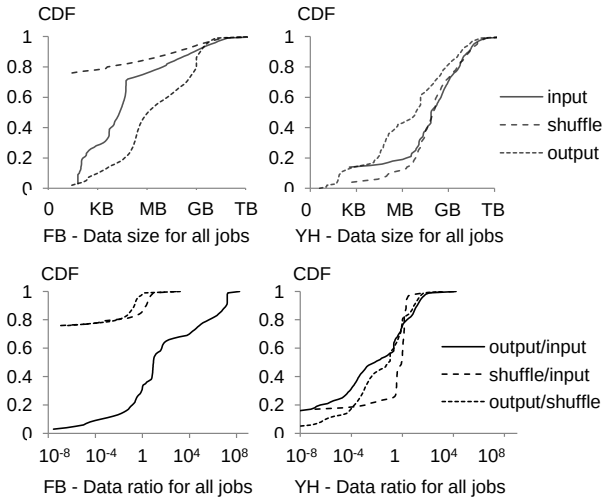

Fig. 1. Data size at input/shuffle/output (top), and data ratio between each stage (botttom).

We compare MapReduce data characteristics (Section II-A), job submission and data movement rates (Section II-B), and common jobs within each trace (Section II-C). The comparison helps us develop a vocabulary to capture key properties of MapReduce workloads (Section II-D).

A. Data Characteristics

MapReduce operates on key-value pairs at input, shuffle (intermediate), and output stages. The size of data at each of these stages provide a first-order indication of "what data the jobs run on". Figure 1 shows the aggregate data sizes and data ratios at the input/shuffle/output stages, plotted as a cumulative distribution function (CDF) of the jobs in each trace.

The input, shuffle, and output data sizes range from KBs to TBs in both traces. Within the same trace, the data sizes at different MapReduce stages follow different distributions. Across the two traces, the same MapReduce stage also has different distributions. Thus, the two MapReduce systems were performing different computations on different data sets. Additionally, many jobs in the FB trace have no shuffle stage - the map outputs are directly written to the Hadoop Distributed File System (HDFS). Consequently, the CDF of the shuffle data sizes has a high density at 0.

The data ratios between the output/input, shuffle/input, and output/shuffle stages also span several orders of magnitude. Interestingly, there is little density around 1 for the FB trace, indicating that most jobs are data expansions (ratio $>>$ 1 for all stages) or data compressions (ratio $<<$ 1 for all stages). The YH trace shows more data transformations (ratio $\approx$ 1).

B. Job Submission and Data Intensity Over Time

Job submission patterns and data intensities indicate "how much work" there is. The YH trace is too short to capture the long-term evolution in job submission and data intensity. Thus, we focus on the FB trace for this analysis.

Figure 2 shows weekly aggregates of job counts and the sum of input, shuffle, and output data sizes over the entire trace. There is no long term growth trend in the number of jobs or

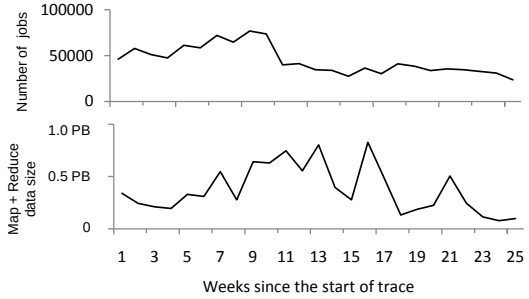

Fig. 2. Weekly aggregate of job counts (top) and sum of map and reduce data size (below). FB trace.

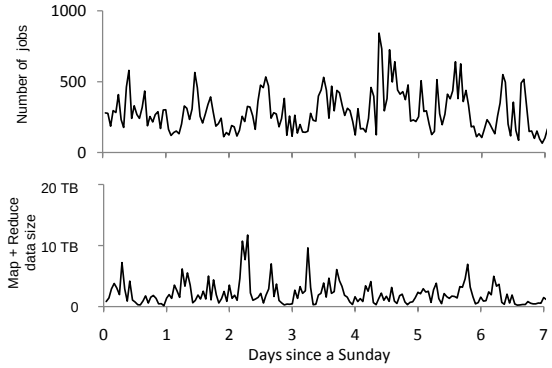

Fig. 3. Hourly job submission rate (top) and sum of input, shuffle, output data size (below) over a randomly selected week. FB trace.

the sum of data size. Also, there is high variation in the sum data size but not in the number of jobs, indicating significant changes in the data processed. We also see a sharp drop in the number of jobs in Week 11, which our Facebook collaborators clarified was due to a change in cluster operations.

Figure 3 shows hourly aggregates of job counts and sum data sizes over a randomly selected week. We do not aggregate at scales below an hour as many jobs take tens of minutes to complete. There is high variation in both the number of jobs and the sum data size. The number of jobs also show weak diurnal patterns, peaking at mid-day and dropping at mid-night. To detect the existence of any cycles, we performed Fourier analysis on the hourly job counts over the entire trace. There are visible but weak cycles at the 12 hour, daily, and weekly frequencies, mixed in with a high amount of noise.

C. Common Jobs Within Each Trace

In this section, we investigate whether we can capture "what the computation is" using a small set of "common jobs". We use k-means, a well-known data clustering algorithm [11], to extract such information. We describe each job with many features (dimensions), and input the array of all jobs into k-means. K-means finds the natural clusters of data points, i.e., jobs. We consider jobs in the same cluster as belonging to a single equivalence class, i.e., a single "common job".

