**Title**
Workload-Driven Design and Evaluation of Large-Scale Data-Centric Systems



**Author**
Chen, Yanpei

**Publication Date**
2012

Peer reviewed|Thesis/dissertation

# Workload-Driven Design and Evaluation of Large-Scale Data-Centric Systems

by

Yanpei Chen

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Engineering — Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Randy H. Katz, Chair
Professor Vern Paxson
Professor Ray R. Larson

Spring 2012

**Abstract**

# Workload-Driven Design and Evaluation of Large-Scale Data-Centric Systems

Yanpei Chen

Doctor of Philosophy in Engineering — Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Randy H. Katz, Chair

Large-scale data-centric systems help organizations store, manipulate, and derive value from large volumes of data. They consist of distributed components spread across a scalable number of connected machines and involve complex software/hardware stacks with multiple semantic layers. These systems help organizations solve established problems involving large amounts of data, while catalyzing new, data-driven businesses such as search engines, social networks, and cloud computing and data storage service providers. The complexity, diversity, scale, and rapid evolution of large-scale data-centric systems make it challenging to develop intuition about these systems, gain operational experience, and improve performance. It is an important research problem to develop a method to design and evaluate such systems based on the empirical behavior of the targeted workloads. Using an unprecedented collection of nine industrial workload traces of business-critical large-scale data-centric systems, we develop a workload-driven design and evaluation method for these systems and apply the method to address previously unsolved design problems. Specifically, the dissertation contributes the following:

1. A conceptual framework of breaking down workloads for large-scale data-centric systems into data access patterns, computation patterns, and load arrival patterns.
2. A workload analysis and synthesis method that uses multi-dimensional, non-parametric statistics to extract insights and produce representative behavior.
3. Case studies of workload analysis for industrial deployments of MapReduce and enterprise network storage systems, two examples of large-scale data-centric systems.
4. Case studies of workload-driven design and evaluation of an energy-efficient MapReduce system and Internet datacenter network transport protocol pathologies, two research topics that require workload-specific insights to address.

Overall, the dissertation develops a more objective and systematic understanding of an emerging and important class of computer systems. The work in this dissertation helps further accelerate the adoption of large-scale data-centric systems to solve real life problems relevant to business, science, and day-to-day consumers.

# Acknowledgements

I would like to sincerely thank the following people, without whose kindness and patience this thesis would not have been possible:

- My research adviser: Professor Randy H. Katz.
- Members of the thesis committee: Professors Vern Paxson and Ray R. Larson.
- My co-authors:

  Sara Alspaugh, Archana Ganapathi, Rean Griffith, Kiran Srinivasan, Garth Goodson, Dhruba Borthakur, Laura Keys, Tracy Wang, David Zats.

- Various faculty at UC Berkeley:

  Anthony D. Joseph, Joseph M. Hellerstein, Ion Stoica, Michael Franklin, David Patterson, Armando Fox, Michael Jordan, David Culler, Seth Sanders, Ken Lutz, Babak Ayazifar, Dan Garcia, Jihong Sanderson, Brian Carver, Andrew Issacs, Cari Kaufman, Jean Walrand, Laurent El Ghaoui, David Wagner.

- Industry sponsors, collaborators, alumni, and other friends of the lab:

  Mike Olson, Amr Awadallah, Jolly Chen, Todd Lipcon, Aaron T. Myers, Eric Sammer, Kaladhar Voruganti, Lakshmi Bairavasundaram, Minlong Shao, Shankar Pasupathy, Val Bercovici, Kim Keaton, Greg Ganger, Shivnath Babu, Garth Gibson, George Porter, Rodrigo Fonseca, Joseph L. Hellerstein, John Wilkes, Srikanth Kandula, Feng Zhao, Owen O'Malley, Jeff Mogul.

- Postdocs and fellow students of the Algorithm, Machines, and People Laboratory and LoCal Project, especially

  Matei Zaharia, Ganesh Anathanarayanan, Ali Ghosi, Andrew Krioukov, Andy Konwinski, Ariel Rabkin, Gunho Lee, Stephen Dawson Haggerty, Charles Reiss, Kay Ousterhout, Justin Ma, Tim Kraska, Fabian Wauthier, Kristal Curtis, Michael Ambrust, Beth Trushkowsky, Timothy Hunter, Ariel Kleiner, Patrick Wendell, Sameer Agarwal, Aurojit Panda, Andrew Wang, Shivaram Venkataraman.

- Various support staff:

  Jon Kuroda, Kattt Atchley, Sean McMahon, Christina Allen, Keith Sklower.

- Friends and family from outside of work - thanks for making life even more fun.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship. This research is supported in part by gifts from Google, SAP, Amazon Web Services, Blue Goji, Cloudera, Ericsson, General Electric, Hewlett Packard, Huawei, IBM, Intel, MarkLogic, Microsoft, NEC Labs, NetApp, Oracle, Quanta, Splunk, VMware and by DARPA (contract #FA8650-11-C-7136). Research supported in part by the DARPA- and SRC-funded MuSyC FCRP Multiscale Systems Center.

Dedicated to

My parents, who guided my baby steps in science,

My teachers, who inspire me by their personal examples, and

My fellow students — may we remain forever friends and forever young.

# Contents

# Chapter 1

# Introduction

*The journey of a thousand miles begins with a single step.*

*— Lao Zi, Dao De Jing.*

**Large-scale data-centric systems** first emerged in large technology companies to helped solve "needle-in-a-haystack" type data problems. Such problems involve organizing diverse, high volume, and potentially disorganized data sources to distill timely and actionable insights. These systems have enabled new solutions to established data problems, while catalyzing new, data-driven businesses such as search engines, social networks, and cloud computing and data storage service providers.

Today, large-scale data-centric systems also contribute to front-line, business critical decisions in diverse industries beyond technology, including finance, manufacturing, retail, media, and others. They perform data indexing and retrieval, pattern matching and anormaly detection, natural language processing for topic identification and sentiment analysis, information extraction from image, audio, or video data, and a variety of other tasks involving "big data".

Characteristics of large-scale data-centric systems include:

1. The ability to store, manipulate, and derive value from large volumes of data.
2. Distributed components spread across a scalable number of connected machines.
3. Complex software/hardware stacks with multiple semantic layers.
4. Diverse use cases and many opportunities for optimization and re-design.

Several technology trends drive the growth of such systems. They include:

1. Increasingly easy and economical access to large scale storage and computation infrastructure [5, 23].

2. Ubiquitous ability to generate, collect, and archive data about both technology systems and the physical world [56].

3. Growing desire and statistical literacy across many industries to understand and derive value from large datasets [7, 37, 91, 76].

It is important to enhance the capability of such systems. Doing so allows society at large to maximize the benefit from new, quantitative knowledge created by understanding large volumes of business, scientific, and consumer data.

One entry point to design and evaluate computer systems in general is to understand their behavior, i.e., their **workloads**. For large-scale data-centric systems, one approach breaks down their workloads into three conceptual components:

1. Data access patterns, i.e., properties of the data.

2. Computation patterns, i.e., manipulations done on the data.

3. Load arrival patterns, i.e., the time-varying combination of the requested data accesses and computation.

Historically, **workload-driven design and evaluation** has complemented intuition-driven design and evaluation. We believe that key properties of large-scale data-centric systems compels the workload-driven approach to be the primary if not the only feasible design and evaluation method. These properties are empirically verified later in the dissertation. They include

1. System complexity.
   It is hard to develop intuitions about multi-layered and distributed large-scale data-centric systems. A purely subjective approach risks incorrect assumptions and mental models. Any conceptual simplifications of these complex systems requires objective, empirical verification. Design and evaluation intuitions thus becomes the outcome of a workload-driven empirical approach.

2. Rapid system evolution.
   Large-scale data-centric systems constantly and rapidly evolve. Such ongoing improvements result from the rapid evolution in the data stored on such systems, which in turn reflects the rapid innovations in business, science, and society at large facilitated by knowledge extracted from large scale data. Consequently, the change in the underlying systems likely outpaces the ability to empirically verify system models and design and evaluation guidelines. In other words, operating "experience" becomes outdated at a pace faster than such experience can be validated. This necessitates design and evaluation methods capable of consuming constantly and rapidly emerging workload insights.

3. System diversity.
   Uses cases for large-scale data-centric systems reflect the diversity of the data they

store and manipulate. Such diversity translates to significant, and sometimes mutually exclusive, variations in workload behavior. The design and evaluation insights developed for one use case may be suboptimal, or even harmful for another use case. Conversely, some previously marginal design options may become top priority subject to emerging understandings of workloads. Existing approaches of "designing for the common case" become problematic, simply because there are many common cases. Thus, it becomes essential to develop methods capable of generating targeted insights for specific workloads.

4. System scale.
   The sheer size of large-scale data-centric systems means that an accurate trace of workload behavior requires large volumes of high-dimensional data. This translates to the need for both dedicated tools to log and manage the data, and analysis methods capable of distilling actionable insights from a seeming over-abundance of data. Fortunately, the rise of large-scale data-centric systems is symbiotic with increased ability to analyze data about such systems, with innovations in one often inspiring mirror innovations in the other. In other words, emerging data science tools that made large-scale data-centric systems useful can also help us understand data about these systems.

## 1.1 Problem Statement

This dissertation seeks to develop a new workload-driven design and evaluation methodology to enable effective design of large-scale data-centric systems subject to increased system complexity, diversity, scale, and speed of system evolution.

The success of this methodology is to give the following new capabilities to future operators of large-scale data-centric systems:

1. The ability to discover workload behavior that are previously unknown.
2. The ability to measure system performance in a previously unavailable, per-workload fashion.
3. The ability to identify design opportunities with a high level of confidence that new features optimize for common behavior in the targeted workload.

## 1.2 Solution Summary

We take advantage of nine **production workload traces** of MapReduce and enterprise network storage, two important examples of large-scale data-centric systems. These traces represent a rare resource, and offer unprecedented quantitative insights into a range of real-life behavior of large-scale data-centric systems. The insights thus motivate

| Trace | System | Length | Date | Trace size[1] |
|---|---|---|---|---|
| NetApp-corporate | Enterprise network storage | 2 months | 2007 | 509,076 |
| NetApp-engineering | Enterprise network storage | 3 months | 2007 | 232,033 |
| Facebook-2009 | MapReduce | 6 months | 2009 | 1,129,193 |
| Facebook-2010 | MapReduce | 1.5 months | 2010 | 1,169,184 |
| Cloudera Customer A | MapReduce | 1 month | 2011 | 5,759 |
| Cloudera Customer B | MapReduce | 9 days | 2011 | 22,974 |
| Cloudera Customer C | MapReduce | 1 month | 2011 | 21,030 |
| Cloudera Customer D | MapReduce | 2+ months | 2011 | 13,283 |
| Cloudera Customer E | MapReduce | 9 days | 2011 | 10,790 |

Table 1.1. Summary of production workload traces used in the dissertation. Notes: [1]Trace size shows the number of user sessions for enterprise network storage, and the number of jobs for MapReduce.

the need for a new, workload-driven design and evaluation methodology. The traces come from front-line systems of up to thousands of machines servicing critical business needs. Combined, the traces cover use cases across multiple industries, include aggregated duration of longer than a year, and cover aggregated data movement of close to two exabytes. Table 1.1 summarizes the traces.

At the time of this dissertation, such collection of large-scale production workload traces is rare. However, we believe the technology trends responsible for the rise of large-scale data-centric systems will also lead to a proliferation of detailed traces about such systems. We expect the methdological insights from this dissertation to translate beyond the workloads analyzed here. Each operator of a large-scale data-centric system is rightly interested in focusing on optimizing for their own workloads only. It remains the responsibility of researchers to put together trace collections covering multiple use cases, and distill insights from both similarities and differences between workloads.

The dissertation contributes the following components of a workload-driven design and evaluation methodology for large-scale data-centric systems:

1. A **conceptual framework** of breaking down workloads for large-scale data-centric systems into data access patterns, computation patterns, and load arrival patterns. This framework helps us systematically understand workload behavior, while allowing the mapping to specific system design and behavior to evolve.

2. A **workload analysis and synthesis method** that uses multi-dimensional, non-parametric statistics to extract insights and produce representative behavior despite system complexity and scale. Such a method also enables workload-specific design and evaluation over diverse use cases.

3. Two case studies of **workload analysis** for MapReduce and enterprise network storage systems. A list of design insights results from both studies. The empirical evidence from these case studies also verify the previously discussed properties with regard to system complexity, rapid evolution, diversity, and scale.

4. Two further case studies of **workload-driven design and evaluation** of energy-

efficient MapReduce systems and Internet datacenter network transport protocol pathologies. Both studies depend on the workload insights developed in the first half of the dissertation.

These contributions form the rationale for the structure of the dissertation.

Overall, the dissertation helps us develp a more objective and systematic understanding of large-scale data-centric systems. Ultimately, we hope the techniques developed here will contribute to symbiotic efforts to develop objective and systematic understanding of the large scale data stored and manipulated by such systems.

## 1.3 Structure of the Dissertation

### 1.3.1 Background and methodology

This dissertation begins with a review of relevant prior studies (Chapter 2). These include (1) workload-driven historical studies on network systems and file systems, two key components of large-scale data-centric systems; (2) emerging research on Internet data centers, the environment in which large-scale data-centric systems operate; and (3) nascent efforts to develop large-scale data-centric performance benchmarks. Historical and emerging work both reveal the desire to understand and improve systems based on empirical evidence, and highlight technology changes that warrant innovations in analysis, design, and evaluation methodology.

The next chapter discusses these methodology innovations (Chapter 3). We break down large-scale data-centric workloads into three conceptual components — data access patterns, computation patterns, and load arrival patterns. We also outline techniques we developed for analyzing such workloads, synthesizing representative behavior, and evaluating multi-dimensional system performance using workload insights. The techniques involve multi-dimensional, non-parametric statistics, and build on previous modeling and analysis approaches. The rest of the dissertation illustrate this methodology.

### 1.3.2 Workload analysis

The following two chapters apply the workload analysis techniques to two types of large-scale data-centric systems.

We first analyze on production workloads from seven instances of MapReduce systems (Chapter 4). The MapReduce programming paradigm has a relatively simple semantic structure of map and reduce computations. Thus, workloads from these systems form a good initial illustration of our workload analysis methods.

We then present a companion analysis of two enterprise network storage workloads (Chapter 5). Enterprise network storage systems have a more complex semantic structure, with storage servers being distinct from storage clients, and each breaking down

further into multiple semantic layers. Such complexity requires an extension of the approach we used for analyzing MapReduce workloads.

For both MapReduce and enterprise network storage systems, an in-depth workload analysis reveals many design insights that validate existing efforts, suggest new opportunities, or necessitate changes in design priorities.

### 1.3.3 Workload-driven design and evaluation

The derived workload insights then translate to the actual design and evaluation of large-scale data-centric systems, as we illustrate over the next two chapters.

We improve the energy efficiency for clusters servicing an emerging class of MapReduce with Interactive Analysis (MIA) workloads (Chapter 6). Energy efficiency mechanisms heavily depend on the particular workload involved. The empirical analysis of MIA workloads motivated the design of the BEEMR (Berkeley Energy Efficient MapReduce) workload manager, which represents a completely different design approach than that for improving energy efficiency for web search centric MapReduce workloads. We also discover that evaluating energy efficiency improvements requires large scale simulations covering long-term historical behavior, further amplifying the need for advances in workload analysis and replay.

We also investigate the TCP incast problem in the context of MapReduce workloads (Chapter 7). TCP incast is a network transport protocol pathology that affects N-to-1 data transfer patterns in Internet data centers. Recent years have seen several attempts to mitigate or avoid incast, while some doubts remained with regard to how much it impacts large-scale data-centric systems such as MapReduce. We contribute workload-driven evaluation results that quantify the performance impact of TCP incast under realistic settings. The workload insights developed earlier in the dissertation further allow us to place TCP incast in the context of improvements to system components that do not involve the network, such as schedulers, or configuration tuning tools.

These two chapters demonstrate that workload-driven design and evaluation of large-scale data-centric systems leads to new opportunities (BEEMR) and informs existing engineering efforts (TCP incast). Conversely, the lack of such methods risks muddled design priorities, as well as mis-engineering for the wrong workload.

### 1.3.4 Broader implications and future work

We close the dissertation by outlining future work and reflecting on the broader implications of the dissertation (Chapter 8). Further research opportunities include increasing the quality of analysis by tracing additional workloads, implementing the various design insights identified, extending the methodology to other large-scale data-centric systems, and applying data-driven design and evaluation to topics beyond computer systems.

# Chapter 2

# Background and Prior Work

*Study the old to divine the new. — Confucius, The Analects.*

The desire for thorough system measurement predates the rise of large-scale data-centric systems. Workload characterization has been invaluable in helping system designers identify problems, analyze causes, and evaluate solutions.

This section gives a quick overview of MapReduce and enterprise network storage systems (Section 2.1), two types of large-scale data-centric systems studied in this dissertation. We then review key previous studies in which workload characterization played a significant part. In particular, we identify three trends in the evolution of existing work:

1. Increasing deployment of large-scale data-centric systems necessitates and enables studies with greater generality than recent single use case analysis (Section 2.2).
2. Emerging multi-dimensional statistical techniques invite more in-depth analysis compared with studies that analyze one workload feature at a time (Section 2.3).
3. Growing diversity and complexity of use cases require a move beyond nascent benchmarking efforts that oversimplify and misrepresent the workload (Section 2.4).

These trends both motivate and facilitate the advances proposed in the dissertation.

## 2.1 Examples of Large-Scale Data-Centric Systems

Characteristics of large-scale data-centric systems include

1. The ability to store, manipulate, and derive value from large volumes of data.
2. Distributed components spread across a scalable number of connected machines.

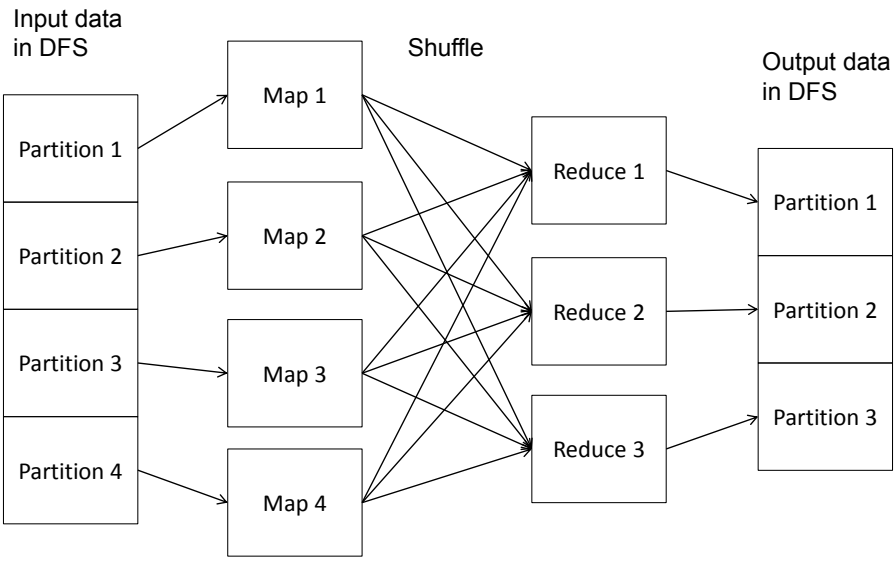



Figure 2.1. Stages in a MapReduce computation. From left to right: Read input data from the DFS. Apply the map function. Shuffle the intermediate data to the reduce tasks. Apply the reduce function. Write the output data to DFS.

3. Complex software/hardware stacks with multiple semantic layers.

4. Diverse use cases and many opportunities for optimization and re-design.

Many types of computer systems potentially qualify for the loosely defined label of "large-scale data-centric systems". This dissertation focuses on two types – MapReduce and enterprise network storage. MapReduce represents a relatively new system that helped spread the know-how to process large scale data beyond technology companies. Enterprise network storage exemplifies a more established type of computer system that has grown to become a kind of large-scale data-centric system due to the changing scale and nature of the data it stores. Hence, the combination of MapReduce and enterprise network storage allows this dissertation to develop design and evaluation methodologies applicable for both emerging and established systems.

### 2.1.1 Background – MapReduce

MapReduce was initially developed at Google for parallel processing of large datasets [53]. Today, MapReduce supports the data indexing needs of Google's flagship web search service, as well as business critical processes at Yahoo!, Facebook, and other companies [6, 7]. Programs written using MapReduce are automatically executed in a parallel fashion on the cluster. Also, MapReduce can run on clusters of cheap commodity machines, an attractive alternative to expensive, specialized clusters. MapReduce is scalable, allowing petabytes of data to be processed on thousands of machines.

At its core, MapReduce has two user-defined functions. The *Map* function takes in a key-value pair, and generates a set of intermediate key-value pairs. The *Reduce* function takes in all intermediate pairs associated with a particular key, and emits a final set of key-value pairs. To limit network traffic, users may additionally specify a Combine function that merges multiple intermediate key-value pairs before sending them to the Reduce worker. Both the input pairs to Map and the output pairs of Reduce are placed in an underlying distributed file system (DFS). The run-time system takes care of retrieving from and outputting to the DFS, partitioning the data, scheduling parallel execution, coordinating network communication, and handling machine failures.

An actual MapReduce computation proceeds as follows. There is a master daemon that coordinates a cluster of workers. The *input* data resides in the DFS. The master divides the input file into many splits, i.e., sub-files, each read and processed by a Map worker. The intermediate key-value pairs are periodically written to the local disk at the Map workers, usually separate machines from the master, and the locations of the pairs are sent to the master. The master forwards these locations to the Reduce workers, who read the intermediate pairs from Map workers using remote procedure call (RPC). This N-to-N movement of intermediate data is called *shuffle*. After a Reduce worker has read all the intermediate pairs, it sorts the data by the intermediate key, applies the Reduce function, and appends the *output* key-value pairs to a final DFS file for the Reduce partition. If any of the Map or Reduce executions lags behind, backup executions are launched, and the execution is considered complete when any of the original or backup workers finishes. An entire MapReduce computation is called a *job*, and the execution of a Map or Reduce function on a worker is called a *task*. Each worker node allocates resources in the form of *slots* and each Map task or Reduce task uses one slot.

Figure 2.1 illustrates the several stages of a MapReduce computation. The conceptual simplicity of the map and reduce data flow invites a mental model of MapReduce workloads built around the data movement and computation semantics of the map and reduce components.

The original MapReduce paper [53] contains several examples of MapReduce programs. One example is count of URL access frequency, essential to computing online advertising click rates: The map function outputs $[URL, 1]$, and the reduce function adds together all values for the same URL and emits $[URL, totalcount]$. Another example is computing the reverse web-link graph, a key component of building web search indices: The map function outputs $[target, source]$ for each link in a webpage, and the reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair $[target, list(source)]$. As the rising popularing of MapReduce demonstrates, the map and reduce programming paradigm can implement a wide range of computations.

For this dissertation, we select the Apache Hadoop implementation of MapReduce [5]. In Hadoop, the user-supplied Map and Reduce functions can be written in many languages. The Hadoop distributed file system (HDFS) implements many features of the Google DFS [64]. The open source nature of Hadoop has made it the frequent target for optimizations, e.g., an improved way to launch backup tasks [142], a fair scheduler for

multi-user environments [141], pipeline task execution and streaming queries [50], and resource managers from multiple computational frameworks that include MapReduce [72].

### 2.1.2 Background – Enterprise Network Storage Systems

Enterprise network storage systems evolved from traditional file systems. The "network" aspect became necessary when single node file systems could no longer economically provide the scale required to store large data sets. The "enterprise" adjective came from the desire to add features to network storage systems that would make them suitable for business-critical use cases. Such features include fault tolerance, data snapshot and archive, auditability, security access controls.

Modern enterprise network storage systems consist of storage clients and servers. Storage *clients* interface directly with users, who create and view content via applications. Separately, *servers* store the content in a durable and efficient fashion. The two communicate via the underlying network. The clients make requests to retrieve or modify the data, while the servers execute those requests. A typical deployment has a few servers connected to a large number of clients.

There are several semantic layers at both clients and servers. At clients, the highest unit of activity is a *session*, bounded by user-initiated connect and logoff requests. Once connected, users may launch a number of *applications*. Each application may perform many *file open-close* operations, which enclose the detailed data retrieval and modification requests.

The semantic layering at servers is identical to that for traditional file systems – *trees* contain *directories*, which in turn contain *files*.

It is necessary to consider these layers, because each layer correponds to a natural semantic enclosure of the workload serviced by the system, and thus forms natural design and evaluation boundaries. Figure 2.2 shows the semantic hierarchy among different access units.

In addition to user generated activity, modern enterprise network storage systems also perform a large range of activities that are invisible to the user. These activities enable the "enterprise" functionalities mentioned earlier, which includes fault tolerance, storage capacity efficiency, data snapshot, file system integrity checks, monitoring and audit, and automatic performance tuning. We do not view such activities as a part of the user generated workload. Rather, they are system functionalities that should be optimized in the context of the user generated workload.

Compared with MapReduce, enterprise network storage systems represent both a simplification and a complication. The simplication comes from the fact that the workload is completely determined by data movement, while MapReduce workloads contain both data movement and computation on the data. The complication comes from the

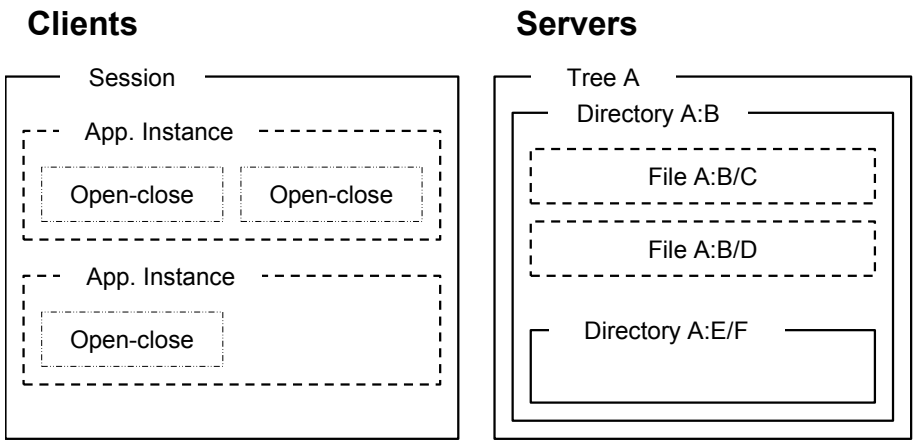



Figure 2.2. Different semantic layers of enterprise network storage systems

existence of different semantic layers at both clients and servers, while MapReduce has a single-layer semantic of jobs.

## 2.2 The Need to Examine Diverse Use Cases

Both MapReduce and enterprise network storage systems heavily rely on network and storage subsystems, as do large-scale data-centric systems in general, according to the definition in this dissertation. This dissertation builds on the knowledge derived from historical studies on network and storage subsystems.

Table 2.1 summarizes some of these studies in which understanding realistic system behavior played a large part. The studies took place over several decades. The network systems studies include [82, 95, 104, 38, 59, 17], and storage systems studies include [101, 134, 38, 92, 30, 43, 27]. We also include some recent work focused on large scale data centers [35, 139, 63, 17, 126, 94, 27, 89, 25], the environment in which large-scale data-centric systems are deployed.

An interesting trend emerges when we order the studies by the date of trace collection. In the late 1980s and early 1990s, measurement studies often capture system behavior for only one setting [101, 82]. We speculate that the stand-alone nature of these studies are due to the emerging measurement tools that created considerable logistical and analytical challenges at the time. Studies that examined multiple organizations and sectors are rare. They appeared from mid-1990s and to mid-2000s [95, 104, 38, 92, 30, 59], possibly the result of improved measurement tools, wide adoption of certain systems, and more engineering resources being devoted to system measurement.

Stand-alone studies have become common again in recent years [43, 17, 126, 94, 27, 89]. This is likely due to the tremendous scale of the systems of interest. Only a few

| Study | Traced system | Trace date | Organizations | Sectors | Findings & contributions |
|---|---|---|---|---|---|
| Ousterhout *et al.* [101] | BSD file system | 1985 | 1 | 1 | Various findings re bandwidth & access patterns |
| Leland *et al.* [82] | Ethernet packets | 1989-1992 | 1 | 1 | Ethernet traffic exhibits self-similarity |
| Wang *et al.* [134] | Disks and disk arrays | 1992-2004 | 2 | 1 | Predict storage device request response times |
| Mogul [95] | HTTP requests | 1994 | 2 | 2 | Persistent connections are vital for HTTP |
| Paxson [104] | Internet traffic | 1994-1995 | 35 | >3 | Various findings re packet loss, re-ordering, & delay |
| Breslau *et al.* [38] | Web caches | 1996-1998 | 6 | 2 | Web caches see various Zipf access patterns |
| Mesnier *et al.* [92] | NFS | 2001-2004 | 2 | 1 | Predict file properties based on name and attributes |
| Bairavasundaram *et al.* [30] | Enterprise storage | 2004-2008 | 100s | Many | Quantifies silent data corruption in reliable storage |
| Feamster *et al.* [59] | BGP configurations | 2005 | 17 | N/A | Discovers BGP misconfigurations by static analysis |
| Leung, *et al.* [83] | Enterprise storage | 2007 | 1 | 1 | Various findings re access patterns |
| Bodik *et al.* [35] | Datacenter perf. logs | 2008 | 1 | 1 | Automated classification of performance crisis |
| Xu *et al.* [139] | Hadoop & web server | 2009 | 1 | 1 | Automated problems detection using console logs |
| Ganapathi *et al.* [63] | Hadoop/Hive | 2009 | 1 | 1 | Use Hadoop job features to predict Hive query performance |
| Alizadeh *et al.* [17] | Datacenter TCP traffic | 2009 | 1 | 1 | Use explicit congestion notification to improve TCP |
| Thereska *et al.* [126] | Storage for web apps | 2009 | 1 | 1 | Save energy by powering down data replicas |
| Mishra *et al.* [94] | Web apps backend | 2009 | 1 | 1 | Task classification for capacity planning & scheduling |
| Anathanarayanan *et al.* [27] | Web search (Dryad) | 2009 | 1 | 1 | Alleviate hotspots by pre-emptive data replication |
| Meisner *et al.* [89] | Web search | 2010 | 1 | 1 | Interactive latency complicates hardware power mngmt. |
| Anathanarayanan *et al.* [25] | Hadoop and Dryad | 2009-2011 | 3 | 1 | Cache data based on heavy-tail access patterns |

Table 2.1. Summary of some prior work. We show here studies that relied on empirical system traces and pertains to either the subcomponents or execution environment of large-scale data-centric systems: network systems [82, 95, 104, 38, 59, 17], storage systems [101, 134, 38, 92, 30, 43, 27], and large-scale Internet data centers [35, 139, 63, 17, 126, 94, 27, 89, 25]. The studies with multiple organizations and sectors appeared in mid-1990s to mid-2000s.

organizations can afford systems of thousands or even millions of machines. Concurrently, the vast improvement in measurement tools create an over-abundance of data, presenting new challenges to derive useful insights from the deluge of trace data.

These technology trends create the pressing need to generalize beyond the studies that consider only single organizations or sectors. As large-scale data-centric systems use diversifies to many industries, system designers need to optimize for common behavior, in addition to improving the particulars of individual use cases.

Some studies amplified their breadth by working with ISPs [104, 59] or enterprise network storage vendors [30], i.e., intermediaries who interact with a large number of end customers. This dissertation takes advantage of similar opportunities to generalize beyond single-point studies of large-scale data-centric systems. As detailed later in the dissertation, we utilize production workload logs collected by vendors for large-scale data-centric systems.

## 2.3 Emerging Multi-Dimensional Techniques

Many studies in Table 2.1 had to confront the challenge of *model multi-dimensionality* [134, 92, 35, 139, 63, 94]. This challenge arises from the simple fact that the systems and workloads being studied are complex, and therefore need to be described in multiple ways, i.e., multiple features, characteristics, or dimensions. For example, a description of a MapReduce job includes at least three dimensions—the amount of input, shuffle, and output data. The studies in Table 2.1 contains models of up to thousands of dimensions, e.g., the collection of per-machine utilization statistics for all machines in a data center. Multi-dimensionality is inevitable for any rigorous attempts to model complex systems and workloads. For the rest of this dissertation, we will interchangeably use the terms features, characteristics, and dimensions.

Working in multiple dimensions create a heavy cognitive load. There are two common approaches to decrease this complexity—analyze workloads only one dimension at a time, or perform dimensionality reduction. The first is inherently problematic, the second is problematic if done poorly.

Most studies in Table 2.1 analyze workloads one dimension at a time to some extent. To illustrate the inherent limitation of the approach, consider study [83], which analyzed file systems. The analysis there revealed that most bytes are transferred from larger files. Although this is an useful observation, it does not reveal other characteristics of such large files: Do they have repeated reads? Do they have overwrites? Do they have many metadata requests? And so on. The resulting insights, although valuable, lead to uniform policies around a single design point in each of the dimensions. This is a potentially problematic outcome, since it assumes that a common case dominates workload behavior, and the common case in one dimension correlates to that in another, e.g., the common case in file size correspond to common case in access sequentiality. In reality, the workload may consist of many complex minority behavior patterns. For example, most bytes are indeed transferred from large files, which further breaks down into bytes from large and sequentially written and overwritten files, large and sequentially read and written once files, large and randomly read files, etc. However, each of the

broken down, truly multi-dimensional workload patterns form a minority when viewed only one dimension at a time.

Another approach to dealing with multi-dimensionality is to perform dimensionality reduction, i.e., judiciously leave out some dimensions in the analysis. This is often done according to heuristic and subjective approaches. However judicious the original intent may be, such ad-hoc dimensionality reduction introduces designer bias. Bias arises from the human system designer's potentially incorrect assumptions and mental models, built up by historical experience, but needs constant re-evaluation as systems and workloads rapidly evolve and increase in scale and complexity. Any subjective choice of which dimensions to keep and what heuristic reasoning to apply inevitably involves some assumptions about how systems and workloads behave.

The common limit to these two seemingly straightforward approaches is the human designer – large-scale data-centric systems have grown to a scale that would be cognitively unmanageable by previously satistifactory techniques. We need techniques that can simultaneously consider many aspects of the system being studied, and capable of simplifying the complexity of the system according to objective, empirical observations.

Fortunately, prior studies have pioneered a set of such techniques [134, 92, 35, 139, 63, 94]. These techniques relies on high-dimensional statistics to perform classification, clustering, regression, correlation analysis, and similar information extraction tasks. The rest of this dissertation applies these increasingly common analysis techniques to systematically understand and optimize large-scale data-centric systems. Chapter 3 reviews the most relevant techniques.

## 2.4 Beyond Nascent Benchmarks

To date, there has been no consensus approach to measuring the performance of large-scale data-centric systems. For some specific examples of such systems, a series of "benchmarks" emerged to measure various aspects of performance. However, there is a severe mismatch between the diversity of real life use cases, and the narrow slivers of artificial behavior covered by such "benchmarks". The mismatch is especially acute for relatively new systems such as MapReduce. The problem is less severe for established systems that have evolved to become large-scale data-centric systems. Even there, the vastly increased scale, diversity, and complexity of the data necessitates a re-think of previously validated benchmarks. In this section, we focus on emerging benchmarks for MapReduce to highlight the need for innovative approaches that more accurately reflect real-life workload characteristics.

The figures of merit for these benchmarks are:

1. Workload representativeness, i.e., whether the benchmark captures realistic data sizes, job arrival rates, and number of jobs for each type,

| | Grid-mix2 | Hive BM | Hi bench | Pig Mix | Grid-mix3 |
|---|---|---|---|---|---|
| Representative data-sizes | | | | | √ |
| Representative job arrival rates | | | | | √ |
| Representative # of jobs for each job type | | | | | √ |
| Easy to generate synthetic workloads | | | | | |
| Cluster & configuration independent | √ | √ | √ | √ | |

Table 2.2. Summary of shortcomings of recent MapReduce benchmarks

2. Ease of construction, i.e., whether it is easy to generate synthetic workload representative of realistic behavior, and

3. System independence, i.e., whether the benchmark allows performance comparison between clusters with different hardware, software, or configurations.

Table 2.2 summarizes five contemporary benchmarks for the MapReduce ecosystem according to these metrics – Gridmix2, Hive Benchmark, Pigmix, Hibench and Gridmix3. None of the these "benchmarks" sufficiently provide what is needed to evaluate system performance subject to real life conditions.

Gridmix2 [3] 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 [55], 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. As we will see later in the dissertation, while stress evaluations are an important aspect of evaluating MapReduce performance, actual production workloads contain many jobs with KB to MB data sizes. Thus, running a representative workload places realistic stress on the system beyond that generated by Gridmix 2.

Hive Benchmark [77] tests the performance of Hive [128], a data warehousing infrastructure built on top of Hadoop MapReduce. It uses datasets and queries derived from those used for comparing MapReduce with relational databases [103]. These queries aim to describe "more complex" analytical workloads and focus on direct translation from relational database queries. It is not clear that these 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 query mixes, interleavings, arrival intensities, data sizes, and other complexities that one would expect in a production deployment of Hive.

HiBench [74] 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 represent a wider diversity of applications than those used in other MapReduce benchmarking efforts. While HiBench includes a wider variety of jobs, it still fails to capture different job mixes and job arrival rates that one would expect in production clusters.

PigMix [110] 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 [4, 55] was motivated in response to situations where improvements yielding dramatic gains on Gridmix2 showed ambiguous or even decreased performance in production [4]. Gridmix3 replays job traces collected via Rumen [116], reproducing the byte and record movement patterns, as well as the job submission sequences, thus producing comparable load on the I/O subsystems.

Although the direct replay approach reproduces inter-arrival rates and the correct mix of job types and data sizes, it introduces other problems. For example, it is challenging to change the workload to add or remove new types of jobs, or to scale the workload along dimensions such as data sizes or arrival intensities. Further, changing the input Rumen traces is difficult, limiting the benchmark's usefulness on clusters with configurations different from that which 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. The thus far limited deployment of Rumen further hinders the usability of Gridmix3.

## 2.5 Ripe Time for a New Methodology

We should not consider the inadequacy of the prior analysis and benchmarking techniques as due to some negligence of the people who designed them. In fact, these tools once were quite helpful when functionality and scalability concerns were top priority, and real life use cases of large-scale data-centric systems were relatively homogenous. We should think of these prior techniques as so successful that they cause their own present demise – the most critical functionality and scalability concerns have been addressed, which in turn resulted in large-scale data-centric systems being adopted by diverse use cases. Consequently, it becomes a pressing concern to understand the performance of such systems subject to more complex and diverse environments. In other words, it is time for a new design and evaluation methodology that reflects the growth and evolution of large-scale data-centric systems.

# Chapter 3

# Methodology

*The method should be determined by the circumstances.*

— *Sun Zi, The Art of War.*

This chapter details some methodology innovations common to subsequent chapters on workload analysis (Chapters 4 and 5) and workload-driven design and evaluation (Chapters 6 and 7). We build on predecessor work on computer system performance measurement and evaluation in general and adapt the concepts there for large-scale data-centric systems.

We organize the chapter around three key steps of a workload-driven design and evaluation method:

1. Analyze a workload (Section 3.1),
2. Synthesize representative behavior (Section 3.2), and
3. Measure performance by workload replay or simulation (Section 3.3).

The concepts of workload analysis, synthesis, and evaluation translate across MapReduce and enterprise storage systems, two examples of large-scale data-centric systems analyzed in the dissertation. The actual semantics of each system determines the precise engineering details of how these concepts are carried out.

## 3.1 Analysis - Conceptual Workload Framework

This section details the conceptual framework for analyzing data centric workloads. We discuss the appropriate level of workload abstraction (Section 3.1.1) and the accompanying workload components (Section 3.1.2). We also briefly provide examples of how these workload analysis concepts translate to actual MapReduce and enterprise network

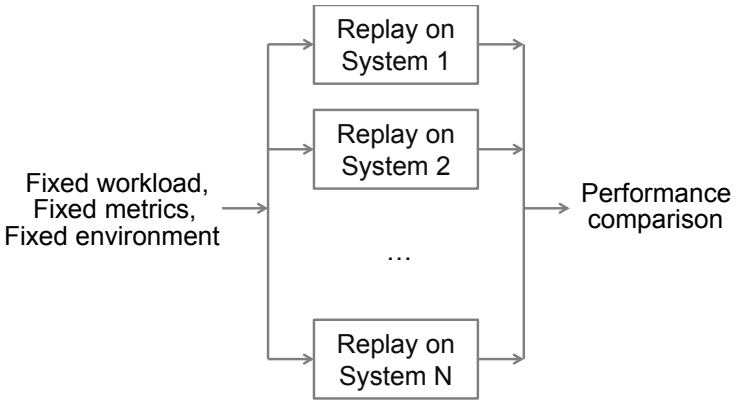



Figure 3.1. Performance comparison for many systems.

storage workloads, two case studies that will be further detailed in Chapter 4 (analysis of MapReduce workloads) and Chapter 5 (analysis of enterprise network storage workloads).

### 3.1.1 Workload abstraction level

The ultimate goal of workload analysis is to facilitate workload-driven design and evaluation, so that we can make claims such as "System $X$ has been engineered to have good performance for workload $Y$." A key part of the effort is to compare performance subject to the same workload for systems that implemented different design options. Figure 3.1 captures the canonical setup for workload driven performance comparisons between equivalent systems. What kind of systems are "equivalent" depends on the abstraction level at which the workload has been described, e.g., at the hardware or application level. Finding a good workload abstration level will enable a large range of equivalent systems to be compared, as discuss below.

