In-Production Benchmarks for Distributed Large-scale Data Processing

Sunghwan Ihm, Ken Goldman, and Jerry Zhao
Google Inc., Mountain View, California

1. Introduction

Rapid progress in databases, distributed file systems, and computing paradigms [1, 2, 3], as well as the sheer volume of data available, has made "Big Data" processing possible and more accessible than a decade ago. For example, at Google, MapReduce [2] processes petabyte-scale data-sets regularly. MapReduce users and systems developers face two common challenges: (1) optimally configuring a MapReduce application to minimize performance and/or cost, and (2) understanding a job's performance deeply enough to make a correct diagnosis when it degrades.

Benchmarks, such as GraySort [4], can help address these challenges. For example, the PBSort and 10PBSort [5] experiments at Google helped reveal issues in throughput, scalability, and resource utilization across all layers and components (e.g., network platform, GFS [1], and task scheduling), and led to guidelines for users configuring jobs appropriately. However, such large-scale tests cannot run regularly due to the high cost of thousands of machine hours. In addition, there are several unique challenges for benchmarking large-scale data processing softwares, like MapReduce, in modern data centers [6]:

- **Dynamic execution environment**: Modern data centers share resources among multiple jobs. Resource isolation or prioritization can provide certain resource guarantee, but these are weakened by job overpacking which intends to increase utilization and reduce cost [6]. Typically, large-scale data processing jobs run at lower priorities than serving jobs with tighter latency requirements. Consequently, a manually tuned configuration may be optimal only under test conditions. Figure 1 demonstrates the impact of background CPU load on a sort benchmark.[1]
- **Diversified performance requirements and characteristics**: MapReduce is used at Google to serve interactive SQL queries, render streetview tiles, and build search indices [2, 7]. Each use case has unique latency or throughput requirements, and poses different I/O and computation workloads. This means default configuration settings cannot be "one size fits all." It is important to understand the sensitivity and difference of their performance in the presence of the dynamics in available resources.
- **Evolving computation infrastructure**: Data processing tools are built upon layers of system components (e.g., distributed file systems, operating systems, and hardware). Upgrades over time at each layer can dramatically change performance characteristics. Any manual performance tuning must be repeated to keep pace with these changes. Figure 2 shows these different performance characteristics across different generations of hardware/software.[2]
- **Impact from fault tolerance**: Distributed large-scale data processing systems often have built-in mechanisms to cope with task failures or lack of resources. For example, when MapReduce detects a slow task, it schedules a backup to minimize impact on total runtime. However, fault-tolerance can mask the root causes of low performance or resource waste.

[1] Figure 1 presents performance of an external sort benchmark from 6 different clusters, 15 trials each.

[2] Figure 2 depicts read/write performance over GFS from 6 different clusters, 15 trials each.

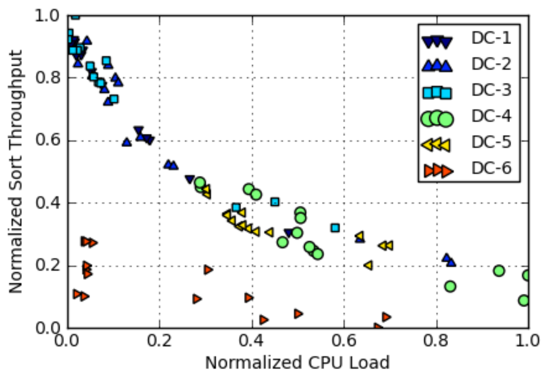

Figure 1. Background CPU Load vs. Performance

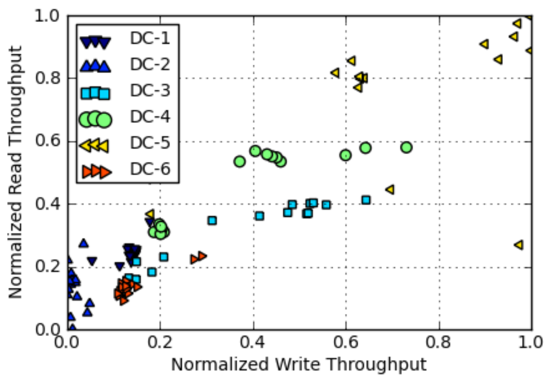

Figure 2. Performance Variance Across Data Centers

2. Benchmarking Large-scale Data Processing in Production

This paper proposes some guidelines for designing and executing benchmarks *in a production environment* for tuning and diagnosing performance of large-scale data processing applications and systems.