We describe each job using all 6 features available from the cluster traces - the job's input, shuffle, output data sizes in bytes, its running time in seconds, and its map and reduce time in task-seconds. We linearly normalize all data dimensions to a range between 0 and 1 to account for the different measurement units in each feature. We increment k, the number of clusters, until there is diminishing improvement in the cluster quality. We measure cluster quality by "*variance explained*", a standard metric computed by the difference between the total variance in all data points and the residual variance between the data points and their assigned cluster centers.

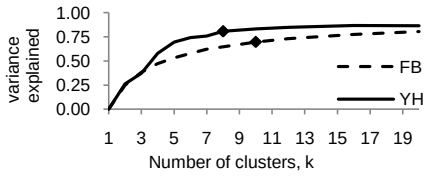

Fig. 4. Cluster quality - % variance explained vs. the number of clusters. The marker indicates the number of clusters used for more detailed analysis.

Figure 4 shows the % variance explained as k increases. Even at small k, we start seeing diminishing improvements. Having too many clusters will lead to an unwieldy number of common jobs, even though the variance explained will be higher. We believe a good place to stop is $k = 10$ for the FB trace, and $k = 8$ for the YH trace, indicated by the markers in Figure 4. At these points, the clustering structure explains 70% (FB) and 80% (YH) of the total variance, suggesting that a small set of "common jobs" can indeed cover a large range of per-job behavior in the trace data.

We can identify the characteristics of these common jobs by looking at the numerical values of the cluster centers. Table I shows the cluster size and our manually applied labels. We derive the labels from looking at the data ratios and durations at various stages, e.g., "aggregate, fast" jobs have shuffle/input and output/shuffle ratios both $<<1$, and relatively small duration compare with the other jobs.

Both traces have a large cluster of small jobs, and several small clusters of various large jobs. Small jobs dominate the total number of jobs, while large jobs dominate all other dimensions. Thus, for performance metrics that place equal weights on all jobs, the small jobs should be an optimization priority. However, large jobs should be the priority for performance metrics that weigh each job according to its "size" either in data, running time, or map and reduce task time.

Also, the FB and YH traces contain different job types. the FB trace contains many data loading jobs, characterized by large output data size $>>$ input data size, with minimal shuffle data. The YH trace does not contain this job type. Both traces contain some mixture of jobs performing data aggregation (input $>$ output), expansion (input $<$ output), transformation (input $\approx$ output), and summary (input $>>$ output), with each job type in varying proportions.

D. To Describe a Workload

Our comparison shows that the two traces capture different MapReduce use cases. Thus, we need a good way to describe each workload and compare it against other workloads.

We believe a good description should focus on the semantics at the MapReduce abstraction level. This includes the data

TABLE I
CLUSTER SIZES, MEDIANS, AND LABELS FOR FB (TOP) AND YH (BELOW). MAP TIME AND REDUCE TIME ARE IN TASK-SECONDS, E.G., 2 TASKS OF 10 SECONDS EACH IS 20 TASK-SECONDS.

# Jobs	Input	Shuffle	Output	Duration	Map time	Reduce time	Label
Facebook trace							
1081918	21 KB	0	871 KB	32 s	20	0	Small jobs
37038	381 KB	0	1.9 GB	21 min	6,079	0	Load data, fast
2070	10 KB	0	4.2 GB	1 hr 50 min	26,321	0	Load data, slow
602	405 KB	0	447 GB	1 hr 10 min	66,657	0	Load data, large
180	446 KB	0	1.1 TB	5 hrs 5 min	125,662	0	Load data, huge
6035	230 GB	8.8 GB	491 MB	15 min	104,338	66,760	Aggregate, fast
379	1.9 TB	502 MB	2.6 GB	30 min	348,942	76,736	Aggregate and expand
159	418 GB	2.5 TB	45 GB	1 hr 25 min	1,076,089	974,395	Expand and aggregate
793	255 GB	788 GB	1.6 GB	35 min	384,562	338,050	Data transform
19	7.6 TB	51 GB	104 KB	55 min	4,843,452	853,911	Data summary
Yahoo trace							
21981	174 MB	73 MB	6 MB	1 min	412	740	Small jobs
838	568 GB	76 GB	3.9 GB	35 min	270376	589385	Aggregate, fast
91	206 GB	1.5 TB	133 MB	40 min	983998	1425941	Expand and aggregate
7	806 GB	235 GB	10 TB	2 hrs 35 min	257567	979181	Transform and expand
35	4.9 TB	78 GB	775 MB	3 hrs 45 min	4481926	1663358	Data summary
5	31 TB	937 GB	475 MB	8 hrs 35 min	33606055	31884004	Data summary, large
1303	36 GB	15 GB	4.0 GB	1 hr	15021	13614	Data transform
2	5.5 TB	10 TB	2.5 TB	4 hrs 40 min	7729409	8305880	Data transform, large

on which the jobs run, the number of jobs there are in the workload and their arrival patterns, and the computation performed by the jobs. A workload description at this level will persist despite any changes in the underlying physical hardware (e.g., CPU/memory/network), or any overlaid functionality extensions (e.g., Hive or Pig).

Data on which the jobs run:

The data size and data ratios at the input/shuffle/output stages give a first-order description of the data. We need to describe both the statistical distribution of data sizes, and the per-job data ratios at each stage. Statistical techniques like k-means can extract the dependencies between various data dimensions. Also, the data format should be captured. Even though our traces contain no such information, data formats help us separate workloads that have the same data sizes and ratios, but different data content.

Number of jobs and their arrival patterns:

The list of job submissions and submission times will give a very specific description. Given that we cannot list submission times for all jobs, we should describe submission patterns using averages, peak-to-average ratios, diurnal patterns, etc.