One approach is to use a *functional view* of the workload [61]. This view aims to facilitate comparison between, say, a MapReduce system and a relational database system that service the equivalent functional goals of some enterprise data warehouse management workload. Such comparisons are essential for technology strategy decisions that commit an organization to alternate kinds of large-scale data-centric systems. To facilitate such comparisons, the functional view describes high level features of the enterprise warehouse management workload in terms that are independent of the engineering specifics of the systems to be compared. This functional workload view enables a large range of equivalent systems to be compared. However, the shortcomings of the approach include the fact that large-scale data-centric systems currently lack tracing capabilities at this level, and that it is challenging to translate insights at this level to concrete engineering improvements for the underlying system.

Another approach is to take a *physical view* of the workload [61]. This view describes a workload in terms of system hardware behavior, i.e., what are the CPU, memory, disk,

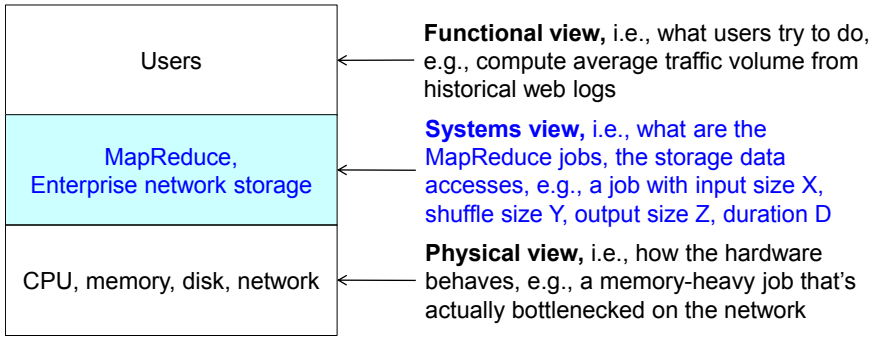



Figure 3.2. Functional, systems, and physical views of workloads. We use the systems view in this dissertation because it strikes a balance between system independence, measurement capability, and ease of translation between workload insights and engineering improvements on the underlying system.

and network activities during workload execution. Analyzing a workload at this level allows the identification of potential hardware bottlenecks, i.e., the hardware components that are fully utilized when other components of the system are not. However, hardware level behavior changes upon hardware, software, or even configuration changes in the system. Thus, characterizing the workload at the physical level precludes any kind of performance comparison — one cannot replay a physical workload on two different systems because the difference in the system alters the physical workload behavior. Prior work on Internet workloads have also identified this concern as the "shaping problem" [62].

We take a middle ground and adopt what we term a *system view* of the workload. This approach captures workload behavior at the highest level of abstraction currently that we can trace in large-scale data-centric systems, which corresponds to the natural, highest level semantic boundaries in the underlying system. For MapReduce, this translates to the stream of jobs and the associated per job characteristics. For enterprise network storage, this is the stream of data accesses at the application, session, file, and directory levels.

The systems view brings several benefits. It expedites workload measurement, since many large-scale data-centric systems already have built-in tracing capabilities at this level. Workload measurements at this level allows us to reason about what the system should be without the burden of what the physical system behavior currently is. The systems view further enables performance comparison across hardware, software, and configuration changes in a system. For example, we can replay the same stream of MapReduce jobs across solutions from different hardware or software vendors for procurement decisions, or tune configurations to a targeted workload on a specific cluster.

We should point out that the systems view does not enable comparisons between two different kinds of systems that service the same goals, e.g., between a MapReduce system and a relational database system that service the functionally equivalent enterprise

warehouse management workload. It remains an open problem to precisely define what exactly is a workload trace that captures behavior at the level of functional user intent.

Figure 3.2 captures the three different workload abstraction layers discussed here and highlights the systems view used in the dissertation.

Next, we detail specific workload components under the system view. We consider large-scale data-centric workloads as being conceptually composed of data access patterns, computation patterns, and load arrival patterns. Section 3.1.2 explains these conceptual components in detail and show how they concretely translate to actual semantic boundaries for MapReduce job streams and enterprise network storage accesses at the application, session, file, and directory levels

### 3.1.2 Workload components

We view the workload for large-scale data-centric systems as composed of data access patterns, computation patterns, and load arrival patterns.

*Data access patterns* represent an essential component of the workload for any systems that operate on data. They describe the following.

1. What the data is, which includes both the size of the data, and if such information is available, the format and content of the data.
2. How the data is organized, which involves unique identifiers for different data sets.
3. How the data is accessed, which translates to concepts such as read, write, sequentiality, repeated access, or other such characteristics.

*Computation patterns* capture what operations are done on the data. The computational code represents the most accurate measure of computation patterns. However, the actual code is often unavailable, and makes the workload code-dependent, i.e., system dependent. We aim to capture computation patterns on the level of data transform, aggregate, expand, filter, join, and other such operations.

*Load arrival patterns* describe the time-varying arrival pattern and sequence of work units. A *work unit* is a conceptual unit of processing. The specifics of a system determine what is an appropriate work unit. For MapReduce, a natural work unit is jobs. For enterprise network storage, there are several natural work units of different granularity, including IO requests, file open-closes, application instances, or user sessions.

Next, we give concrete examples of what are the MapReduce and enterprise network storage data access patterns, computation patterns, and load arrival patterns.

#### 3.1.2.1 Example - components of a MapReduce workload

We model a MapReduce workload as a sequence of jobs. Other possible ways to package a workload into work units are sequences of tasks of workflows. A *job* breaks down into

a series of parallel *tasks*, and some computations require a series of mutually dependent jobs executed in a *workflow*, e.g., a data analysis workflow with a data join job followed by a data selection job followed by a data transform job. Tasks are not an appropriate work unit, because the break down of jobs into tasks is a system-dependent behavior. Workflows are an appropriate work unit, but common MapReduce systems currently lack thorough tracing capabilities at that level.

Load arrival patterns consist of the time arrival sequence of jobs.

Data access patterns consist of the Hadoop Distributed File System (HDFS) input and output paths, which serve as unique identifiers for data sets, as well as the data size for the input, shuffle, and output stages. The format and content of the data also form valid data access patterns. However, common MapReduce systems currently lack thorough tracing of data format and contents.

Computation patterns consist of a six dimensional vector of `[input data size, shuffle data size, output data size, job duration, map task time, reduce task time]`. This six-tuple uses per-job statistics collected by current MapReduce tracing tools. They serve as a proxy for computation patterns. As we will see later in Chapter 4, these per-job six-tuples allow us to identify some meaningful computation patterns. There are some rudimentary capabilities to explicitly trace data operations at the transform, aggregate, expand, filter, or join level. It is not yet precisely known what these tools should trace. The advances in this dissertation should help clarify future tracing needs (See Section 8.1 for discussion of future work).

Later in the dissertation, Chapter 4 analyzes several industrial MapReduce workloads, and in doing so, expand upon the material here.

#### 3.1.2.2 Example - components of a enterprise network storage workload

Enterprise network storage systems also offer several natural boundaries for defining work units—IO requests, file open-closes, application instances, or user sessions. We end up using just application instances and user sessions. These work units lead to some useful workload insights, such as data caching or pre-fetching algorithms that rely on statistics for each application instance or user session. We also analyzed IO requests and file open-closes, but the insights were uninteresting and did not reveal system re-design opportunities.

Data access patterns follow the definition for traditional file systems - read/write, sequential/random, single/repeated access, file sizes, file types. Unique identifiers for datasets are at the file, directory, and tree levels. This is a departure from some perspectives that assume only a block-based view of file systems.

Computation patterns are not applicable for enterprise network storage, since the chief purpose of such systems is to store and retrieve the data without manipulation.

Load arrival patterns consist of the time arrival sequence of application instances or

user sessions, each may last over some time, and contains within its duration the load arrival pattern of finer granularity work units.

Chapter 5 analyzes several industrial enterprise network storage workloads, and in doing so, expand upon the material here.

## 3.2 Synthesis - Generating Representative Behavior

In addition to workload analysis, we also synthesize a representative workload for a particular use case and execute it to evaluate performance for a specific configuration. As discussed in Chapter 2, this approach offers more relevant insights than those gathered from one-size-fits-all benchmarks, because those benchmarks reflect only a small subset of possible workload behavior. We describe here a mechanism to synthesize representative behavior from traces of large-scale data-centric workloads.

We identify two design goals:

*1. The workload synthesis and execution framework should be agnostic to hardware/software/configuration choices, as well as system size and specific system implementation.*

We may intentionally vary any of these factors to quantify the performance impact of hardware choices, software optimizations, configuration differences, cluster capacity increases, functionally equivalent system choices, e.g., open source versus proprietary implementations.

*2. The framework should synthesize representative workloads that execute in a short duration of hours to days.*

Such synthetic workloads lead to rapid performance evaluations, i.e., rapid design iterations. 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 replay in full.

Our approach is to take continuous snapshots of the workload time series data, then concatenate the snapshots to form a synthetic workload. The snapshots span the multi-dimensional descriptions of each work unit contained in the original trace. Section 3.2.1 details this approach. Section 3.2.2 explains how our method departs from prior studies on workload synthesis using parametric models. Section 3.2.3 offers a specific illustration of the approach for MapReduce workloads.

### 3.2.1 Multi-dimensional time series snapshot

The workload synthesizer takes as input a large-scale data-centric system trace over a time period of length $L$, and the desired synthetic workload duration $W$, with $W < L$. The trace is a list of work units, with each list item being a multi-dimensional vector describing the work unit arrival time, data properties, and compute properties.

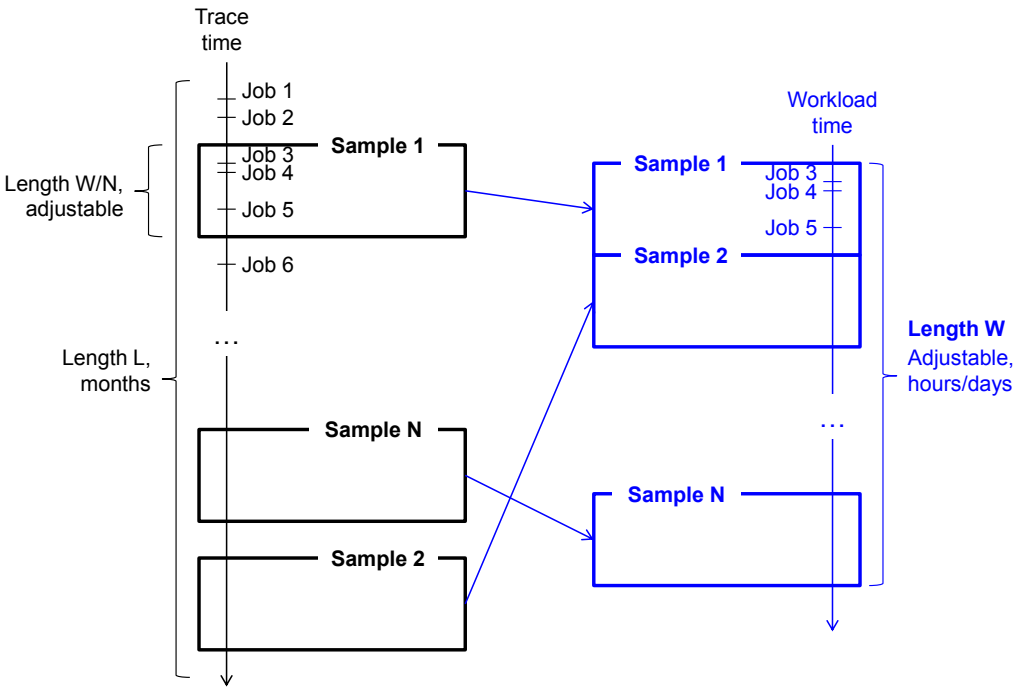



Figure 3.3. Workload synthesis process. Showing the concatenation of continuous time windows from the trace. The sampling preserves the inter-job arrival times, and the multi-dimensional data and compute patterns for each job.

The workload synthesis process seeks to capture both time-independent statistics and workload behavior variations over time. We can capture time-independent statistics using the fact that memoryless Poisson sampling captures time averages [137]. However, purely Poisson sampling would fail to capture workload variation over time. A continuous length of trace reflects workload variation over time, but fails to capture behavior beyond the trace period. After some experimentation, we developed a middle-ground approach that takes memoryless samples of continuous time windows.

We divide the synthetic workload into $N$ non-overlapping segments, each of length $W/N$. Each segment will be filled with a randomly sampled segment of time duration $W/N$, 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. Sampling without replacement can ensure there are non-overlapping samples, but represents a deviation from the memoryless sampling process.

The workload synthesis process returns a list of jobs in the same format as the initial trace, with each item containing the work unit arrival time, data properties, and compute properties. The primary difference is that the synthetic workload is of duration $W < L$. Figure 3.3 shows this process pictorially.

We satisfy design Requirement 1 by including in the input trace only cluster-independent information. Requirement 2 is satisfied by adjusting $W$ and $N$. 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 the time-dependent and time-independent dimensions.

### 3.2.2 Non-parametric models

The synthesis approach uses a non-parametric model of the workload. In other words, the cluster trace is the empirical model of the workload. This approach represents a conceptual departure from established techniques that use parametric models of workloads.

Parametric approaches seek to model workload behavior using some analytically tractable statistical models. For example, a common parametric model for arrival patterns is the Poisson or memoryless arrival model, used decades ago to generate network traffic [107]. A common parametric model for data patterns is the Zipf or long-tail frequency model, used for populating synthetic databases [65]. These models involve a small number of shape parameters, which must be fitted to empirical data. The models are successful for systems whose behavior do indeed follow the structure of these models.

Parametric models work less well for large-scale data-centric systems, because of the complexity, diversity, and rapidly changing nature of such systems. As we will see later in Chapters 4 and 5, the workload behaviors do not fit any statistical processes with a small number of parameters. One could adopt more complex models with additional parameters to fit the empirically observed behavior, such as Poisson models with time varying average event arrival rates.

A fully empirical, non-parametric model is simply an extension of the process of introducing more model parameters. We can view an empirical model as one that contains as many parameters as there are data points in the workload trace. This approach works well for large-scale data-centric systems for several reasons.

1. Such systems are usually instrumented, making the empirical workload traces more easily available.
2. It is easier to observe the system behavior than to fully understand the generative process behind the behavior, which we need for good parametric models.
3. The models easily cover diverse and evolving use cases — an updated workload trace represents an updated model.

In the terminology of [60], we sacrifice the compactness of the model to gain representativeness, flexibility, system independence, and simplicity of construction.

This shift in approach has already started in some prior work. For example, [107] showed that TELNET and FTP session arrivals followed Poisson models, with the Poisson average arrival rates being empirical constants that change at the hourly or finer

granularity. This already represents a partially empirical model. The multi-dimensional empirical models used in this dissertation represents an extension of this method from time varying arrival rates to the arrival times and sequences reflected in the trace, and also from arrival patterns to the multi-dimensional descriptions data and computation patterns.

We should caution that both analytical and emprical models are still *models*, i.e., a partial reflection of the entire universe of real life behavior. Prior studies on Internet traffic modeling have found that both give similiar errors in modeling complex, non-stationary system behavior [105]. We expect large-scale data-centric systems to also exhibit complex, non-stationary behavior. Hence, one should be cautious and not over-interpret the representativeness of either kind of models.

### 3.2.3 Example - synthesizing MapReduce workloads

We illustrate the workload synthesis process using MapReduce workloads. Doing so requires translating the constructs in Section 3.2.1, which apply to any large-scale data-centric systems, to specific artifacts for MapReduce.

By representative, we mean that the synthetic workload should reproduce from the original trace the distribution of input, shuffle, and output data sizes i.e., the 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 2009 Facebook traces (Table 1.1) and our synthesis tools. Chapter 4 contains a more detailed analysis of this workload.

*Data characteristics*

Figure 3.4 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. The results (Figure 3.5) are intuitive - when the synthetic workload length is fixed, shorter sample lengths result in 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 [136]. In other words, when we fix $W$, the synthetic workload length, then increasing by 4 times the number of continuous time window samples would half the error. Figure 3.5 shows that when we increase $n$ by 4 times and then by 4 times again, the synthesis error, i.e., the horizontal distance between the dashed lines, halved and then halved again.

Also, the sampling method could be modified to accommodate different metrics of "representativeness". For example, to capture daily diurnal patterns, the sampling

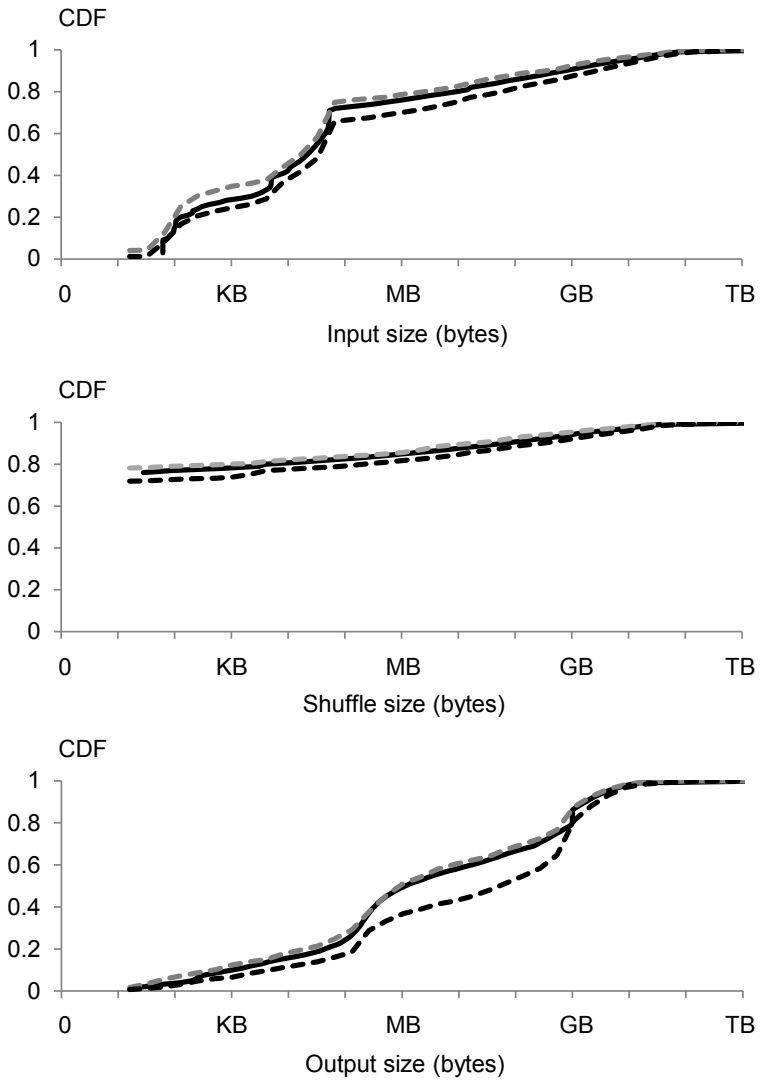



Figure 3.4. Distributions of data sizes in synthesized workload using 1-hr samples. Showing that the data characteristics are representative – min. and max. distributions for the synthetic workload (dashed lines) bound the distribution computed over the entire trace (solid line).

method could use day-long continuous sample windows. Alternately, we could perform conditional sampling of hour-long windows, e.g., the first hour in synthetic trace samples from the midnight to 1AM time window of all days. Other conditional sampling methods can capture behavior changes over different time periods, job streams from different organizations, or other ways of constructing sub-workloads.

*Job submission patterns*

Our intuition is that the 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 performing memoryless sampling, with the job submission rate fluctuating in a narrow

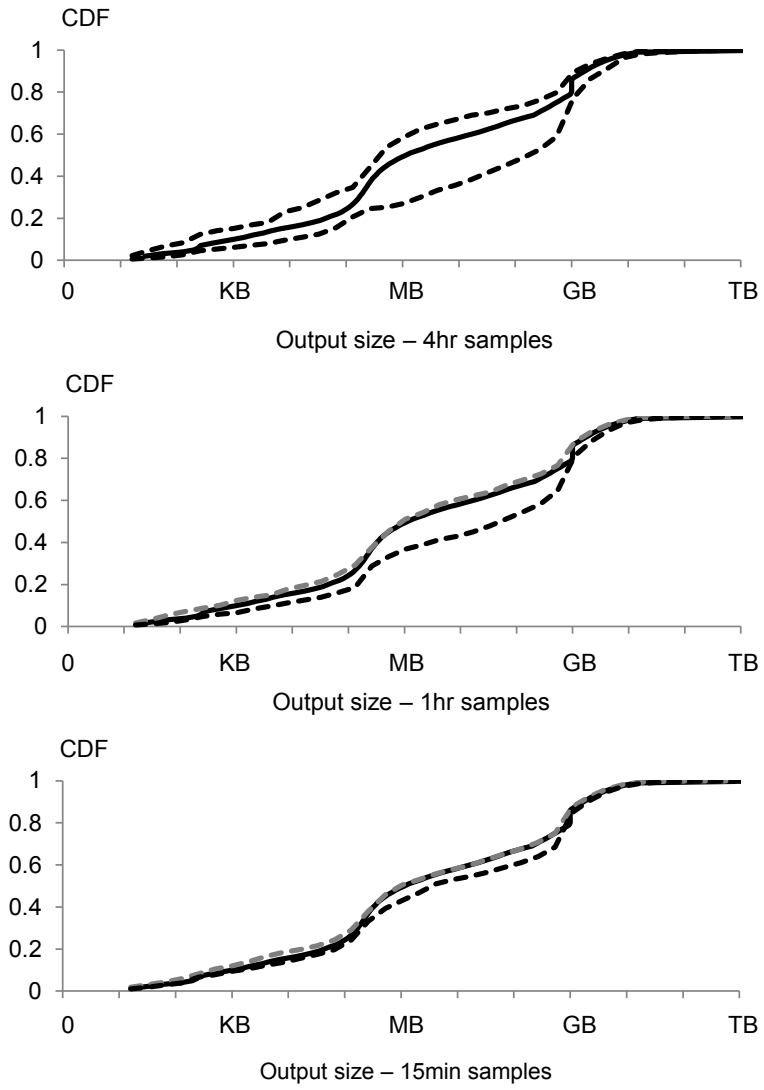



Figure 3.5. Distributions of output sizes in synthesized workload using different sample lengths. For fixed-length synthetic workload, the horizontal gap between the min. and max. distributions for the synthetic workload (dashed lines) and the distribution for the entire trace (solid line) decreases by 2× when the sampling window shortens by 4×.

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 3.6 confirms this intuition. The figure shows the jobs submitted per hour for workloads synthesized by various sample windows lengths. We see that the workload synthesized using 4-hour samples has 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 300 jobs per

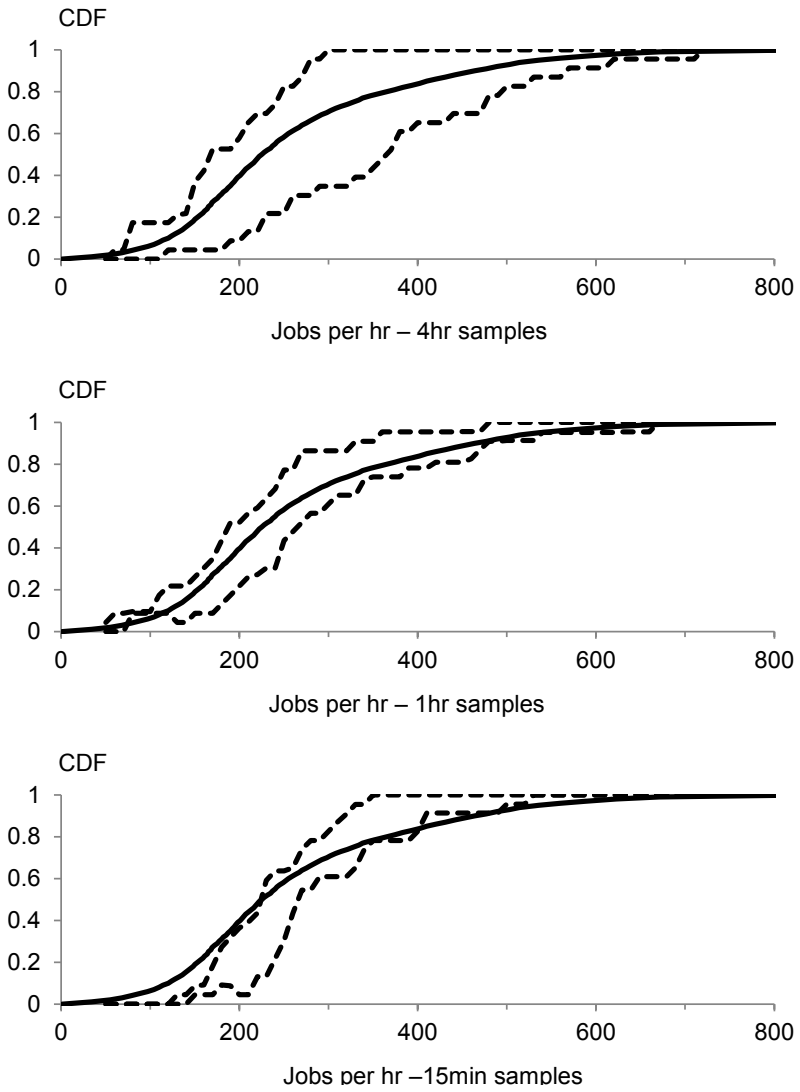



Figure 3.6. Distributions of jobs per hour in synthetic workload. Short samples distort variations in job submit rates – min. and max. distributions for synthetic workload (dashed lines) bound the distribution for the entire trace (solid line) for 1 & 4-hour samples only.

hour, which is the long-term average job submission rate. Thus, while shorter sample windows result in more representative data characteristics, they distort variations in job submission rates.

*Common jobs*

Figure 3.7 shows the frequency of common jobs in the synthetic workload, expressed as fractions of the frequencies in the original trace. Chapter 4 details how we actually identify these common jobs. A representative workload has 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. Note that the

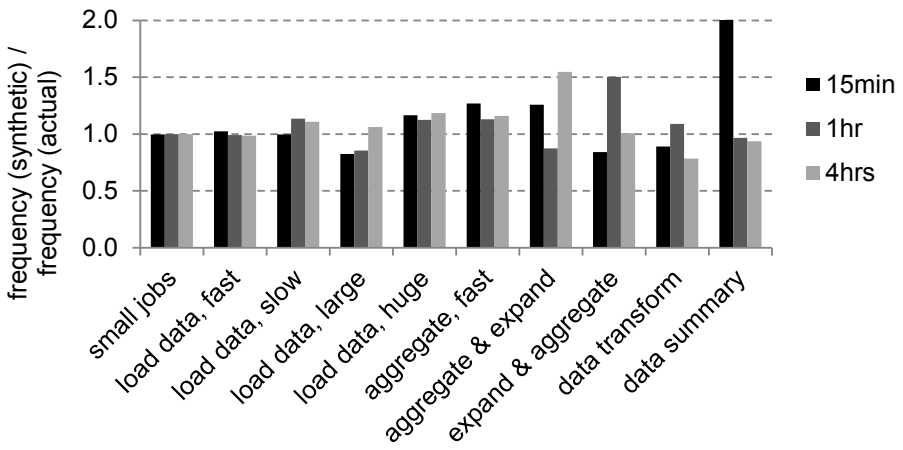



Figure 3.7. Frequency of common jobs in the synthetic workload as fractions of the frequencies in the original trace. Showing that workloads synthesized using continuous samples of 15min, 1hr, and 4hrs all have common jobs frequencies similar to the original trace.

Law of Large Numbers guarantee that the sample average converges to the true average, but the frequencies of the most common jobs converge the fastest to their actual values.

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. It turns out that those jobs have very low frequencies (for details see Chapter 4 and Table 4.3). 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 has 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 the representativeness of the synthetic workload, i.e., how close are all of the statistics of the synthetic workload compared with the actual workload. We can increase workload representativeness by synthesizing longer workloads.

Now that we can synthesize representative workloads, we need a way to actually run them on real systems and evaluate performance. This is the subject of the next section.

## 3.3 Evaluate - Workload Replay and Simulation

In this section, we turn to the process of workload replay, either on an actual system or in simulation. Recall from earlier in the chapter that the canonical setup for workload driven performance comparisons seeks to replay a workload on two equivalent systems (Figure 3.1). We can apply this setup to compare competing products for procurement decisions, evaluate new features for performance testing, optimize configurations for workload-specific tuning, or accomplish other similar goals. The resulting claims will

be phrased as "System $n$ has the best performance, subject to workload X, performance metric Y, and execution environment Z."

It is impossible to replay large-scale data-centric workloads at production scale, for long durations, and using production data and code. Doing so requires replicating the production system, data, and code at full scale, a economically and logistically impossible task, even though it gives an accurate measurement of how systems perform under real life conditions. Doing measurements on the actual production system achieves the same purpose, but involves disrupting front line, business critical processes. Hence, we need to address the following concerns to evaluate systems by replaying workloads without production data, code, and system scale.

- Synthesize representative workload (Covered previously in Section 3.2.3).
- Generate synthetic input data, possibly scaled down.
- Run artificial jobs to generate the load on the system.
- Remove output data, necessary if the system on which the workload is replayed is not production scale, and therefore potentially unable to archive the full output data.
- Actual workload execution mechanism to generate synthetic input data, followed by launching the artifical jobs at the appropriate arrival times.
- Verify that the workload execution mechanism introduces low performance overhead.

The particular implementation of these mechanisms depend on the system in question. In Section 3.3.1, we give an example of the replay mechanisms for MapReduce workloads. We then discuss in Section 3.3.2 context in which simulations are still needed despite the ability to replay real life workloads.

### 3.3.1 Example - replay MapReduce workloads

We translate the job list from the synthetic workload to concrete MapReduce jobs that can be executed on artificially generated data. We then use a script to actually generate the input data, and execute the MapReduce workload.

#### 3.3.1.1 Workload execution script

We use the following script.

```
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 &

...
```

The line `HDFS randomwrite(max_input_size)` writes the input test data to the underlying file system (Section 3.3.1.2). The lines `RatioMapReduce inputFiles[*] output* shuffleInputRatio[*] outputShuffleRatio[*]` launch artificial load generating jobs (Section 3.3.1.3). The lines `HDFS -rmr output* &` removes the output data (Section 3.3.1.4). The lines `sleep interval[1]` preserves the job submission intensity in the workload. Section 3.3.1.5 empirically verifies that these replay mechanisms introduce little performance overhead.

#### 3.3.1.2 Generate synthetic input 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 as default HDFS block size. We believe this setting is reasonable because our input files become as granular as the underlying HDFS. We validated that there is negligible overhead when concurrent jobs read from the same HDFS input (Section 3.3.1.5).

#### 3.3.1.3 Artificial load generating MapReduce job

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, as below.

```
class RatioMapReduce {

x = shuffleInputRatio

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

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

```
  for each v in values {
    repeat floor(y) times {
      output.collect(new K3(randomKey), new V3(randomValue));
    }
  }
  if (randomFloat(0,1) < decimal(y)) {
    output.collect(new K3(randomKey), new V3(randomValue));
  }
}

} // end class RatioMapReduce
```

#### 3.3.1.4 Removing output data

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 3.3.1.5).

#### 3.3.1.5 Verifying workload replay has low performance overhead

Since workload replay aims to measure performance, it is vital to verify that the replay mechanisms do not introduce performance overhead. There are two sources of potential overhead due to concurrent processing in our workload replay 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 quantify the overhead by running, say, the full scale Facebook workload with non-overlapping input data or no removal of workload output, and compare the performance against a setup in which we have overlapping input and background removal of output. Doing so presents a logistical challenge — we require a system with up to 200TB of disk space, which is the 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 have low impact on HDFS reads, we repeat 10 times the following 10GB sort 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
Job 3: 10 GB sort, input HDFS/directoryA
Job 4: 10 GB sort, input HDFS/directoryA
```

| | |
|---|---|
| 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 3.1. Simultaneous HDFS read. Jobs 1 and 2 perform concurrent reads of two different directories. Jobs 3 and 4 perform concurrent reads of the same directory. We report the average and 95% confidence interval from 10 repeated measurements. Results show concurrent reads of the same directory has low overhead, since there is considerable overlap in the confidence intervals.

| | |
|---|---|
| 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 |

Table 3.2. Background HDFS remove. Jobs 1-4 perform read, write, shuffle, and sort without background delete. Jobs 5-8 perform read, write, shuffle, and sort with background delete. We report the average and 95% confidence interval from 10 repeated measurements. Results indicate background deletes introduce low overhead, since the confidence intervals overlap.

Jobs 1 and 2 give the baseline performance, while Jobs 3 and 4 identify any potential overhead. The running times are in Table 3.1. The finishing times are completely within the 95% confidence intervals of each other. Thus, our data input mechanism imposes no measurable overhead.

We repeat the experiment with up to 20 concurrent read jobs. There, the MapReduce task schedulers and placement algorithms introduce large variance in job completion time, since job execution becomes bottlenecked on the number of available task slots on the cluster. A queue of waiting jobs builds up. The performance becomes a function of the overall cluster drain rate, and the average job durations again falling within confidence intervals of each other. Thus, at higher read concurrency levels, performance is dominated by effects other than our data input mechanism.