Realistic: A good in-production benchmark shares the same principles as any good benchmark. It should incur a deterministic, yet sufficiently representative, load that captures the nature of real applications. It often highlights certain aspects of the applications' performance characteristics. For example, a Mapper with synthetic inputs.

Specific and detailed: An in-production benchmark must capture not only overall performance metrics (e.g., total runtime), but also detailed application-specific metrics (e.g., the time spent in each read/sort/write stage). For a good understanding of impact from dynamics in the environment, the operating system must support collecting detailed local execution environment profiles (e.g., background CPU loads, disk or network I/O loads).

Configurable: In-production benchmarks should also be highly configurable to explore options offered by the specific platforms. For example, in sorting, in addition to sort-specific settings such as internal buffer sizes or number of threads, the benchmark could also be configured with different data encodings supported by the underlying file systems, or with different task priorities.

Monitored over time: To capture long term evolution of underlying platform over time, benchmarks must be executed regularly, or even continuously to explore different combinations in configuration parameter space. Continuous benchmark results provide guidelines for resource budgeting, and initial setup recommendations responsive to (possibly unforeseen) systems changes. Additionally comparison to historical execution results of the same configuration can also reveal the detailed impact from such changes.

Embedded for probabilistic execution: Continuously running a benchmark can be expensive. To address that, the benchmark can be "embedded" in production jobs: The implementation of a benchmark is linked in, or packaged together with production binaries. It can be triggered probabilistically to gather data over time, to reveal correlation to specific users or application. Also since the embedded benchmark is consistent across all users and applications, the results can be aggregated together to broadly improve job configuration.

Targeted: Benchmark execution should also be triggered when it is likely to provide insights into the overall platform stack. Additional information from large-scale data processing systems can provide hints to achieve such goals. For example, a MapReduce scheduler can detect slow tasks based on different algorithms and heuristics. Benchmarking tasks can be started immediately, or scheduled for later execution on the same machine as well. The dataset collected can help identify potential issues hidden by the built-in fault tolerance mechanisms.

3. Related and Future Work

Configuring individual Hadoop/MapReduce instances is a well-known challenge. Several recent proposals focus on configuring jobs based on historical runs of the real applications [10]. The approach described in this paper distinguishes itself from them, in that consistent benchmarks are used. The dataset collected can be broadly reused for broader analysis to resolve issues across multiple users and applications, and even across different data processing frameworks [11, 12]. There are also proposals that consider system heterogeneity when scheduling jobs in cloud [9]. This paper emphasizes that the underlying platform could and should be more transparent to provide richer context to understand and improve specific performance behavior.

Others have conducted research on systems for collecting traces and statistics from production environments, and correlating information across multiple layers to diagnose subtle performance bugs [6, 8]. Such prior work is orthogonal to the approach described here. Similar techniques can be applied for manual or automatic analysis of the benchmark data.

Beyond what is sketched in this paper, some topics merit further exploration: For example, is it practical to automatically generate in-production benchmarks to cover a class of data processing applications, based on microbenchmark tests used during application development? Can we extend the framework described to collect and analyze data continuously in order to automatically reconfigure production jobs? What measurements in the production operating system "stack" are most critical for performance tuning, and how can these be exposed effectively and efficiently?

References

1. S. Ghemawat, H. Gobioff, and S-T. Leung, "The Google file system", SOSP'03
2. J. Dean, S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters", OSDI'04
3. Hadoop. http://hadoop.apache.org/
4. GraySort. http://sortbenchmark.org/
5. http://googleresearch.blogspot.com/2011/09/sorting-petabytes-with-mapreduce-next.html
6. L. A. Barroso and U. Hölzle. "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines", Synthesis Lectures on Computer Architecture, 2009
7. B. Chattopadhyay, et al. "Tenzing: A SQL Implementation on the MapReduce Framework", PVLDB'11.
8. E. Pinheiro, et al. "Failure trends in a large disk drive population", FAST'07
9. G. Lee, et al. "Heterogeneity-Aware Resource Allocation and Scheduling in the Cloud", HotCloud'11
10. K. Kambatla, et al. "Towards Optimizing Hadoop Provisioning in the Cloud", HotCloud'09
11. G. Malewicz, et al. "Pregel: a system for large-scale graph processing", SIGMOD'10
12. S. Melnik, et al. "Dremel: Interactive Analysis of Web-Scale Datasets", PVLDB'10