Computation performed by the jobs:

The actual code for map and reduce tasks represents the most accurate description the computation done. However, the code is often unavailable due to logistical and confidentiality reasons. We believe that a good alternative is to identify classes of common jobs in the workload, similar to our k-means analysis. Here, we identify common jobs using data characteristics, job durations, and task run times. When the code is available, it would be helpful to add some semantic description, e.g., text parsing, reversing indices, image processing, detecting statistical outliers, etc. We expect this information would be available within the organizations directly managing the MapReduce clusters.

The above facilitates a qualitative description. We introduce a quantitative description in Section IV when we describe how to synthesize a representative workload.

TABLE II
SUMMARY OF SHORTCOMINGS OF RECENT MAPREDUCE BENCHMARKS, COMPARED AGAINST WORKLOAD SUITES (RIGHT-MOST COLUMN).

	Grid-mix2	Hive BM	Hi bench	Pig Mix	Grid-mix3	WL suites
Diverse job types			√	√	√	√
Right # of jobs for each job type					√	√
Variations in job submit intensity					√	√
Representative data-sizes					√	√
Easy to generate scaled/anticipated workloads						√
Easy to generate consolidated workloads						√
Cluster & config. independent	√	√	√	√		√

III. SHORTCOMINGS OF MAPREDUCE BENCHMARKS

In this section, we discuss why existing benchmarks are insufficient for evaluating MapReduce performance. Our thesis is that existing benchmarks are not representative. They capture narrow slivers of a rich space of workload characteristics. What is needed is a framework for constructing workloads that allows us to select and combine various characteristics.

Table II summarizes the strengths and weaknesses of five contemporary MapReduce benchmarks – Gridmix2, Hive Benchmark, Pigmix, Hibench and Gridmix3. Below, we discuss each in detail. None of the existing benchmarks come close to the flexibility and functionality of workload suites.

Gridmix2 [2] includes stripped-down versions of "common jobs" – sorting text data and SequenceFiles, sampling from large compressed datasets, and chains of MapReduce jobs exercising the combiner. Gridmix 2 is primarily a saturation tool [3], which emphasizes stressing the framework at scale. As a

result, jobs produced from Gridmix tend towards the jobs with 100s of GBs of input, shuffle, and output data. While stress evaluations are an important aspect of evaluating MapReduce performance, the production workloads in Section II contain many jobs with KB to MB data sizes. Also, as we show later in Section V-C, running a representative workload places realistic stress on the system beyond that generated by Gridmix 2.

Hive Benchmark [5] tests the performance of Hive, a data warehousing infrastructure built on top of Hadoop MapReduce. It uses datasets and queries derived from those used in [12]. These queries aim to describe "more complex analytical workloads" and focus on "apples-to-apples" comparison against parallel databases. It is not clear that the queries in the Hive Benchmark reflect actual queries performed in production Hive deployments. Even if the five queries are representative, running Hive Benchmark does not capture different mixes, interleavings, arrival intensities, data sizes, etc., that one would expect in a production deployment of Hive.

HiBench [13] consists of a suite of eight Hadoop programs that include synthetic microbenchmarks and real-world applications – Sort, WordCount, TeraSort, NutchIndexing, PageRank, Bayesian Classification, K-means Clustering, and EnhancedDFSIO. These programs are presented as representing a wider diversity of applications than those used in prior MapReduce benchmarking efforts. While HiBench includes a wider variety of jobs, it still fails to capture the different job mixes and job arrival rates that one would expect in production MapReduce clusters.

PigMix [4] is a set of twelve queries intended to test the latency and the scalability limits of Pig – a platform for analyzing large datasets that includes a high-level language for constructing analysis programs and the infrastructure for evaluating them. While this collection of queries may be representative of the types of queries run in Pig deployments, there is no information on representative data sizes, query mixes, query arrival rate etc. to capture the workload behavior seen in production environments.

Gridmix3 [14], [3] was driven by situations where improvements measured to have dramatic gains on Gridmix2 showed ambiguous or even negative effects in production [14]. Gridmix3 replays job traces collected via Rumen [15] with the same byte and record patterns and submits them to the cluster in matching time intervals, thus producing comparable load on the I/O subsystems and preserving job inter-arrival intensities.

The direct replay approach reproduces inter-arrival rates and the correct mix of job types and data sizes, but introduces other shortcomings. For example, it is challenging to change the workload to add or remove new types of jobs, or to scale the workload along one or more dimensions of interest (data sizes, arrival patterns). Further, changing the input Rumen traces is difficult, limiting the benchmark's usefulness on clusters with configurations different from the cluster that initially generated the trace. For example, the number of tasks-per-job is preserved from the traces. Thus, evaluating the appropriate configuration of task size and task number is difficult. Misconfigurations of the original cluster would be replicated. Similarly, it is challenging to use Gridmix3 to explore the performance impact of combining or separating workloads, e.g., through consolidating the workload from many clusters, or separating a combined workload into specialized clusters.

IV. Workload Synthesis and Execution

We would like to synthesize a representative workload for a particular use case and execute it to evaluate MapReduce performance for a specific configuration. For MapReduce cluster operators, this approach offers more relevant insights than those gathered from one-size-fits-all benchmarks. We describe here a mechanism to synthesize representative workloads from MapReduce traces (Section IV-B), and a mechanism to execute the synthetic workload on a target system (Section IV-C).

A. Design Goals

We identify two design goals:

1. The workload synthesis and execution framework should be agnostic to hardware/software/configuration choices, cluster size, specific MapReduce implementation, and the underlying file system. We may intentionally vary any of these factors to quantify the performance impact of hardware choices, software (e.g., task scheduler) optimizations, configuration differences, cluster capacity increases, MapReduce implementation choices (open source vs. proprietary), or file system choices (distributed vs. local vs. in-memory).