*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, which is a legacy, basic, but still relatively stable and full featured Hadoop distribution.

```
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 HDFS -rmr in background
  Wait for job to finish
Job 6: Read 10 GB from HDFS, with HDFS -rmr in background
  Wait for job to finish
Job 7: Shuffle 10 GB, with HDFS -rmr in background
  Wait for job to finish
Job 8: Sort 10 GB, with 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. In particular, we delete a 10GB pre-existing file. The running times are in Table 3.2. The finishing times are 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. The space is truly reclaimed after at least 6 hours [8]. The datanodes can perform the reclaim operation when it detects that it is not servicing other data access requests. Thus, even an in-thread, non-background HDFS remove impose low overhead. This "background delete" would work less well when clusters temporarily have very little spare storage capacity. Clusters are usually provisioned with the intent to make such events are rare.

### 3.3.2 Simulate where replay cannot scale in time or size

There are situations where simulations play an important role, even though workload replay on real systems allows us to measure performance "closer to real life". The necessity of simulation has been well discussed for Internet systems [62]. Simulations are complementary to experiment, measurement, and analysis. They allow the exploration of complicated scenarios that are difficult or impossible to analyze. Real-life use cases for large-scale data-centric systems are full of such scenarios.

Chapter 6 involves one such scenario. There, we present a system with the following characeristics that make it hard to do performance measurement by replaying a real life workload:

1. The performance depends on behavior across continuous periods of hours or days. Within the workload synthesis framework in Section 3.2.3, we need many continuous time windows of length hours or days to obtain a representative workload. Hence, workload replay according to Section 3.3 requires many days to measure just a single system setting.

2. The system involves many configuration and policy settings that impact performance. It is necessary to scan a large configuration and policy space to identify

the optimal setting. It becomes impossible to replay days-long workloads to scan such a large space.

3. It takes considerable effort to fully implement the system and incorporate it into the constraints of a production system. Cluster operators need to understand the performance gains to set appropriate design priorities. Hence, workload replay on a fully implemented system is impossible.

These concerns arise out of the scale, complexity, and rapid workload evolution of large-scale data-centric systems. They necessitate replaying the workload in simulation. We believe they also apply to settings outside that covered in Chapter 6.

Traditionally, we judge a simulation to be "good" if it is accurate. Simulations for large-scale data-centric systems need to consider additional performance metrics. Prior work on Internet simulations already identified that beyond certain system scale and complexity, simulation speed and scalability are also important metrics [113, 87]. These concerns translate to large-scale data-centric systems. As we will discuss in Chapter 6, some current simulators focus on accuracy only, which increasees simulator complexity and limit simulation scalability and speed. We are compelled to construct our own simulators to find a more appropriate tradeoff point (details see Sections 6.5 and 6.7).

## 3.4 Applying the Method Later in the Dissertation

The remainder of the dissertation more concretely illustrates dimensions of the methodology presented in this chapter. Chapters 4 and 5 present analyses of MapReduce and enterprise network storage workloads, and make use of the analysis component of the methodology. Chapters 6 and 7 applies these workload insights to solve two challenging system design problems — MapReduce energy efficiency and TCP incast. Both involve using our workload synthesis and replay tools. The study on MapReduce energy efficiency further illustrates the workload simulation part of the methodology.

# Chapter 4

# Workload Analysis and Design Insights - Enterprise MapReduce

*Measure yourself before buying clothes. — Mo Zi.*

This is the first of two chapters that apply the workload analysis techniques to large-scale data-centric systems. Here, we **analyze production workloads from seven MapReduce use cases**. Workloads from these systems form a good initial illustration of our workload analysis methods, because the MapReduce programming paradigm has a relatively simple semantic structure of map and reduce computations. The analysis leads to a series of insights that validate existing efforts, suggest new opportunities, or necessitate a change in engineering focus.

The chapter is organized as follows. We begin by motivating the importance of analyzing MapReduce traces (Section 4.1) and describing the traces covered in the chapter (Section 4.2). We then discuss in depth three independent axes of workload behavior – data access patterns (Section 4.3), load arrival patterns (Section 4.4), and computation patterns (Section 4.5). We close the chapter by highlighting some data analysis trends revealed by the production workloads (Section 4.6), and summarizing the broader implications of the study with regard to design and evaluation approaches (Section 4.7).

## 4.1 Motivation

The MapReduce computing paradigm is gaining widespread adoption across diverse industries as a platform for large-scale data analysis, accelerated by open-source implementations such as Apache Hadoop. The scale of such systems and the depth of their software stacks create complex challenges for the design and operation of MapReduce clusters. A number of recent studies looked at MapReduce-style systems within a handful of large technology companies [43, 17, 126, 94, 89, 27]. These studies look at one industry sector

| Section | Observations | Implications/Interpretations |
|---|---|---|
| 4.3.1 | Input data forms 51-77% of all bytes, shuffle 8-35%, output 8-23%. | Need to re-assess sole focus on shuffle-like traffic for datacenter networks. |
| 4.3.2 | The majority of job input sizes are MB-GB. | TB-scale benchmarks such as TeraSort are not representative. |
| 4.3.2 | For the Facebook workload, over a year, input and shuffle sizes increase by over 1000× but output size decreases by roughly 10×. | Growing customers and raw data size over time, distilled into possibly the same set of metrics. |
| 4.3.3 | For all workloads, data accesses follow the same Zipf distribution. | Tiered storage is beneficial. Uncertain cause for the common patterns. |
| 4.3.3 | 90% of jobs access small files that make up 1-16% of stored bytes. | Cache data if file size is below a threshold. |
| 4.3.3 | Up to 78% of jobs read input data that is recently read or written by other jobs. 75% of re-accesses occur within 6 hrs. | LRU or threshold-based cache eviction viable. Trade-offs between eviction policies need further investigation. |
| 4.4.1 | There is a high level of burstiness in job submit patterns in all workloads. | Online prediction of data and computation needs will be challenging. |
| 4.4.1 | Daily diurnal pattern evident in some workloads. | Human-driven interactive analysis, and/or daily automated computation. |
| 4.4.2 | Workloads are bursty, with peak to median hourly job submit rates of 9:1 or greater. | Scheduling and placement optimizations essential under high load. Increase utilization or conserve energy during inherent workload troughs. |
| 4.4.2 | For Facebook, over a year, peak to median job submit rates decrease from 31:1 to 9:1, while more internal organizations use MapReduce. | Multiplexing many workloads help smooth out bustiness. However, utilization remains low. |
| 4.4.2 | Workload intensity is multi-dimensional (jobs, I/O, task-times, etc.). | Need multi-dim. metrics for burstiness and other time series properties. |
| 4.4.3 | Avg. temporal correlation between job submit & data size is 0.21; for job submit & compute time it is 0.14; for data size & compute time it is 0.62. | Schedulers need to consider metrics beyond number of active jobs. MapReduce workloads remain data- rather than compute-centric. |
| 4.5.1 | Workloads on avg. spend 68% of time in map, 32% of time in reduce. | Optimizing read latency/locality should be a priority. |
| 4.5.1 | Most workloads have task-seconds per byte of $1 \times 10^{-7}$ to $7 \times 10^{-7}$; one workload has task-seconds per byte of $9 \times 10^{-4}$. | Build balanced systems for each workload according to task-seconds per byte. Develop benchmarks to probe a given value of task-seconds per byte. |
| 4.5.2 | For all workloads, task durations range from seconds to hours. | Need mechanisms to choose uniform task sizes across jobs. |
| 4.5.3.1 | <10 job name types make up >60% of all bytes or all compute time. | Per-job-type prediction and manual tuning are possible. |
| 4.5.3.1 | All workloads are dominated by a 2-3 frameworks. | Meta-schedulers need to multiplex only a few frameworks, e.g., Hive, Pig, in addition to core MapReduce. |
| 4.5.3.2 | Jobs touching <10GB of total data make up >92% of all jobs, and often have <10 tasks or even a single task. | Schedulers should design for small jobs. Re-assess the priority placed on addressing task stragglers. |
| 4.5.3.2 | Five out of seven workloads contain map-only jobs. | Not all jobs benefit from network optimizations for shuffle. |
| 4.5.3.2 | Common job types change significantly over a year. | Periodic re-tuning and re-optimizations is necessary. |

Table 4.1. Summary of observations from the workload, and associated design implications and interpretations.

or sometimes even one company at a time. Consequently, the associated design insights may not generalize across use cases, an important concern given the emergence of MapReduce users in different industries [7]. There is a need to develop systematic knowledge of MapReduce behavior at both established users within technology companies, and at recent adopters in other industries.

In this chapter, we analyze seven MapReduce workload traces from production clusters at Facebook and at Cloudera's customers in e-commerce, telecommunications, media, and retail. Cumulatively, these traces comprise over a year's worth of data, covering over

two million jobs that moved approximately 1.6 exabytes spread over 5000 machines (Table 4.2). They are collected using standard tools in Hadoop, and facilitate comparison both across workloads and over time.

The key findings of our analysis are as follows:

1. There is a new class of MapReduce workloads for interactive, semi-streaming analysis that differs considerably from the original MapReduce use case targeting purely batch computations.
2. There is a wide range of behavior within this workload class, such that we must exercise caution in regarding any aspect of workload dynamics as "typical".
3. SQL-like programatic frameworks on top of MapReduce such as Hive and Pig make up a considerable fraction of activity in all workloads we analyzed.
4. Some prior assumptions about MapReduce such as uniform data access, regular diurnal patterns, and prevalence of large jobs no longer hold.

Subsets of these observations have emerged in several studies that each looks at only one kind of MapReduce workload [128, 37, 27]. Identifying these characteristics across a rich and diverse set of workloads shows that the observations are applicable to a range of use cases.

Table 4.1 summarizes our findings and serves as a roadmap for the rest of the chapter. Our methodology extends [43, 44, 40], and organizes the analysis according to three conceptual aspects of a MapReduce workload: data access, load arrival, and compute patterns. Section 4.3 looks at data patterns. This includes aggregate bytes in the MapReduce input, shuffle, and output stages, the distribution of per-job data sizes, and the per-file access frequencies and intervals. Section 4.4 focuses on temporal patterns. It analyzes variation over time across multiple workload dimensions, quantifies burstiness, and extracts temporal correlations between different workload dimensions. Section 4.5 examines computate patterns. It analyzes the balance between aggregate compute and data size, the distribution of task sizes, and the breakdown of common job categories both by job names and by multi-dimensional job behavior. Our contributions are:

- Analysis of seven MapReduce production workloads from five industries totaling over two million jobs,
- Derivation of design and operational insights, and
- Methodology of analysis and the deployment of a public workload repository with workload replay tools.

## 4.2 Workload Traces Overview

We analyze seven workloads from various Hadoop deployments. All seven come from clusters that support business critical processes. Five are workloads from Cloudera's enterprise customers in e-commerce, telecommunications, media, and retail. Two others are

| Trace | Machines | Length | Date | Jobs | Total input, shuffle, output bytes |
|---|---|---|---|---|---|
| CC-a | <100 | 1 month | 2011 | 5,759 | 80 TB |
| CC-b | 300 | 9 days | 2011 | 22,974 | 600 TB |
| CC-c | 700 | 1 month | 2011 | 21,030 | 18 PB |
| CC-d | 400-500 | 2+ months | 2011 | 13,283 | 8 PB |
| CC-e | 100 | 9 days | 2011 | 10,790 | 590 TB |
| FB-2009 | 600 | 6 months | 2009 | 1,129,193 | 9.4 PB |
| FB-2010 | 3000 | 1.5 months | 2010 | 1,169,184 | 1.5 EB |
| Total | >5000 | ≈ 1 year | - | 2,372,213 | 1.6 EB |

Table 4.2. Summary of traces. CC is short for "Cloudera Customer". FB is short for "Facebook". Bytes touched is computed by sum of input, shuffle, and output data sizes for all jobs. Note that a precise count of the machines for CC-a was not available due to fast expansion of the cluster during the traced period.

Facebook workloads on the same cluster across two different years. These workloads offer a rare opportunity to survey Hadoop use-cases across several technology and traditional industries (Cloudera customers), and track the growth of a leading Hadoop deployment (Facebook).

Table 4.2 gives some detail about these workloads. The trace lengths are limited by the logistical challenges of shipping trace data for offsite analysis. The Cloudera customer workloads have raw logs sizes approaching 100GB. The raw logs are on the 10s of TB scale for the Facebook workloads, logistically difficult to transfer over the Internet, and required us to query Facebook's internal monitoring tools for a pre-processed format. Combined, the workloads contain over a year's worth of trace data, covering over 2 millions jobs and 1.6 exabytes processed by the clusters.

The data is logged by standard tools in Hadoop; no additional tracing tools were necessary. The workload traces contain per-job statistics for job ID (numerical key), job name (string), input/shuffle/output data sizes (bytes), duration, submit time, map/reduce task time (slot-seconds), map/reduce task counts, and input/output file paths. We call each of these characteristic a numerical *dimension* of a job. Some traces have some data dimensions unavailable.

We obtained the Cloudera traces by doing a time-range selection of per-job Hadoop history logs based on the file timestamp. The Facebook traces come from a similar query on Facebook's internal log database. The traces reflect no logging interruptions, except for the cluster in CC-d, which was taken offline several times due to cluster maintenance. There are some inaccuracies at trace start and termination, due to incomplete jobs at the trace boundaries. The length of our traces far exceeds the typical job length on these systems, leading to negligible errors. To capture weekly behavior for CC-b and CC-e, we queried for 9 days of data to allow for inaccuracies at trace boundaries.

### 4.2.1 Questions on Workload Behavior

One can develop questions about workload behavior based on prior work. Below are some key questions to ask about any MapReduce workloads.

1. For optimizing the underlying storage system:
   - How uniformly or skewed are the data accesses?
   - How much temporal locality exists?
2. For workload-level provisioning and load shaping:
   - How regular or unpredictable is the cluster load?
   - How large are the bursts in the workload?
3. For job-level scheduling and execution planning:
   - What are the common job types?
   - What are the size, shape, and duration of these jobs?
   - How frequently does each job type appear?
4. For optimizing SQL-like programming frameworks:
   - What % of cluster load come from these frameworks?
   - What are the common uses of each framework?
5. For performance comparison between systems:
   - How much variation exists between workloads?
   - Can we distill features of a representative workload?

Using the original MapReduce use-case of data indexing in support of web search [53] and the workload assumptions behind common microbenchmarks of stand-alone, large-scale jobs [13, 3, 103], one would expect answers to the above to be: (1) Some data access skew and temporal locality exists, but there is no information to speculate on how much. (2) The load is sculpted to fill a predictable web search diurnal with batch computations; bursts are not a concern since new load would be admitted conditioned on spare cluster capacity. (3) The workload is dominated by large-scale jobs with fixed computation patterns that are repeatedly and regularly run. (4) We lack information to speculate how and how much SQL-like programming frameworks are used. (5) We expect small variation between different use-cases, and the representative features are already captured in publications on the web indexing use-case and existing microbenchmarks.

Several recent studies offered single use-case counterpoints to the above mental model [128, 37, 27]. The data in this paper allow us to look across use-cases from several industries to identify an alternate workload class. What surprised us the most is (1) the tremendous diversity within this workload class, which precludes an easy characterization of representative behavior, and (2) that some aspects of workload behavior are polar opposites of the original large-scale data indexing use-case, which warrants efforts to revisit some MapReduce design assumptions.

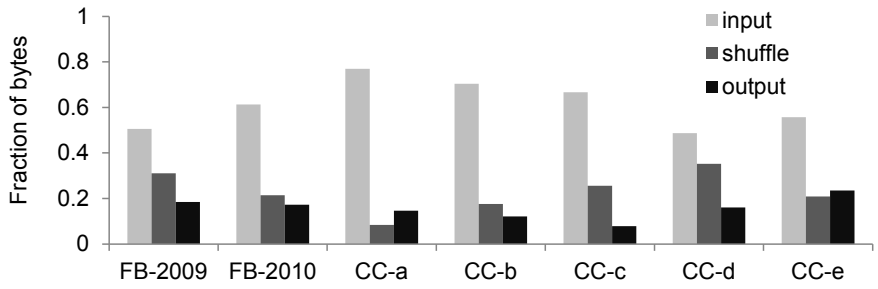



Figure 4.1. Aggregate fraction of all bytes that come from input, shuffle, or output of each workload.

## 4.3 Data Access Patterns

Data movement is a key function of these clusters, so understanding data access patterns is crucial. This section looks at data patterns in aggregate (Section 4.3.1), by jobs (Section 4.3.2), and per-file (Section 4.3.3).

### 4.3.1 Aggregate input/shuffle/output sizes

Figure 4.1 shows the aggregate input, shuffle, and output bytes. These statistics reflect I/O bytes seen from the MapReduce API. Different MapReduce environments lead to two interpretations with regard to actual bytes moved in hardware.

One interpretation is to optimize for N-to-N traffic patterns for datacenter networks [16, 17, 66, 47]. This interpretation comes from the assumptions that the cluster is so large that task placement is compelled to be random, and locality is negligible for all three input, shuffle, and output stages. Under these assumptions, all three MapReduce data movement stages involve network traffic, and the aggregate traffic looks like N-to-N shuffle traffic for all three stages.

Another interpretation is to optimize for data locality and minimize network traffic. Hadoop by default attempts to place map tasks on machines that already have the input data. Hadoop also tries to combine or compress map outputs and optimize placement for reduce tasks to increase rack locality for shuffle. By default, for every API output block, HDFS stores one copy locally, another within-rack, and a third cross-rack. Under these assumptions, data movement would be dominated by input reads, with read locality optimizations being worthwhile [140]. Once data and network locality has been optimized, HDFS output becomes a target for further improvement, since HDFS output produces cross-rack replication traffic, which has yet to be optimized.

Facebook uses HDFS RAID, which employs Reed-Solomon erasure codes to tolerate 4 missing blocks with 1.4× storage cost [36, 117]. Parity blocks are placed in a non-random fashion. This design contasts with the Hadoop default of three-fold data replication to tolerate 3 missing blocks with 3× the storage cost. Combined with efforts to improve

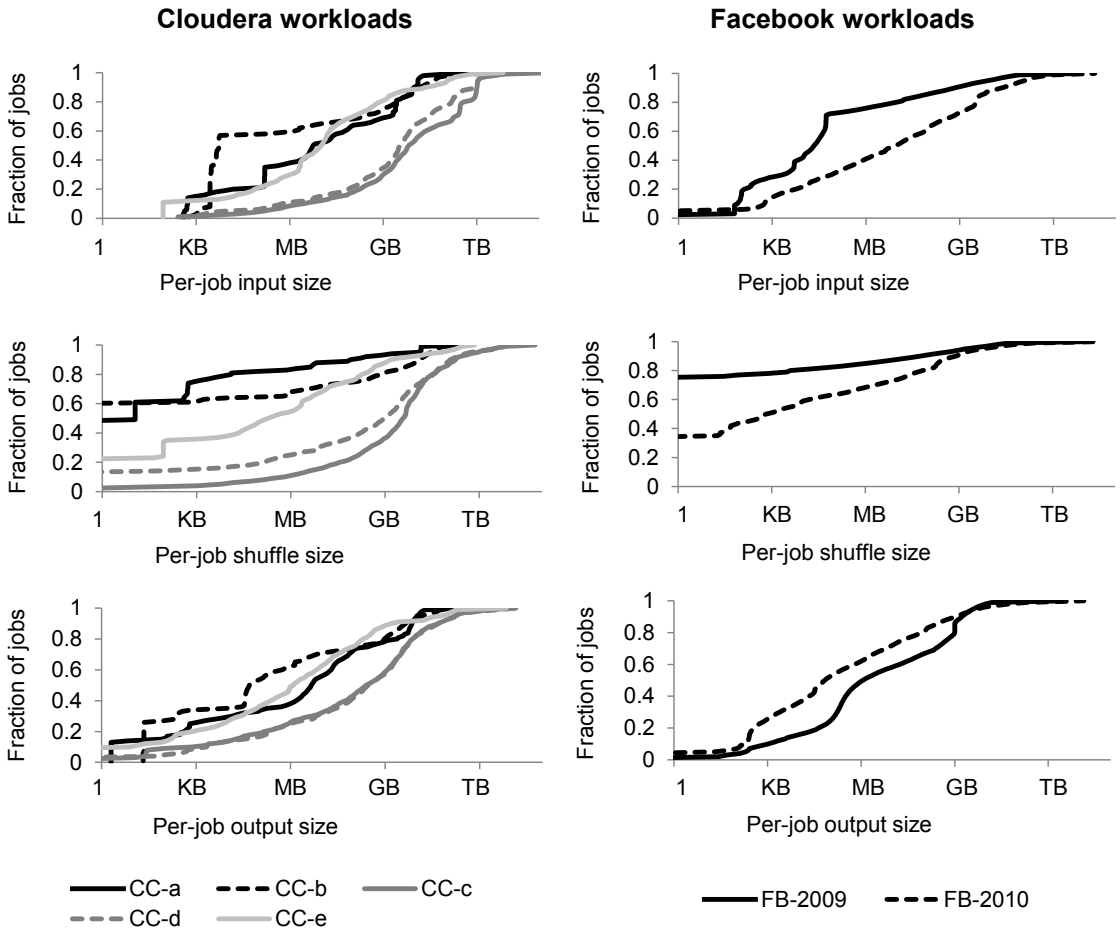



Figure 4.2. Data size for each workload. Showing input, shuffle, and output size per job.

locality, the HDFS RAID creates another environment in which we need to reassess optimization priority between MapReduce API input, shuffle, and output.

### 4.3.2 Per-job data sizes

Figure 4.2 shows the distribution of per-job input, shuffle, and output data sizes for each workload. The median per-job input, shuffle, and output size respective differ by 6, 8, and 4 orders of magnitude. Most jobs have input, shuffle, and output sizes in the MB to GB range. Thus, benchmarks of TB and above [13, 3] capture only a narrow set of input, shuffle, and output patterns.

From 2009 to 2010, the Facebook workloads' per-job input and shuffle size distributions shift right (become larger) by several orders of magnitude, while the per-job output size distribution shifts left (becomes smaller). Raw and intermediate data sets have grown while the final computation results have become smaller. One possible ex-

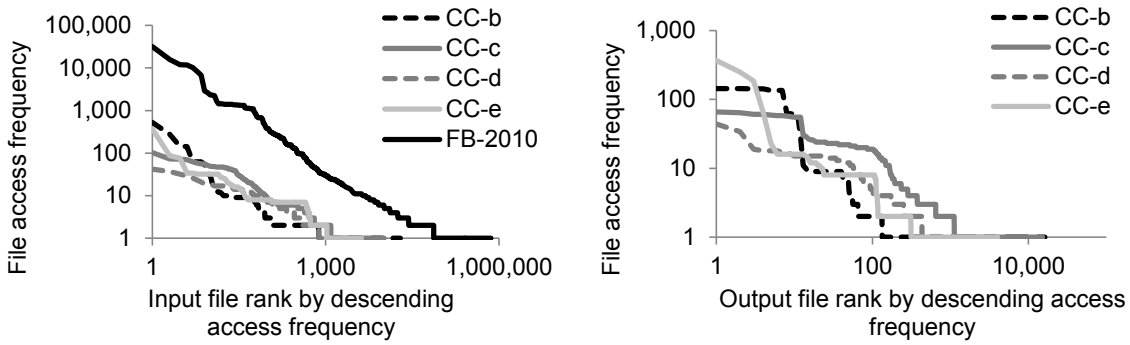



Figure 4.3. Log-log file access frequency vs. rank. Showing Zipf distribution, i.e., straight-lines on a log-log graph, of the same shape (slope) but different sizes (vertical shift) for all workloads.

planation is that Facebook's customer base (raw data) has grown, while the final metrics (output) to drive business decisions have remained the same.

### 4.3.3 Access frequency and intervals

This section analyzes HDFS file access frequency and intervals based on hashed file path names. The `FB-2009` and `CC-a` traces do not contain path names, and the `FB-2010` trace contains path names for input only.

Figure 4.3 shows the distribution of HDFS file access frequency, sorted by rank according to non-decreasing frequency. Note that the distributions are graphed on log-log axes, and form approximate straight lines. This indicates that the file accesses follow a Zipf distribution.

The generalized Zipf distribution has the form in Equation 4.1 [38], where $f(r;\alpha,N)$ is the access frequency of $r^{th}$ ranked file, $N$ is the total size of the distribution, i.e., number of unique files in the workload, and $\alpha$ is the shape parameter of the distribution. Also, $H_{N,\alpha}$ is the $N^{th}$ generalized harmonic number. Figure 4.3 graphs $log(f(r;\alpha,N))$ against $log(r)$. Thus, a linear log-log graph indicates a distribution of the form in Equation 4.1, with $N/H_{N,\alpha}$ being a vertical shift given by the size of the distribution, and $\alpha$ reflects the slope of the line.

$$f(r;\alpha,N) = \frac{N}{r^{\alpha}H_{N,\alpha}} \quad H_{N,\alpha} = \sum_{k=1}^{N}\frac{1}{k^{\alpha}} \tag{4.1}$$

Figure 4.3 indicates that few files account for a very high number of accesses. Thus, *any* data caching policy that includes those files will bring considerable benefit.

Further, the slope, i.e., $\alpha$ parameter of the distributions, are all approximately 5/6, across workloads and for both inputs and outputs. Thus, file access patterns are Zipf

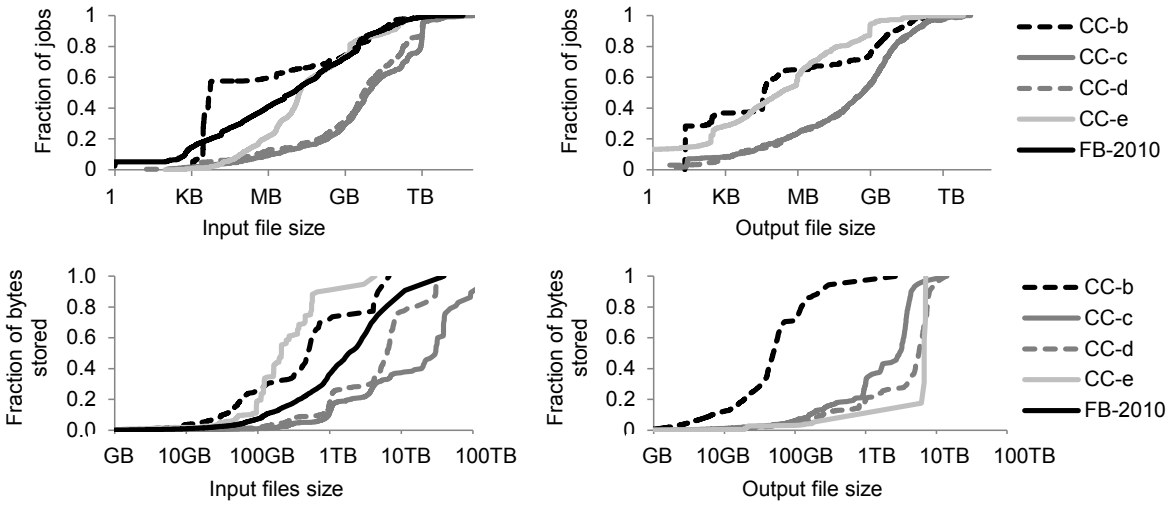



Figure 4.4. Access patterns vs. input/output file size. Showing cummulative fraction of jobs with input/output files of a certain size (top) and cummulative fraction of all stored bytes from input/output files of a certain size (bottom). Note the truncated horizontal axis of the lower graphs.

distributions of the same shape. Figure 4.3 suggests the existence of common computation needs that leads to the same file access behavior across different industries.

The above observations indicate only that caching helps. If there is no correlation between file sizes and access frequencies, maintaining cache hit rates would require caching a fixed fraction of bytes stored. This design is not sustainable, since caches intentionally trade capacity for performance, and cache capacity grows slower than full data capacity. Fortunately, further analysis suggests more viable caching policies.

Figure 4.4 shows data access patterns plotted against file sizes. The distributions for fraction of jobs versus file size vary widely (top graphs), but converge in the upper right corner. In particular, 90% of jobs access files of less than a few GBs (note the log-scale horizontal axis). These files account for up to only 16% of bytes stored (bottom graphs). Thus, a viable cache policy would be to cache files whose size is less than a threshold. This policy would allow cache capacity growth rates to be detached from the growth rate in data.

Further analysis also suggest cache eviction policies. Figure 4.5 indicates the distribution of time intervals between data re-accesses. 75% of the re-accesses take place within 6 hours. Thus, a possible cache eviction policy would be to evict entire files that have not been accessed for longer than a workload specific threshold duration.

Figure 4.6 further shows that up to 78% of jobs involve data re-accesses (`CC-c`, `CC-d`, `CC-e`), while for other workloads, the fraction is lower. Thus, the same cache eviction policy potentially translates to different benefits for different workloads.

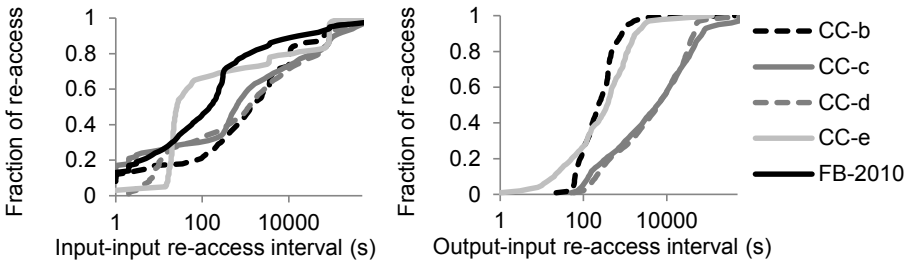



Figure 4.5. Data re-accesses intervals. Showing interval between when an input file is re-read (left), and when an output is re-used as the input for another job (right).

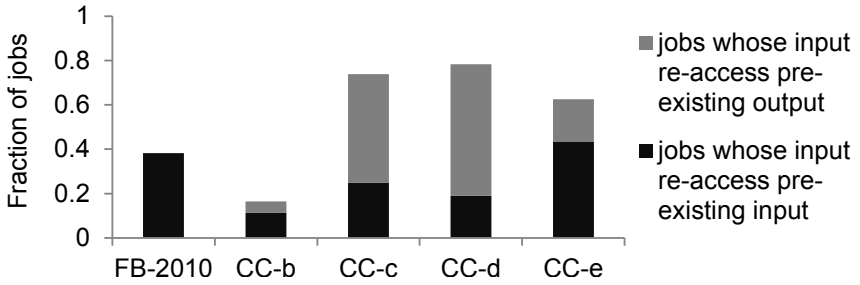



Figure 4.6. Fraction of jobs that read pre-existing input path. Note that output path information is missing from `FB-2010`.

## 4.4 Workload Variation Over Time

The intensity of a MapReduce workload depends on the job submission rate, as well as the computation and data access patterns of the jobs that are submitted. System occupancy depends on the combination of these multiple time-varying dimensions. This section looks at workload variation over a week (Section 4.4.1), quantifies burstiness, a common feature for all workloads (Section 4.4.2), and computes temporal correlations between different workload dimensions (Section 4.4.3).

### 4.4.1 Weekly time series

Figure 4.7 depicts the time series of four dimensions of workload behavior over a week. The first three columns respectively represent the cumulative job counts, amount of I/O (again counted from MapReduce API), and computation time of the jobs submitted in that hour. The last column shows cluster utilization, which depends on the cluster hardware and execution environment, and reflects how the cluster services the submitted workload describes by the preceding columns. Where possible, we try to have the y-axes of the graphs have the same range to enable easy visual comparisons. However, the difference in the scale of the workloads means that graphing all workloads on the same axes range would hide variations in the smaller workloads.

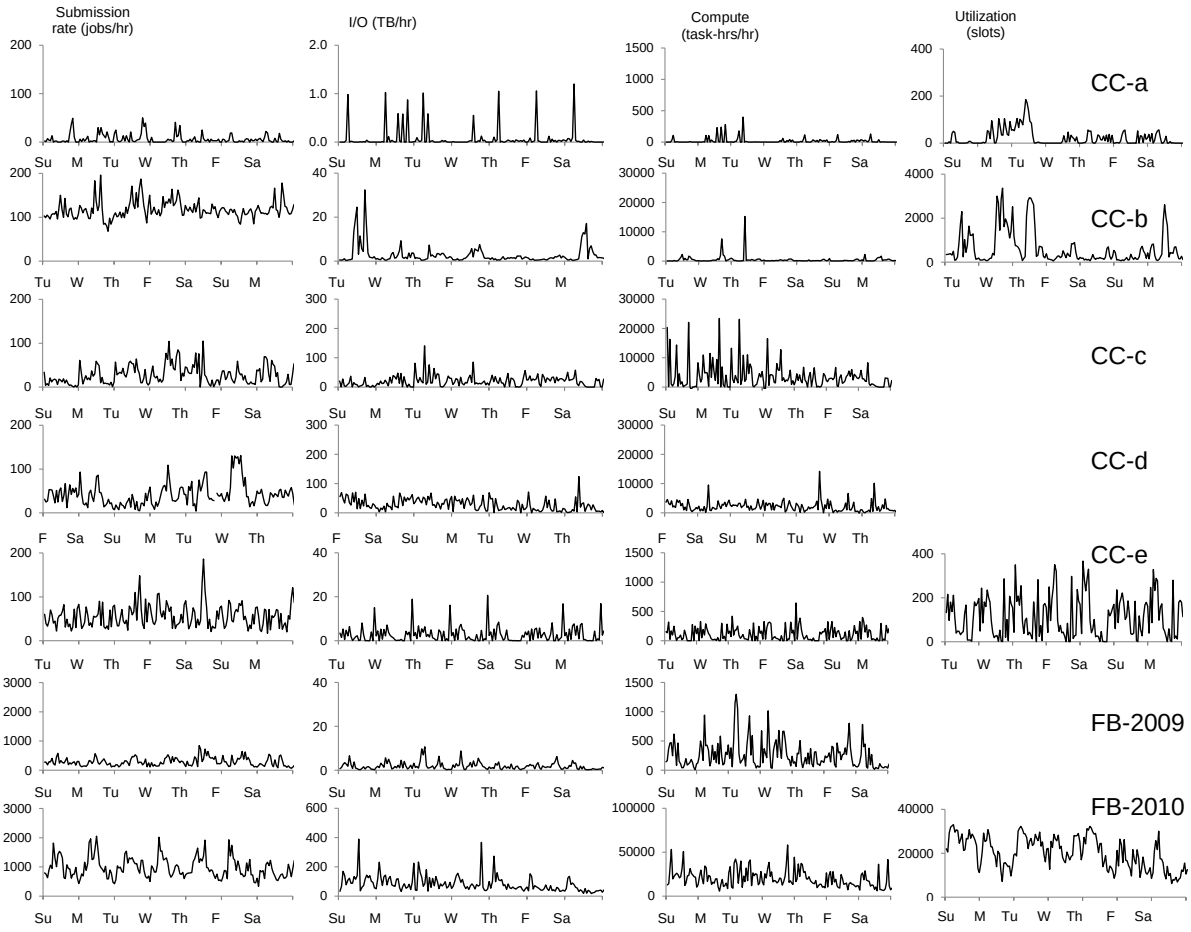



Figure 4.7. Workload behavior over a week. From left to right: (1) Jobs submitted per hour. (2) Aggregate I/O (i.e., input + shuffle + output) size of jobs submitted. (3) Aggregate map and reduce task time in task-hours of jobs submitted. (4) Cluster utilization in average active slots. From top row to bottom, showing `CC-a`, `CC-b`, `CC-c`, `CC-d`, `CC-e`, `FB-2009`, and `FB-2010` workloads. Note that for `CC-c`, `CC-d`, and `FB-2009`, the utilization data is not available from the traces. Also note that not all workloads begin on the same day of the week due to short, week-long trace lengths (`CC-b` and `CC-e`), or gaps from missing data in the trace (`CC-d`).

The first feature to observe in the graphs of Figure 4.7 is that noise is high. This means that even though the number of jobs submitted is known, it is challenging to predict how many I/O and computation resources will be needed as a result. Also, standard signal process methods to quantify the signal to noise ratio would be challenging to apply to these time series, since neither the signal nor noise models are known.

Some workloads exhibit daily diurnal patterns, revealed by Fourier analysis, and for some cases, are visually identifiable (e.g., jobs submission for `FB-2010`, utilization for `CC-e`). In Section 4.6, we combine this observation with several others to speculate that there is an emerging class of interactive and semi-streaming workloads.

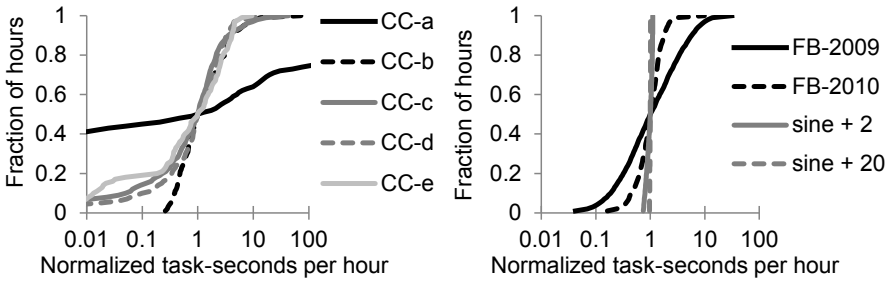



Figure 4.8. Workload burstiness. Showing cummulative distribution of task-time (sum of map time and reduce time) per hour. To allow comparison between workloads, all values have been normalized by the median task-time per hour for each workload. For comparison, we also show burstiness for artificial sine submit patterns, scaled with min-max range the same as mean (sine + 2) and 10% of mean (sine + 20).

### 4.4.2 Burstiness

Another feature of Figure 4.7 is the bursty submission patterns in all dimensions. Burstiness is an often discussed property of time-varying signals, but it is not precisely measured. A commonly used metric is the peak-to-average ratio, even though, strictly speaking, it fails to cover the temporal component of bursts. There are also domain-specific metrics, such as for bursty packet loss on wireless links [122]. Here, we extend the concept of peak-to-average ratio to quantify a key workload property: burstiness.

We start defining burstiness first by using the median rather than the arithmetic mean as the measure of "average". Median is statistically robust against data outliers, i.e., extreme but rare bursts [69]. For two given workloads with the same median load, the one with higher peaks, that is, a higher peak-to-median ratio, is more bursty. We then observe that the peak-to-median ratio is the same as the $100^{th}$-percentile-to-median ratio. While the median is statistically robust to outliers, the $100^{th}$-percentile is not. This implies that the $99^{th}$, $95^{th}$, or $90^{th}$-percentile should also be calculated. We extend this line of thought and compute the general $n^{th}$-percentile-to-median ratio for a workload. We can graph this vector of values, with $\frac{n^{th}-percentile}{median}$ on the x-axis, versus $n$ on the y-axis. The resultant graph can be interpreted as a cumulative distribution of arrival rates per time unit, normalized by the median arrival rate. This graph indicate the burstiness of the time series. A more horizontal line corresponds to a more bursty workload; a vertical line represents a workload with a constant arrival rate.

Figure 4.8 graphs this metric for one of the dimensions of our workloads. We also graph two different sinusoidal signals to illustrate how common signals appear under this burstiness metric. Figure 4.8 shows that for all workloads, the highest and lowest submission rates are orders of magnitude from the median rate. This indicates a level of burstiness far above the workloads examined by prior work, which have more regular diurnal patterns [126, 89]. For the workloads here, scheduling and task placement policies

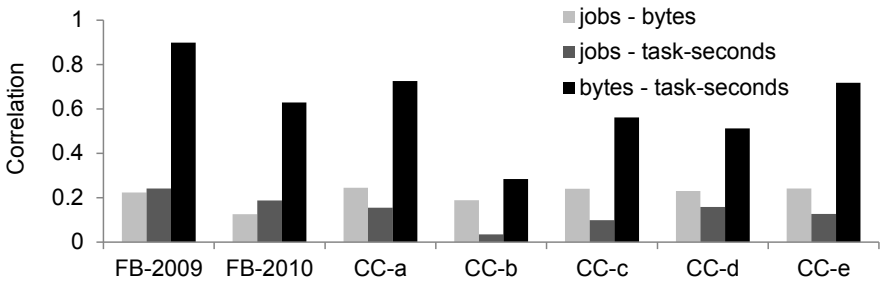



Figure 4.9. Correlation between different submission pattern time series. Showing pair-wise correlation between jobs per hour, (input + shuffle + output) bytes per hour, and (map + reduce) task times per hour.

will be essential under high load. Conversely, mechanisms for conserving energy would be beneficial during periods of low utilization.

For the Facebook workloads, over a year, the peak-to-median-ratio dropped from 31:1 to 9:1, accompanied by more internal organizations adopting MapReduce. This shows that a decrease in burstiness is correlated with multiplexing many workloads, i.e., running on the same clusters workloads from more organizations. In absolute terms, however, the workload remains bursty.

### 4.4.3 Time series correlations

We also computed the correlation between the workload submission time series in all three dimensions, shown in Figure 4.9. The average temporal correlation between job submit and data size is 0.21; for job submit and compute time it is 0.14; for data size and compute time it is 0.62. The correlation between data size and compute time is by far the strongest. We can visually verify this by the $2^{nd}$ and $3^{rd}$ columns for `CC-e` in Figure 4.9. This indicates that MapReduce workloads remain data-centric rather than compute-centric. Also, schedulers and load balancers need to consider dimensions beyond number of active jobs.

## 4.5 Computation Patterns

MapReduce is designed as a combined storage and computation system. This section examines computation patterns by task times (Section 4.5.1), task granularity (Section 4.5.2), and common job types by both job names (Section 4.5.3.1) and a multi-dimensional analysis (Section 4.5.3.2).

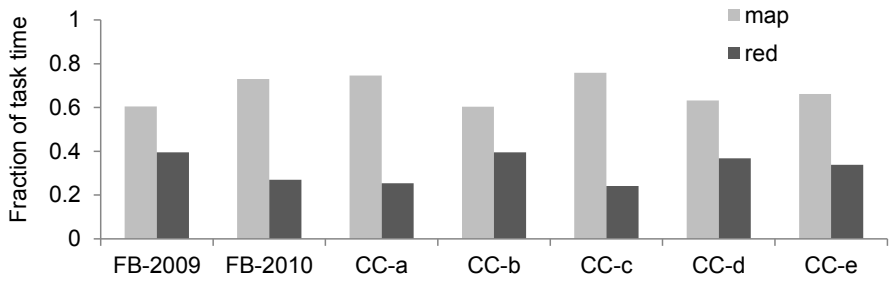



Figure 4.10. Aggregate fraction of all map and reduce task times for each workload.

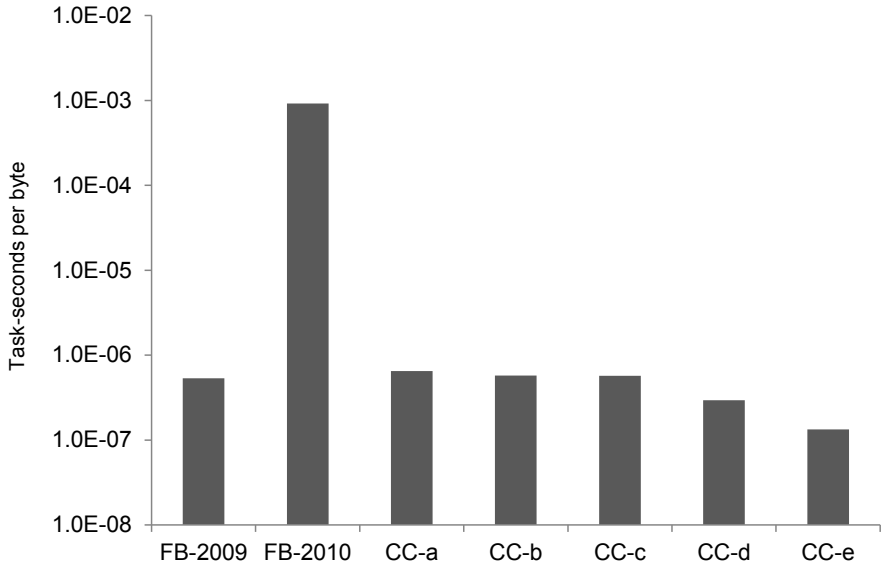



Figure 4.11. Task-seconds per byte for each workload.

### 4.5.1 Aggregate task times

Figure 4.10 shows the aggregate map and reduce task durations for each workload. On average, workloads spend 68% of time in map and 32% of time in reduce. For Facebook, the fraction of time spent mapping increases by 10% over a year. Thus, a design priority should be to optimize map task components, such as read locality, read bandwidth, and map output combiners.

Figure 4.11 shows for each workload the ratio of aggregate task durations (map time + reduce time) over the aggregate bytes (input + shuffle + output). This ratio aims to capture the amount of computation per data in the absence of CPU/disk/network level logs. The unit of measurement is task-seconds per byte. Task-seconds measures the computation time of multiple tasks, e.g., two tasks of 10 seconds equals 20 task-seconds. This is a good unit of measurement if parallelization overhead is small; it approximates the amount of computational resources required by a job that is agnostic to the degree of

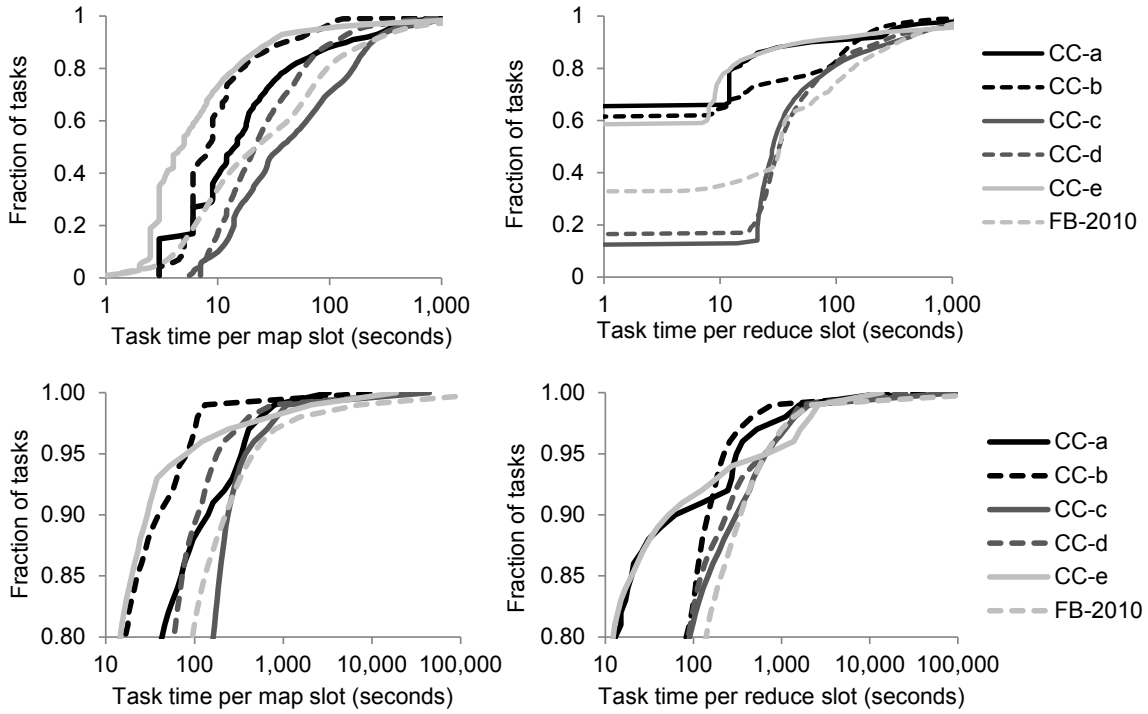



Figure 4.12. Task lengths per workload. Showing the entire distribution (top) and top 20% of the distribution (bottom). Discontinuity at the y-axis for reduce task lengths is due to map-only jobs, i.e., jobs with 0 reduce task times.

parallelism (e.g., $X$ task-seconds divided into $T_1$ tasks equals to $X$ task-seconds divided into $T_2$ tasks).

Figure 4.11 shows that the ratio of computation per data ranges from $1 \times 10^{-7}$ to $7 \times 10^{-7}$ task-seconds per byte, with the `FB-2010` workload having $9 \times 10^{-4}$ task-seconds per byte. Task-seconds per byte clearly separates the workloads into two categories of "compute heavy" workloads (FB-2010) and "compute-light" workloads (all the others). A balanced system should be provisioned specifically to service the task-seconds per byte of a particular workload. Existing hardware benchmarks should also probe a given value of this metric to be more relevant to MapReduce-like systems that combine data movements and computation.

### 4.5.2 Variance in task times and task granularity

Many MapReduce workload management mechanisms make decisions based on task-level information [140, 27, 25]. The underlying assumption is that different jobs break down into tasks in a regular fashion, with all tasks from all jobs being roughly "equal". The default policy in Hadoop MapReduce seeks to achieve this by assigning one map task per HDFS block of input, and one reduce task per 1GB of input. Many MapReduce operators

override this policy, both intentionally and accidentally. Therefore, it is important to empirically verify the assumption of regular task granularity.

Figure 4.12 shows the cummulative distribution of task durations per workload. The distribution is long tailed. Approximately 50% of the tasks have durations of less than a minute. The remaining tasks have durations of up to hours. Thus, the traces do not support the assumption that tasks are regularly sized.

Absent an enforcement of task size, any task-level scheduling or placement decisions are likely to be sub-optimal and prone to be undermined. For example, the Hadoop fair scheduler [140] can be undermined in the following way. The fair scheduler ensures a fair share of task slots. An operator with a very large job could divide her job into very large tasks. During job submission troughs, this job would consume an increasingly large share of the cluster. When job submission peaks again, the large tasks would be incomplete. Absent preemptive task termination, the job with large tasks would have circumvented the intended fair share constraints. Furthermore, preemptive termination of long running tasks potentially results in large amounts of wasted work. MapReduce workload managers should decompose jobs into regularly sized tasks in addition to optimizing execution scheduling and placement.

### 4.5.3 Common job types

There are two complementary ways of grouping jobs:

1. By the job names submitted to MapReduce, which serves as a qualitative proxy for unavailable proprietary code (Section 4.5.3.1), and
2. By the multi-dimensional job description according to per-job data sizes, duration, and task times, which serve as a quantitative proxy (Section 4.5.3.2).

#### 4.5.3.1 By job names

Job names are user-supplied strings recorded by MapReduce. Some computation frameworks built on top of MapReduce, such as Hive [128], Pig [99], and Oozie [2] generate the job names automatically. MapReduce does not currently impose any structure on job names. To simplify analysis, we focus on the first word of job names, ignoring any capitalization, numbers, or other symbols.

Figure 4.13 shows the most frequent first words in job names for each workload, weighted by number of jobs, the amount of I/O, and task-time. The `FB-2010` trace does not have this information. The top figure shows that the top handful of words account for a dominant majority of jobs. When these names are weighted by I/O, Hive queries such as `insert` and other data-centric jobs such as data extractors dominate; when weighted by task-time, the pattern is similar, unsurprising given the correlation between I/O and task-time.

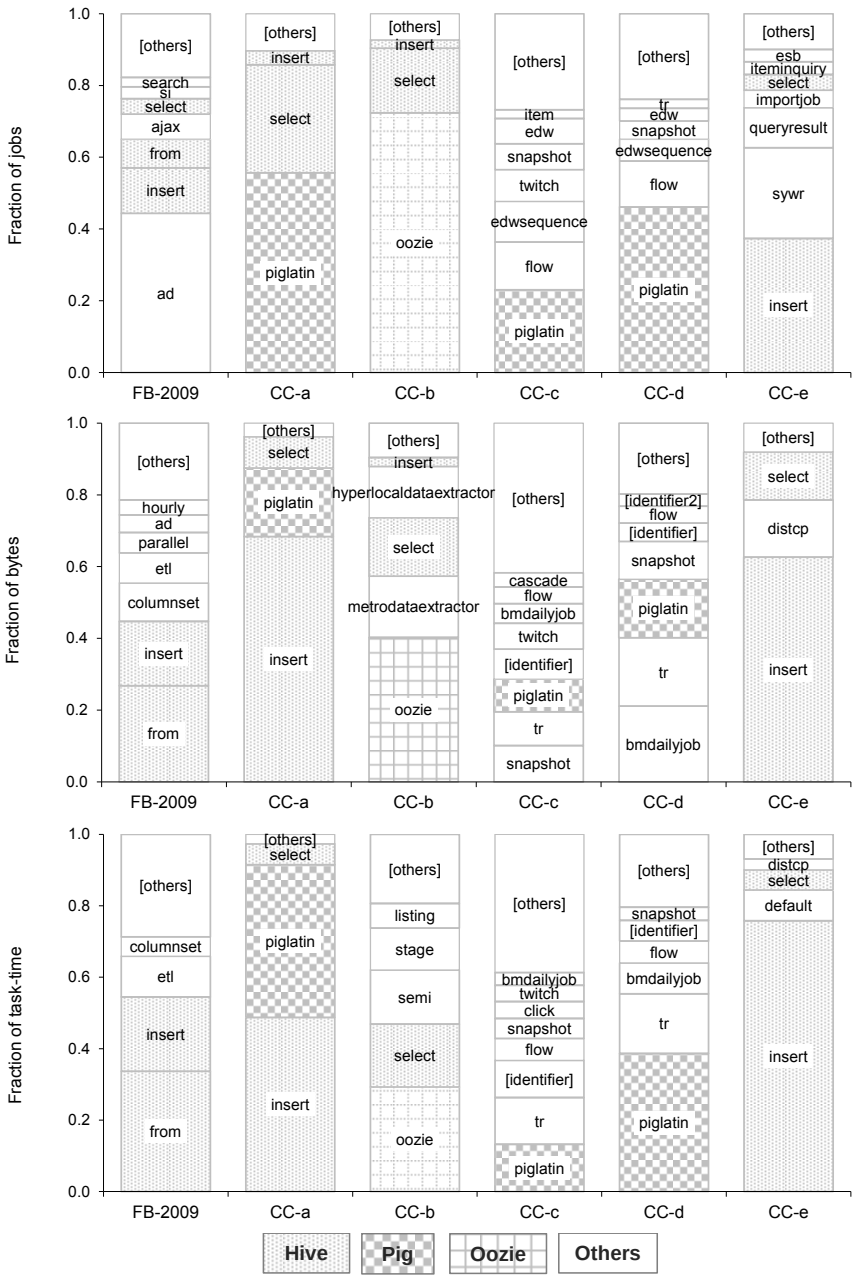



Figure 4.13. The first word of job names for each workload, weighted by the number of jobs beginning with each word (top), total I/O in bytes (middle), and map/reduce task-time (bottom). For example, 44% of jobs in the `FB-2009` workload have a name beginning with "ad", a further 12% begin with "insert"; 27% of all I/O and 34% of total task-time comes from jobs with names that begin with "from" (middle and bottom). The `FB-2010` trace did not contain job names.

Figure 4.13 also implies that each workload consists of only a small number of common computation types. The reason is that job names are either automatically generated, or

assigned by human operators using informal but common conventions. Thus, job names beginning with the same word likely performs similar computation. The small number of computation types represent targets for static or even manual optimization. This would greatly simplify workload management problems, such as predicting job duration or resource use, and optimizing scheduling, placement, or task granularity.

Each workload services only a small number of MapReduce frameworks: Hive, Pig, Oozie, or similar layers on top of MapReduce. Figure 4.13 shows that for all workloads, two frameworks account for a dominant majority of jobs. There is ongoing research to achieve well-behaved multiplexing between different frameworks [71]. The data here suggests that multiplexing between two or three frameworks already covers the majority of jobs in all workloads here. We believe this observation to remain valid in the future. The reason is that as new frameworks develop, enterprise MapReduce users are likely to converge on a small set of mature and well-supported frameworks for business critical computations.

#### 4.5.3.2 By multi-dimensional job behavior

Another way to group jobs is by their multi-dimensional behavior. Each job can be represented as a six-dimensional vector described by input size, shuffle size, output size, job duration, map task time, and reduce task time. One way to group similarly behaving jobs is to find clusters of vectors close to each other in the six-dimensional space. We use a standard data clustering algorithm, k-means [21]. K-means enables quick analysis of a large number of data points and facilitates intuitive labeling and interpretation of cluster centers [44, 94, 43].

We use a standard technique to choose $k$, the number of job type clusters for each workload: increment $k$ until there is diminishing rate in the decrease of intra-cluster variance, i.e., residual variance. Our previous work [44, 43] contains additional details of this methodology.

Table 4.3 summarizes our k-means analysis results. We have assigned labels using common terminology to describe the one or two data dimensions that separate job categories within a workload. A system optimizer would use the full numerical descriptions of cluster centroids.

We see that jobs touching <10GB of total data make up >92% of all jobs. These jobs are capable of achieving interactive latency for analysts, i.e., durations of less than a minute. The dominance of these jobs validate research efforts to improve the scheduling time and the interactive capability of large scale computation frameworks [76, 91, 37].

The dominance of small jobs complicates efforts to rein in stragglers [26], tasks that execute significantly slower than other tasks in a job and delay job completion. Comparing the job duration and task time columns indicate that small jobs contain only a handful of small tasks, sometimes a single map task and a single reduce task. Having few comparable tasks makes it difficult to detect stragglers, and also blurs the definition of a straggler. If the only task of a job runs slowly, it becomes impossible to tell whether the

| | # Jobs | Input | Shuffle | Output | Duration | Map time | Reduce time | Label |
|---|---|---|---|---|---|---|---|---|
| CC-a | 5,525 | 51 MB | 0 | 3.9 MB | 39 sec | 33 | 0 | Small jobs |
| | 194 | 14 GB | 12 GB | 10 GB | 35 min | 65,100 | 15,410 | Transform |
| | 31 | 1.2 TB | 0 | 27 GB | 2 hrs 30 min | 437,615 | 0 | Map only, huge |
| | 9 | 273 GB | 185 GB | 21 MB | 4 hrs 30 min | 191,351 | 831,181 | Transform and aggregate |
| CC-b | 21,210 | 4.6 KB | 0 | 4.7 KB | 23 sec | 11 | 0 | Small jobs |
| | 1,565 | 41 GB | 10 GB | 2.1 GB | 4 min | 15,837 | 12,392 | Transform, small |
| | 165 | 123 GB | 43 GB | 13 GB | 6 min | 36,265 | 31,389 | Transform, medium |
| | 31 | 4.7 TB | 374 MB | 24 MB | 9 min | 876,786 | 705 | Aggregate and transform |
| | 3 | 600 GB | 1.6 GB | 550 MB | 6 hrs 45 min | 3,092,977 | 230,976 | Aggregate |
| CC-c | 19,975 | 5.7 GB | 3.0 GB | 200 MB | 4 min | 10,933 | 6,586 | Small jobs |
| | 477 | 1.0 TB | 4.2 TB | 920 GB | 47 min | 1,927,432 | 462,070 | Transform, light reduce |
| | 246 | 887 GB | 57 GB | 22 MB | 4 hrs 14 min | 569,391 | 158,930 | Aggregate |
| | 197 | 1.1 TB | 3.7 TB | 3.7 TB | 53 min | 1,895,403 | 886,347 | Transform, heavy reduce |
| | 105 | 32 GB | 37 GB | 2.4 GB | 2 hrs 11 min | 14,865,972 | 36,9846 | Aggregate, large |
| | 23 | 3.7 TB | 562 GB | 37 GB | 17 hrs | 9,779,062 | 14,989,871 | Long jobs |
| | 7 | 220 TB | 18 GB | 2.8 GB | 5 hrs 15 min | 66,839,710 | 758,957 | Aggregate, huge |
| CC-d | 12,736 | 3.1 GB | 753 MB | 231 MB | 67 sec | 7,376 | 5,085 | Small jobs |
| | 214 | 633 GB | 2.9 TB | 332 GB | 11 min | 544,433 | 352,692 | Expand and aggregate |
| | 162 | 5.3 GB | 6.1 TB | 33 GB | 23 min | 2,011,911 | 910,673 | Transform and aggregate |
| | 128 | 1.0 TB | 6.2 TB | 6.7 TB | 20 min | 847,286 | 900,395 | Expand and Transform |
| | 43 | 17 GB | 4.0 GB | 1.7 GB | 36 min | 6,259,747 | 7,067 | Aggregate |
| CC-e | 10,243 | 8.1 MB | 0 | 970 KB | 18 sec | 15 | 0 | Small jobs |
| | 452 | 166 GB | 180 GB | 118 GB | 31 min | 35,606 | 38,194 | Transform, large |
| | 68 | 543 GB | 502 GB | 166 GB | 2 hrs | 115,077 | 108,745 | Transform, very large |
| | 20 | 3.0 TB | 0 | 200 B | 5 min | 137,077 | 0 | Map only summary |
| | 7 | 6.7 TB | 2.3 GB | 6.7 TB | 3 hrs 47 min | 335,807 | 0 | Map only transform |
| FB-2009 | 1,081,918 | 21 KB | 0 | 871 KB | 32 s | 20 | 0 | Small jobs |
| | 37,038 | 381 KB | 0 | 1.9 GB | 21 min | 6,079 | 0 | Load data, fast |
| | 2,070 | 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 |
| | 6,035 | 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 |
| FB-2010 | 1,145,663 | 6.9 MB | 600 B | 60 KB | 1 min | 48 | 34 | Small jobs |
| | 7,911 | 50 GB | 0 | 61 GB | 8 hrs | 60,664 | 0 | Map only transform, 8 hrs |
| | 779 | 3.6 TB | 0 | 4.4 TB | 45 min | 3,081,710 | 0 | Map only transform, 45 min |
| | 670 | 2.1 TB | 0 | 2.7 GB | 1 hr 20 min | 9,457,592 | 0 | Map only aggregate |
| | 104 | 35 GB | 0 | 3.5 GB | 3 days | 198,436 | 0 | Map only transform, 3 days |
| | 11,491 | 1.5 TB | 30 GB | 2.2 GB | 30 min | 1,112,765 | 387,191 | Aggregate |
| | 1,876 | 711 GB | 2.6 TB | 860 GB | 2 hrs | 1,618,792 | 2,056,439 | Transform, 2 hrs |
| | 454 | 9.0 TB | 1.5 TB | 1.2 TB | 1 hr | 1,795,682 | 818,344 | Aggregate and transform |
| | 169 | 2.7 TB | 12 TB | 260 GB | 2 hrs 7 min | 2,862,726 | 3,091,678 | Expand and aggregate |
| | 67 | 630 GB | 1.2 TB | 140 GB | 18 hrs | 1,545,220 | 18,144,174 | Transform, 18 hrs |

Table 4.3. Job types in each workload as identified by k-means clustering, with cluster sizes, centers, and labels. Map and reduce time are in task-seconds, i.e., a job with 2 map tasks of 10 seconds each has map time of 20 task-seconds. Note that the small jobs dominate all workloads. We ran k-means with 100 random instantiations of cluster centers, which averages to over 1 bit of randomness in each of the 6 data dimensions.

task is inherently slow, or abnormally slow. The importance of stragglers as a problem also requires re-assessment. Any stragglers would seriously hamper jobs that have a single wave of tasks. However, if it is the case that stragglers occur randomly with a fixed probability, fewer tasks per job means only a few jobs would be affected. We do not yet know whether stragglers occur randomly. Finding out would involve detecting straggler tasks, then investigating whether they distribute uniformly over time, between jobs, and across different machines in the cluster.

Interestingly, map functions in some jobs aggregate data, reduce functions in other jobs expand data, and many jobs contain data transformations in either stages. Such data ratios reverse the original intuition behind map functions as expansions and reduction functions as aggregates [53].

Also, map-only jobs appear in all but two workloads. They form 7% to 77% of all bytes, and 4% to 42% of all task times in their respective workloads. Some are launcher

jobs from Oozie, a system that manages Hadoop workflows consisting of many mutually dependent jobs. Others are maintenance jobs that operate on very little data. Compared with other jobs, map-only jobs benefit less from datacenter networks optimized for shuffle patterns [16, 17, 66, 47].

The FB-2009 and FB-2010 workloads in Table 4.3 show that job types at Facebook changed significantly over one year. The small jobs remain, and several kinds of map-only jobs remain. However, the job profiles changed in several dimensions. Thus, for Facebook, any policy parameters need to be periodically revisited.

## 4.6 Data Analysis Trends

The non-trivial workloads we analyzed and our conversations with Facebook and Cloudera provide the opportunity to speculate on MapReduce-related data analysis trends, subject to verification as more comprehensive data becomes available. Overall, the trends reflect a desire for timely, high quality insights, extracted from growing and complex data sets, and using as little technical expertise as possible.

**Increasing semi-streaming analysis.** Streaming analysis describes continuous computation processes, which often updates some time-aggregation metric [49]. For MapReduce, a common substitute for truly streaming analysis is to setup automated jobs that regularly operates on recent data. Since "recent" data is intentionally smaller than "historical" data, this type of semi-streaming analysis partially accounts for the large number of small jobs in all workloads.

**Increasing interactive analysis.** Another source of small jobs is interactive large-scale data analysis. Small jobs are capable of low latency, compared with larger jobs. Evidence suggesting interactive analysis include diurnal workload patterns, identified by both visual inspection and Fourier analysis, e.g., in jobs submitted per hour for CC-c, FB-2009, and FB-2010; and the presence across all workloads of frameworks such as Hive and Pig, one of whose design goals was ease of use by human analysts familiar with SQL. Not all small jobs in Hive and Pig would be interactively generated. However, over half of small jobs remain after we remove jobs whose names contain time stamps (semi-streaming analysis) and jobs submitted in the early morning (outside work hours).

At Facebook, a common use case is interactive data exploration using ad-hoc queries [40, 127]. Several recent research efforts at web search companies also focus on achieving interactive processing latency [91, 76].

**Frameworks on top of MapReduce.** All workloads include many jobs from Hive, Pig, or Oozie. Frameworks like these allow organizations to devote intellectual resources to understanding the data being processed, versus learning how to process the data using native map and reduce functions. Thus, these frameworks are likely to see increased adoption, both for established users such as Facebook, and especially for emerging users in media, retail, finance, manufacturing, and other industries.

**Storage capacity as a constraint.** The comparison between FB-2009 and FB-2010 workloads reveals orders of magnitude increase in per-job data sizes and the aggregate active data set touched. Such growth compels the heavy use of data compression and RAID-style error correction codes, which are more efficient with regard to storage capacity versus HDFS replication [36, 117]. As data generation and collection capabilities improve across industries, more MapReduce use cases would need to address the storage capacity challenge.

## 4.7 Chapter Conclusions

The analysis in this chapter has several repercussions. First, MapReduce has evolved to the point where performance claims should be qualified with the underlying workload assumptions, e.g., by replaying a suite of workloads. Second, system engineers should regularly re-assess design priorities subject to changing use cases. Prerequisites to these efforts are workload replay tools and a public workload repository, so that engineers can share insights across different enterprise MapReduce deployments.

We have developed and deployed SWIM, a Statistical Workload Injector for MapReduce (https://github.com/SWIMProjectUCB/SWIM/wiki). This is a set of workload replay tools, under the New BSD License, that can pre-populate HDFS using synthetic data, scaled to cluster size, and replay the workload using synthetic MapReduce jobs. The workload replay methodology is further discussed in [44]. The SWIM repository already includes the FB-2009 and FB-2010 workloads. Cloudera has allowed us to contact the end customers directly and seek permission to make public their traces. We invite MapReduce operators to add to the workload repository developed in this dissertation. We hope the repository can help others pursue the future work opportunities identified in the chapter, and contribute to a scientific approach to designing large-scale data processing systems such as MapReduce.

This chapter relates to the remainder of the dissertation as follows. Chapter 5 presents a workload analysis of enterprise network storage systems, and demonstrates that the analysis methodology in this chapter and Chapter 3 generalize beyond MapReduce. enterprise network storage systems are semantically different to MapReduce, and thus require a separate treatment, even though the analysis method is conceptually the same. Still later in the dissertation, Chapters 6 and 7 apply these workload insights to solve two challenging system design problems — MapReduce energy efficiency and TCP incast. Both involve running the workload synthesis, replay, and simulation tools on the traces presented in this chapter.

# Chapter 5

# Workload Analysis and Design Insights - Enterprise Network Storage

*Measure the ground before raising cities.*

— *The Histories of the Kingdoms of Wu and Yue.*

This is the second of two chapters that apply the workload analysis techniques to large-scale data-centric systems. Here, we **analyze two enterprise network storage workloads**. The analysis shares the same goals as Chapter 4, i.e., to discovery design and evaluation insights not otherwise available. While both MapReduce and enterprise network storage are examples of large-scale data-centric systems, they differ in the following ways:

1. Enterprise network storage systems have a more complex structure, with storage servers being distinct from storage clients, and each breaking down further into multiple semantic layers.

2. The workload does not contain computation patterns as that for MapReduce, since the primary purpose of the system is to storage and retrieve data.

3. Enterprise network storage represents an established type of computer system that has grown to become a kind of large-scale data-centric system due to the changing scale and nature of the data it stores. This is unlike MapReduce, which has been purposefully designed as a large scale data processing system.

Hence, this chapter serves as an illustration of the workload analysis beyond MapReduce. The combination of the two chapters demonstrates that our workload analysis methodology can be effective for both emerging and established systems.

The chapter is organized as follows. We begin by motivating the importance of analyzing enterprise network storage workloads (Section 5.1) and describing the traces and methodology extensions in the chapter (Section 5.2). We then discuss workload behavior in terms of client-side data access patterns (Section 5.3) and server-side data access patterns (Section 5.4). We further detail the long-term evolution of data access patterns, which helps increase confidence that the method translates to enterprise network storage use cases beyond the ones examined in this chapter (Section 5.5). We close the chapter by highlighting some enterprise network storage design architecture trends revealed by the production workloads (Section 5.6), and summarizing the broader implications of the chapter with regard to design and evaluation approaches (Section 5.7).

## 5.1 Motivation

Enterprise storage systems are designed around a set of data access patterns. The storage system can be specialized by designing to a specific data access pattern; e.g., a storage system for streaming video supports different access patterns than a document repository. The better the access pattern is understood, the better the storage system design. Insights into access patterns have been derived from the analysis of existing file system workloads, typically through trace analysis studies [15, 31, 100, 115, 133]. While this is the correct general strategy for improving storage system design, past approaches have critical shortcomings, especially given recent changes in technology trends. In this chapter, we present a new design methodology to overcome these shortcomings.

The data stored on enterprise network-attached storage systems is undergoing changes due to a fundamental shift in the underlying technology trends. We have observed three such trends, including:

- *Scale*: Data size grows at an alarming rate [75], due to new types of social, business and scientific applications [123], and the desire to "never delete" data.
- *Heterogeneity*: The mix of data types stored on these storage systems is becoming increasingly complex, each having its own requirements and access patterns [131].
- *Consolidation*: Virtualization has enabled the consolidation of multiple applications and their data onto fewer storage servers [51, 132]. These virtual machines (VMs) also present aggregate data access patterns more complex than those from individual clients.

Better design of future storage systems requires insights into the changing access patterns due to these trends. While past trace studies have been used to derive data access patterns, we believe that they have the following shortcomings:

- *Unidimensional:* Although existing methods analyze many access characteristics, they do so one at a time, without revealing cross-characteristic dependencies.
- *Expertise bias:* Past analyses were performed by storage system designers looking for specific patterns based on prior mental models. This introduces a bias that needs to be revisited based on the new technology trends.

| Client side observations and design implications | Server side observations and design implications |
|---|---|
| 1. Client sessions with IO sizes >128KB are read only or write only. ⇒ Clients can consolidate sessions based on only the read-write ratio. | 7. Files with >70% sequential read or write have no repeated reads or overwrites. ⇒ Servers should delegate sequentially accessed files to clients to improve IO performance. |
| 2. Client sessions with duration >8 hours do ≈10MB of IO. ⇒ Client caches can already fit an entire day's IO. | 8. Engineering files with repeated reads have random accesses. ⇒ Servers should delegate repeatedly read files to clients; clients need to store them in flash or memory. |
| 3. Number of client sessions drops off linearly by 20% from Monday to Friday. ⇒ Servers can get an extra "day" for background tasks by running at appropriate times during week days. | 9. Most files are active (have opens, IO, and metadata access) for only 1-2 hours in a few months. ⇒ Servers can use file idle time to compress or deduplicate to increase storage capacity. |
| 4. Applications with <4KB of IO per file open and many opens of a few files do only random IO. ⇒ Clients should always cache the first few KB of IO per file per application. | 10. All files have either all random access or >70% sequential access. (Seen in past studies too) ⇒ Servers can select the best storage medium for each file based on only access sequentiality. |
| 5. Applications with >50% sequential read or write access entire files at a time. ⇒ Clients can request file prefetch (read) or delegation (write) based on only the IO sequentiality. | 11. Directories with sequentially accessed files almost always contain randomly accessed files as well. ⇒ Servers can change from per-directory placement policy (default) to per-file policy upon seeing any sequential IO to any files in a directory. |
| 6. Engineering applications with >50% sequential read and sequential write are doing code compile tasks, based on file extensions. ⇒ Servers can identify compile tasks; server should cache the output of these tasks. | 12. Some directories aggregate only files with repeated reads and overwrites. ⇒ Servers can delegate these directories entirely to clients, tradeoffs permitting. |

Table 5.1. **Summary of design insights**, separated into insights derived from client access patterns and server access patterns. The workload analysis gives us high confidence that the proposed improvements will bring a benefit. We defer to future work the implementation of these system changes and the quantification of performance gains per workload.

- *Storage server centric:* Past file system studies focused primarily on storage servers. This creates a critical knowledge gap regarding client behavior.

To overcome these shortcomings, we propose a new design methodology backed by the analysis of storage system traces. We present a *method that simultaneously analyzes multiple characteristics and their cross dependencies*. We use a multi-dimensional, statistical correlation technique, called k-means [21], that is completely agnostic to the characteristics of each access pattern and their dependencies. The K-means algorithm can analyze hundreds of dimensions simultaneously, providing added objectivity to our analysis. To further reduce expertise bias, we involve as many relevant characteristics as possible for each access pattern. In addition, we analyze patterns at different granularities (e.g., at the user session, application, file level) on the storage server as well as the client, thus addressing the need for understanding client patterns. The resulting design insights enable policies for building new storage systems.

We analyze two recent network-attached storage file system traces from a production enterprise datacenter. Table 5.1 summarizes our key observations and design implications; they will be detailed later in the chapter. Our methodology leads to observations that would be difficult to extract using past methods. We illustrate two such access patterns, one showing the value of multi-granular analysis (Observation 1 in Table 5.1) and another showing the value of multi-feature analysis (Observation 8).

First, we observe (Observation 1) that *sessions with more than 128KB of data reads*

*or writes are either read-only or write-only.* This observation affects shared caching and consolidation policies across sessions. Specifically, client OSs can detect and co-locate cache sensitive sessions (read-only) with cache insensitive sessions (write-only) using just one parameter (read-write ratio). This improves cache utilization and consolidation (increased density of sessions per server).

Similarly, we observe (Observation 8) that *files with >70% sequential read or sequential write have no repeated reads or overwrites.* This access pattern involves four characteristics: read sequentiality, write sequentiality, repeated read behavior, and overwrite behavior. The observation leads to a useful policy: sequentially accessed files do not need to be cached at the server (no repeated reads), which leads to an efficient buffer cache.

These observations illustrate that our methodology can derive unique design implications that leverage the correlation between different characteristics. To summarize, our contributions are:

- Identify storage system access patterns using a multi-dimensional statistical analysis technique.
- Build a framework for analyzing traces at different granularity levels at both server and client.
- Analyze our specific traces and present the access patterns identified.
- Derive design implications for various storage system components from the access patterns.

## 5.2 Traces and Methodology Extensions

In this section, we describe our analysis method in detail. We start with a description of the traces we analyzed, followed by a description of the access units selected for our study. Next, we describe key steps in our analysis process, including selecting the right features for each access unit, using the k-means data clustering algorithm to identify access patterns, and additional information needed to interpret and generalize the results.

### 5.2.1 Traces Analyzed

We collected Common Internet File System (CIFS) traces from two large-scale, enterprise-class file servers deployed at our corporate datacenters. One server covers roughly 1000 employees in marketing, sales, finance, and other corporate roles. We call this the *corporate trace*. The other server covers roughly 500 employees in various engineering roles. We call this the *engineering trace*. The trace collecting infrastructure is described in [83].

The corporate trace reflects activities on 3TB of active storage from 09/20/2007 to 11/21/2007. It contains activity from many Windows applications. The engineering trace reflects activities on 19TB of active storage from 08/10/2007 to 11/14/2007. It interleaves

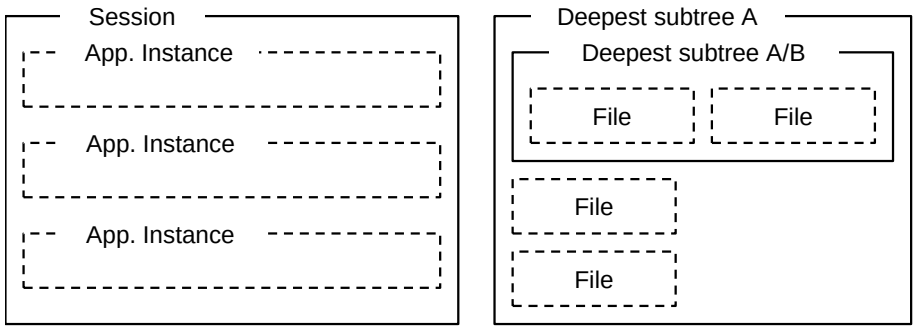



Figure 5.1. **Access units analyzed.** At clients, each session contains many application instances. At servers, each subtree contains many files.

activity from both Windows and Linux applications. In both traces, many clients use virtualization technologies. Thus, we believe we have representative traces with regards to the technology trends in scale, heterogeneity, and consolidation. Also, since protocol-independent users, applications, and stored data remain the primary factors affecting storage system behavior, we believe our analysis is relevant beyond CIFS.

### 5.2.2 Access Units

We analyze access patterns at multiple access units at the server and the client. Selecting access units is subjective. We chose access units that form clear semantic design boundaries. On the client side, we analyze two access units:

- *Sessions*: Sessions reflect aggregate behavior of a user. A CIFS session is bounded by matching session connect and logoff requests. CIFS identifies it by a tuple — {client IP address, session ID}.
- *Application instance*: Analysis at this level leads to application specific optimizations in client VMs. CIFS identifies each application instance by the tuple - {client IP address, session ID, and process ID}.

We also analyzed file open-closes, but obtained uninteresting insights. Hence we omit that access unit from the chapter.

We also examined two server side access units:

- *File*: Analyzing file level access patterns facilitates per-file policies and optimization techniques. Each file is uniquely identified by its full path name.
- *Deepest subtree*: This access unit is identified by the directory path immediately containing the file. Analysis at this level enables per-directory policies.

Figure 5.1 shows the semantic hierarchy among different access units. At clients, each session contains many application instances. At servers, each subtree contains many files.

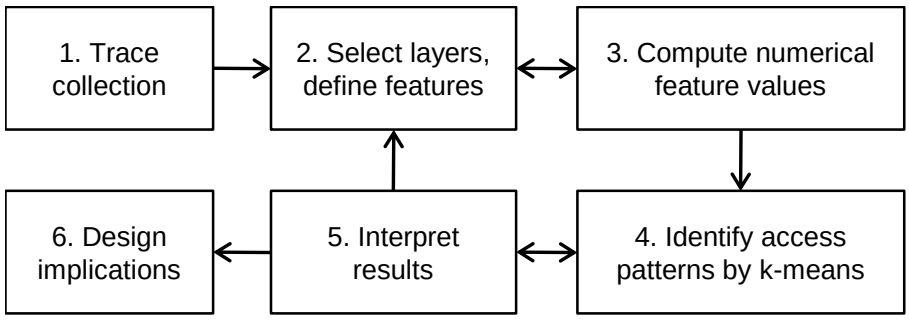



Figure 5.2. **Methodology overview.** The two-way arrows and the loop from Step 2 through Step 5 indicate our many iterations between the steps.

### 5.2.3 Analysis Process

Our method (Figure 5.2) involves the following steps:

1. Collect network storage system traces (Section 5.2.1).
2. Define the descriptive features for each access unit. This step requires domain knowledge about storage systems (Section 5.2.3.1).
3. Extract multiple instances of each access unit, and compute from the trace the corresponding numerical feature values of each instance.
4. Input those values into k-means, a multi-dimensional statistical data clustering technique (Section 5.2.3.2).
5. Interpret the k-means output and derive access patterns by looking at only the relevant subset of features. This step requires knowledge of both storage systems and statistics. We also need to extract considerable additional information to support our interpretations (Section 5.2.3.3).
6. Translate access patterns to design insights.

We give more details about Steps 2, 4, and 5 in the following subsections.

#### 5.2.3.1 Selecting features for each access unit

Selecting the set of descriptive features for each access unit requires domain knowledge about storage systems (Step 2 in Figure 5.2). It also introduces some subjectivity, since the choice of features limits how one access pattern can differ from another. The human designer needs to select some basic features initially, e.g., total IO size and read-write ratio for a file. We will not know whether we have a good set of features until we have completed the entire analysis process. If the analysis results leave some design choice ambiguities, we need to add new features to clarify those ambiguities, again using domain knowledge. For example, for the deepest subtrees, we compute various percentiles (25th,

50th, and 75th) of certain features like read-write ratio because the average value for those features did not clearly separate the access patterns. We then repeat the analysis process using the new feature set. This iterative process leads to a long feature set for all access units, somewhat reducing the subjective bias of a small feature set. We list in Section 5.3 and 5.4 the chosen features for each access unit.

Most of the features used in our analysis are self-explanatory. Some ambiguous or complex features require precise definitions, such as:

*I/O:* We use "I/O" as a substitute for "read and write".

*Sequential reads or writes:* We consider two read or writes requests to be sequential if they are consecutive in time, and the file offset + request size of the first request equals the file offset of the second request. A single read or write request is by definition not sequential.

*Repeated reads or overwrites:* We track accesses at 4KB block boundaries within a file, with the offset of the first block being zero. A read is considered repeated if it accesses a block that has been read in the past half hour. We use an equivalent definition for overwrites.

#### 5.2.3.2 Identifying access patterns via k-means

A key part of our methodology is the k-means multi-dimensional correlation algorithm. We use it to identify access patterns simultaneously across many features (Step 4 in Figure 5.2). Recall from Chapter 4 that k-means is a well-known, statistical correlation algorithm. It identifies sets of data points that congregate around a region in n-dimensional space. These congregations are called *clusters*.