2. The framework should synthesize representative workloads that execute in a short duration. Such workloads lead to rapid performance evaluations, i.e., rapid design loops. It is challenging to achieve both representativeness and short duration. For example, a trace spanning several months forms a workload that is representative by definition, but practically impossible to execute in full.

B. Workload Synthesis

The workload synthesizer takes as input a MapReduce trace over a time period of length L, and the desired synthetic workload duration W, with $W < L$. The MapReduce trace is a list of jobs, with each item containing the job submit time, input data size, shuffle/input data ratio, and output/shuffle data ratio. Traces with this information allow us to synthesize a workload with representative job submission rate and patterns, as well as data size and ratio characteristics. The data ratios also serve as an approximation to the actual computation being done. The approximation is a good one for a large class of IO-bound MapReduce computations.

The workload synthesizer divides the synthetic workload into N non-overlapping segments, each of length W/N. Each segment will be filled with a randomly sampled segment of length W/N, taken from the input trace. Each sample contains a list of jobs, and for each job the submit time, input data size, shuffle/input data ratio, and output/shuffle data ratio. We concatenate N such samples to obtain a synthetic workload of length W. The synthetic workload essentially samples the trace for a number of time segments. If $W << L$, the samples have low probability of overlapping.

The workload synthesis returns a list of jobs in the same format as the initial trace, with each item containing the job submit time, input data size, shuffle/input data ratio, and output/shuffle data ratio. The primary difference is that the synthetic workload is of duration $W < L$.

We satisfy design Requirement 1 by the choice of what data to include in the input trace – the data included contains no cluster-specific information. In Section V, we demonstrate that Requirement 2 is also satisfied. Intuitively, when we increase W, we get more samples, hence a more representative workload. When we increase W/N, we capture more representative job submission sequences, but at the cost of fewer samples within a given W. Adjusting W and N allows us to tradeoff representativeness in one characteristic versus another.

C. Workload Execution

The workload executor translates the job list from the synthetic workload to concrete MapReduce jobs that can be executed on artificially generated data. The workload executor runs a shell script that writes the input test data to the underlying file system (HDFS in our case), launches jobs with specified data sizes and data ratios, and sleeps between successive jobs to account for gaps between job submissions:

```
HDFS randomwrite(max_input_size)

sleep interval[0]
RatioMapReduce inputFiles[0] \
               output0 \
               shuffleInputRatio[0] \
               outputShuffleRatio[0]
HDFS -rmr output0 &

sleep interval[1]
RatioMapReduce inputFiles[1] \
               output1 \
               shuffleInputRatio[1] \
               outputShuffleRatio[1]
HDFS -rmr output1 &

...
```

Populate the file system with test data:

We write the input data to HDFS using the RandomWriter example included with recent Hadoop distributions. This job creates a directory of fixed size files, each corresponding to the output of a RandomWriter reduce task. We populate the input data only once, writing the maximum per-job input data size for our workload. Jobs in the synthetic workload take as their input a random sample of these files, determined by the input data size of each job. The input data size has the same granularity as the file sizes, which we set to be 64MB, the same size as default HDFS blocks. We believe this setting is reasonable because our input files would be as granular as the underlying HDFS. We validated that there is negligible overhead when concurrent jobs read from the same HDFS input (Section V).

MapReduce job to preserve data ratios:

We wrote a MapReduce job that reproduces job-specific shuffle-input and output-shuffle data ratios. This RatioMapReduce job uses a straightforward probabilistic identity filter to enforce data ratios. We show only the map function below. The reduce function uses an identical algorithm.

```
class RatioMapReduce {

  x = shuffleInputRatio

  map(K1 key, V1 value, <K2, V2> shuffle) {
    repeat floor(x) times {
      shuffle.collect(new K2(randomKey),
                      new V2(randomValue));
    }
    if (randomFloat(0,1) < decimal(x)) {
      shuffle.collect(new K2(randomKey),
                      new V2(randomValue));
    }
  } // end map()

  reduce(K2 key, <V2> values, <K3, V3> output) {
  ...
  }

} // end class RatioMapReduce
```

Removing HDFS output from synthetic workload:

We need to remove the data generated by the synthetic workload. Otherwise, the synthetic workload outputs accumulate, quickly reaching the storage capacity on a cluster. We used a straightforward HDFS remove command, issued to run as a background process by the main shell script running the workload. We also experimentally ensured that this mechanism imposes no performance overhead (Section V).

V. Evaluation

We believe an evaluation of a MapReduce workload suite should demonstrate three things – that the synthesized workload is actually *representative* (Section V-A), that the workload execution framework has *low overhead* (Section V-B), and that executing the synthesized workloads give cluster operators *new capabilities* otherwise unavailable (Sections V-C, V-D). We demonstrate all three by synthesizing a day-long workload using the Facebook traces and executing it to identify workload-specific system size bottlenecks and to inform workload-specific choice of MapReduce task schedulers.

A. Representativeness of the Synthetic Workload

By "representative", we mean that the synthetic workload should reproduce from the original trace the distribution of input, shuffle, and output data sizes (representative data characteristics), the mix of job submission rates and sequences, and the mix of common job types. We demonstrate all three by synthesizing day-long "Facebook-like" workloads using the Facebook trace and our workload synthesis tools.

Data characteristics

Figure 5 shows the distributions of input, shuffle, and output data sizes of the synthetic workload, compared with that in the original Facebook trace. To observe the statistical properties of the trace sampling method, we synthesized 10 day-long workloads using 1-hour continuous samples . We see that sampling does introduce a degree of statistical variation, but bounded around the aggregate statistical distributions of the entire trace. In other words, our workload synthesis method gives representative data characteristics.

We also repeated our analysis for different sample window lengths. We omit the graphs due to space limits. The results are

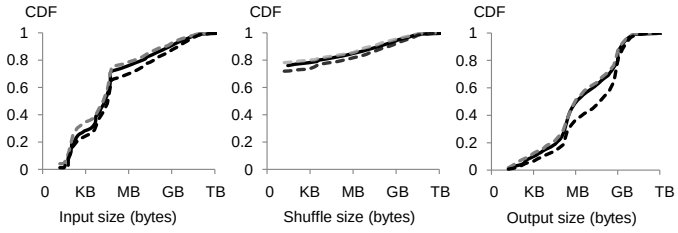

Fig. 5. Distributions of data sizes in synthesized workload using 1-hr samples. Showing that the data characteristics are representative – min. and max. distributions for the synthetic workload (dashed lines) bound the distribution computed over the entire trace (solid line).

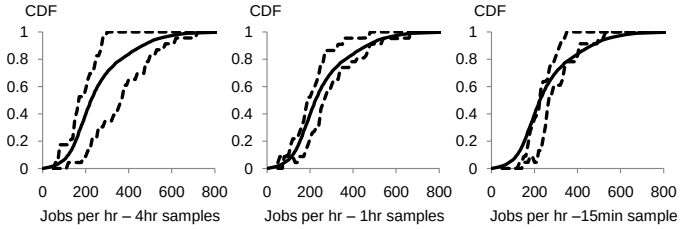

Fig. 6. Distributions of jobs per hour in synthetic workload. Short samples distort variations in job submit rates – min. and max. distributions for synthetic workload (dashed lines) bound the distribution for the entire trace (solid line) for 1 & 4hrs samples only.

intuitive - when the synthetic workload length is fixed, shorter sample lengths mean more samples and more representative distributions. In fact, according to statistics theory, the CDFs for the synthetic workloads converge towards the "true" CDF, with the bounds narrowing at O($n^{-0.5}$), where n is the number of samples we take from the original trace [16]. Thus, shorter sample lengths correspond to synthetic workloads that are more representative of the data characteristics.

Job submission patterns

Our intuition is that job submission-rate per time unit is faithfully reproduced only if the length of each sample is longer than the time unit involved. Otherwise, we would be essentially doing memoryless sampling, with the job submission rate fluctuating in a narrow range around the long term average, thus failing to reproduce workload spikes in the original trace. If the job sample window is longer than the time unit, then more samples would lead to a more representative mix of behavior, as we discussed previously.

Figure 6 confirms this intuition. The figure shows the jobs submitted per hour for workloads synthesized by sample windows of different lengths. We see that the workload synthesized using 4-hour samples has very loose bounds around the overall distribution, while the workload synthesized using 1-hour samples has closer bounds. However, the workload synthesized using 15-minute samples does not bound the overall distribution. In fact, the 15-minute sample synthetic workload has a narrow distribution around roughly 300 jobs per hour, the long-term average job submission rate. Thus, while shorter sample windows give us more representative data characteristics, they distort variations in job submission rates.

Common jobs

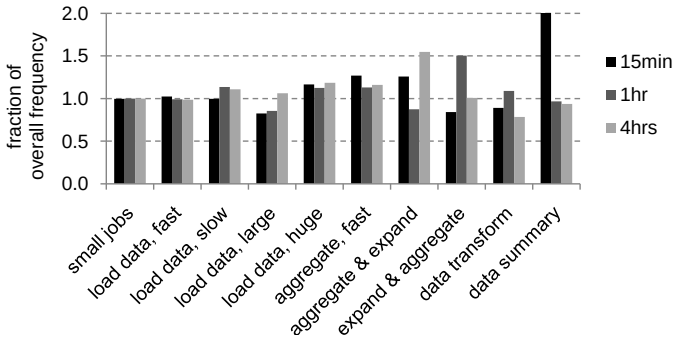

Fig. 7. Frequency of common jobs in the synthetic workload as fractions of the frequencies in the original trace. Showing that workloads synthesized using continuous samples of 15min, 1hr, and 4hrs all have similar frequencies of common jobs as the original trace.

Figure 7 shows the frequency of common jobs in the synthetic workload, expressed as fractions of the frequencies in the original trace. A representative workload would have the same frequencies of common jobs as the original trace, i.e., fractions of 1. To limit statistical variation, we compute average frequencies from 10 instances of a day-long workload.

We see that regardless of the sample window length, the frequencies are mostly around 1. A few job types have fractions deviating considerably from 1. Table I indicates that those jobs have very low frequencies. Thus, the deviations are statistical artifacts – the presence or absence of even one of those jobs can significantly affect the frequency.

Interestingly, the sample window length appears to have no impact on how much the frequencies deviate. This differs from the data characteristics and submission patterns, where the sample window length has a clear impact on how representative is the synthetic workload. Here, we can still increase workload representativeness by synthesizing longer workloads.

B. Low Workload Execution Overhead

There are two sources of potential overhead in our workload execution framework. First, concurrent reads by many jobs on the same input files could potentially affect HDFS read performance. Second, the background task to remove workload output could affect both HDFS read and write performance.

Ideally, we would quantify the overhead by running the Facebook-like workload with non-overlapping input data or no removal of workload output, and compare the performance against a setup in which we do have overlapping input and background removal of output. Doing so requires a system with up to 200TB of disk space (sum of per-day input, shuffle, output size, multiplied by 3-fold HDFS replication). Thus, we evaluate the overhead using simplified experiments.