For each access unit, we extract different instances of it from the trace, i.e., all session instances, application instances, files, and directories. For each instance, we compute the numerical values of all its features. This gives us a data array in which each row corresponds to an instance, i.e., a data point, and each column corresponds to a feature, i.e., a dimension. We input the array into k-means, and the algorithm finds the natural clusters across all data points. *We consider all data points in a cluster as belonging to a single equivalence class, i.e., a single access pattern.* The numerical values of the cluster centers indicate the characteristics of each access pattern.

We choose k-means for two reasons. First, k-means is algorithmically simple. This allows rapid processing on large data sets. We used a modified version of the k-means C library [79], in which we made some minor edits to limit the memory footprint when processing large data sizes. Second, k-means leads to intuitive labels of the cluster centers. This helps us translate the statistical behavior extracted from the traces into tangible insights. Thus, we prefer k-means to other clustering algorithms such as hierarchical clustering and k-means derivatives [21].

K-means requires us to specify $k$, the number of clusters. This is a difficult task since we do not know a priori the number of "natural" clusters in the data. As in

Chapter 4, we use a standard technique to set $k$: increment $k$ until there is diminishing rate in the decrease of intra-cluster variance, i.e., residual variance. Further details of the methodology appears in [44, 43].

We get a qualitative measure of the clarity of cluster boundaries by looking at the reassignment of data points as we increment $k$. If the clusters boundaries are clear, i.e., the data points fall into natural clusters, then incrementing $k$ would lead to a single cluster split into two, and the data points in the other clusters remain stationary. Conversely, unclear cluster boundaries are indicated by $N$ to $N + 1$ rearrangements in data points with $N \geq 2$. The cluster boundaries are clear for all the results presented in this chapter. There are one or two instances of a double split, i.e., two clusters split into three, but all three of the clusters are preserved when we further increment $k$. Such a situation represents three roughly equidistant natural clusters being identified. In aggregate, clear cluster boundaries would be indicated by a tree-like structure in the movement of data points as we increase $k$. Figure 5.3 shows such a graphical representation that includes a double split. We verified that such a tree-like structure exists for all the k-means results presented in this chapter.

#### 5.2.3.3 Interpreting and generalizing the results

The k-means algorithm gives us a set of access patterns with various characteristics. We need additional information to understand the significance of the results. This information comes from computing various secondary data outside of k-means analysis (Step 5 in Figure 5.2:

- We gathered the start and end times of each session instance, aggregated by times of the day and days of the week. This gave us insight into how users launch and end sessions.
- We examine filename extensions of files associated with every access pattern belonging to these access units: application instances, files, and deepest subtrees. This information connects the access patterns to more easily recognizable file extensions.
- We perform correlation analysis between the file and deepest subtrees access units. Specifically, we compute the number of files of each file access pattern that is located within directories in each deepest subtree access pattern. This information captures the organizations of files in directories.

Such information gives us a detailed picture about the semantics of the access patterns, resulting in human understandable labels to the access patterns. Such labels help us translate observations to design implications.

Furthermore, after identifying the design implications, we explore if the design insights can be extrapolated to other trace periods and other storage system use cases. We accomplish this by repeating our exact analysis over multiple subsets of the traces, for example, a week's worth of traces at a time. This allow us to examine how our analysis would be different had we obtained only a week's trace. Access patterns that are consis-

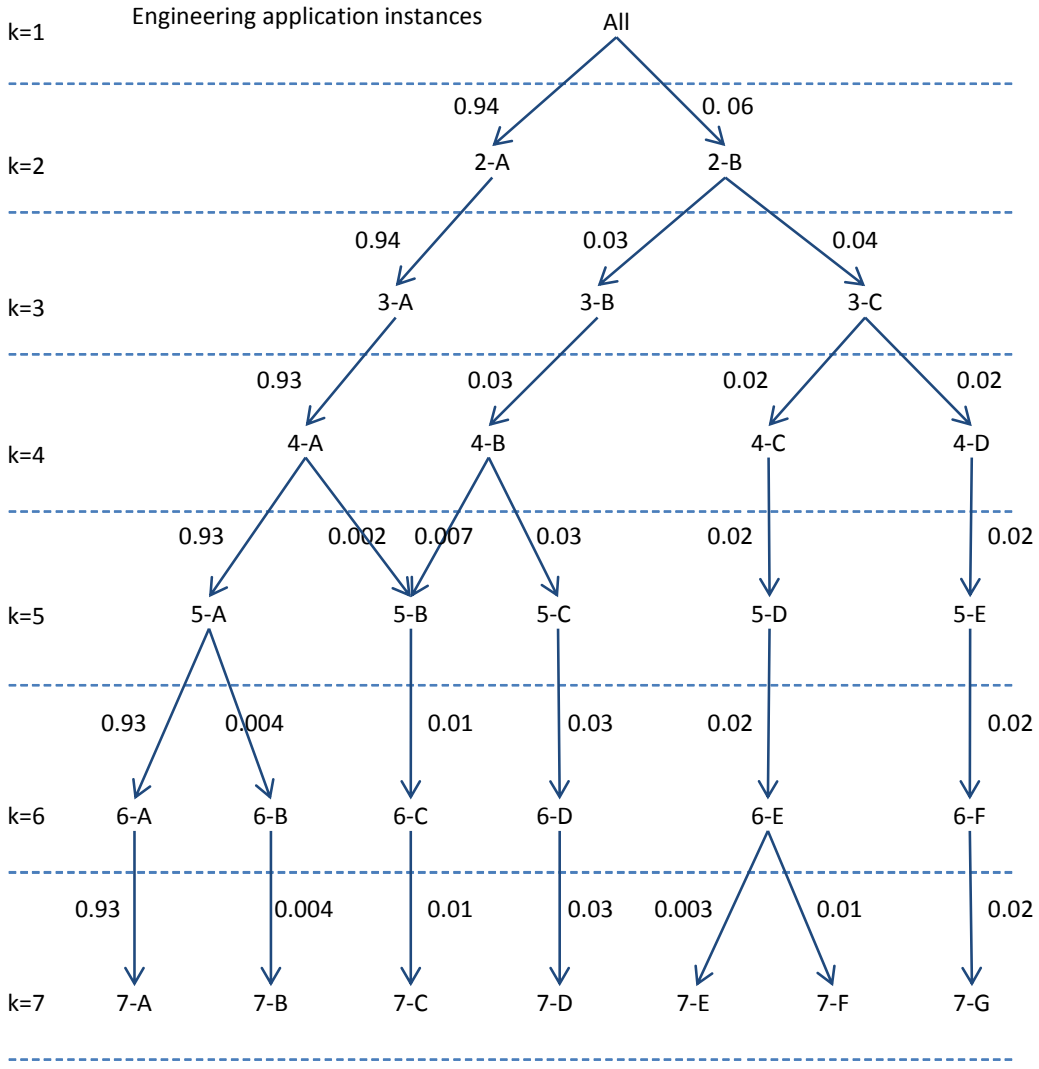



Figure 5.3. **Movement of data points for k-means as we increase $k$.** Different horizontal levels indicate clusters at each $k$. The clusters are represented as nodes in the tree. An edge from node $\alpha$ to $\beta$ represents movement of data points. Edge weights represent the fraction of all data points associated with the movement between clusters.

tent, stable across different weeks would indicate that they are likely to be more general than just our tracing period or our use cases.

| (a). Descriptive features for each session | | |
|---|---|---|
| Duration | Avg. time between I/O requests | Unique trees accessed |
| Total I/O size | Read sequentiality | File opens |
| Read:write ratio by bytes | Write sequentiality | Unique files opened |
| Total I/O requests | Repeated read ratio | Directories accessed |
| Read:write ratio by requests | Overwrite ratio | Application instances seen |
| Total metadata requests | Tree connects | |

| (b). Corporate session access patterns | Full day work | Half day content viewing | Short content viewing | Short content generate | Supporting metadata | Supporting read-write |
|---|---|---|---|---|---|---|
| % of all sessions | 0.5% | 0.7% | 1.2% | 0.2% | 96% | 1.4% |
| Duration | 8 hrs | 4 hrs | 10 min | 70 min | 7 sec | 10 sec |
| Total I/O size | 11 MB | 3 MB | 128 KB | 3 MB | 0 | 420 B |
| Read:write ratio by bytes | 3:2 | 1:0 | 1:0 | 0:1 | 0:0 | 1:1 |
| Metadata requests | 3000 | 700 | 230 | 550 | 1 | 20 |
| Read sequentiality | 70% | 80% | 0% | - | - | 0% |
| Write sequentiality | 80% | - | - | 90% | - | 0% |
| File opens:files | 200:40 | 80:15 | 30:7 | 50:15 | 0:0 | 6:3 |
| Tree connect:Trees | 5:2 | 3:2 | 2:2 | 2:2 | 1:1 | 2:2 |
| Directories accessed | 10 | 7 | 4 | 6 | 0 | 2 |
| Application instances | 4 | 3 | 2 | 2 | 0 | 1 |

| (c). Engineering session access patterns | Full day work | Human edit small files | App. generated backup | Short content generate | Supporting metadata | Machine generated update |
|---|---|---|---|---|---|---|
| % of all sessions | 0.4% | 1.0% | 4.4% | 0.4% | 90% | 3.6% |
| Duration | 1 day | 2 hrs | 1 min | 1 hr | 10 sec | 10 sec |
| Total I/O size | 5 MB | 5 KB | 2 MB | 2 MB | 0 | 36 B |
| Read:write ratio | 7:4 | 1:1 | 1:0 | 0:1 | 0:0 | 1:0 |
| Metadata requests | 1700 | 130 | 40 | 200 | 1 | 0 |
| Read sequentiality | 60% | 0% | 90% | - | - | 0% |
| Write sequentiality | 70% | 0% | - | 90% | - | - |
| File opens:files | 130:20 | 9:2 | 6:5 | 15:6 | 0:0 | 1:1 |
| Tree connect:Trees | 1:1 | 1:1 | 1:1 | 1:1 | 1:1 | 1:1 |
| Directories accessed | 7 | 2 | 1 | 3 | 0 | 1 |
| Application instances | 4 | 2 | 1 | 1 | 0 | 1 |

Table 5.2. **Session access patterns.** (a): Full list of descriptive features. (b) and (c): Short names and descriptions of sessions in each access pattern; listing only the features that help separate the access patterns.

## 5.3 Client Side Access Patterns

This and the coming sections present the access patterns we identified and the accompanying design insights. We discuss client and serve side access patterns (Section 5.3 and 5.4). We also check if these patterns persist across time (Section 5.5).

For each access unit, we list the descriptive features (only some of which help separate access patterns), outline how we derived the high-level name (label) for each access pattern, and discuss relevant design insights.

As mentioned in Section 5.2.2, we analyze sessions and application instances at clients.

### 5.3.1 Sessions

Sessions reflect aggregate behavior of human users. We used 17 features to describe sessions (Table 5.2). The corporate trace has 509,076 sessions, and the engineering trace has 232,033.

In Table 5.2, we provide quantitative descriptions and short names for all the session access patterns. We derive the names from examining the significant features: duration, read-write ratio, and I/O size.

We also looked at the aggregate session start and end times to get additional semantic knowledge about each access pattern. Figures 5.4 and 5.5 show the start and end times for selected session access patterns. The start times of corporate full-day work sessions correspond exactly to the U.S. work day – 9AM start, 12PM lunch, 5PM end. Corporate content generation sessions show slight increase in the evening and towards Friday, indicating rushes to meet daily or weekly deadlines. In the engineering trace, the application-generated backup and machine-generated update sessions depart significantly from human workday and work-week patterns, leading us to label them as application and machine (client OS) generated.

One surprise was that the 'supporting metadata' sessions account for >90% of all sessions in both traces. We believe these sessions are not humanly generated. They last roughly 10 seconds, leaving little time for human mediated interactions. Also, the session start rate averages to roughly one per employee per minute. We are certain that users are not connecting and logging off every minute of the entire day. However, the shape of the start-time graphs has a strong correlation with the human work day and work week. We call these supporting metadata sessions – machine generated in support of human user activities. These metadata sessions form a sort of "background noise" to the storage system. We observe the same background noise at other layers both at clients and servers.

*Observation 1: The sessions with I/O sizes greater than 128KB are either read-only or write-only, except for the full-day work sessions (Table 5.2).* These observations correspond to the "half day content viewing," " short content viewing," and "short content generate" sessions in the corporate trace, as well as "application generated backup" and "short content generate" sessions in the engineering trace. Among these sessions, only read-only sessions utilize buffer cache for repeated reads and prefetches. Write-only sessions only use the cache to buffer writes. Thus, if we have a cache eviction policy that recognizes their write-only nature and releases the buffers immediately on flushing dirty data, we can satisfy many write-only sessions with relatively little buffer cache space. We can attain better consolidation and buffer cache utilization by managing the ratio of co-located read-only and write-only sessions. This insight can be used by virtualization managers and client operating systems to manage a shared buffer cache between sessions. Recognizing such read-only and write-only sessions is easy. Examining a session's total read size and write size reveals their read-only or write-only nature. *Implication 1: Clients can consolidate sessions efficiently based only on the read-write ratio.*

*Observation 2: The "full-day work" sessions do ≈10MB of I/O (Table 5.2).* This

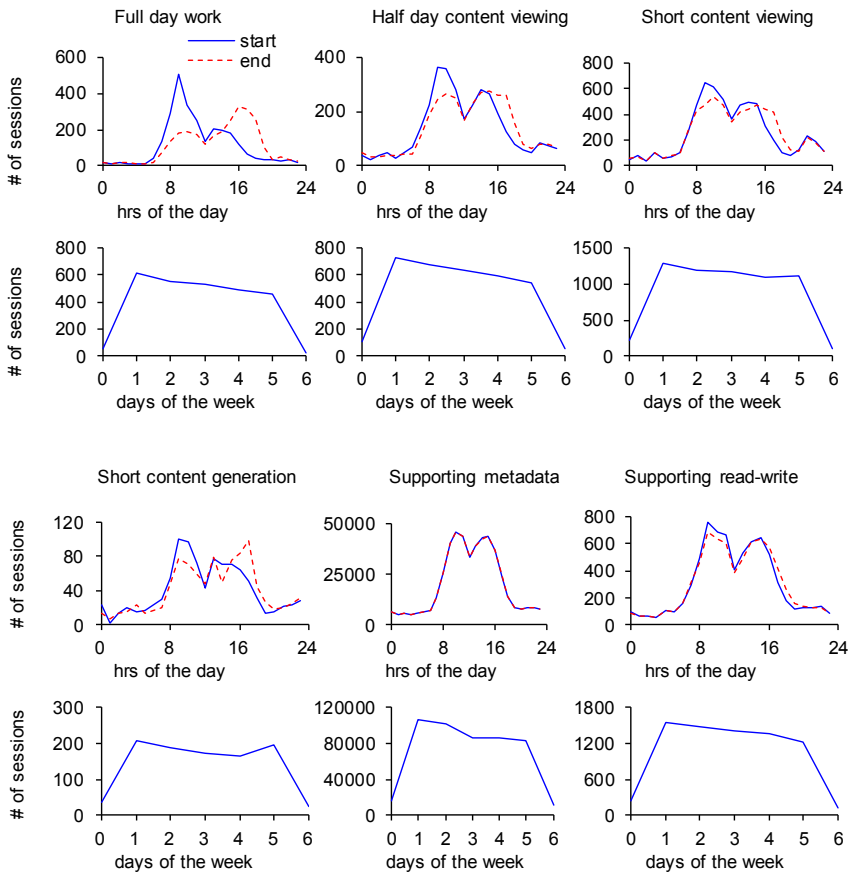



Figure 5.4. **Number of corporate sessions that start or end at a particular time.** Number of session starts and ends versus time of the day (1st and 3rd rows) and session starts versus days of the week (2nd and 4th rows).

means that a client cache of 10s of MB can fit the working set of a day for most sessions. Given the growth of flash devices on clients for caching, despite large-scale consolidation, clients should easily cache a day's worth of data for all users. In such a scenario, most I/O requests would be absorbed by the cache, reducing network latency and bandwidth utilization, and load on the server. Moreover, complex cache eviction algorithms are unnecessary. *Implication 2: Clients caches can already fit an entire day's I/O.*

*Observation 3: The number of human-generated sessions and supporting sessions peaks on Monday and decreases steadily to 80% of the peak on Friday (Figures 5.4 and 5.5).* There is considerable "slack" in the server load during evenings, lunch times, and even during working hours. This implies that the server can perform background tasks such as consistency checks, maintenance, or compression/deduplication, at appropriate times during the week. A simple count of active sessions can serve as an effective start and stop signal. By computing the area under the curve for session start times by days of the week, we estimate that background tasks can squeeze out roughly one extra day's worth of processing without altering the peak demand on the system. This is a

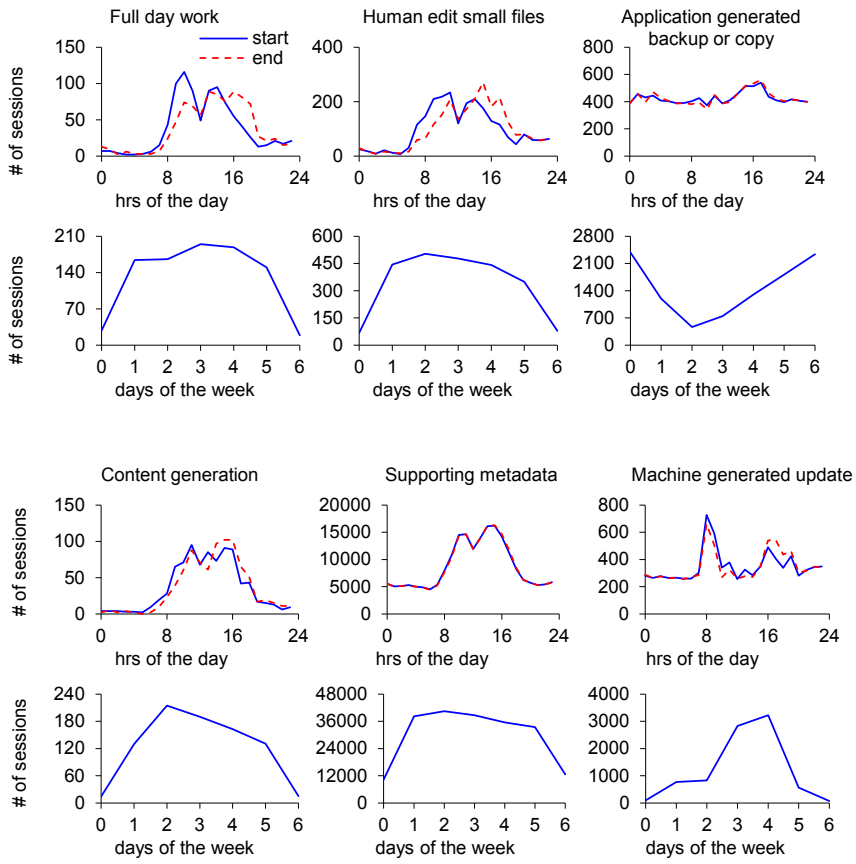



Figure 5.5. **Number of engineering sessions that start or end at a particular time.** Number of session starts and ends versus time of the day (1st and 3rd rows) and session starts versus days of the week (2nd and 4th rows).

50% improvement over a setup that performs background tasks only during weekends. In the engineering trace, the application generated backup or copy sessions seem to have been already designed this way. *Implication 3: Servers get an extra "day" for background tasks by running them at appropriate times during week-days.*

### 5.3.2 Application instances

Application instance access patterns reflect application behavior, facilitating application specific optimizations. We used 16 features to describe application instances (Table 5.3). The corporate trace has 138,723 application instances, and the engineering trace has 741,319.

Table 5.3 provides quantitative descriptions and short names for all the application instance access patterns. We derive the names from examining the read-write ratio, I/O size, and file extensions accessed (Figures 5.6 and 5.7).

| (a). Descriptive features for each application instance | | |
|---|---|---|
| Total I/O size | Read sequentiality | File opens |
| Read:write ratio by bytes | Write sequentiality | Unique files opened |
| Total I/O requests by bytes | Repeated read ratio | Directories accessed |
| Read:write ratio by requests | Overwrite ratio | File extensions accessed |
| Total metadata requests | Tree connects | |
| Avg. time between I/O requests | Unique trees accessed | |

| (b). Corp. app. instance access patterns | Viewing app generated content | Supporting metadata | App generated file updates | Viewing human generated content | Content update app |
|---|---|---|---|---|---|
| % of all app instances | 16% | 56% | 14% | 8.8% | 5.1% |
| Total I/O | 100 KB | 0 | 1 KB | 800 KB | 3.5 MB |
| Read:write ratio | 1:0 | 0:0 | 1:1 | 1:0 | 2:3 |
| Metadata requests | 130 | 5 | 50 | 130 | 500 |
| Read sequentiality | 5% | - | 0% | 80% | 50% |
| Write sequentiality | - | - | 0% | - | 80% |
| Overwrite ratio | - | - | 0% | - | 5% |
| File opens:files | 19:4 | 0:0 | 10:4 | 20:4 | 60:11 |
| Tree connect:Trees | 2:2 | 0:0 | 2:2 | 2:2 | 2:2 |
| Directories accessed | 3 | 0 | 3 | 3 | 4 |
| File extensions accessed | 2 | 0 | 2 | 2 | 3 |

| (c). Eng. app. instance access patterns | Compilation app | Supporting metadata | Content update app - small | Viewing human generated content | Content viewing app - small |
|---|---|---|---|---|---|
| % of all app instances | 1.6% | 93% | 0.9% | 2.0% | 2.5% |
| Total I/O | 2 MB | 0 | 2 KB | 1 MB | 3 KB |
| Read:write ratio | 9:1 | 0:0 | 0:1 | 1:0 | 1:0 |
| Metadata requests | 400 | 1 | 14 | 40 | 15 |
| Read sequentiality | 50% | - | - | 90% | 0% |
| Write sequentiality | 80% | - | 0% | - | - |
| Overwrite ratio | 20% | - | 0% | - | - |
| File opens:files | 145:75 | 0:0 | 3:1 | 5:4 | 2:1 |
| Tree connect:Trees | 1:1 | 0:0 | 1:1 | 1:1 | 1:1 |
| Directories accessed | 15 | 0 | 1 | 1 | 1 |
| File extensions accessed | 5 | 0 | 1 | 1 | 1 |

Table 5.3. **Application instance access patterns.** (a): Full list of descriptive features. Almost identical to those for user sessions, with a new feature "file extensions accessed", and two features no longer applicable — "duration" and "application instances seen". (b) and (c): Short names and descriptions of application instances in each access pattern; listing only the features that help separate the access patterns.

We see again the metadata background noise. The supporting metadata application instances account for the largest fraction, and often do not even open a file.

There are many files without a file extension, a phenomenon also observed in recent storage system snapshot studies [93]. We notice that file extensions turn out to be poor indicators of application instance access patterns. This is not surprising because we separate access patterns based on read/write properties. A user could either view a `.doc` or create a `.doc`. The same application software has different read/write patterns. This speaks to the strength of our multi-layer framework. Aggregating I/O by application instances gives clean separation of patterns; while aggregating just by application software or file extensions will not.

We also find it interesting that most file extensions are immediately recognizable. This means that *what* people use network storage systems for, i.e., the file extensions,

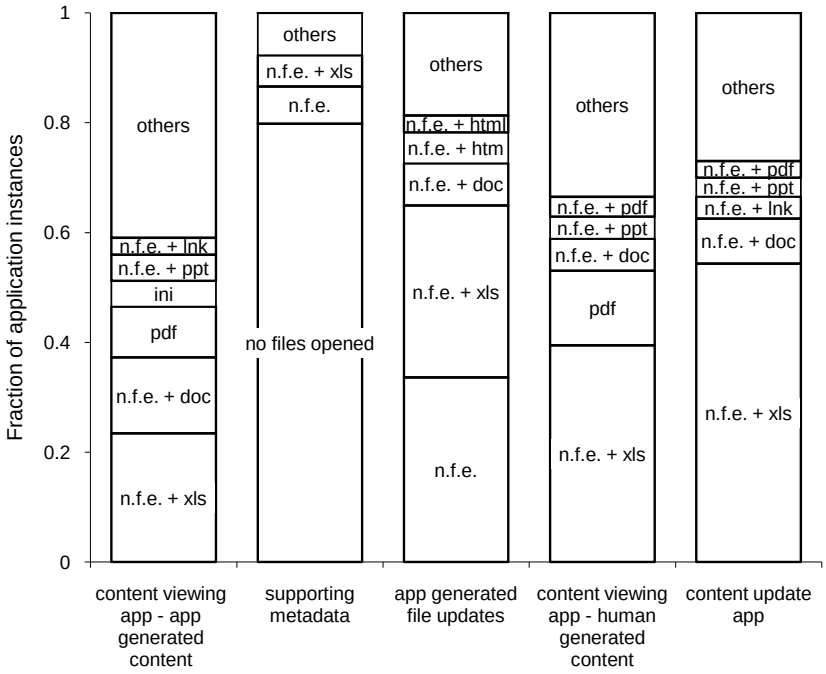



Figure 5.6. **File extensions for corporate application instance access patterns.** For each access pattern (column), showing the fraction of the two most frequent file extensions that are accessed together within a single application instance. "n.f.e." denotes files with "no file extension". Application instances labeled with just one file extension accessed files with only one extension.

remains easily recognizable, even though *how* people use network storage systems, i.e., the access patterns, is ever changing and becoming more complex.

*Observation 4: The "small content viewing" and "content update" application instances have <4KB total reads per file open and access a few unique files many times (Table 5.3).* The small read size and multiple reads from the same files means that clients should prefetch and place the files in a cache optimized for random access (flash/SSD/memory). The trend towards flash caches on clients should enable this transfer.

Application instances have bi-modal total I/O size - either less than 10KB, or 100KB-10MB. Thus, a simple cache management algorithm suffices; we always keep the first 2 blocks of 4KB in cache. If the application instance does more I/O, it is likely to have I/O size in the 100KB-1MB range, so we evict it from the cache. We should note that such a policy makes sense even though we proposed earlier to cache all 11MB of a typical day's working set - 11MB of cache becomes a concern when we have many consolidated clients. *Implication 4: Clients should always cache the first few KB of I/O per file per application.*

*Observation 5: We see >50% sequential read and write ratio for the "content update" applications instances for corporate and the "content viewing" applications instances for human-generated content for both corporate and engineering (Table 5.3).* Dividing the

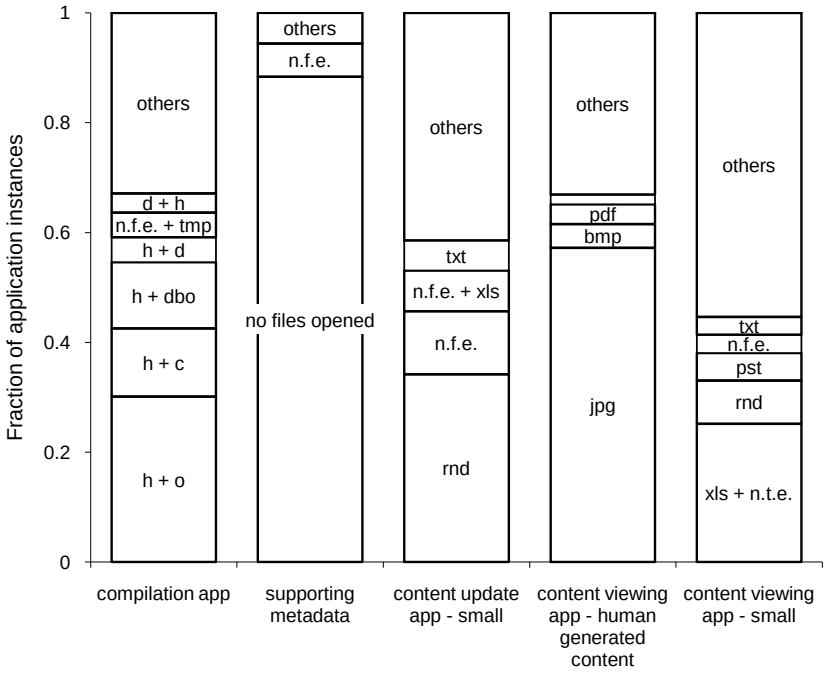



Figure 5.7. **File extensions for engineering application instance access patterns.** For each access pattern (column), showing the fraction of the two most frequent file extensions that are accessed together within a single application instance. "n.f.e." denotes files with "no file extension". Application instances labeled with just one file extension accessed files with only one extension.

total I/O size by the number of file opens suggest that these application instances are sequentially reading and writing entire files for office productivity (`.xls`, `.doc`, `.ppt`, `.pdf`) and multimedia applications.

This implies that the files associated with these applications should be prefetched and delegated to the client. Prefetching means delivering the whole file to the client before the whole file is requested. Delegation means giving a client temporary, exclusive access to a file, with the client periodically synchronizing to server to ensure data durability. CIFS does delegation using opportunistic locks, while NFSv4 has a dedicated operation for delegation. Prefetching and delegation of such files will improve read and write performance, lower network traffic, and lighten server load.

The access patterns again offer a simple, threshold-based decision algorithm. If an application instance does more than 10s of KB of sequential I/O, and has no overwrite, then it is likely to be a content viewing or update application instance; such files are prefetched and delegated to the clients. *Implication 5: Clients can request file prefetch (read) and delegation (write) based on only I/O sequentiality.*

*Observation 6: Engineering applications with >50% sequential reads and >50% sequential writes are doing code compile tasks.* We know this from looking at the file extensions in Figure 5.7. These compile processes show read sequentiality, write sequen-

tiality, a significant overwrite ratio and large number of metadata requests. They rely on the server heavily for data accesses. We need more detailed client side information to understand why client caches are ineffective in this case. However, it is clear that the server cache needs to prefetch the read files for these applications. The high percentage of sequential reads and writes gives us another threshold-based algorithm to identify these applications. *Implication 6: Servers can statistically identify compile tasks by the presence of both sequential reads and writes; server has to cache the output of these tasks.*

## 5.4 Server Side Access Patterns

As mentioned in Section 5.2.2, we analyzed two kinds of server side access units: files and deepest subtrees.

### 5.4.1 Files

File access patterns help storage server designers develop per-file placement and optimization techniques. We used 25 features to describe files (Table 5.4). Note that some of the features include different percentiles of a characteristic, e.g., read request size as percentiles of all read requests. We believe including different percentiles rather than just the average would allow better separation of access patterns. The corporate trace has 1,155,099 files, and the engineering trace has 1,809,571.

In Table 5.4, we present quantitative descriptions and short names for all the file access patterns. Figures 5.8 and 5.9 give the most common file extensions in each. We derived the names by examining the read-write ratio and I/O size. For the engineering trace, examining the file extensions also proved useful, leading to labels such as "edit code and compile output", and "read-only log/backup".

We see that there are groupings of files with similar extensions. For example, in the corporate trace, the small random read access patterns include many file extensions associated with web browser caches. Also, multi-media files like `.mp3` and `.jpg` congregate in the sequential read and write access patterns. In the engineering trace, code libraries group under the sequential write files, and read only log/backup files contain file extensions `.0` to `.99`. However, the most common file extensions in each trace still spread across many access patterns, e.g., office productivity files in the corporate trace and code files in the engineering trace.

*Observation 7: For files with >70% sequential reads or sequential writes, the repeated read and overwrite ratios are close to zero (Table 5.4).* This implies that there is little benefit in caching these files at the server. They should be prefetched as a whole and delegated to the client. Again, the bimodal I/O sequentiality offers a simple algorithm for the server to detect which files should be prefetched and delegated – if a file has any sequential access, it is likely to have a high percentage of sequential access, therefore it should be prefetched and delegated to the client. Future storage servers can suggest

| (a). Descriptive features for each file | |
|---|---|
| Number of hours with 1, 2-3, or 4 file opens | Read sequentiality |
| Number of hours with 1-100KB, 100KB-1MB, or >1MB reads | Write sequentiality |
| Number of hours with 1-100KB, 100KB-1MB, or >1MB writes | Read:write ratio by bytes |
| Number of hours with 1, 2-3, or 4 metadata requests | Repeated read ratio |
| Read request size - 25th, 50th, and 75th percentile of all requests | Overwrite ratio |
| Write request size - 25th, 50th, and 75th percentile of all requests | |
| Avg. time between I/O requests - 25th, 50th, and 75th percentile of all request pairs | |

| (b). Corp. file access patterns | Metadata only | Sequential write | Sequential read | Small random write | Smallest random read | Small random read |
|---|---|---|---|---|---|---|
| % of all files | 59% | 4.0% | 4.1% | 4.7% | 19% | 9.2% |
| # hrs with opens | 2hrs | 1hr | 1hr | 1hr | 1hr | 1hr |
| Opens per hr | 1 open | 2-3 opens | 2-3 opens | 2-3 opens | 1 open | 1 open |
| # hrs with reads | 0 | 0 | 1hr | 0 | 1hr | 1hr |
| Reads per hr | - | - | 100KB-1MB | - | 1-100KB | 1-100KB |
| # hrs with writes | 0 | 1hr | 0 | 1hr | 0 | 0 |
| Writes per hr | - | 100KB-1MB | - | 1-100KB | - | - |
| Read request size | - | - | 4-32KB | - | 2KB | 32KB |
| Write request size | - | 60KB | - | 4-22KB | - | - |
| Read sequentiality | - | - | 70% | - | 0% | 0% |
| Write sequentiality | - | 80% | - | 0% | - | - |
| Read:write ratio | 0:0 | 0:1 | 1:0 | 0:1 | 1:0 | 1:0 |

| (c). Eng. file access patterns | Metadata only | Sequential write | Small random read | Edit code & compile | Sequential read | Read-only log/ backup |
|---|---|---|---|---|---|---|
| % of all files | 42% | 1.9% | 32% | 7.3% | 8.3% | 8.1% |
| # hrs with opens | 1hr | 1hr | 1hr | 1hr | 1hr | 2hrs |
| Opens per hr | 1 open | 2-3 opens | 2-3 opens | 2-3 opens | 2-3 opens | 2-3 opens |
| # hrs with reads | 0 | 0 | 1hr | 1hr | 1hr | 2hrs |
| Reads per hr | - | - | 1-100KB | 1-100KB | 1-100KB | 1-100KB |
| # hrs with writes | 0 | 1hr | 0 | 0 | 0 | 0 |
| Writes per hr | - | >1MB | - | - | - | - |
| Read request size | - | - | 3-4KB | 4KB | 8-16KB | 1KB |
| Write request size | - | 64KB | - | - | - | - |
| Read sequentiality | - | - | 0% | 0% | 70% | 0% |
| Write sequentiality | - | 90% | - | - | - | - |
| Repeated read ratio | - | - | 0% | 50% | 0% | 0% |
| Read:write ratio | 0:0 | 0:1 | 1:0 | 1:0 | 1:0 | 1:0 |

Table 5.4. **File access patterns.** (a): Full list of descriptive features. (b) and (c): Short names and descriptions of files in each access pattern; listing only the features that help separate the access patterns.

such information to clients, leading to delegation requests. *Implication 7: Servers should delegate sequentially accessed files to clients to improve I/O performance.*

*Observation 8: In the engineering trace, only the edit code and compile output files have a high % of repeated reads (Table 5.4).* Those files should be delegated to the clients as well. The repeated reads do not show up in the engineering application instances, possibly because a compilation process launches many child processes repeatedly reading the same files. Each child process reads "fresh data," even though the server sees repeated reads. With larger memory or flash caches at clients, we expect this behavior to drop. The working set issues that lead to this scenario need to be examined. If the repeated reads come from a single client, then the server can suggest that the client cache the appropriate files.

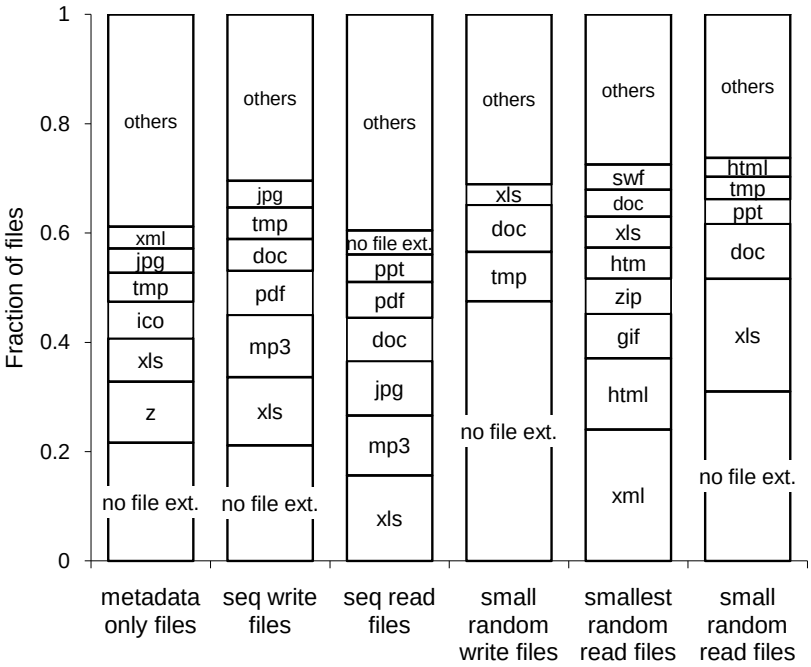



Figure 5.8. **File extensions for corporate files.** Fraction of file extensions in each file access pattern.

We can again employ a threshold-based algorithm. Detecting any repeated reads at the server signals that the file should be delegated to the client. At worst, only the first few reads will hit the server. Subsequent repeated reads are stopped at the client. *Implication 8: Servers should delegate repeatedly read files to clients.*

*Observation 9: Almost all files are active (have opens, I/O, and metadata access) for only 1-2 hours over the entire trace period, as indicated by the typical opens/read/write activity of all access patterns (Table 5.4).* There are some regularly accessed files, but they are so few that they do not affect the k-means analysis. The lack of regular access for most files means that there is room for the server to employ techniques to increase capacity by doing compaction on idle files.

Common techniques include deduplication and compression. The activity on these files indicate that the I/O performance impact should be small. Even if run constantly, compaction has a low probability of affecting an active file. Since common libraries like `gzip` optimize for decompression [67], decompressing files at read time should have only slight performance impact. *Implication 9: Servers can use file idle time to compress or deduplicate data to increase storage capacity.*

*Observation 10: All files have either all random access or >70% sequential access (Table 5.4).* The small random read and write files in both traces can benefit from being placed on media with high random access performance, such as solid state drives (SSDs). Files with a high percentage of sequential accesses can reside on traditional hard disk drives (HDDs), which already optimize for sequential access. The bimodal I/O

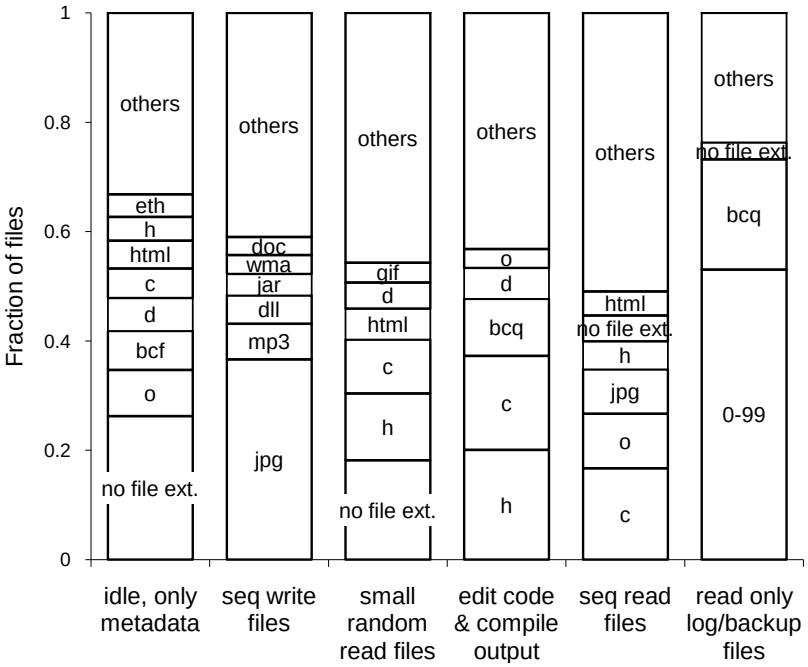



Figure 5.9. **File extensions for engineering files.** Fraction of file extensions in each file access pattern.

sequentiality offers yet another threshold-based placement algorithm – if a file has any sequential access, it is likely to have a high percentage of sequential access; therefore place it on HDDs. Otherwise, place it on SSDs. We note that there are more randomly accessed files than sequentially accessed files. Even though sequential files tend to be larger, we still need to do a working set analysis to determine the right size of server SSDs for each use case. *Implication 10: Servers can select the best storage medium for each file based only on access sequentiality.*

### 5.4.2 Deepest subtrees

Deepest subtree access patterns help storage server designers develop per-directory policies. We use the phrase "deepest subtree" as a more precise alternative to "directories". In particular, for hierarchical directory structures of the form `A/B`, data accesses to files `A/file` are counted as deepest subtree `A`, while data accesses to files `A/B/file` are counted separately as deepest subtree `A/B`.

We used 40 features to describe deepest subtrees (Table 5.5). Some of the features include different percentiles of a characteristic, e.g., per file read sequentiality as percentile of all files in a directory. Including different percentiles rather than just the average allows better separation of access patterns. The corporate trace has 117,640 deepest subtrees, and the engineering trace has 161,858.

In Table 5.5, we provide quantitative descriptions and short names for all the deepest

| **(a). Descriptive features for each subtree** |
|---|
| Number of hours with 1, 2-3, or 4 file opens |
| Number of hours with 1-100KB, 100KB-1MB, or >1MB reads |
| Number of hours with 1-100KB, 100KB-1MB, or >1MB writes |
| Number of hours with 1, 2-3, or 4 metadata requests |
| Read request size - 25th, 50th, and 75th percentile of all requests |
| Write request size - 25th, 50th, and 75th percentile of all requests |
| Avg. time between I/O requests - 25th, 50th, and 75th percentile of all request pairs |
| Read sequentiality - 25th, 50th, and 75th percentile of files in the subtree, and aggregated across all files |
| Write sequentiality - 25th, 50th, and 75th percentile of files in the subtree, and aggregated across all files |
| Read:write ratio - 25th, 50th, and 75th percentile of files, and aggregated across all files |
| Repeated read ratio - 25th, 50th, and 75th percentile of files, and aggregated across all files |
| Overwrite ratio - 25th, 50th, and 75th percentile of files, and aggregated across all files |

| **(b). Corp. subtree access patterns** | Temp real data | Client cacheable | Mixed read | Meta-data only | Mixed write | Small random read |
|---|---|---|---|---|---|---|
| % of all subtrees | 2.3% | 4.1% | 5.6% | 64% | 3.5% | 21% |
| # hrs with opens | 3hrs | 3hrs | 2hrs | 2hrs | 1hr | 1hr |
| Opens per hr | >4 opens | 1 open | 1 open | 1 open | >4 opens | >4 opens |
| # hrs with reads | 3hrs | 2hrs | 1hr | 0 | 0 | 1hr |
| Reads per hr | 1-100KB | 1-100KB | 1-100KB | - | - | 1-100KB |
| # hrs with writes | 2hrs | 0 | 0 | 0 | 1hr | 0 |
| Writes per hr | 0.1-1MB | - | - | - | >1MB | - |
| Read request size | 4KB | 4-10KB | 4-32KB | - | - | 1-8KB |
| Write request size | 4KB | - | - | - | 64KB | - |
| Read sequentiality | 10-30% | 0% | 50-70% | - | - | 0% |
| Write sequentiality | 50-70% | - | - | - | 70-80% | - |
| Repeat read ratio | 20-50% | 50% | 0% | - | - | 0% |
| Overwrite ratio | 30-70% | - | - | - | 0% | - |
| Read:write ratio | 1:0 to 0:1 | 1:0 | 1:0 | 0:0 | 0:1 | 1:0 |

| **(b). Eng. subtree access patterns** | Meta-data only | Small random read | Client cacheable | Mixed read | Sequen-tial write | Temp real data |
|---|---|---|---|---|---|---|
| % of all subtrees | 59% | 25% | 6.1% | 7.1% | 1.9% | 1.3% |
| # hrs with opens | 1hr | 1hr | 1hr | 1hr | 1hr | 3hrs |
| Opens per hr | 2-3 pens | >4 opens | >4 opens | >4 opens | >4 opens | >4 opens |
| # hrs with reads | 0 | 1hr | 1hr | 1hr | 0 | 3hrs |
| Reads per hr | - | 1-100KB | 1-100KB | 0.1-1MB | - | 1-100KB |
| # hrs with writes | 0 | 0 | 0 | 0 | 1hr | 1hr |
| Writes per hr | - | - | - | - | 0.1-1MB | 1-100KB |
| Read request size | - | 1-4KB | 2-4KB | 8-10KB | - | 4-32KB |
| Write request size | - | - | - | - | 32-60KB | 4-60KB |
| Read sequentiality | - | 0% | 0% | 40-70% | - | 10-65% |
| Write sequentiality | - | - | - | - | 70-90% | 60-80% |
| Repeat read ratio | - | 0% | 50-60% | 0% | - | 0-40% |
| Overwrite ratio | - | - | - | - | 0% | 0-30% |
| Read:write ratio | 0:0 | 1:0 | 1:0 | 1:0 | 0:1 | 1:0 to 0:1 |

Table 5.5. **Deepest subtree access patterns.** (a): Full list of descriptive features. Compared with the features list in Table 5.4, the changes are that read sequentiality, write sequentiality, read:write ratio, repeated read ratio, and overwrite ratio changed from a single statistic for a file to 25th, 50th, and 75th percentile of all files aggregated in a directory. (b) and (c): Short names and descriptions of subtrees in each access pattern; listing only the features that help separate access patterns.

subtree access patterns. We derive the names using two types of information. First, we analyze the file extensions in each access pattern (Figures 5.10 and 5.11). Second, we examine how many files of each file access patterns are within each subtree pattern (Figures 5.12 and 5.13).

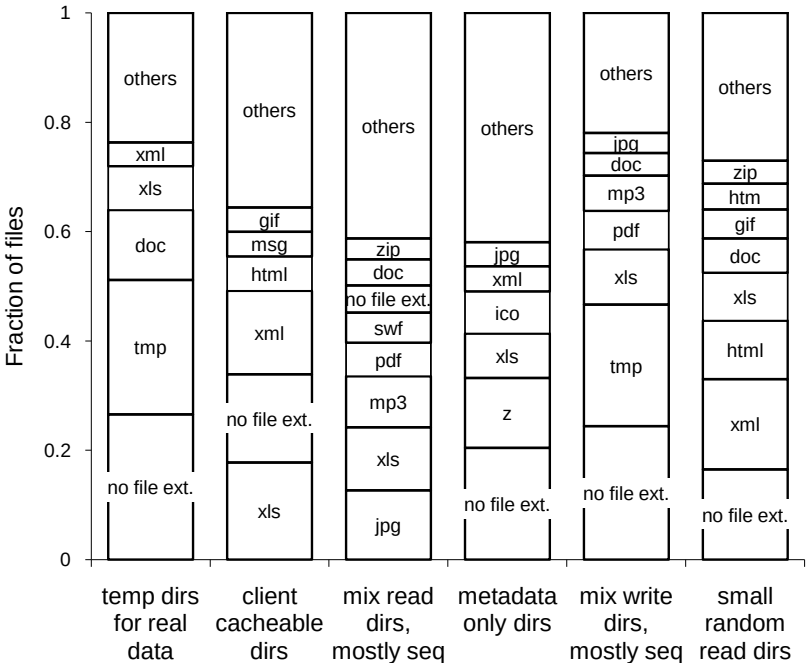



Figure 5.10. **File extensions for corporate deepest subtrees.** Fraction of file extensions in deepest subtree access patterns.

For example, the "random read" and "client cacheable" labels come from looking at the I/O patterns. "Temporary directories" accounted for the `.tmp` files in those directories. "Mix read" and "mix write" directories considered the presence of both sequential and randomly accessed files in those directories.

The metadata background noise remains visible at the subtree layer. The spread of file extensions is similar to that for file access patterns – some file extensions congregate and others spread evenly. Interestingly, some subtrees have a large fraction of files for which only the metadata gets accessed.