Concurrent reads

To verify that concurrent reads of the same input files has low impact on HDFS read performance, we repeat 10 times the following experiment on a 10-machine cluster running Hadoop 0.18.2.

```
Job 1: 10 GB sort, input HDFS/directoryA
Job 2: 10 GB sort, input HDFS/directoryB
Wait for both to finish
```

TABLE III
SIMULTANEOUS HDFS READ, SHOWING LOW OVERHEAD.

Job 1	597 s ± 56 s
Job 2	588 s ± 46 s
Job 3	603 s ± 56 s
Job 4	614 s ± 50 s

TABLE IV
BACKGROUND HDFS REMOVE, ALSO SHOWING LOW OVERHEAD.

Job 1	206 s ± 14 s
Job 2	106 s ± 10 s
Job 3	236 s ± 8 s
Job 4	447 s ± 18 s
Job 5	206 s ± 11 s
Job 6	102 s ± 8 s
Job 7	218 s ± 16 s
Job 8	417 s ± 9 s

```
Job 3: 10 GB sort, input HDFS/directoryA
Job 4: 10 GB sort, input HDFS/directoryA
```

Jobs 1 and 2 give the baseline performance, while Jobs 3 and 4 identify any potential overhead. The running times are in Table III. The finishing times are completely within the confidence intervals of each other. Thus, our data input mechanism imposes no measurable overhead.

We repeat the experiment with more concurrent read jobs. In those experiments, the MapReduce task schedulers and task placement algorithms introduce large variance in job completion time, with the performance difference again falling within confidence intervals of each other. Thus, our data input mechanism has no measurable overhead at even higher concurrency levels of reads.

Background deletes

To verify that the background task to remove workload output has low impact on HDFS read and write performance, we repeat 10 times the following experiment on a 10-machine cluster running Hadoop 0.18.2.

```
Job 1: Write 10 GB to HDFS
Wait for job to finish
Job 2: Read 10 GB from HDFS
Wait for job to finish
Job 3: Shuffle 10 GB
Wait for job to finish
Job 4: Sort 10 GB
Wait for job to finish

Job 5: Write 10 GB to HDFS \
  with large HDFS -rmr in background
Wait for job to finish
Job 6: Read 10 GB from HDFS \
  with large HDFS -rmr in background
Wait for job to finish
Job 7: Shuffle 10 GB \
  with large HDFS -rmr in background
Wait for job to finish
Job 8: Sort 10 GB \
  with large HDFS -rmr in background
Wait for job to finish
```

Jobs 1-4 provide the baseline for write, read, shuffle and sort. Jobs 5-8 quantify the performance impact of background deletes. The running times are in Table IV. The finishing times are completely within the confidence intervals of each other. Again, our data removal mechanism imposes no measurable overhead. This is because recent HDFS versions implement delete by renaming the deleted file to a file in the `/trash` directory, with the space being truly reclaimed only after 6 hours [17]. Thus, even an in-thread, non-background HDFS remove would impose low overhead.

C. New Capability 1 - Identify Workload-Specific Bottlenecks

We run the day-long Facebook-like workload at scale on a 200-machine cluster on Amazon Elastic Computing Cloud (EC2) [18], running Hadoop 0.18.2 with default configurations. Each machine is a m1.large machine instance with 7.5GB memory, 4×1.0-1.2GHz equivalent CPU capacity, 400GB storage capacity, and "high" IO performance.

When we run the workload, many of the jobs failed to complete. We suspected that there is a system sizing issue because we run on a 200-machine cluster a workload that originally came from a 600-machine cluster. Thus, we decreased by a factor of 3 the size of input/shuffle/output for all jobs. Even then, 8.4% of the jobs still failed, with the failed jobs appearing in groups of similar submission times, but with the groups dispersed throughout the workload. It turns out that a subtle system sizing issue is the bottleneck.

What happens is that when there is a mix of large and small jobs, and the large jobs have reduce tasks that take a long time to complete, the small jobs complete their map tasks, with the reduce tasks remaining on queue. When this happens, the map tasks keep completing, allowing newly submitted jobs to begin. We get an increasingly long queue of jobs that completed the map phase but wait for the reduce phase. The cluster is compelled to store the shuffle data of all these active jobs, since the reduce step requires the shuffle data. It is this increasing set of active shuffle data that makes the system run out of storage space. Once that happens, jobs that attempt to write shuffle data or output data will fail.

MapReduce recovers gracefully from this failure. Once a job fails, MapReduce reclaims the space for intermediate data. Thus, when enough jobs have failed, there would be enough reclaimed disk space for MapReduce to resume executing the workload as usual, until the failure repeats. Hence the failures appear throughout the workload.

The ability to identify this bottleneck represents a new capability because the failure occurs only when the workload contains a specific sequence of large and small jobs, and specific ratios of map versus reduce times. A benchmark cannot identify this bottleneck because it does not capture the right job submission sequences. A direct trace replay does, but potentially takes longer. For example, if the pathological job submission sequence happens frequently, but only in the second half of the trace, then we need to replay the entire first half of the trace before identifying the bottleneck.

Increasing the disk size would address this failure mode, for example having a cluster with 200TB of storage (sum of input, shuffle, output sizes, multiplied by 3-fold HDFS replication). However, this would be a wasteful over-provision. The real culprit is the FIFO task scheduler, which creates a long queue of jobs that are starved of reduce slots. The Hadoop fair scheduler was designed specifically to address this issue [19]. Thus, we are not surprised that the fair schedule came out of a direct collaboration with Facebook.

As a natural follow-up, we investigate how much the fair scheduler actually benefits this workload.