Some subtrees contain only files of a single access pattern (e.g., small random-read subtrees in Figures 5.12). There, we can apply the design insights from the file access patterns to the entire subtree. For example, the small random-read subtrees can reside on SSDs. Since there are more files than subtrees, per-subtree policies can lower the amount of policy information kept at the server.

In contrast, the mix read and mix write directories contain both sequential and randomly accessed files. Those subtrees need per-file policies: Place the sequentially accessed files on HDDs and the randomly accessed files on SSDs. Soft links to files can preserve the user-facing directory organization, while allowing the server to optimize per-file placement. The server should automatically decide when to apply per-file or per-subtree policies.

*Observation 11: Directories with sequentially accessed files almost always contain randomly accessed files also (Figures 5.12 and 5.13).* Conversely, some directories with

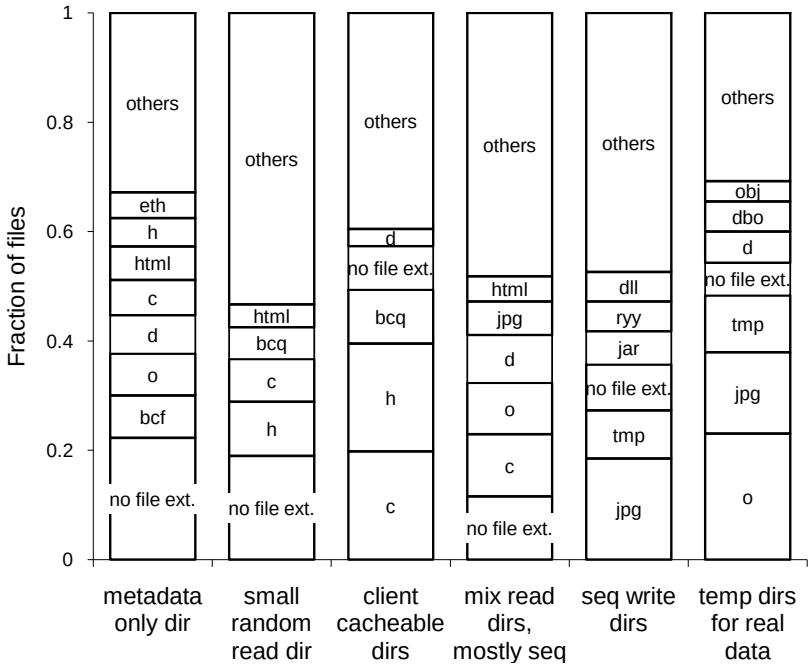



Figure 5.11. **File extensions for engineering deepest subtrees.** Fraction of file extensions in deepest subtree access patterns.

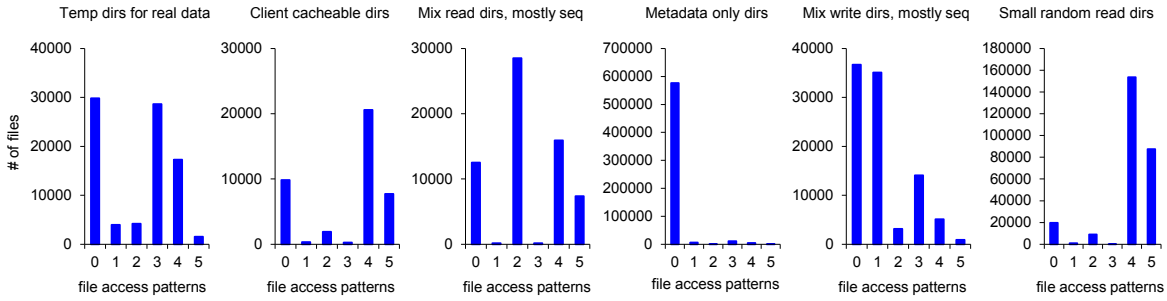



Figure 5.12. **Corporate file access patterns within each deepest subtree.** For each deepest subtree access pattern (i.e., each graph), showing the number of files belonging to each file access pattern that belongs to subtrees in the subtree access pattern. Corporate file access pattern indices: 0. metadata-only files; 1. sequential-write files; 2. sequential-read files; 3. small random-write files; 4. small random-read files; 5. less-small random-read files.

randomly access files will not contain sequentially accessed files. Thus, we can default all subtrees to per-subtree policies. Concurrently, we track the I/O sequentiality per subtree. If the sequentiality is above some threshold, then the subtree switches to per-file policies. *Implication 11: Servers can change from per-directory placement policy (default) to per-file policy upon seeing any sequential I/O to any files in a directory.*

*Observation 12: The "client cacheable" subtrees and temporary subtrees aggregate files*

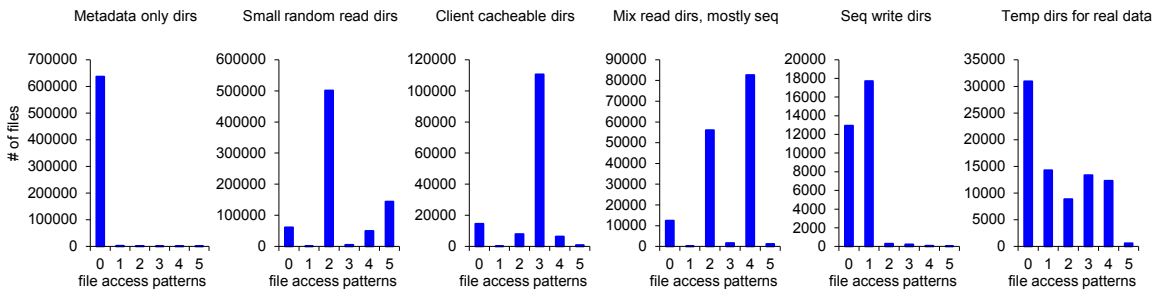



Figure 5.13. **Engineering file access patterns within each deepest subtree.** For each deepest subtree access pattern (i.e., each graph), showing the number of files belonging to each file access pattern that belongs to subtrees in the subtree access pattern. Engineering file access pattern indices: 0. metadata-only files; 1. sequential-write files; 2. small random-read files; 3. edit code and compile output files; 4. sequential-read files; 5. read-only log/backup files.

*with repeated reads or overwrites (Table 5.5 and Figures 5.12 and 5.13).* Additional computation showed that the repeated reads and overwrites almost always come from a single client. Thus, it is possible for the entire directory to be prefetched and delegated to the client. Delegating entire directories can preempt all accesses that are local to a directory, but consumes client cache space. We need to understand the tradeoffs through a more in-depth working set and temporal locality analysis at both the file and deepest subtree levels. *Implication 12: Servers can delegate repeated read and overwrite directories entirely to clients, tradeoffs permitting.*

## 5.5 Access Pattern Evolutions Over Time

We want to know if the access patterns are restricted to our particular tracing period or if they persist across time. Only if the design insights remain relevant across time can we rationalize their existence in similar use cases.

We do not have enough traces to generalize beyond our monitoring period, which is 3 months. This trace length allows us to investigate the reverse problem - if we had to analyze traces from only a subset of our tracing period, how would our results differ? We divided our traces into weeks and repeated the analysis for each week. We present the results for weekly analysis of corporate application instances and files. These two layers have yielded the most interesting design insights and they highlight separate considerations at the client and server.

Figure 5.14 shows the result for files. All the large access patterns remain steady across the weeks. However, the access pattern corresponding to the smallest number of files, the small random write files, comes and goes week to week. There are exactly two, temporary, previously unseen access patterns that are very similar to the small random files. Furthermore, the numerical values of the descriptive features for each access pattern

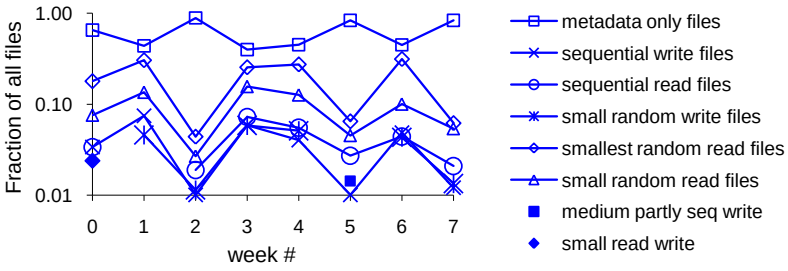



Figure 5.14. **Corporate file access patterns over 8 weeks.** All patterns remain (hollow markers), but the fractional weight of each changes greatly between weeks. Some small patterns temporarily appear and disappear (solid markers).

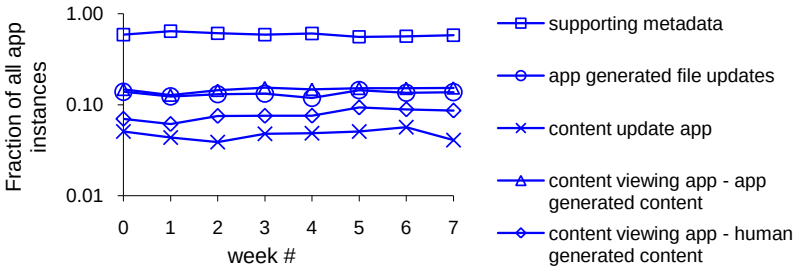



Figure 5.15. **Corporate application instance access patterns over 8 weeks.** All patterns remain with near constant fractional weight. No new patterns appear.

vary in a moderate range. For example, the write sequentiality of the sequentiality write files ranges from 50% to 90%.

Figure 5.15 shows the result for application instances. We see no new access patterns, and the fractional weight of each access pattern remains nearly constant, despite holidays. Furthermore, the numerical values of descriptive features also remain nearly constant. For example, the write sequentiality of the content update applications varies in a narrow range from 80% to 85%.

Thus, if we had done our analysis on just a week's traces, we would have gotten nearly identical results for application instances, and qualitatively similar result for files. We believe that the difference comes from the limited duration of client sessions and application instances, versus the long-term persistence of files and subtrees.

Based on our results, we are confident that the access patterns are not restricted just to our particular trace period. Future storage systems should continuously monitor the access patterns at all levels, automatically adjusting policies as needed, and notify designers of previously unseen access patterns.

We should always be cautious when generalizing access patterns from one use case to another. For use cases with the same applications running on the same OS file API, we expect to see the same application instance access patterns. Session access patterns such

as daily work sessions are also likely to be general. For the server-side access patterns, we expect the most common files and subtrees to appear in other use cases.

## 5.6 Architectural Trends

Sections 5.3 and 5.4 offered many specific optimizations for placement, caching, delegation, and consolidation decisions. We combine the insights here to speculate on the architecture of future enterprise storage systems.

We see a clear separation of roles for clients and servers. The client design can target high I/O performance by a combination of efficient delegation, prefetching and caching of the appropriate data. The servers should focus on increasing their aggregated efficiency across clients: collaboration with clients (on caching, delegation, etc.) and exploiting user patterns to schedule background tasks. Automating background tasks such as offline data deduplication delivers capacity savings in a timely and hassle-free fashion, i.e., without system downtime or explicit scheduling. Regarding caching at the server, we observe that very few access patterns suggest how to improve server's buffer cache for data accesses. Design insights 4-6, 8 and 12 indicate a heavy role for the client cache and Design insight 7 suggests how *not* to use the server buffer cache - caching metadata only and acting as a warm/backup cache for clients would result in lower latencies for many access patterns.

We also see simple ways to take advantage of new storage media such as SSDs. The clear identification of sequential and random access file patterns enables efficient device-specific data placement algorithms (Design insights 10 and 11). Also, the background metadata noise seen at all levels suggests that storage servers should both optimize for metadata accesses and redesign client-server interactions to decrease the metadata chatter. Depending on the growth of metadata and the performance requirements, we also need to consider placing metadata on low latency, non-volatile media like flash or SSDs.

Furthermore, we believe that storage systems should introduce many monitoring points to dynamically adjust the decision thresholds of placement, caching, or consolidation policies. We need to monitor both clients and servers. For example, when repeated read and overwrite files have been properly delegated to clients, the server would no longer see files with such access patterns. Without monitoring points at the clients, we would not be able to quantify the file delegation benefits. Storage systems should make extensible tracing APIs to expedite the collection of long-term future traces. This will facilitate future work similar to ours.

## 5.7 Chapter Conclusions

We must address the storage technology trends toward ever-increasing scale, heterogeneity, and consolidation. Current storage design paradigms that rely on existing trace analysis methods are ill equipped to meet the emerging challenges because they are unidi-

mensional, focus only on the storage server, and are subject to designer bias. We showed that a multi-dimensional, multi-layered trace-driven design methodology leads to more objective design points with highly targeted optimizations at both storage clients and servers. Using our corporate and engineering use cases, we present a number of insights that inform future designs. We described in some detail the access patterns we observed, and we encourage fellow storage system designers to extract further insights from our observations.

Storage system designers face an increasing challenge to anticipate access patterns. This chapter builds the case that system designers can no longer accurately anticipate access patterns using intuition only. We believe that the corporate and engineering traces from the NetApp corporate headquarters would have similar use cases at other traditional and high-tech businesses. Other use cases would require us to perform the same trace collection and analysis process to extract the same kind of empirical insights. We also need similar studies at regular intervals to track the evolving use of storage system. We hope that this chapter contributes to an objective and principled design approach targeting rapidly changing data access patterns.

In the broader context of the dissertation, the analysis here and in Chapter 4 applies the methodology presented in Chapter 3 to helps us understand the behavior of different kinds of large-scale data-centric systems. The subsequent chapters leverage the workload insights developed thus far to solve some system design problems beyond those that immediately follow from the workload behavior observations. Specifically, Chapters 6 seeks to improve MapReduce energy efficiency. Chapter 7 quantifies the performance implications of TCP incast. While these two design problems are not immediately related to workload management, an understanding of workload behavior is vital to developing a solution.

# Chapter 6

# Workload-Driven Design and Evaluation - Energy Efficient MapReduce

*Learning culminates in action. — Xun Zi.*

This is the first of two chapters that illustrate how insights gained from workload analysis helps the actual design and evaluation of large-scale data-centric systems. Both chapters address research problems that are not directly related to workload management for large-scale data-centric systems. Rather, empirical workload insights allow us to identify a new workload class and simplify a challenging design topic (this chapter), or evaluate the importance of a known problem in the context of common and future workloads (next chapter).

Here, we seek to **improve the energy efficiency of large scale MapReduce clusters**. Energy efficiency mechanisms heavily depend on the particular workload involved. The empirical workload analysis, carried over from Chapter 4, allows us to identify an emerging class of MapReduce with Interactive Aanlysis (MIA) workloads. Properties of MIA workloads motivated the design of the BEEMR (Berkeley Energy Efficient MapReduce) workload manager. It represents a completely different design approach than that for improving energy efficiency for web search centric MapReduce workloads. We also discovered that evaluating energy efficiency improvements requires large scale simulations covering long-term historical behavior, further amplifying the need for advances in workload analysis and replay.

The chapter is organized as follows. We motivate the need to improve MapReduce energy efficiency (Section 6.1). We focus one of the MapReduce workloads analyzed earlier to distill the key features of MIA workloads (Section 6.2). Next we outline how prior work has been limited by the lack of such empirical insights (Section 6.3). The BEEMR architecture follows (Section 6.4). Evaluating BEEMR requires some advances

in methodology, and involves developing scalable MapReduce simulators and replaying workloads to empirically verify the simulators (Section 6.5). We present the evaluation results next (Section 6.6), along with a further discussion on logistical, generality, and methodology concerns (Section 6.7). We close the chapter by reflecting on the broader implications of workload-driven design and evaluation with regard to energy efficiency of large-scale data-centric systems (Section 6.8).

## 6.1 Motivation

Massive computing clusters are increasingly being used for data analysis. The sheer scale and cost of these clusters make it critical to improve their operating efficiency, including energy. Energy costs are a large fraction of the total cost of ownership of datacenters [68, 33]. Consequently, there is a concerted effort to improve energy efficiency for Internet datacenters, encompassing government reports [129], standardization efforts [125], and research projects in both industry and academia [114, 58, 89, 54, 90, 120, 34, 84, 81, 78].

Approaches to increasing datacenter energy efficiency depend on the workload in question. One option is to increase machine utilization, i.e., increase the amount of work done per unit energy. This approach is favored by large web search companies such as Google, whose machines have persistently low utilization and waste considerable energy [32]. Clusters implementing this approach would service a mix of interactive and batch workloads [108, 52, 94], with the interactive services handling the external customer queries [89], and batch processing building the data structures that support the interactive services [53]. This strategy relies on predictable diurnal patterns in web query workloads, using latency-insensitive batch processing drawn from an "infinite queue of low-priority work" to smooth out diurnal variations, to keep machines at high utilization [32, 52, 108].

This chapter focuses on an alternate use case—what we call *MapReduce with Interactive Analysis (MIA)* workloads. MIA workloads contain interactive services, traditional batch processing, and large-scale, latency-sensitive processing. The last component arises from human data analysts interactively exploring large data sets via ad-hoc queries, and subsequently issuing large-scale processing requests once they find a good way to extract value from the data [37, 140, 91, 76]. Such human-initiated requests have flexible but not indefinite execution deadlines.

MIA workloads require a very different approach to energy-efficiency, one that focuses on decreasing the amount of energy used to service the workload. As we will show by analyzing traces of a front-line MIA cluster at Facebook, such workloads have arrival patterns beyond the system's control. This makes MIA workloads unpredictable: new data sets, new types of processing, and new hardware are added rapidly over time, as analysts collect new data and discover new ways to analyze existing data [37, 140, 91, 76]. Thus, increasing utilization is insufficient: First, the workload is dominated by human-initiated jobs, so low-priority batch jobs only partially smooth out the workload variation, given that the cluster must be provisioned for peak to maintain good service

level objectives (SLOs); Second, the workload has unpredictable high spikes compared with regular diurnal patterns for web queries, resulting in wasted work from batch jobs being preempted upon sudden spikes in the workload.

MIA-style workloads have already appeared in several organizations, including both web search and other businesses [37, 91, 76]. Several technology trends help increase the popularity and generality of MIA workloads:

- Industries ranging from e-commerce, finance, and manufacturing are increasingly adopting MapReduce as a data processing and archival system [7].
- It is increasingly easy to collect and store large amounts of data about both virtual and physical systems [37, 78, 56].
- Data analysts are gaining expertise using MapReduce to process big data sets interactively for real-time analytics, event monitoring, and stream processing [37, 91, 76].

In short, MapReduce has evolved far beyond its original use case of high-throughput batch processing in support of web search-centric services, and it is critical that we develop energy efficiency mechanisms for MIA workloads.

This chapter presents BEEMR (Berkeley Energy-Efficient MapReduce), an energy-efficient MapReduce system motivated by an empirical analysis of a real-life MIA workload at Facebook. This workload requires BEEMR to meet stringent design requirements, including minimal impact on interactive job latency, write bandwidth, write capacity, memory set size, and data locality, as well as compatibility with distributed file system fault tolerance using error correction codes rather than replication. BEEMR represents a new design point that combines batching [81], zoning [84], and data placement [78] with new analysis-driven insights to create an efficient MapReduce system that saves energy while meeting these design requirements. The key insight is that although MIA clusters host huge volumes of data, the interactive jobs operate on just a small fraction of the data, and thus can be served by a small pool of dedicated machines; whereas the less time-sensitive jobs can run in a batch fashion on the rest of the cluster. These defining characteristics of MIA workloads both motivate and enable the BEEMR design. BEEMR increases cluster utilization while batches are actively run, and decreases energy waste between batches because only the dedicated interactive machines need to be kept at full power. The contributions of this chapter are:

- An analysis of Facebook cluster trace to quantify the empirical behavior of a MIA workload.
- The BEEMR framework which combines novel ideas with existing MapReduce energy efficiency mechanisms.
- An improved evaluation methodology to quantify energy savings and accounts for the complexity of MIA workloads.
- An identification of a set of general MapReduce design issues that warrant more study.

We show energy savings of 40-50%. BEEMR highlights the need to design for an important class of data center workloads, and represents an advance over existing MapReduce energy-efficiency proposals [84, 81, 78]. Systems like BEEMR become more important

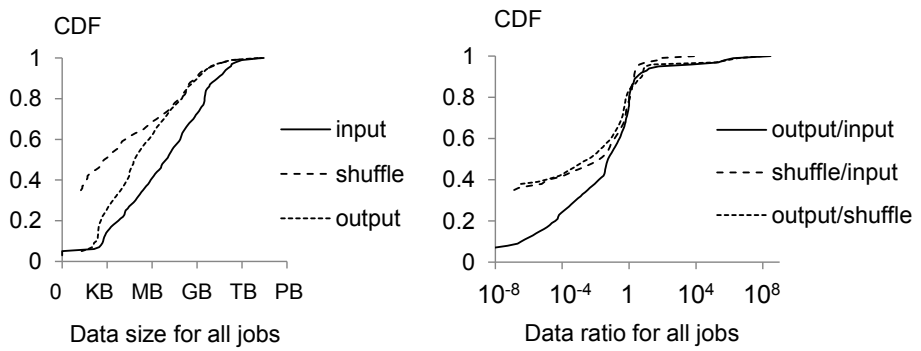



Figure 6.1. CDFs of input/shuffle/output sizes and ratios for the entire 45-day Facebook trace. Both span several orders of magnitudes. Energy efficiency mechanisms must accommodate this range.

as the need for energy efficiency continues to increase, and more use cases approach the scale and complexity of the Facebook MIA workload.

## 6.2 The Facebook Workload

Facebook is a social network company that allows users to create profiles and connect with each other. The Facebook workload provides a detailed case study of the growing class of MIA workloads. Chapter 4 has already analyzed the Facebook workload in depth along with several Cloudera customer workloads. We repeat here observations with regard to the `Facebook-2010` workload that are especially relevant to energy efficiency. Combined with some specific operational requirements at Facebook, which we also detail, the following workload characteristics motivate the BEEMR design and highlight where previous solutions fall short.

Figure 6.1 shows the distribution of per-job data sizes and ratios between input, shuffle, and output data sizes. The data sizes span several orders of magnitude, and most jobs have data sizes in the KB to GB range. The data ratios also span several orders of magnitude. 30% of the jobs are map-only, and thus have 0 shuffle data. Any effort to improve energy efficiency must account for this range of data sizes and data ratios.

Figure 6.2 shows the workload variation over two weeks. The number of jobs is diurnal, with peaks around midday and troughs around midnight. All three time series have a high peak-to-average ratio, especially map and reduce task times. Most hardware is not power proportional, i.e., the power consumption is roughly constant regardless of the level of hardware utilization [32]. Hence, a cluster provisioned for peak would experience considerable periods of below peak activity running at near-peak power.

To distinguish among different types of jobs in the workload, we can perform statisti-

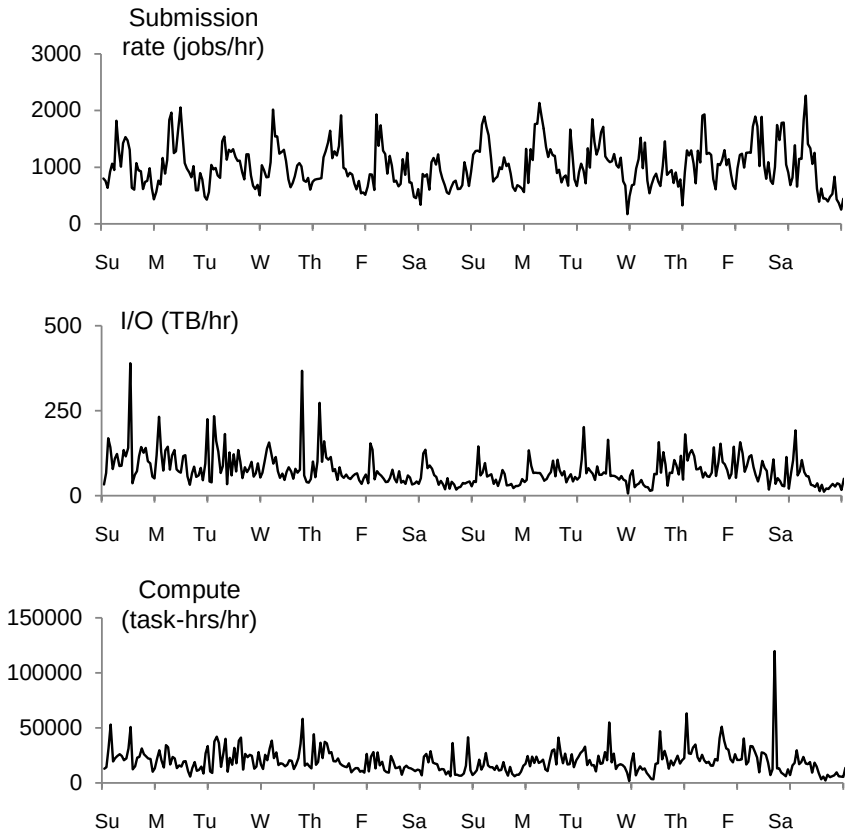



Figure 6.2. Hourly workload variation over two weeks. The workload has high peak-to-average ratios. A cluster provisioned for the peak would be often underutilized and waste a great deal of energy.

cal data clustering analysis. This analysis treats each job as a multi-dimensional vector, and finds clusters of similar numerical vectors, i.e., similar jobs. Our traces give us six numerical dimensions per job — input size, shuffle size, output size, job duration, map time, and reduce time. Table 4.3 shows the results using the k-means algorithm, in which we labeled each cluster based on the numerical value of the cluster center.

Most of the jobs are small and interactive. These jobs arise out of ad-hoc queries initiated by internal human analysts at Facebook [37, 127]. There are also jobs with long durations but small task times (map only, GB-scale, many-day jobs). These jobs have inherently low levels of parallelism, and take a long time to complete, even if they have the entire cluster at their disposal. Any energy-efficient MapReduce system must accommodate many job types, each with their own unique characteristics.

Figures 6.3 and 6.4 show the data access patterns as indicated by the HDFS path of the input data set of each job. Unfortunately the Facebook-2010 traces do not contain the HDFS path of the output data sets. Figure 6.3 shows that the input path accesses follow a Zipf distribution, i.e., a few input paths account for a large fraction of all accesses.

| # Jobs | Input | Shuffle | Output | Duration | Map time | Reduce time | Label |
|---|---|---|---|---|---|---|---|
| 1,145,663 | 6.9 MB | 600 B | 60 KB | 1 min | 48 | 34 | Small jobs |
| 7,911 | 50 GB | 0 | 61 GB | 8 hrs | 60,664 | 0 | Map only transform, 8 hrs |
| 779 | 3.6 TB | 0 | 4.4 TB | 45 min | 3,081,710 | 0 | Map only transform, 45 min |
| 670 | 2.1 TB | 0 | 2.7 GB | 1 hr 20 min | 9,457,592 | 0 | Map only aggregate |
| 104 | 35 GB | 0 | 3.5 GB | 3 days | 198,436 | 0 | Map only transform, 3 days |
| 11,491 | 1.5 TB | 30 GB | 2.2 GB | 30 min | 1,112,765 | 387,191 | Aggregate |
| 1,876 | 711 GB | 2.6 TB | 860 GB | 2 hrs | 1,618,792 | 2,056,439 | Transform, 2 hrs |
| 454 | 9.0 TB | 1.5 TB | 1.2 TB | 1 hr | 1,795,682 | 818,344 | Aggregate and transform |
| 169 | 2.7 TB | 12 TB | 260 GB | 2 hrs 7 min | 2,862,726 | 3,091,678 | Expand and aggregate |
| 67 | 630 GB | 1.2 TB | 140 GB | 18 hrs | 1,545,220 | 18,144,174 | Transform, 18 hrs |

Table 6.1. Job types in the workload as identified by k-means clustering, with cluster sizes, medians, and labels. Map and reduce time are in task-seconds, i.e., a job with 2 map tasks of 10 seconds each has map time of 20 task-seconds. Notable job types include small, interactive jobs (top row) and jobs with inherently low levels of parallelism that take a long time to complete (bottom row). We ran k-means with 100 random instantiations of cluster centers, which averages to over 1 bit of randomness in each of the 6 data dimensions. We determine $k$, the number of clusters by incrementing $k$ from 1 and stopping upon diminishing decreases in the intra-cluster "residual" variance.

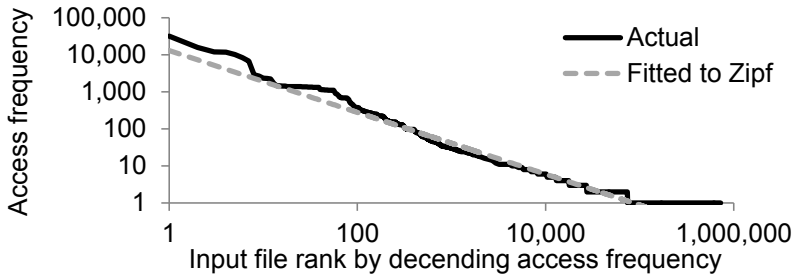



Figure 6.3. Log-log plot of workload input file path access frequency. This displays a Zipf distribution, meaning that a few input paths account for a large fraction of all job inputs.

Figure 6.4 shows that small data sets are accessed frequently; input paths of less than 10s of GBs account for over 80% of jobs, but only a tiny fraction of the total size of all input paths. Prior work has also observed this behavior in other contexts, such as web caches [39] and databases [65]. The implication is that *a small fraction of the cluster is sufficient to store the input data sets of most jobs.*

Other relevant design considerations are not evident from the traces. First, some applications require high write throughput and considerable application-level cache, such as Memcached. This fact was reported by Facebook in [37] and [127]. Second, the cluster is storage-capacity constrained, so Facebook's HDFS achieves fault tolerance through error correcting codes instead of replication, which brings the physical replication factor down from three to less than two [117]. Further, any data hot spots or decreased data locality would increase MapReduce job completion times [27].

Table 6.2 summarizes the design constraints. They represent a superset of the requirements considered by existing energy-efficient MapReduce proposals.

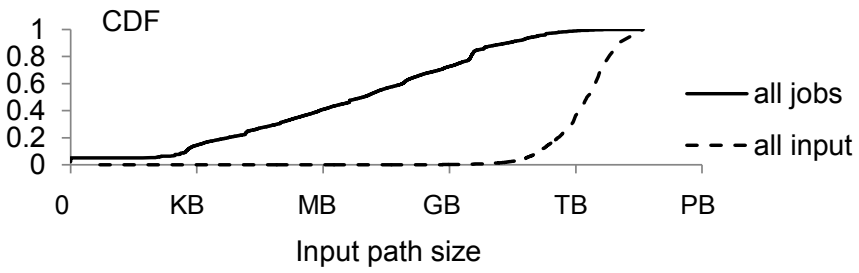



Figure 6.4. CDF of both (1) the input size per job and (2) the size per unique HDFS path of the input data set for each job. This graph indicates that small input paths are accessed frequently, i.e., data sets of less than 10s of GBs account for over 80% of jobs, and such data sets are a tiny fraction of the total data stored on the cluster.

| **Desirable Property** | **Covering subset** [84] | **All-In** [81] | **Hot & Cold Zones** [78] | **BEEMR** |
|---|---|---|---|---|
| Does not delay interactive jobs | ✔ | | ✔ | ✔ |
| No impact on write bandwidth | | ✔ | | ✔ |
| No impact on write capacity | | ✔ | ✔ | ✔ |
| No impact on available memory | | ✔ | | ✔ |
| Does not introduce data hot spots nor impact data locality | ✔ | ✔ | | ✔ |
| Improvement preserved when using ECC instead of replication | | ✔ | ✔ | ✔ |
| Addresses long running jobs with low parallelism | | | | Partially |
| Energy savings | 9-50%[1] | 0-50%[2] | 24%[3] | 40-50% |

Table 6.2. Required properties for energy-saving techniques for Facebook's MIA workload. Prior proposals are insufficient. Notes: [1] The reported energy savings used an energy model based on linearly extrapolating CPU utilization while running the GridMix throughput benchmark [3] on a 36-node cluster. [2] Reported only relative energy savings compared with the covering subset technique, and for only two artificial jobs (Terasort and Grep) on a 24-node experimental cluster. We recomputed absolute energy savings using the graphs in that study. [3] Reported simulation-based energy *cost* savings, assumed an electricity cost of $0.063/KWh and 80% capacity utilization.

## 6.3 Prior Work on Cluster Energy Efficiency

Prior work includes both energy-efficient MapReduce schemes as well as strategies that apply to other workloads.

### 6.3.1 Energy-Efficient MapReduce

Existing energy-efficient MapReduce systems fail to meet all the requirements in Table 6.2. We review them here.

The covering subset scheme [84] keeps one replica of every block within a small subset of machines called the covering subset. This subset remains fully powered to preserve data availability while the rest is powered down. Operating only a fraction of the cluster decreases write bandwidth, write capacity, and the size of available memory. More critically, this scheme becomes unusable when error correction codes are used instead of replication, since the covering subset becomes the whole cluster.

The all-in strategy [81] powers down the entire cluster during periods of inactivity, and runs at full capacity otherwise. Figure 6.2 shows that the cluster is never completely inactive. Thus, to power down at any point, the all-in strategy must run incoming jobs in regular batches, an approach we investigated in [46]. All jobs would experience some delay, an inappropriate behavior for the small, interactive jobs in the MIA workload (Table 4.3).

Green HDFS [78] partitions HDFS into disjoint hot and cold zones. The frequently accessed data is placed in the hot zone, which is always powered. To preserve write capacity, Green HDFS fills the cold zone using one powered-on machine at a time. This scheme is problematic because the output of every job would be located on a small number of machines, creating a severe data hotspot for future accesses. Furthermore, running the cluster at partial capacity decreases the available write bandwidth and memory.

The prior studies in Table 6.2 also suffer from several methodological weaknesses. Some studies quantified energy efficiency improvements by running stand-alone jobs, similar to [114]. This is the correct initial approach, but it is not clear that improvements from stand-alone jobs translate to workloads with complex interference between concurrent jobs. More critically, for workloads with high peak-to-average load (Figure 6.2), per-job improvements fail to eliminate energy waste during low activity periods.

Other studies quantified energy improvements using trace-driven simulations. Such simulations are essential for evaluating energy-efficient MapReduce at large scale. However, the simulators used there were not empirically verified, i.e., there were no experiments comparing simulated versus real behavior, nor simulated versus real energy savings. Section 6.6.8 demonstrates that an empirical validation reveals many subtle assumptions about simulators, and puts into doubt the results derived from unverified simulators.

These shortcomings necessitate a new approach in designing and evaluating energy-efficient MapReduce systems.

### 6.3.2 Energy Efficient Web Search-Centric Workloads

MIA workloads require a different approach to energy efficiency than previously considered workloads.

In web search-centric workloads, the interactive services achieve low latency by using data structures in-memory, requiring the entire memory set to be always available [89]. Given hardware limits in power proportionality, it becomes a priority to increase utilization of machines during diurnal troughs [32]. One way to do this is to use batch

processing to consume any available resource. This policy makes the combined workload *closed-loop*, i.e., the system controls the amount of admitted work. Further, the combined workload becomes more *predictable*, since the interactive services display regular diurnal patterns, and with batch processing smoothing out most diurnal variations [58, 32, 89].

These characteristics enable energy efficiency improvements to focus on *maximizing the amount of work done subject to the given power budget*, i.e., maximizing the amount of batch processing done by the system. Idleness is viewed as waste. Opportunities to save energy occur on short time scales, and require advances in hardware energy efficiency and power proportionality [58, 89, 32, 54, 90, 120, 34].

These techniques remain helpful for MIA workloads. However, the open-loop and unpredictable nature of MIA workloads necessitates additional approaches. Human-initiated jobs have both throughput and latency constraints. Thus, the cluster needs to be provisioned for peak load, and idleness is inherent to the workload. Machine-initiated batch jobs can only partially smooth out transient activity peaks. Improving hardware power proportionality helps, but remains a partial solution since state-of-the-art hardware is still far from perfectly power proportional. Thus, absent policies to constrain the human analysts, improving energy efficiency for MIA workloads requires *minimizing the energy needed to service the given amount of work*.

More generally, energy concerns complicate capacity provisioning, a challenging topic with investigations dating back to the time-sharing era [118, 29, 28]. This chapter offers a new perspective informed by MIA workloads.

## 6.4 BEEMR Architecture

BEEMR is an energy-efficient MapReduce workload management system. It takes advantage of the Zipf-distributed data access patterns and the small nature of interactive jobs. The key insight is that the interactive jobs can be served by a small pool of dedicated machines with their associated storage, while the less time-sensitive jobs can run in a batch fashion on the rest of the cluster using full computation bandwidth and storage capacity. This setup leads to energy savings and meets all the requirements listed in Table 6.2.

### 6.4.1 Design

Figure 6.5 shows the BEEMR architecture. It is similar to a typical Hadoop MapReduce cluster, with important differences in the allocation of resources to jobs.

BEEMR splits the cluster into disjoint interactive and batch zones. The interactive zone makes up a small, fixed percentage of cluster resources — task slots, memory, disk capacity, network bandwidth, similar to the design in [29]. The interactive zone is always fully powered. The batch zone makes up the rest of the cluster, and is put into a very low power state between batches [70].

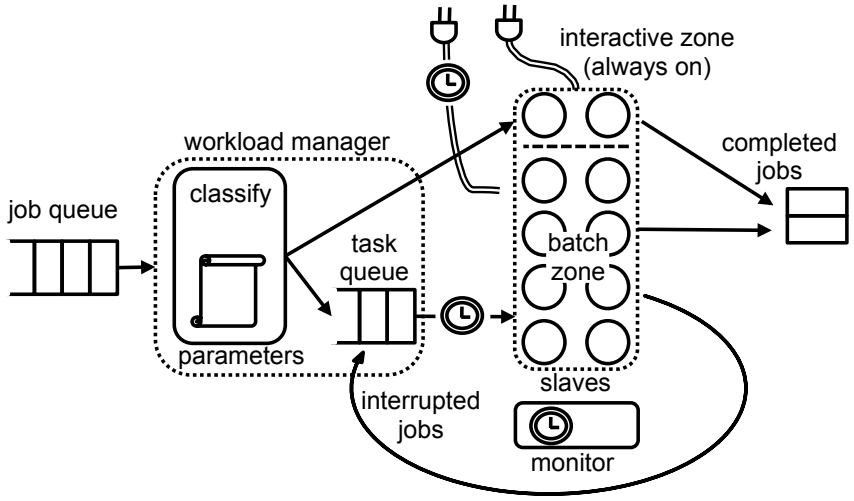



Figure 6.5. The BEEMR workload manager (i.e., job tracker) classifies each job into one of three classes which determines which cluster zone will service the job. Interactive jobs are serviced in the interactive zone, while batchable and interruptible jobs are serviced in the batch zone. Energy savings come from aggregating jobs in the batch zone to achieve high utilization, executing them in regular batches, and then transitioning machines in the batch zone to a low-power state when the batch completes.

As jobs arrive, BEEMR classifies them as one of three job types. Classification is based on empirical parameters derived from the analysis in Section 6.2. If the job input data size is less than some threshold `interactive`, it is classified as an interactive job. BEEMR seeks to service these jobs with low latency. If a job has tasks with task duration longer than some threshold `interruptible`, it is classified as an interruptible job. Latency is not a concern for these jobs, because their long-running tasks can be check-pointed and resumed over multiple batches. All other jobs are classified as batch jobs. Latency is also not a concern for these jobs, but BEEMR makes a best effort to run them by regular deadlines. Such a setup is equivalent to deadline-based policies where the deadlines are the same length as the batch intervals.

The interactive zone is always in a full-power ready state. It runs all of the interactive jobs and holds all of their associated input, shuffle, and output data (both local and HDFS storage). Figures 6.3 and 6.4 indicate that choosing an appropriate value for `interactive` can allow most jobs to be classified as interactive and executed without any delay introduced by BEEMR. This `interactive` threshold should be periodically adjusted as workloads evolve. For example, a growth in the data size for all jobs would require `interactive` threshold to be steadily increased over time.

The interactive zone acts like a data cache. When an interactive job accesses data that is not in the interactive zone (i.e., a cache miss), BEEMR migrates the relevant data from the batch zone to the interactive zone, either immediately or upon the next batch. Since most jobs are small, and small data sets are reaccessed frequently, cache misses occur infrequently. Also, BEEMR requires storing the ECC parity or replicated

blocks within the respective zones, e.g., for data in the interactive zone, their parity or replication blocks would be stored in the interactive zone also.

Upon submission of batched and interruptible jobs, all tasks associated with the job are put in a wait queue. At regular intervals, the workload manager initiates a batch, powers on all machines in the batch zone, and run all tasks on the wait queue using the whole cluster. The machines in the interactive zone are also available for batch and interruptible jobs, but interactive jobs retain priority there. After a batch begins, any batch and interruptible jobs that arrive would wait for the next batch. Once all batch jobs complete, the job tracker assigns no further tasks. Active tasks from interruptible jobs are suspended, and enqueued to be resumed in the next batch. Machines in the batch zone return to a low-power state. If a batch does not complete by start of the next batch interval, the cluster would remain fully powered for consecutive batch periods. The high peak-to-average load in Figure 6.2 indicates that on average, the batch zone would spend considerable periods in a low-power state.

BEEMR improves over prior batching and zoning schemes by combining both, and uses empirical observations to set the values of policy parameters, which we describe next.

#### 6.4.1.1 Parameter Space

BEEMR involves several design parameters whose values need to be optimized. These parameters are:

- `totalsize`: the size of the cluster in total (map and reduce) task slots.
- `mapreduceratio`: the ratio of map slots to reduce slots in the cluster.
- `izonesize`: the percentage of the cluster assigned to the interactive zone.
- `interactive`: the input size threshold for classifying jobs as interactive.
- `interruptible`: task duration threshold for classifying jobs as interruptible.
- `batchlen`: the batch interval length.
- `taskcalc`: the algorithm for determining the number of map and reduce tasks to assign to a job.

Table 6.3 shows the parameter values we will optimize for the Facebook workload. For other workloads, the same tuning process can extract a different set of values. Note that `totalsize` indicates the size of the cluster in units of task slots, which differs from the number machines. One machine can run many task slots, and the appropriate assignment of task slots per machine depends on hardware capabilities.

Another parameter worth further explanation is `taskcalc`, the algorithm for determining the number of map and reduce tasks to assign to a job. An algorithm that provides appropriate task granularity ensures that completion of a given batch is not held up by long-running tasks from some jobs.

BEEMR considers three algorithms: *Default* assigns 1 map per 128 MB of input and 1 reduce per 1 GB of input; this is the default setting in Hadoop. *Actual* assigns the same number of map and reduce tasks as given in the trace and corresponds to settings at

Facebook. *Latency-bound* assigns a number of tasks such that no task will run for more than 1 hour. This policy is possible provided that task execution times can be predicted with high accuracy [96, 63].

#### 6.4.1.2 Requirements Check

We argue that BEEMR meets the requirements in Table 6.2.

1. Write bandwidth is not diminished because the entire cluster is fully powered when batches execute. Table 4.3 indicates that only batch and interruptible jobs require large write bandwidth. When these jobs are running, they have access to all of the disks in the cluster.

2. Similarly, write capacity is not diminished because the entire cluster is fully powered on during batches. Between batches, the small output size of interactive jobs (Table 4.3) means that an appropriate value of `izonesize` allows those job outputs to fit in the interactive zone.

3. The size of available memory also remains intact, again because the memory of the entire cluster is accessible to batch and interruptible jobs. Only those jobs have a large in-memory working set. For interactive jobs, the default or actual (Facebook) `taskcalc` algorithms for determining tasks per job will assign few tasks, resulting in a small in-memory working set.

4. Interactive jobs are not delayed. The interactive zone is always fully powered, and designated specifically to service interactive jobs without delay.

5. BEEMR spreads data evenly within both zones, and makes no changes that impact data locality. Nonetheless, Figure 6.3 suggests that there will be some hotspots inherent to the Facebook workload, independent of BEEMR.

6. BEEMR improves energy efficiency via batching. There is no dependence on ECC or replication, thus preserving energy savings regardless of fault tolerance mechanism.

7. Long jobs with low levels of parallelism remain a challenge, even under BEEMR. These jobs are classified as interruptible jobs if their task durations are large, and batch jobs otherwise. If such jobs are classified as batch jobs, they could potentially prevent batches from completing. Their inherent low levels of parallelism cause the batch zone to be poorly utilized when running only these long jobs, resulting in wasted energy. One solution is for experts to label such jobs a priori so that BEEMR can ensure that these jobs are classified as interruptible.

### 6.4.2 Implementation

BEEMR involves several extensions to Apache Hadoop.

The job tracker is extended with a wait queue management module. This module holds all incoming batch jobs, moves jobs from the wait queue to the standard scheduler upon each batch start, and places any remaining tasks of interruptible jobs back on the wait queue when batches end. Also, the scheduler's task placement mechanism is modified such that interactive jobs are placed in the interactive zone, and always have first priority to any available slots.

The namenode is modified such that the output of interactive jobs is assigned to the interactive zone, and the output of batch and interruptible jobs is assigned to the batch zone. If either zone approaches storage capacity, it must adjust the fraction of machines in each zone, or expand the cluster.

The Hadoop master is augmented with a mechanism to transfer all slaves in the batch zone in and out of a low-power state, e.g., sending a "hibernate" command via `ssh` and using Wake-on-LAN or related technologies [85]. If batch intervals are on the order of hours, it is acceptable for this transition to complete over seconds or even minutes.

Accommodating interruptible jobs requires a mechanism that can suspend and resume active tasks. Current Hadoop architecture makes it difficult to implement such a mechanism. However, suspend and resume is a key component of failure and resume under Next Generation Hadoop [97], which is on the product release roadmap for some enterprise Hadoop vendors [143]. We can re-purpose for BEEMR the mechanisms there.

These extensions will create additional computation and IO at the Hadoop master node. The current Hadoop master has been identified as a scalability bottleneck [119]. Thus, it is important to monitor BEEMR overhead at the Hadoop master to ensure that we do not affect cluster scalability. This overhead would become more acceptable under Next Generation Hadoop, where the Hadoop master functionality would be spread across several machines [97].

## 6.5 Evaluation Methodology

The evaluation of our proposed algorithm involves running the Facebook MIA workload both in simulation and on clusters of hundreds of machines on Amazon EC2 [23].

The Facebook workload provides a level of validation not obtainable through stand-alone programs or artificial benchmarks. It is logistically impossible to replay this workload on large clusters at full duration and scale. The high peak-to-average nature of the workload means that at time scales of less than weeks, there is no way to know whether the results capture transient or average behavior. Enumerating a multi-dimensional design space would also take prohibitively long. Gradient ascent algorithms are not possible, simply because there is no guarantee that the performance behavior is convex. Combined, these concerns compel us to use experimentally validated simulations.

The simulator is optimized for simulation scale and speed by omitting certain details: job startup and completion overhead, overlapping map and reduce phases, speculative execution and stragglers, data locality, and interference between jobs. This differs from

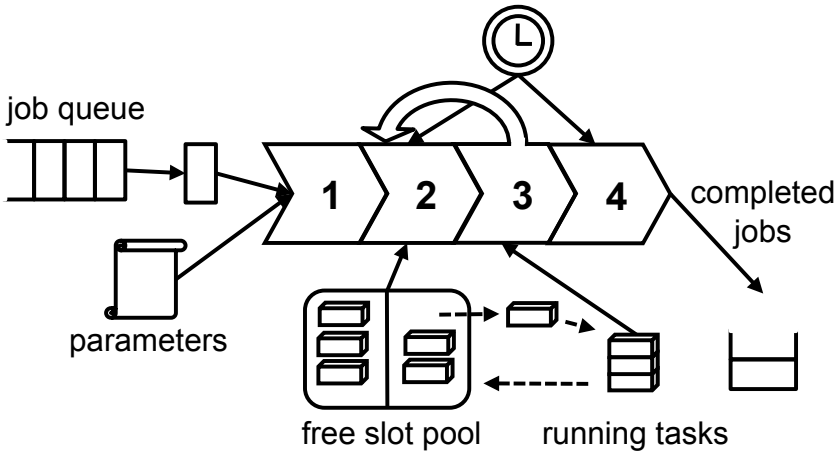



Figure 6.6. A high-level view of the simulation algorithm. For each simulated second, the following executes: 1. The simulator dequeues newly arrived jobs (arrival pattern given in the trace), classifies the job as interactive, batch, or interruptible, and applies the task granularity policy. 2. The simulator checks for available map or reduce slots, checks batch policy to see which jobs can be run at the present time, and assigns slots to jobs in round robin, fair scheduler fashion. 3. The simulator removes completed tasks and returns the corresponding slot back in the free slot pool. For each active job, it checks to see if the job has more tasks to run (go back to step 2) or is complete (go to step 4). 4. The job is marked complete and the job duration recorded.

existing MapReduce simulators [10, 135], whose focus on details make it logistically infeasible to simulate large scale, long duration workloads. The simulator assumes a simple, fluid-flow model of job execution, first developed for network simulations as an alternative to packet-level models [113, 87]. There, the motivation was also to gain simulation scale and speed. Section 6.6.8 demonstrates that the impact on accuracy is acceptable.

Simulated job execution is a function of job submit time (given in the trace), task assignment time (depends on a combination of parameters, including batch length, and number of map and reduce slots), map and reduce execution times (given in the trace), and the number of mappers and reducers chosen by BEEMR (a parameter). Figure 6.6 shows how the simulator works at a high level.

We empirically validate the simulator by replaying several day-long workloads on a real-life cluster (Section 6.6.8). This builds confidence that simulation results translate to real clusters. The validation employs previously developed methods to replay MapReduce workloads independent of hardware [44]. The techniques there replay the workload using synthetic data sets, and reproduces job submission sequences and intensities, as well as the data ratios between each job's input, shuffle, and output stages.

We model the machines as having "full" power when active and negligible power when in a low power state. Despite recent advances in power proportionality [32], such models

| Parameter | Units or Type | Values |
|---|---|---|
| `totalsize` | thousand slots | 32, 48, 60, 72 |
| `mapreduceratio` | map:reduce slots | $1:1$, $27:14$, ($\approx 2.0$), $13:5$ ($\approx 2.6$) |
| `izonesize` | % total slots | 10 |
| `interactive` | GB | 10 |
| `interruptible` | hours | 6, 12, 24 |
| `batchlen` | hours | 1, 2, 6, 12, 24 |
| `taskcalc` | algorithm | default, actual, latency-bound |

Table 6.3. Design space explored. The values for `izonesize` and `interactive` are derived from the analysis in Section 6.2. We scan at least three values for each of the other parameters. The somewhat quirky values of `mapreduceratio` comes from starting with a 1:1 ratio of map and reduce task slots, and then increasing the number of slots assigned to map tasks while decreasing the number of slots assigned to reduce tasks. For example, for `totalsize` of 72,000, we get 13:5 by starting from 36,000 slots each for map and reduce tasks, then converting 16,000 reduce slots to map slots.

remain valid for Hadoop. In [42], we used wall plug power meters to show that machines with power ranges of 150W-250W draw 205W-225W when running Hadoop. The chattiness of the Hadoop/HDFS stack means that machines are active at the hardware level even when they are idle at the Hadoop workload level. The simple power model allows us to scale the experiments in size and in time.

Several performance metrics are relevant to energy efficient MapReduce:

1. Energy savings: Under our power model, this would be the duration for which the cluster is fully idle.

2. Job latency (analogous to "turn around time" in multiprogramming literature [57]): We measure separately the job latency for each job class, and quantify any tradeoff against energy savings;

3. System throughput: Under the MIA open-loop workload model, the historical system throughput would be the smaller of `totalsize` and the historical workload arrival rate. We examine several values of `totalsize` and quantify the interplay between latency, energy savings, and other policy parameters.

Table 6.3 shows the parameter values used to explore the BEEMR design space.

## 6.6 Results

The evaluation spans the multi-dimensional design space in Table 6.3. Each dimension illustrates subtle interactions between BEEMR and the Facebook workload.

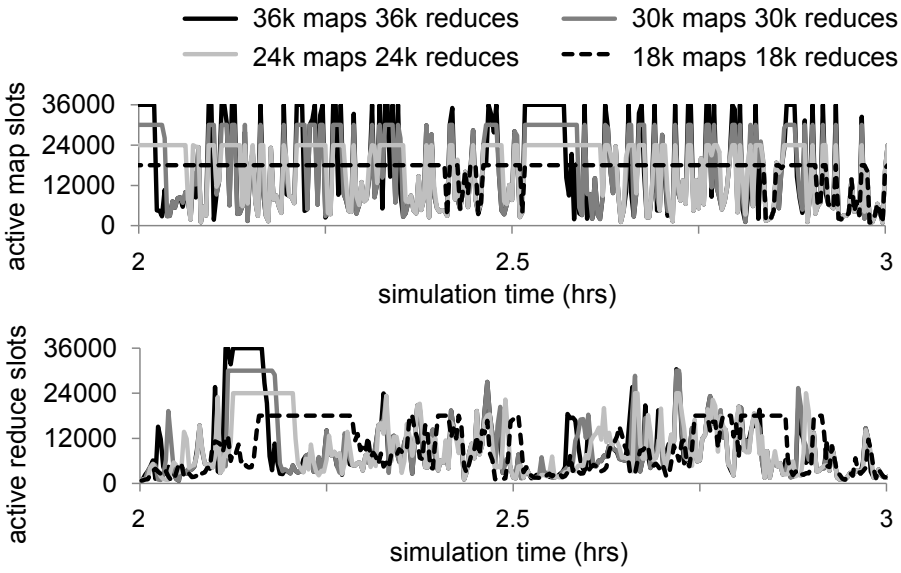



Figure 6.7. The number of concurrently active tasks for clusters of different sizes (in terms of total task slots, `totalsize`).

### 6.6.1 Cluster Size

Cluster size is controlled by `totalsize`. Underprovisioning a cluster results in long queues and high latency during workload peaks; overprovisioning leads to arbitrarily high baseline energy consumption and waste. Over the 45-days trace, the Facebook workload has an average load of 21,029 map tasks and 7,745 reduce tasks. Since the workload has a high peak-to-average ratio, we must provision significantly above the average. Figure 6.7 shows the detailed cluster behavior for several cluster sizes without any of the BEEMR improvements. We pick a one-to-one map-to-reduce-slot ratio because that is the default in Apache Hadoop, and thus forms a good baseline. A cluster with only 32,000 total slots cannot service the historical rate, being pegged at maximum slot occupancy; larger sizes still see transient periods of maximum slot occupancy. A cluster with at least 36,000 map slots (72,000 total slots) is needed to avoid persistent long queues, so we use this as a baseline.

### 6.6.2 Batch Interval Length

Energy savings are enabled by batching jobs and transitioning the batch zone to a low-power state between batches. The ability to batch depends on the predominance of interactive analysis in MIA workloads (Section 6.2). We consider here several static batch interval lengths. A natural extension would be to have dynamically adjusted batch intervals to enable various deadline driven policies.

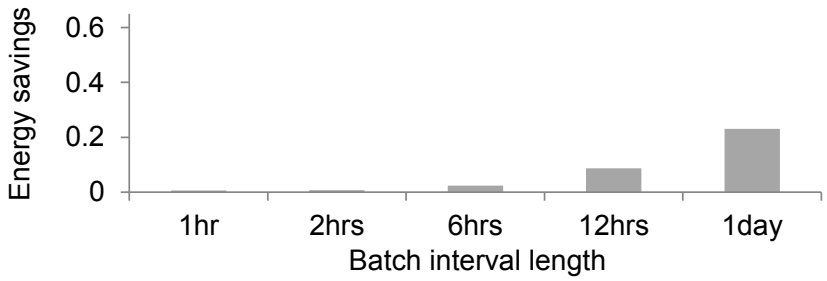



Figure 6.8. Energy savings for different batch interval lengths as given by `batchlen`. Energy savings are non-negligible for large batch intervals only. Note that `taskcalc` is set to the default task granularity policy for Hadoop, `mapreduceratio` is set to 1:1, `totalsize` is set to 72000 slots, and `interruptible` is set to 24 hours.

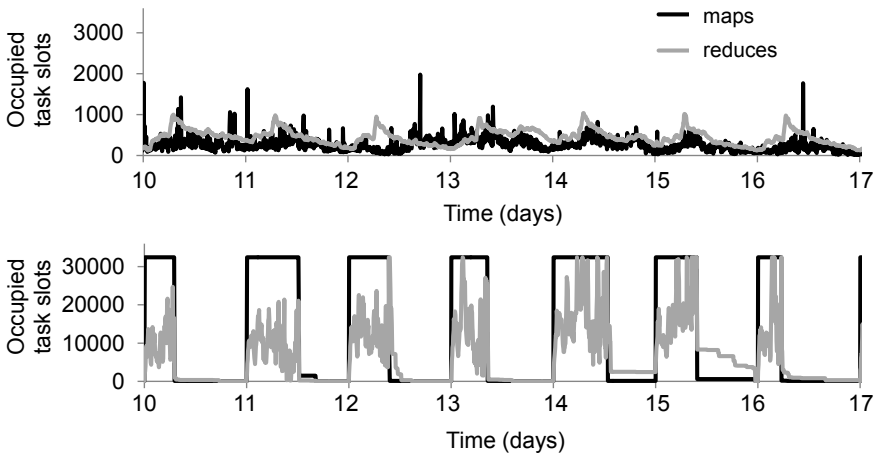



Figure 6.9. Active slots for a `batchlen` of 24 hours. Showing slot occupancy in the interactive zone (top) and in the batch zone (bottom). Showing one week's behavior. Note that `taskcalc` is set to the default task granularity policy for Hadoop, `mapreduceratio` is set to 1:1, `totalsize` is set to 72000 slots, and `interruptible` is set to 24 hours. Note that energy savings come from the times when there are 0 active map task slots and 0 active reduce task slots, i.e., the times when we can power off the batch zone.

We vary `batchlen`, the length of the interval during which arriving batch jobs would be queued, while holding the other parameters fixed. Figure 6.8 shows that energy savings, expressed as a fraction of the baseline energy consumption, become non-negligible only for batch lengths of 12 hours or more. Figure 6.9 shows map tasks execute in near-ideal batch fashion, with maximum task slot occupancy for a fraction of the batch interval and no further tasks in the remainder of the interval. However, reduce slot occupancy rarely reaches full capacity, while "dangling" reduce tasks often run for a long time at very low cluster utilization. There are more reduce tasks slots available, but the algorithm for choosing the number of task slots limits the amount of parallelism. Dur-

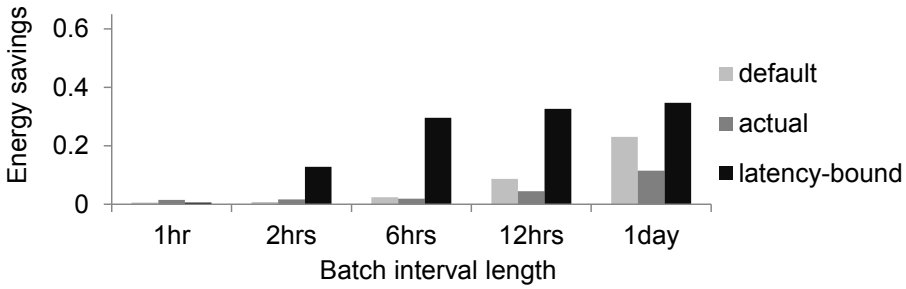



Figure 6.10. Energy savings for different `taskcalc` algorithms. Note that `mapreduceratio` is set to 1:1, and `interruptible` is set to 24 hours. The *actual* (Facebook) algorithm does worst and the *latency-bound* algorithm does best.

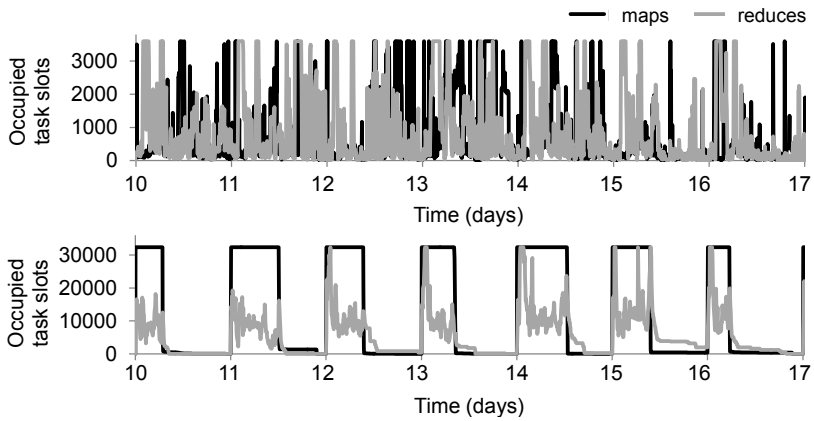
ing the fifth and sixth days, such dangling tasks cause the batch zone to remain at full power for the entire batch interval. Fixing this requires improving both the algorithm for calculating the number of tasks for each job and in the ratio of map-to-reduce slots.

### 6.6.3 Task Slots Per Job

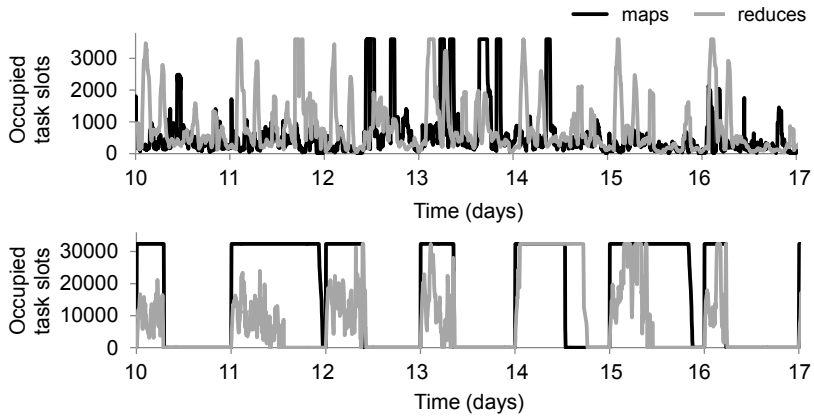
The evaluation thus far considered only the default algorithm for computing the number of tasks per job, as specified by `taskcalc`. Recall that we consider two other algorithms: *Actual* assigns the same number of map and reduce tasks as given in the trace and corresponds to settings at Facebook. *Latency-bound* assigns a number of tasks such no task will run for more than 1 hour. Figure 6.10 compares the default versus actual and latency-bound algorithms. The actual policy does the worst, which is unsurprising because the task assignment algorithm at Facebook is not yet optimized for energy efficiency. The latency-bound policy does the best; this indicates that good task execution time prediction can improve task assignment and achieve greater energy savings.

Observing task slot occupancy over time provides insight into the effects of `taskcalc`. Using the actual algorithm (Figure 6.11(a)), slots in the interactive zone reach capacity more frequently, suggesting that the Facebook algorithm seeks to increase parallelism to decrease the amount of computation per task and lower the completion latency of interactive jobs. In contrast, tasks in the batch zone behave similarly under the default and Facebook algorithm for the week shown in Figure 6.11(a). Aggregated over the entire trace, the actual policy turns out to have more dangling tasks overall, diminishing energy savings.

In contrast, task slot occupancy over time for the latency-bound policy eliminates all dangling tasks of long durations (Figure 6.11(b)). This results in high cluster utilization during batches, as well as clean batch completion, allowing the cluster to be transitioned into a low-power state at the end of a batch. There is still room for improvement in Figure 6.11(b): the active reduce slots are still far from reaching maximum task slot




(a) *Actual* (Facebook)




(b) *Latency-bound*

Figure 6.11. Slot occupancy over time in the interactive zone (top graphs) and batch zone (bottom graphs). Showing one week's behavior. Note that `batchlen` is set to 24 hours, `mapreduceratio` is set to 1:1, and `interruptible` is set to 24 hours.

capacity. This suggests that even if we keep the total number of task slots constant, we can harness more energy savings by changing some reduce slots to map slots.

### 6.6.4 Map to Reduce Slot Ratio

The evaluation thus far illustrates that reduce slots are utilized less than map slots. Changing `mapreduceratio` (i.e., increasing the number of map slots and decreasing the number of reduce slots while keeping cluster size constant) should allow map tasks in each batch to complete faster without affecting reduce tasks completion rates. Figure 6.12 shows that doing so leads to energy efficiency improvements, especially for the latency-bound algorithm.

Viewing the task slot occupancy over time reveals that this intuition about the map-

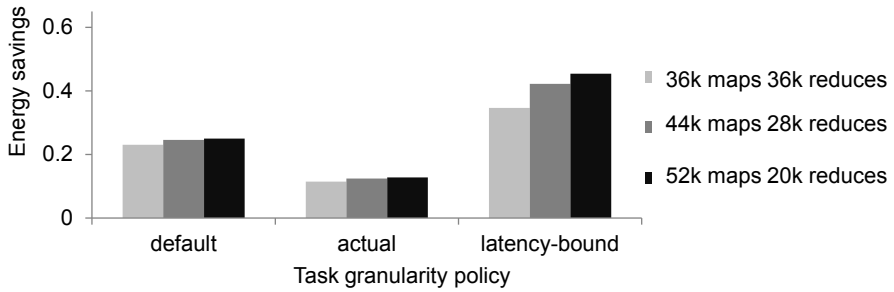



Figure 6.12. Energy savings for different values of `mapreduceratio`. Increasing the number of map slots increase energy savings for all `taskcalc` algorithms, with the improvement for *latency-bound* being the greatest. Note that `totalsize` is set to 72000 slots, `batchlen` is set to 24 hours, and `interruptible` is set to 24 hours.

to-reduce-slot ratio is correct. Figure 6.13(a) compares batch zone slot occupancy for two different ratios using the default algorithm. With a larger number of map slots, the periods of maximum map slot occupancy are shorter, but there are still dangling reduce tasks. The same ratio using the latency-bound algorithm avoids these dangling reduce tasks, as shown in Figure 6.13(b), achieving higher energy savings.

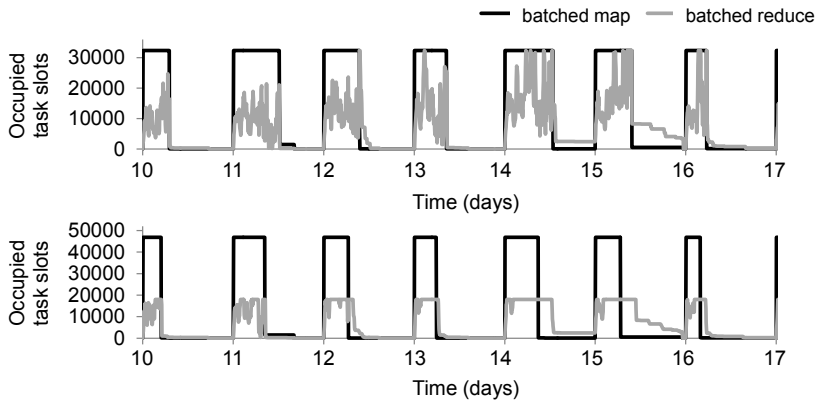
Nevertheless, the latency-bound algorithm still has room for improvement. During the fifth and sixth days in Figure 6.13(b), the batches are in fact limited by available reduce slots. Figure 6.14 shows that the previously discussed static policies for map versus task ratios achieve the best savings for all days. A dynamically adjustable ratio of map and reduce slots is best. A dynamic ratio can ensure that every batch is optimally executed, bottlenecked on neither map slots nor reduce slots.

### 6.6.5 Interruptible Threshold

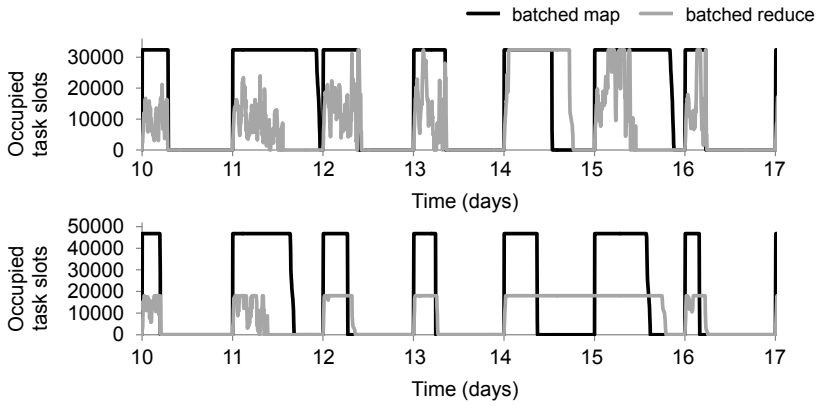
The last dimension to evaluate is `interruptible`, the task duration threshold that determines when a job is classified as interruptible. In the evaluation so far, `interruptible` has been set to 24 hours. Decreasing this threshold should cause more jobs to be classified as interruptible, and fewer jobs as batch. A lower interruptible threshold allows faster batch completions and potentially more capacity for the interactive zone, at the cost of higher average job latency, as more jobs are spread over multiple batches.

Figure 6.15 shows the energy saving improvements from lowering `interruptible`. (The latency-bound algorithm, by design, does not result in any interruptible jobs, unless the `interruptible` is set to less than an hour, so the energy savings for the latency-bound algorithm are unaffected.) Actual and default algorithms show considerable energy savings improvements, at the cost of longer latency for some jobs. It would be interesting to see how many cluster users and administrators are willing to make such trades.

Lowering `interruptible` too much would cause the queue of waiting interruptible




(a) *Default*




(b) *Latency-bound*

Figure 6.13. Batch zone slot occupancy over time using a `mapreduceratio` of 1:1 for the top graphs, and a `mapreduceratio` of 13:5 for the bottom graphs. Showing one week's behavior. Note that `batchlen` is set to 24 hours and `interruptible` is set to 24 hours.

jobs to build without bound. Consider the ideal-case upper bound on possible energy savings. The Facebook workload has a historical average of 21029 active map tasks and 7745 active reduce tasks. A cluster of 72000 task slots can service 72000 concurrent tasks at maximum. Thus, the best case energy savings is $1 - (21029 + 7745)/72000 = 0.60$. As we lower `interruptible`, any energy "savings" above this ideal actually represents the wait queue building up.

The best policy combination we examined achieves energy savings of 0.55 fraction of the baseline, as shown Figure 6.15, with `taskcalc` set to default and `interruptible` set to 6 hours. This corresponds to 92% of this ideal case.

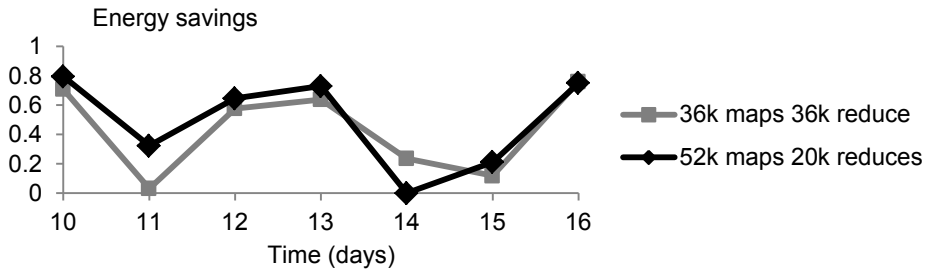



Figure 6.14. Energy savings per day for the latency-bound policy comparison in Figure 6.13(b). Daily energy savings range from 0 to 80%. Neither static policy achieves best energy savings for all days. The policy for 52k map and 20k reduces almost does, except for Day 14.

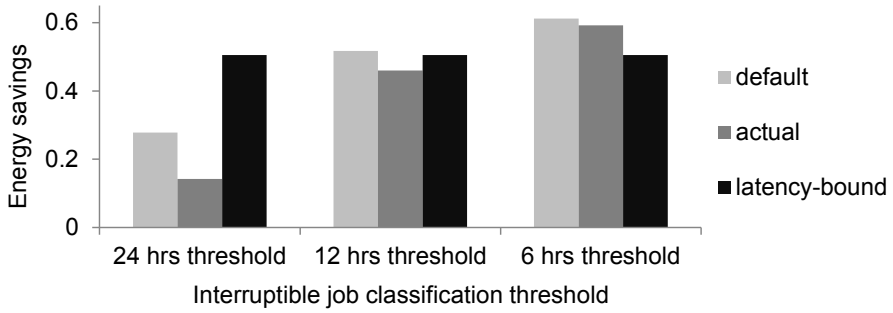



Figure 6.15. Energy savings for different values of `interruptible`. Lowering the threshold leads to increased energy savings for actual and default algorithms. Note that `mapreduceratio` is set to 13:5 and `batchlen` is set to 24 hours. Note that for actual and default algorithms, having a low `interruptible` causes the queue for waiting interrupted jobs to grow without limit; the latency-bound policy is preferred despite seemingly lower energy savings (Section 6.6.5).

### 6.6.6 Overhead

The energy savings come at the cost of increased job latency. Figure 6.16 quantifies the latency increase by looking at normalized job durations for each job type. BEEMR achieves minimal latency overhead for interactive jobs, and some overhead for other job types. This delayed execution overhead buys us energy savings for non-interactive jobs.

For interactive jobs, more than 60% of jobs have ratio of 1.0, approximately 40% of jobs have ratio less than 1.0, and a few outliers have ratio slightly above 1.0. This indicates that a dedicated interactive zone can lead to either unaffected job latency, or even improved job latency from having dedicated resources. The small number of jobs with ratio above 1.0 is caused by peaks in interactive job arrivals. This suggests that it would be desirable to increase the capacity of the interactive zone during workload peaks.

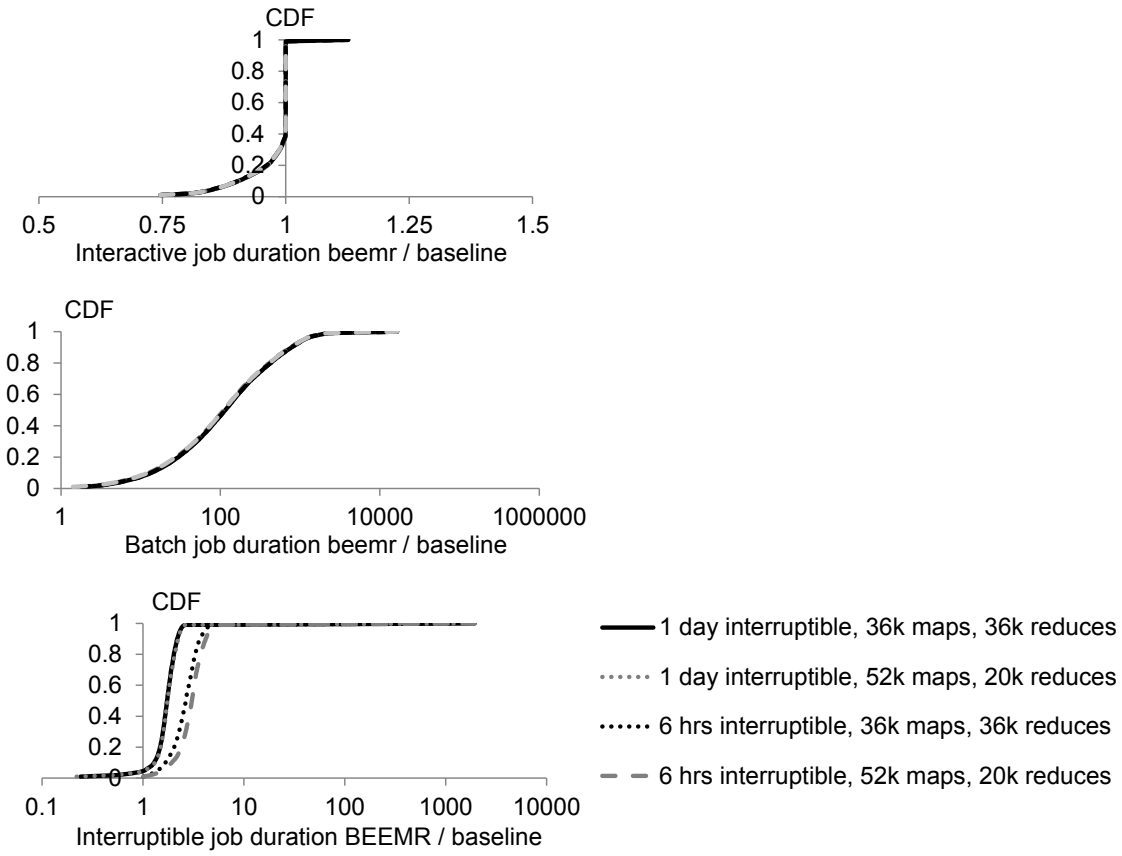



Figure 6.16. Latency ratio by job type between BEEMR with `totalsize` set to 72000, `taskcalc` set to default, `batchlen` set to 24 hours, and (1) `interruptible` set to 24 hours, `mapreduceratio` set to 1:1, or (2) `interruptible` set to 6 hours, `mapreduceratio` set to 13:5; versus the baseline with no batching. A ratio of 1.0 indicates no overhead. Some interactive jobs see improved performance (ratio $< 1$) due to dedicated resources. Some batch jobs have very long delays, the same behavior as delayed execution under deadline-based policies. Interruptible jobs have less overhead than batch jobs, indicating these jobs are truly long running, since delaying them by more than a day translates to relatively low overhead. The delayed execution in non-interactive jobs buys us energy savings.

For batched jobs, the overhead spans a large range. This is caused by the long batch interval, and is acceptable as a matter of policy. A job that arrives just after the beginning of one batch would have a delay of at least one batch interval, leading to large latency. Conversely, a job that arrives just before a batch starts will have almost no delay. This is the same delayed execution behavior as policies in which users specify, say, a daily deadline.

For interruptible jobs, the overhead is also small for most jobs. This is surprising because interruptible jobs can potentially execute over multiple batches. The result in-

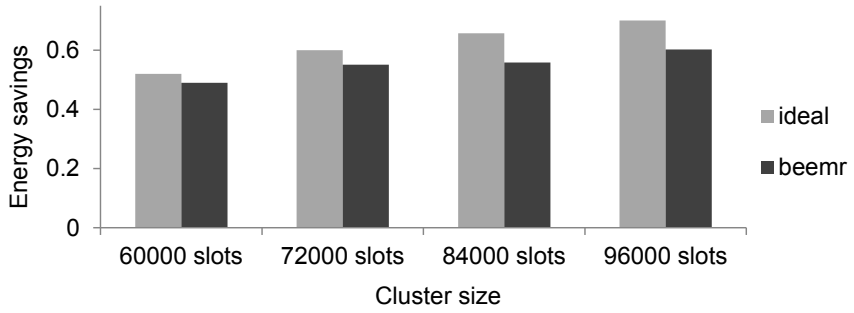



Figure 6.17. Ideal and observed energy savings for different cluster sizes. Both increase as cluster size increases. Note that `batchlen` is set to 24 hours, `taskcalc` is set to default, `mapreduceratio` is set to 13:5, and `interruptible` is set to 6 hours.

dicates that interruptible jobs are truly long running jobs. Executing them over multiple batches imposes a modest overhead.

### 6.6.7 Sensitivity

The size of the cluster affects the amount of energy savings. For example, had we set a cluster size of an arbitrarily large number of task slots, then for the same workload, we could get the trivial result of energy savings arbitrarily close to 100%. Thus, it is important to perform a sensitivity analysis on the energy savings versus cluster size.

So far, we have set a `totalsize` of 72000 task slots and discovered the best parameter values based on this setting. A cluster size of 72000 forms a conservative baseline for energy consumption. Using BEEMR on larger clusters yields the energy savings in Figure 6.17. The ideal energy savings is calculated as 1 - (historical average # of active map tasks + historical average # of reduce tasks) / `totalsize`.

BEEMR extracts most, but not all, of the ideal energy savings. The discrepancy arises from long tasks that hold up batch completion (Section 6.6.2) and transient imbalance between map and reduce slots (Section 6.6.4). If the fraction of time that each batch runs at maximum slot occupancy is already small, then the effects of long tasks and map/reduce slot imbalance are amplified. Thus, as cluster size increases, the gap between BEEMR energy savings and the ideal also increases. One way to narrow the gap would be to extend the batch interval length, thus amortizing the overhead of long tasks holding up batch completion and transient map/reduce slot imbalance. In the extreme case, BEEMR can achieve arbitrarily close to ideal energy savings by running the historical workload in one single batch.

### 6.6.8 Validation

Empirical validation of the simulator provides guidance on how simulation results translate to real clusters. The BEEMR simulator explicitly trades simulation scale and speed for accuracy, making it even more important to quantify the simulation error.

We validate the BEEMR simulator using an Amazon EC2 cluster of 200 "m1.large" instances [22]. We ran three experiments:

1. A series of stand-alone sort jobs,
2. Replay several day-long Facebook workloads using the methodology in [44], which reproduces arrival patterns and data sizes using synthetic MapReduce jobs running on synthetic data, and
3. Replay the same workloads in day-long batches.

For Experiments 1 and 2, we compare the job durations from these experiments to those obtained by a simulator configured with the same number of task slots and the same policies regarding task granularity. For Experiment 3, we compare the energy savings predicted by the simulator to that from the EC2 cluster. These experiments represent an essential validation step before deployment on the actual front-line Facebook cluster running live data and production code.

Figure 6.18 shows the results from stand-alone sort jobs. This ratio is bounded on both ends and is very close to 1.0 for sort jobs of size 100s of MB to 10s of GB. The simulator underestimates the run time (the ratio is less than 1.0) for small sort sizes. There, the overhead of starting and terminating a job dominates; this overhead is ignored by the simulator. The simulator overestimates the run time (the ratio is greater than 1.0) for large sort sizes. For those jobs, there is non-negligible overlap between map and reduce tasks; this overlap is not simulated. The simulation error is bounded for both very large and very small jobs.

Also, there is low variance between different runs of the same job, with 95% confidence intervals from 20 repeated measurements being barely visible in Figure 6.18. Thus, pathologically long jobs caused by task failures or speculative/abandoned executions are infrequent; not simulating these events causes little error.

Figure 6.19 shows the results of replaying one day's worth of jobs, using three different day-long workloads. The ratio is again bounded, and close to 0.75 for the majority of jobs. This is because most jobs in the workload have data sizes in the MB to GB range (Figure 6.1). As explained previously, job startup and termination overhead lead to the simulator to underestimate the duration of these jobs.

Figure 6.20 shows the validation results from batching the three day-long workloads. The simulation error varies greatly between three different days. The average error is 22% of the simulated energy savings (top graph in Figure 6.20). We identify two additional sources of simulator error.

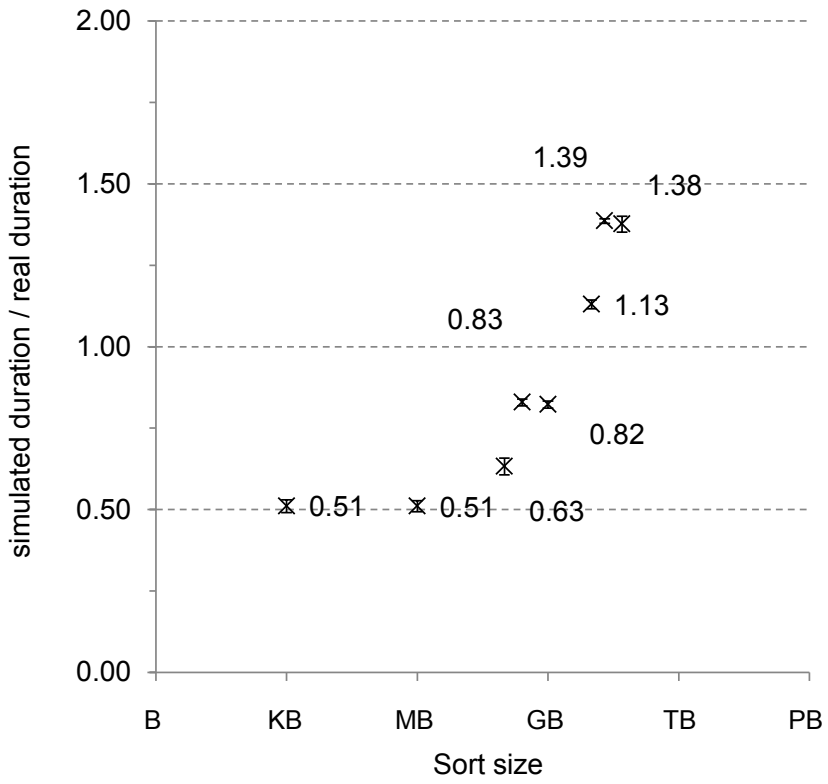



Figure 6.18. Simulator validation for stand-alone jobs. Showing the ratio between simulated job duration and average job duration from 20 repeated measurements on a real cluster. The ratio is bounded for both large and small jobs and is very close to 1.0 for sort jobs of size 100s of MB to 10s of GB.

First, the BEEMR simulator assumes that all available task slots are occupied during the batches. However, on the EC2 cluster, the task slot occupancy averages from 50% to 75% of capacity, a discrepancy again due to task start and termination overhead — the scheduler simply cannot keep all task slots occupied. Adjusting the simulator by using a lower cluster size than the real cluster yields the bottom graph in Figure 6.20, with the error decreased to 13% of the simulated energy savings.

Second, the BEEMR simulator assumes that task times remain the same regardless of whether the workload is executed as jobs arrive, or executed in batch. Observations from the EC2 cluster reveals that during batches, the higher real-life cluster utilization leads to complex interference between jobs, with contention for disk, network, and other resources. This leads to longer task times when a workload executes in batch, and forms another kind of simulation error that is very hard to model.

Overall, these validation results mean that the simulated energy savings of 50-60% (Section 6.6.5) would likely translate to 40-50% on a real cluster.

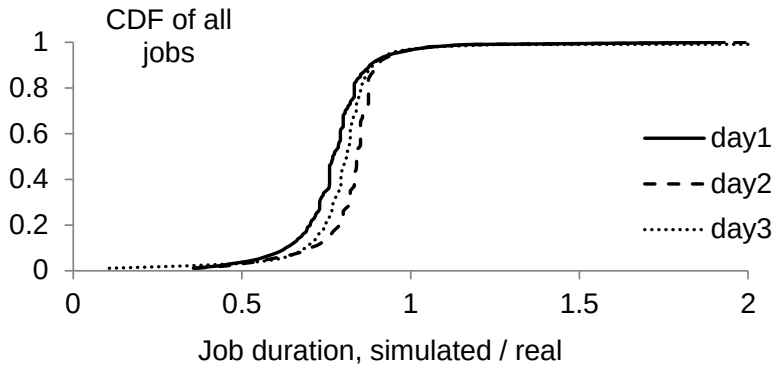



Figure 6.19. Simulator validation for three day-long workloads, without batching. Showing the ratio between simulated and real job duration. This ratio is bounded on both ends and is very close to 0.75 for the vast majority of jobs. Whether this effect biases the energy saving results would depend on further experiments measuring the workload being run in a batch fashion.

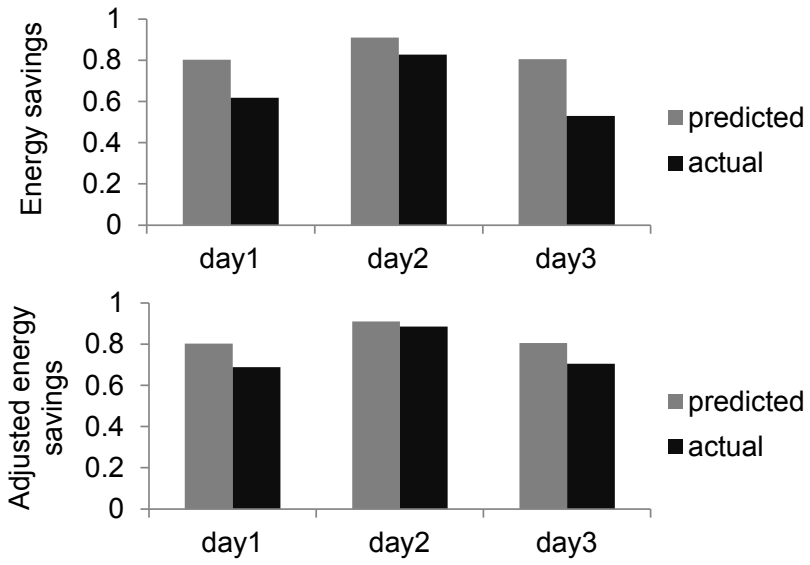



Figure 6.20. Simulator validation for three different day-long workloads, with `batchlen` set to 24 hours. Showing the predicted versus actual energy savings (top graph, average 22% simulation error), and the predicted versus actual energy savings after adjusting for the slot occupancy capacity on the real-life cluster (bottom graph, average 13% simulation error).