D. New Capability 2 - Select Workload-Specific Schedulers

Briefly, MapReduce task schedulers work as follows. Each job breaks down into many map and reduce tasks, operating on a partition of the input, shuffle, and output data. These tasks execute in parallel on different machines. Each machine has a fixed number of task slots, by default 2 map slots and 2 reduce slots per machine. The task scheduler receives job submission requests and assigns tasks to worker machines. The FIFO scheduler assigns task slots to jobs in FIFO order, while the fair schedule seeks to give each job a concurrent fair share of the task slots. The biggest performance difference occurs when the job stream contains many small jobs following a large job. Under the FIFO scheduler, the large job takes up all the task slots, with the small jobs enqueued until the large job completes. Under the fair scheduler, the large and small jobs share the task slots equally, with the large jobs taking longer, but small jobs being able to run immediately.

We run the day-long Facebook-like workload on the cluster of 200 m1.large EC2 instances. We compare the behavior when the cluster runs Hadoop 0.18.2, which has the FIFO scheduler, with Hadoop 0.20.2, which has the fair scheduler. We observed three illustrative kinds of behavior. We analyze each, and then combine the observations to discuss why the choice of schedulers should depend on the workload.

Disk "bottleneck"

Figure 8 captures a snapshot of 100 consecutive jobs in our day-long workload of roughly 6000 jobs. The horizontal axis indicates the job indices in submission order, i.e., the first job in the workload has index 0. There are several bursts of large jobs that cause many jobs to fail for the FIFO scheduler. These failed jobs have no completion time, leaving a missing marker in the graph. We know there are bursts of large jobs because the jobs take longer to complete under the fair scheduler. We see two such bursts - Jobs 4570-4580, 4610-4650. This is the failure mode we discussed in Section V-C. The fair scheduler is clearly superior, due to the higher job completion rate.

Small jobs after large jobs, no failures

This is the precise job arrival sequence for which the fair scheduler was designed. Figure 9 captures another 100 consecutive jobs in the day-long workload. Here, when both the FIFO and fair schedulers exhibit no job failures, the fair scheduler is still far superior. Several very large jobs arrive in succession (high completion times around Job 4820 and just beyond Job 4845). Each arrival brings a large jump in the FIFO scheduler completion time of subsequent jobs. This is again due to FIFO head-of-queue blocking. Once the large job completes, all subsequent small jobs complete in rapid succession, leading to the horizontal row of markers. The fair scheduler, in contrast, shows small jobs with unaffected running times, sometimes orders of magnitude faster than their FIFO counterpart. Such improvements agree with the best-case improvement reported in the original fair scheduler study [19], but far higher than the average improvement reported there.

Long sequence of small jobs

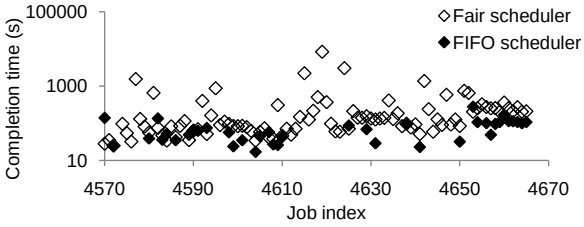

Fig. 8. A snapshot of 100 jobs in a day-long Facebook-like workload, showing job failures in FIFO scheduler (missing markers, i.e., jobs without a completion time).

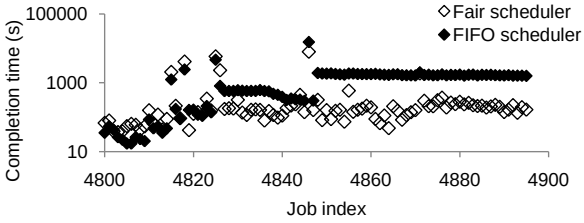

Fig. 9. Job submit pattern of small jobs after large jobs from a snapshot of 100 jobs in a day-long Facebook-like workload. The fair scheduler gives lower completion times and is also superior.

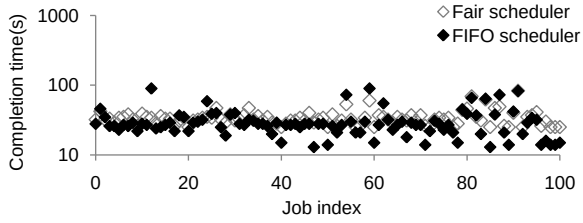

Fig. 10. Long sequence of small jobs from a snapshot of 100 jobs in a day-long Facebook-like workload. The FIFO scheduler gives lower completion times and is superior.

Figure 10 captures 100 consecutive jobs that are all small jobs with fast running times. For this job submission pattern, Hadoop 0.20.2 is slower than Hadoop 0.18.2, unsurprising given the many added features since 0.18.2. The fair scheduler brings little benefit. Small jobs dominate this workload (Table I). The vast improvements for small jobs after large jobs would be amortized across performance penalties for long sequences of small jobs.

Workload-specific choice of schedulers

Our experiments show that the choice of schedulers depends on both the performance metric and the workload. The fair scheduler would be a clear winner if the metric is the worst-case job running time or the variance in job running time. However, if average job running time is the metric, then the FIFO scheduler would be preferred if long sequences of small jobs dominate the workload. Thus, even though cluster users benefit from the fairness guarantees of the fair scheduler, cluster operators may find that fairness guarantees are rarely needed, and adopt the FIFO scheduler instead.

The ability to make workload-specific choice of schedulers represents a new capability because scheduler performance depends on the frequencies of various job submission patterns. The right choice of scheduler for one workload would not imply the right choice for another. The original fair scheduler study [19] used a synthetic workload with frequent interleaving

of large and small jobs, leading to the conclusion that the fair scheduler should be unequivocally preferred. Here, we execute a workload with a more representative interleaving between large and small jobs. This leads us to a more nuanced, workload-specific choice of MapReduce task schedulers.

VI. Towards MapReduce Workload Suites

We must go beyond using one-size-fits-all benchmarks for MapReduce performance evaluations. Our comparison of two production traces shows great differences between MapReduce use cases. No single benchmark can capture such diverse behavior. We advocate for performance evaluations using representative workloads and presented a framework to generate and execute such workloads. We demonstrated that running realistic workloads gives cluster operators new ways to identify system bottlenecks and evaluate system configurations.

We believe that having representative workloads can assist many recent MapReduce studies. For example, studies on how to predict MapReduce job running times [20], [21] can evaluate their mechanisms on realistic job mixes. Studies on MapReduce energy efficiency [22], [23] can quantify energy savings under realistic workload fluctuations. Various efforts to develop effective MapReduce workload management schemes [24], [7] can generalize their findings across a different realistic workloads. In short, having realistic workloads allow MapReduce researchers better understand the strengths and limitations of the proposed optimizations.

Our work complements efforts to develop MapReduce simulators [25], [26]. Having realistic workloads allows cluster operators to run simulations with realistic inputs, amplifying the benefit of MapReduce simulators. Our approach differs from that taken by benchmarks that focuses on IO byte stream properties [27]. We focus on MapReduce-level semantics, such as input/shuffle/output data sizes, because doing so would produce more immediate MapReduce design insights.

Many open problems remain for future work. One simplification we made is that it is sufficient to replicate only the data characteristics when we execute the workload. This simplification is acceptable here because we know through direct conversations that the Facebook production cluster runs many Extract-Transform-Load (ETL) jobs, whose behavior is dominated by data movements. Future work should go beyond this simplification. Ideally, we would construct common jobs with more complete semantics than just the data ratios.

Looking forward, we expect that we would eventually understand the full taxonomy of MapReduce use cases. At that point, we can move beyond the highly-targeted, one-per-use-case workload suites that we proposed here. The next step in performance evaluations would move towards standardized workload suites, which serves as a richer kind of "benchmark".

The first step towards that goal is to understand more MapReduce workloads. To that end, we invite all MapReduce cluster operators to publish their production workloads. We have received clearance to release the synthetic Facebook-like workloads. We expect to complete the necessary user documentation within two weeks of submitting this paper, and we will link to the workloads and execution tools from the first author's website. At the time of writing, we have already received requests from several research groups and commercial companies to use our synthesis and execution tools. We hope our workload description vocabulary can provide an initial format for releasing traces. We also hope that our workload synthesis tools can assure MapReduce operators that they can release representative workloads without compromising confidential information about their production clusters.

References

[1] Hadoop Wiki, "Hadoop Power-By Page," http://wiki.apache.org/hadoop/PoweredBy.

[2] "Gridmix," HADOOP-HOME/mapred/src/benchmarks/gridmix in Hadoop 0.21.0 onwards.

[3] C. Douglas and H. Tang, "Gridmix3 Emulating Production Workload for Apache Hadoop," http://developer.yahoo.com/blogs/hadoop/posts/2010/04/gridmix3_emulating_production/.

[4] Pig Wiki, "Pig Mix benchmark," http://wiki.apache.org/pig/PigMix.

[5] Y. Jia and Z. Shao, "A Benchmark for Hive, PIG and Hadoop," https://issues.apache.org/jira/browse/hive-396.

[6] "Amazon Elastic MapReduce," http://aws.amazon.com/elasticmapreduce/.

[7] 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 *NSDI 2011*.

[8] S. Agrawal, "The Next Generation of Hadoop Map-Reduce," Apache Hadoop Summit India 2011.

[9] I. Stoica, "A Berkeley View of Big Data: Algorithms, Machines and People," UC Berkeley EECS Annual Research Symposium, 2011.

[10] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, January 2008.

[11] E. Alpaydin, *Introduction to Machine Learning*. Cambridge, Massachusetts: MIT Press, 2004.

[12] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *SIGMOD 2009*.

[13] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *ICDEW 2010*.

[14] "Gridmix3," HADOOP-HOME/mapred/src/contrib/gridmix in Hadoop 0.21.0 onwards.

[15] "Rumen: a tool to extract job characterization data from job tracker logs," https://issues.apache.org/jira/browse/MAPREDUCE-751.

[16] L. Wasserman, *All of Statistics*. New York, New York: Springer, 2004.

[17] "HDFS Architecture Guide," http://hadoop.apache.org/hdfs/docs/current/hdfs_design.html.

[18] Amazon Web Services, "Amazon Elastic Computing Cloud," http://aws.amazon.com/ec2/.

[19] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys 2010*.

[20] K. Morton, M. Balazinska, and D. Grossman, "Paratimer: a progress indicator for mapreduce dags," in *SIGMOD 2010*.

[21] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, "Statistics-driven workload modeling for the cloud," in *SMDB 2010*.

[22] J. Leverich and C. Kozyrakis, "On the Energy (In)efficiency of Hadoop Clusters," in *HotPower 2009*.

[23] W. Lang and J. Patel, "Energy management for mapreduce clusters," in *VLDB 2010*.

[24] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the Outliers in Map-Reduce Clusters using Mantri," in *OSDI 2010*.

[25] "Mumak: Map-Reduce Simulator," https://issues.apache.org/jira/browse/MAPREDUCE-728.

[26] G. Wang, A. Butt, P. Pandey, and K. Gupta, "A simulation approach to evaluating design decisions in MapReduce setups," in *MASCOTS 2009*.

[27] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC 2010*.