## 6.7 Discussion

The results in Section 6.6 raise many interesting questions. Some additional issues require further discussion below.

### 6.7.1 Power Cycles versus Reliability

Transitioning machines to low-power states is one way to achieve power proportionality for MIA workloads while more power proportional hardware is being developed. Large scale adoption of this technique has been limited by worries that power cycling increases failure rates.

There have been few published, large-scale studies that attribute increased failure rates to power cycling. The authors in [111] observed a correlation between the two for disk drives, but point out that correlation could come simply from failed systems needing more reboots to restore. To identify a causal relationship would require a more rigorous methodology, comparing mirror systems servicing the same workload, with the only difference being the frequency of power cycles.

One such comparison experiment ran for 18 months on 100s of machines, and found that power cycling has no effect on failure rates [102]. Larger scale comparisons have been stymied by the small amount of predicted energy savings, and uncertainty about how those energy savings translate to real systems. BEEMR gives empirically validated energy savings of 40-50%. This represents more rigorous data to justify further exploring the thus far unverified relationship between power cycles and failure rates.

### 6.7.2 MIA Generality Beyond Facebook

MIA workloads beyond Facebook lend themselves to a BEEMR-like approach. We analyzed four additional Hadoop workloads from e-commerce, telecommunications, media, and retail companies. These traces come from production clusters of up to 700 machines, and cover 4 cluster-months of behavior. The following gives a summary of the data. Detailed analysis appears in Chapter 4.

One observation that motivated BEEMR is that most jobs access small files that make up a small fraction of stored bytes. (Figure 6.4). This access pattern allows a small interactive zone to service many jobs. Figure 6.21 reproduces the data in Figure 4.4, and shows that such access patterns exist for all workloads. For FB-2010, input paths of $<$ 10GB account for 88% of jobs and 1% of stored bytes. For workloads A and D, the same threshold respectively accounts for 87% and 87% of jobs, and 4% and 2% of stored bytes. For workloads B and C, input paths of $<$ 1TB accounts for 86% and 91% of jobs, as well as 12% and 17% of stored bytes.

Another source of energy savings comes from the high peak-to-average ratio in workload arrival patterns (Figure 6.2). The cluster has to be provisioned for the peak, which makes it important to achieve energy proportionality either in hardware or by workload managers such as BEEMR. For the five workloads (Facebook and workloads A through D), the peak-to-average ratios are: 8.9, 30.5, 23.9, 14.5, and 5.9. BEEMR potentially extracts higher energy savings from workloads with higher peak-to-average arrival ratios, though the exact energy savings and the tradeoff between policy parameters is workload

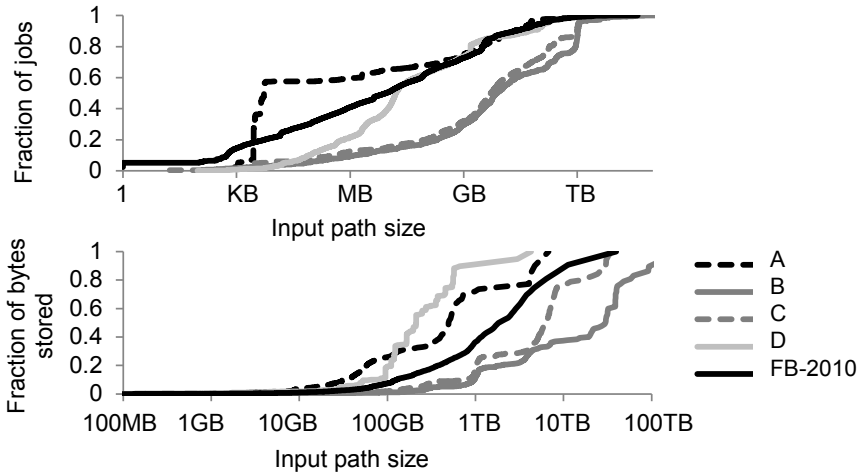



Figure 6.21. Access patterns vs. input path size. Showing fraction of jobs with input paths of a certain size (top) and cummulative fraction of all stored bytes from input paths of a certain size (bottom). Contains data from Figure 6.4 for the FB-2010 workload, and four additional workloads from e-commerce, telecommunications, media, and retail companies. Note the truncated horizontal axis of the lower graph.

specific. These additional workloads give us confidence that the BEEMR architecture can generalize beyond the Facebook workload.

### 6.7.3 Methodology Reflections

Evaluating the energy efficiency of large scale distributed systems presents significant methodological challenges. This chapter strikes a balance between scale and accuracy. Future work could improve on our techniques.

*Simulation vs. replay.* The inherent difference between MIA and other workloads suggest that the best energy efficiency mechanisms would be highly workload dependent. Even for MIA workloads, the behavior varies between use cases (Figure 6.21) and over time (Figure 6.14). Thus, only evaluation over long durations can reveal the true historical savings (Figure 6.14). Days or even weeks-long experiments are unrealistic, especially to explore multiple design options at large scale. Hence, we are compelled to use simulations.

*Choice of simulator.* We considered using Mumak [10] and MRPerf [135]. Mumak requires logs generated by the Rumen tracing tool [116], which is not yet in universal use and not used at Facebook. MRPerf generates a simulation event per data packet and control messages, which limits simulation scale and speed. Neither simulator has been verified at the multi-job workload level. Thus, we developed the BEEMR simulator, which intentionally trades simulation detail and accuracy to gain scale and speed. We also verify the simulator at the workload level (Section 6.6.8).

*Choice of power model.* One accurate way to measure system power is by a power meter

attached at the machine wall socket [42]. This method does not scale to clusters of 1000s of machines. The alternative is to use empirically verified power models, which are yet to be satisfactorily developed for MapReduce. The translation between SPECpower [121] measurements and MapReduce remains unknown, as it is between MapReduce workload semantics and detailed CPU, memory, disk, and network activity. We chose an on-off power model, i.e., machines have "max" power when on and "zero" power when off. This simple model allow us to scale the experiments in size and in time, as well as to capture the fact that machines running Hadoop have a narrow dynamic power range.

*Towards improved methodology.* The deliberate tradeoffs we had to make reflect the nascent performance understanding and modeling of large scale systems such as MapReduce. We encourage the research community to seek to overcome the methodology limitations of this study.

## 6.8 Chapter Conclusions

BEEMR is able to cut the energy consumption of a cluster almost *in half* (after adjusting for empirically quantified simulation error) without harming the response time of latency-sensitive jobs or relying on storage replication, while allowing jobs to retain the full storage capacity and compute bandwidth of the cluster. BEEMR achieves such results because its design was guided by a thorough analysis of a real-world, large-scale instance of the targeted workload. We dubbed this widespread yet under-studied workload MIA. The key insight from our analysis of MIA workloads is that although MIA clusters host huge volumes of data, the interactive jobs operate on just a small fraction of the data, and thus can be served by a small pool of dedicated machines; the less time-sensitive jobs can run in a batch fashion on the rest of the cluster.

In the broader context of the dissertation, this chapter illustrates an example of the workload-driven design and evaluation process for large-scale data-centric systems. We apply the workload behavior insights from earlier in the dissertation to improve energy efficiency for a type of large-scale data-centric system. This problem is inherently workload dependent, and it is natural to adopt the systems view of the workload developed in Chapter 3 and expand in Chapters 4 and 5. In the next chapter, we provide another illustration of the workload-driven design and evaluation process. The problem to be solved is TCP incast, a pathological behavior of the network transport protocol. Again, the problem is inherently workload dependent, and the systems view of the workload allow us to translate low level network effects to higher workload-level performance metrics.

# Chapter 7

# Workload-Driven Design and Evaluation - TCP Incast

*Prescribe the medicine to fit the disease.*

— *Hua Tuo, quoted in The Histories of the Three Kingdoms.*

This is the second of two chapters that illustrate how workload analysis insights translate to the actual design and evaluation of large-scale data-centric systems. The chapter further illustrates that the design and evaluation methods developed in the dissertation helps engineers evaluate the importance of a known problem in the context of large-scale data-centric workloads.

We **investigate the TCP incast pathology in the context of large-scale data-centric workloads**. TCP incast is a recently identified network transport problem that affects many-to-one communication pattnerns in Internet datacenters. Recent years has seen several attempts to mitigate or avoid incast. At the same time, some doubts remained with regard to how much it impacts large-scale data-centric systems such as MapReduce. We present a quantitative, empirically verified model for predicting TCP incast throughput collapse. We contribute workloads-driven evaluation results that quantify the performance impact of TCP incast under realistic settings. The workload insights developed earlier in the dissertation further allow us to assess TCP incast in the context of future design priorities.

The first half of this chapter develops and validates a quantitative model that accurately predicts the onset of incast and TCP behavior both before and after incast occurs (Section 7.2). The second half of this chapter investigates how incast affects the Apache Hadoop implementation of MapReduce, an important example of a large-scale data-centric application (Section 7.3). We close the chapter by reflecting on some technology and data analysis trends surrounding large-scale data-centric systems, speculating on how these trends interact with incast (Section 7.4), making recommendations for datacenter operators, and discussing the broader implications of the work (Section 7.5).

## 7.1 Motivation

TCP incast is a recently identified network transport pathology that affects many-to-one communication patterns in datacenters [109, 45, 17]. It is caused by a complex interplay between datacenter applications, the underlying switches, network topology, and TCP, which was originally designed for wide area networks. Incast increases the queuing delay of flows, and decreases application-level throughput to far below the link bandwidth. The problem especially affects computing paradigms in which distributed processing cannot progress until all parallel threads in a stage complete. Examples of such paradigms include distributed file systems, web search, advertisement selection, and other applications with partition or aggregation semantics [109, 45, 17].

There have been many proposed solutions for incast. Representative approaches include modifying TCP parameters [130, 45] or its congestion control algorithm [138], optimizing application-level data transfer patterns [109, 80], switch level modifications such as larger buffers [109] or explicit congestion notification (ECN) capabilities [17], and link layer mechanisms such as Ethernet congestion control [9, 18]. Application-level solutions are the least intrusive to deploy; they can be deployed without changing the underlying Internet datacenter infrastructure, but require modifying each and every application. Switch and link level solutions require modifying the underlying datacenter infrastructure, and are likely to be logistically feasible only during hardware upgrades.

Unfortunately, despite these solutions, we still have no quantitatively accurate and empirically validated model to predict incast behavior. Similarly, despite many studies demonstrating incast for microbenchmarks, we still do not understand how incast impacts application-level performance subject to real life complexities in configuration, scheduling, data size, and other environmental and workload properties. These concerns create justified skepticism on whether we truly understand incast at all, whether it is even an important problem for a wide class of workloads, and whether it is worth the effort to deploy various incast solutions in front-line, business-critical datacenters.

We seek to understand how incast impacts the emerging class of large-scale data-centric workloads. These workloads help solve needle-in-a-haystack type problems and extract actionable insights from large scale, potentially complex and unformatted data. We do not propose in this chapter yet another solution for incast. Rather, we focus on developing a deep understanding of one existing solution: modifying the TCP stack in the OS kernel to reduce the mininum length of TCP retransmission time out (RTO) [106] from 200ms to 1ms [130, 45]. TCP incast is fundamentally a transport layer problem, thus a solution at this level is best.

## 7.2 Towards an Analytical Model

We use a simple network topology and workload to develop an analytical model for incast, shown in Figure 7.1. This is the same setup as that used in prior work [109, 130, 45], and is representative of the network traffic patterns on enterprise network storage systems

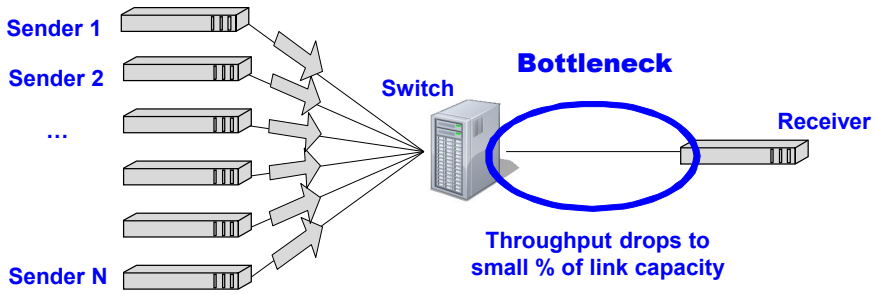



Figure 7.1. Simple setup to observe incast. The receiver requests $k$ blocks of data from a set of $N$ storage servers. Each block is striped across $N$ storage servers. For each block request received, a server responds with a fixed amount of data. Clients do not request block $k+1$ until all the fragments of block $k$ have been received.

that store files in partitions or stripes across different servers. We choose this topology and workload to make the analysis tractable.

The workload is as follows. The receiver requests $k$ blocks of data from a set of $N$ storage servers — in our experiments $k = 100$ and $N$ varies from 1 to 48. Each block is stored in partitions or stripes across $N$ storage servers. For each block request received, a server responds with a fixed amount of data. Clients do not request block $k+1$ until all the fragments of block $k$ have been received — this leads to a *synchronized read pattern* of data requests [109, 130, 45]. We re-use the storage server and client code in [109, 130, 45]. The performance metric for these experiments is *application-level goodput*, i.e., the total bytes received from all senders divided by the finishing time of the *last* sender.

We conduct our experiments on the DETER Lab testbed [124], where we have full control over the non-virtualized node OS, as well as the network topology and speed. We used 3GHz dual-core Intel Xeon machines with 1Gbps network links. The nodes run standard Linux 2.6.28.1. This was the most recent mainline Linux distribution in late 2009, when we obtained our prior results [45]. We perform experiments using both a relatively shallow-buffered Nortel 5500 switch (4KB per port) [98], and a more deeply buffered HP Procurve 5412 switch (64KB per port) [73].

### 7.2.1 Flow rate models

The simplest model for incast is based on two competing behaviors as we increase $N$, the number of concurrent senders. The first behavior occurs before the onset of incast, and reflects the intuition that goodput is the block size divided by the transfer time. Ideal transfer time is just the sum of a round trip time (RTT) and the ideal send time. Equation 7.1 captures this idea.

$$
\begin{aligned}
Goodput_{beforeIncast} &= idealGoodputPerSender \times N \\
&= \frac{blockSize}{idealTransferTime} \times N \\
&= \frac{blockSize}{RTT + \frac{blockSize}{perSenderBandwidth}} \times N \\
&= \frac{blockSize}{RTT + \frac{blockSize \times N}{linkBandwidth}} \times N
\end{aligned}
\tag{7.1}
$$

Incast occurs when there are some $N > 1$ concurrent senders, and the goodput drops significantly. After the onset of incast, TCP retransmission time out (RTO) represents the dominant effect. Transfer time becomes RTT + RTO + ideal send time, as captured in Equation 7.2. The goodput collapse represents a transition between the two behavior modes. We see in Section 7.2.3 that TCP fast retransmit [20] does not get triggered, making RTO the primary effect.

$$
\begin{aligned}
Goodput_{incast} &= goodputPerSender \times N \\
&= \frac{blockSize}{RTO + idealTransferTime} \times N \\
&= \frac{blockSize}{RTO + RTT + \frac{blockSize}{perSenderCapacity}} \times N \\
&= \frac{blockSize}{RTO + RTT + \frac{blockSize \times N}{linkCapacity}} \times N
\end{aligned}
\tag{7.2}
$$

Figure 7.2 gives some intuition with regard to Equations 7.1 and 7.2. We substitute $blockSize$ = 64KB, 256KB, 1024KB, and 64MB, as well as $RTT$ = 1ms, and $RTO$ = 200ms. Before the onset of incast (Equation 7.1), the goodput increases as $N$ increases, though with diminishing rate, asymptotically approaching the full link bandwidth. The curves move vertically upwards as block size increases. This reflects the fact that larger blocks result in a larger fraction of the ideal transfer time spent transmitting data, versus waiting for an RTT to acknowledge that the transmission completed. After incast occurs (Equation 7.2), RTO dominates the transfer time for small block sizes. Again, larger blocks lead to RTO forming a smaller ratio versus ideal transmission time. The curves move vertically upwards as block size increases.

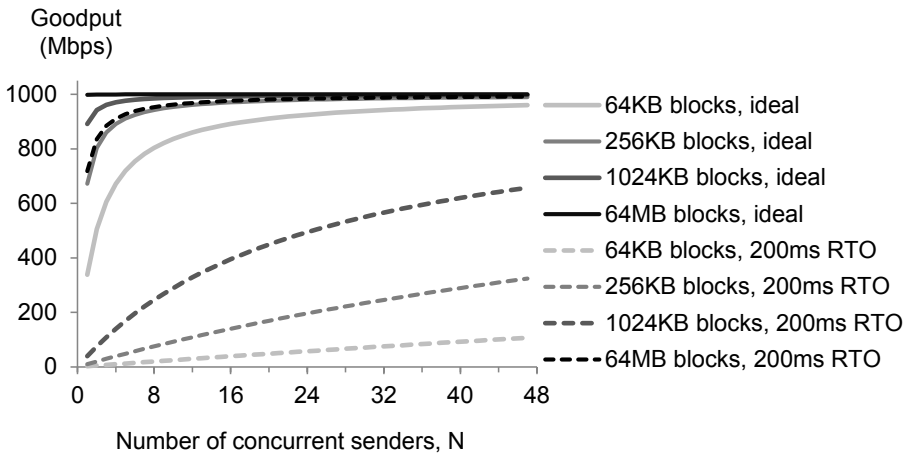



Figure 7.2. Flow rate model for incast. Showing ideal behavior (solid lines, Equation 7.1) and incast behavior caused by RTOs (dotted lines, Equation 7.2). We substitute $blockSize$ = 64KB, 256KB, 1024KB, and 64MB, as well as $RTT$ = 1ms, and $RTO$ = 200ms. The incast goodput collapse comes from the transition between the two TCP operating modes.

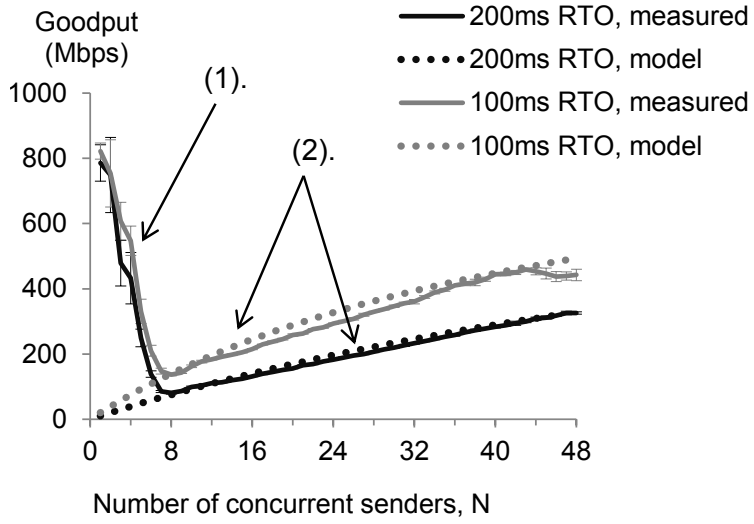



Figure 7.3. Empirical verification of flow rate incast model. The graph shows our previously presented data in [45]. The $blockSize$ is 256KB, $RTO$ is set to 100ms and 200ms, and the model uses $RTT$ = 1ms. Error bars represent 95% confidence interval around the average of 5 repeated measurements. The switch is a Nortel 5500 (4KB per port). Showing (1) incast goodput collapse begins at $N$ = 2 senders, and (2) behavior after goodput collapse verifies Equation 7.2.

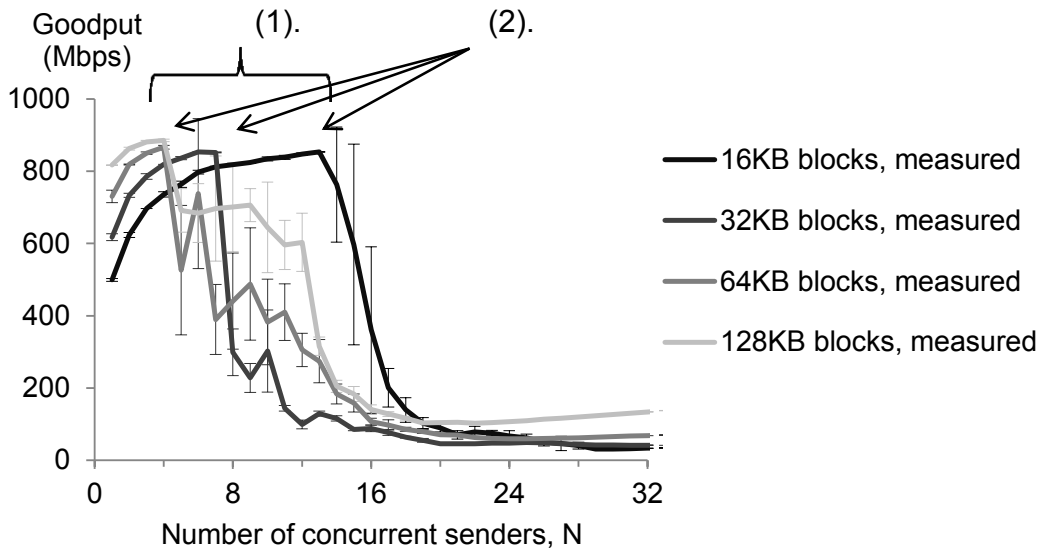



Figure 7.4. Empirical verification of flow rate TCP model before onset of incast. Measurements done on HP Procurve 5412 switches (64KB per port). *RTO* is 200ms. Error bars represent 95% confidence interval around the average of 5 repeated measurements. Showing (1) behavior before goodput collapse verifies Equation 7.1, and (2) onset of incast goodput collapse predicted by switch buffer overflow during slow start (Equation 7.3).

### 7.2.2 Empirical verification

This model matches well with our empirical measurements. Figure 7.3 superpositions the model on our previously presented data in [45]. There, we fix *blockSize* at 256KB and set *RTO* to 100ms and 200ms. The switch is a Nortel 5500 (4KB per port). For simplicity, we use $RTT = 1$ms for the model. Goodput collapse begins at $N = 2$, and we observe behavior for Equation 7.2 only. The empirical measurements (solid lines) match the model (dotted-lines) almost exactly.

We use a more deeply buffered switch to verify Equation 7.1. As we discuss later, the switch buffer size determines the onset of incast. Figure 7.4 shows the behavior using the HP Procurve 5412 switch (64KB per port). Behavior before goodput collapse verifies Equation 7.1 — the goodput increases as $N$ increases, though with diminishing rate; the curves move vertically upwards as block size increases. We can see this graphically by comparing the curves in Figure 7.4 before the goodput collapse to the corresponding curves in Figure 7.2.

*Insight: Flow rate model of Equation 7.1 captures behavior before onset of incast. TCP RTO dominates behavior after onset of incast (Equation 7.2).*

| Round trip # | Cwnd size, 16KB blocks | Cwnd size, 32KB blocks | Cwnd size, 64KB blocks | Cwnd size, 128KB blocks |
|---|---|---|---|---|
| 1 | 1,448 | 1,448 | 1,448 | 1,448 |
| 2 | 2,896 | 2,896 | 2,896 | 2,896 |
| 3 | 5,792 | 5,792 | 5,792 | 5,792 |
| 4 | 5,864 | 11,584 | 11,584 | 11,584 |
| 5 | | 10,280 | 23,168 | 23,168 |
| 6 | | | 19,112 | 46,336 |
| 7 | | | | 36,776 |

Table 7.1. TCP slow start congestion window size in bytes versus number of round trips. Showing the behavior for $blockSize$ = 16KB, 32KB, 64KB, 128KB. We verified using `sysctl` that Linux begins at (2× base MSS) = 1448 bytes.

### 7.2.3 Predicting the onset of incast

Figure 7.4 also shows that goodput collapse occurs at different $N$ for different block sizes. We can predict the location of the onset of goodput collapse by detailed modeling of TCP slow start and buffer occupancy. Table 7.1 shows the slow start congestion window sizes versus each packet round trip. For 16KB blocks, 12 concurrent senders of the largest congestion window of 5864 bytes would require 70368 bytes of buffer, larger than the available buffer of 64KB per port. Goodput collapse begins after $N = 13$ concurrent senders. The discrepancy of 1 comes from the fact that there is additional "buffer" on the network beyond the packet buffer on the switch, e.g., packets in flight or buffer at the sender machines. According to this logic, goodput collapse should take place according to Equation 7.3. The equation accurately predicts that for Figure 7.4, the goodput collapse for 16KB, 32KB, and 64KB blocks begin at 13, 7, and 4 concurrent senders, and for Figure 7.3, the goodput collapse is well underway at 2 concurrent senders.

$$N_{initialGoodputCollapse} = \left\lceil \frac{perSenderBuffer}{largestSlowStartCwnd} \right\rceil + 1 \tag{7.3}$$

This analysis also indicates why TCP fast retransmit [20] fails to prevent RTOs from taking place. Fast retransmit requires three duplicate acknowledgements to trigger. Table 7.1 shows that packet loss likely occurs during the last or second last round trips. Hence, connections with little additional data to send after a packet loss would likely observe only a single or double duplicate ACK. Fast retransmit does not get triggered, and the connection enters RTO soon after.

*Insight: For small flows, the switch buffer space determines the onset of incast.*

### 7.2.4 Second order effects

Figure 7.4 also suggests the presence of second order effects not explained by Equations 7.1 to 7.3. Equation 7.3 predicts that goodput collapse for 128KB blocks should

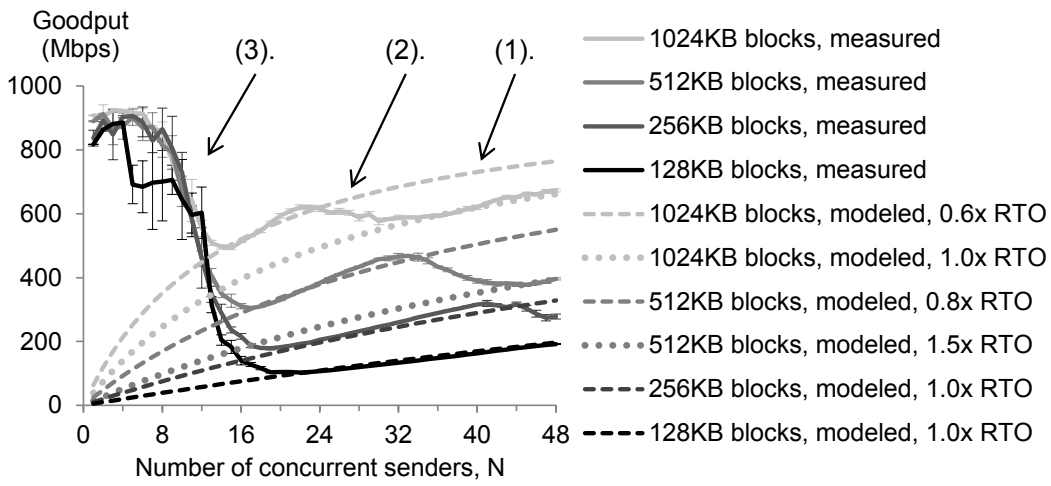



Figure 7.5. 2nd order effects other than RTO during slow start. Measurements done on HP Procurve 5412 switches (64KB per port). *RTO* is 200ms. Error bars represent 95% confidence interval around the average of 5 repeated measurements. Showing (1) partial RTOs more accurately modeling incast behavior for large blocks, (2) transition between single and multiple partial RTOs, and (3) gradual onset of incast that is independent of $blockSize$.

begin at $N = 2$ concurrent senders, while the empirically observed goodput collapse begins at $N = 4$ concurrent senders. It turns out that block sizes of 128KB represent a transition point from RTO-during-slow-start to more complex modes of behavior.

We repeat the experiment for $blockSize$ = 128KB, 256KB, 512KB, and 1024KB. Figure 7.5 shows the results, which includes several interesting effects.

First, for $blockSize$ = 512KB and 1024KB, the goodput immediately after the onset of incast is given by Equation 7.4. It differs from Equation 7.2 by the multiplier $\alpha$ for the $RTO$ in the denominator. This $\alpha$ is an empirical constant, and represents a behavior that we call partial RTO. What happens is as follows. When RTO takes place, TCP SACK (turned on by default in Linux) allows transmission of further data, until the congestion window can no longer advance due to the lost packet [88]. Hence, the link is idle for a duration of less than the full RTO value. Hence we call this effect partial RTO. For $blockSize$ = 1024KB, $\alpha$ is 0.6, and for $blockSize$ = 512KB, $\alpha$ is 0.8.

$$Goodput_{incast} = \frac{blockSize}{\alpha \times RTO + RTT + \frac{blockSize \times N}{linkBandwidth}} \times N \tag{7.4}$$

Second, beyond a certain number of concurrent senders, $\alpha$ transitions to something

that approximately doubles its initial value (0.6 to 1.0 for $blockSize$ = 1024KB, 0.8 to 1.5 for $blockSize$ = 512KB). This simply represents that two partial RTOs have occurred.

Third, the goodput collapse for $blockSize$ = 256KB, 512KB, and 1024KB is more gradual compared with the cliff-like behavior in Figure 7.4. Futher, this gradual goodput collapse has the same slope across different $blockSize$. Two factors explain this behavior. First, flows with $blockSize \geq$ 128KB have a lot more data to send even after the buffer space is filled with packets sent during slow start (Equation 7.3 and Table 7.1). Second, even when the switch drops packets, TCP can sometimes recover. Empirical evidence of this fact exists in Figure 7.4. There, for $blockSize$ = 16KB and $N$ = 13 to 16 concurrent senders, at least one of five repeated measurements manages to get goodput close to 90% of link capacity, as reflected by the large error bars. Goodput collapse happens for other runs because the packets are dropped in a way that a connection with little additional data to send would observe only a single or double duplicate ACK, and go into RTO soon after. Larger blocks suffer less from this problem because the ongoing data transfers triggers triple duplicate ACK with higher probability. Thus, the connection retransmits, enters congestion avoidance, and avoids RTO. Hence the gradual goodput collapse.

The limited transmit mechanism [19] potentially addresses the problem of single or double duplicate ACK followed by RTO. The limited transmit mechanism sends a new data segment in response to each of the first two duplicate acknowledgments that arrive at the sender. Turning on this mechanism potentially creates spurious retransmission, since duplicate ACKs could be caused by either packet losses or packet re-ordering. A measurement of packet re-ordering properties in Internet datacenter networks would clarify the cost-benefit tradeoffs.

We should also point out that SACK semantics are independent of duplicate ACKs, since SACK is layered on top of existing cummulative ACK semantics [88].

*Insight: Second order effects include partial RTO due to SACK, multiple partial RTOs, and triple duplicate ACKs causing more gradual onset of incast.*

### 7.2.5 Good enough model

Unfortunately, some parts of the model remain qualitative. We admit that the full interaction between triple duplicate ACKs, slow start, and available buffer space requires elaborate treatment far beyond the flow rate and buffer occupancy analysis presented here (Equations 7.1 to 7.4).

That said, the models here represent the first time we quantitatively explain major features of the incast goodput collapse. Comparable results in related work [138, 109] can be explained by our models also. The analysis allows us to reason about the significance of incast for future large-scale data-centric workloads later in the chapter.

## 7.3 Incast in Hadoop MapReduce

Hadoop represents an interesting case study of how incast affects application level behavior. Network traffic in Hadoop consists of small flows carrying control packets for various cluster coordination protocols, and larger flows carrying the actual data being processed. Incast potentially affects Hadoop in complex ways. Further, Hadoop may well mask incast behavior, because network data transfers forms only a part of the overall computation and data flow. Our goal for this section is to answer whether incast affects Hadoop, by how much, and under what circumstances.

We perform two sets of experiments. First, we run stand-alone, artificial Hadoop jobs to find out how much incast impacts each component of the MapReduce data flow. Second, we replay a scaled-down, real life production workload using previously published tools [44] and cluster traces from Facebook, a leading Hadoop user, to understand the extent to which incast affects whole workloads. These experiments take place on the same DETER machines as those the previous section. We use only the large buffer Procurve switch for these experiments.

### 7.3.1 Stand-alone jobs

Table 7.2 lists the Hadoop cluster settings we considered. The actual stand-alone Hadoop jobs are `hdfsWrite`, `hdfsRead`, `shuffle`, and `sort`. The first three jobs stress one part of the Hadoop IO pipeline at a time. Sort represents a job with 1-1-1 ratio between read, shuffled, and written data. We implement these jobs by modifying the `randomwriter` and `randomtextwriter` examples that are pre-packaged with recent Hadoop distributions. We set the jobs to write, read, shuffle, or sort 20GB of terasort format data [13] on 20 machines.

#### 7.3.1.1 Experiment setup

The TCP versions are the same as before – standard Linux 2.6.28.1, and modified Linux 2.6.28.1 with `tcp_rto_min` set to 1ms. We consider Hadoop versions 0.18.2 and 0.20.2. Hadoop 0.18.2 is considered a legacy and basic, but still relatively stable and mature distribution. Hadoop 0.20.2 is a more fully featured distribution that introduces some performance overhead for small jobs [44]. Subsequent Hadoop improvements have appeared on several disjoint branches that are currently being merged, and 0.20.2 represents the last time there was a single mainline Hadoop distribution [143].

The rest of the parameters are detailed Hadoop configuration settings. Tuning these parameters can considerably improve performance, but requires specialist knowledge about the interaction between Hadoop and the cluster environment. The first value for each configuration parameter in Table 7.2 represents the default setting. The remaining values are tuned values, drawn from a combination of Hadoop sort benchmarking [1], suggestions from enterprise Hadoop vendors [12], and our own experiences tuning Hadoop

| Parameter | Values |
|---|---|
| Hadoop jobs | hdfsWrite, hdfsRead, shuffle, sort |
| TCP version | Linux-2.6.28.1, 1ms-min-RTO |
| Hadoop version | 0.18.2, 0.20.2 |
| Switch model | HP Procurve 5412 |
| Number of machines | 20 workers and 1 master |
| fs.inmemory.size.mb | 75, 200 |
| io.file.buffer.size | 4096, 131072 |
| io.sort.mb | 100, 200 |
| io.sort.factor | 10, 100 |
| dfs.block.size | 67108864, 536870912 |
| dfs.replication | 3, 1 |
| mapred.reduce.parallel.copies | 5, 20 |
| mapred.child.java.opts | -Xmx200m, -Xmx512M |

Table 7.2. Hadoop parameter values for experiments with stand-alone jobs. The `mapred.child.java.opts` parameter sets the maximum virtual memory of the Java child processes to 200MB and 512MB.

for performance. One configuration worth further explaining is `dfs.replication`. It controls the degree of data replication in HDFS. The default setting is three-fold data replication to achieve fault tolerance. For use cases constrained by storage capaciy, the preferred method is to use HDFS RAID [36], which achieves fault tolerance with 1.4× overhead, much closer to the ideal one-fold replication.

#### 7.3.1.2 Results

Figure 7.6 shows the results for Hadoop 0.18.2. We consider two performance metrics — job completion time, and incast overhead. We define incast overhead according to Equation 7.5, i.e., difference between job completion time under default and 1ms-min-RTO TCP, normalized by the job completion time for 1ms-min-RTO TCP. The default Hadoop has very high incast overhead, while for tuned Hadoop, the incast overhead is barely visible. However, the tuned Hadoop-0.18.2 setting leads to considerably lower job completion times.

$$\begin{aligned} t &= jobCompletionTime \\ IncastOverhead &= \frac{t_{defaultTCP} - t_{1ms-min-RTO}}{t_{1ms-min-RTO}} \end{aligned} \tag{7.5}$$

The results illustrate a subtle form of Amdahl's Law, which explains overall improvement to a system when only a part of the system is being improved. Here, the amount of incast overhead depends on how much network data transfers contribute to the overall job completion time. The default Hadoop configurations lead to network transfers contributing to a large fraction of the overall job completion time. Thus, incast overhead is

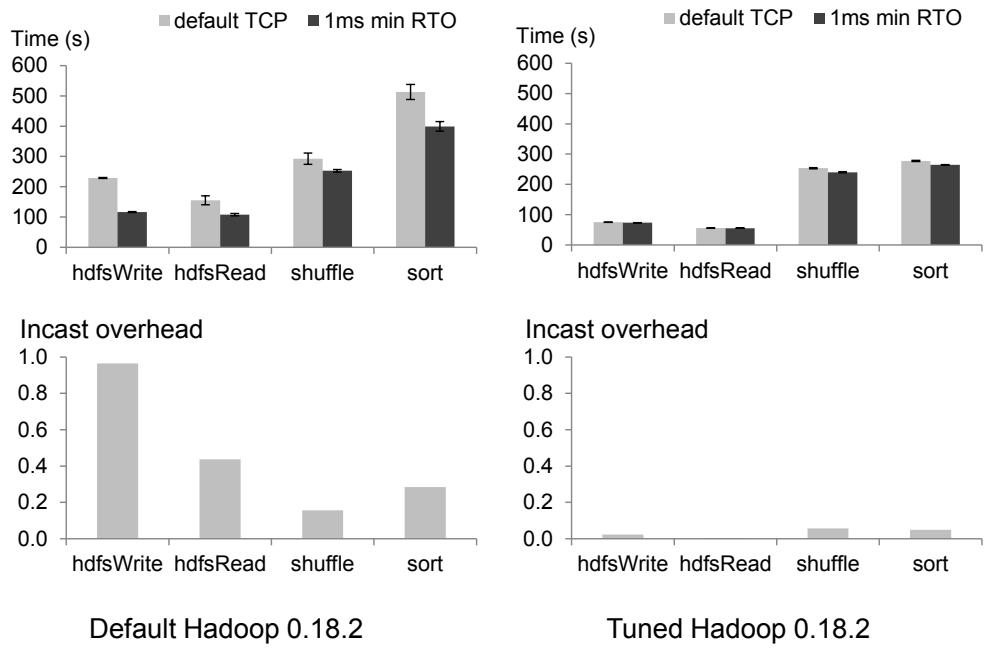



Figure 7.6. Hadoop stand alone job completion times. HP Procurve 5412 switches. Showing job completion times (top) and overhead introduced by incast (bottom) for default Hadoop-0.18.2 (left), and tuned Hadoop-0.18.2 (right). The error bars show 95% confidence intervals from 20 repeated measurements. The tuned Hadoop-0.18.2 leads to considerably lower job completion times. The default Hadoop has higher incast overhead.

clearly visible. Conversely, for tuned Hadoop, overall job completion time is already low. Incast overhead is barely visible because the network transfer time is low.

We repeat these measurements on Hadoop 0.20.2. Compared with Hadoop 0.18.2, the more recent version of Hadoop sees a performance improvement for the default configuration. For the optimized configuration, Hadoop 0.20.2 sees performance overhead of around 10 seconds for all four job types. This result is in line with our prior comparisons between Hadoop versions 0.18.2 and 0.20.2 [44]. Unfortunately, 10 seconds is also the performance improvement for using TCP with 1ms-min-RTO. Hence, the performance overhead in Hadoop 0.20.2 masks the benefits of addressing incast.

*Insight: Incast does affect Hadoop. The performance impact depends on cluster configurations, as well as data and compute patterns in the workload.*

### 7.3.2 Real life production workloads

The results in the above subsection indicate that to find out how much incast *really* affects Hadoop, we must compare the default and 1ms-min-RTO TCP while replaying real life production workloads.

Previously, such evaluation capabilities are exclusive to enterprises that run large scale production clusters. Recent years have witnessed a slow but steady growth of public knowledge about front line production workloads [140, 27, 44, 40, 25], as well as emerging tools to replay such workloads in the absence of production data, code, and hardware [44, 41].

#### 7.3.2.1 Workload analysis

We obtained seven production Hadoop workload traces from five companies in social networking, e-commerce, telecommunications, and retail. Among these companies, only Facebook has so far allowed us to release their name and synthetic versions of their workload. The following present some summary statistics of the more detailed analysis in Chapter 4.

Several observations are especially relevant to incast. Consider Figure 7.7, reproduced from Figure 4.2, which shows the distribution of per job input, shuffle, and output data for all workloads. First, all workloads are dominated by jobs that involve data sizes of less than 1GB. For jobs so small, scheduling and coordination overhead dominate job completion time. Therefore, incast will make a difference only if the workload intensity is high enough that Hadoop control packets alone would overwhelm the network. Second, all workloads contain jobs at the 10s TB or even 100s TB scale. This compels the operators to use Hadoop 0.20.2. This version of Hadoop is the first to incorporate the Hadoop fair scheduler [140]. Without it, the small jobs arriving behind very large jobs would see FIFO head-of-queue blocking, and suffer wait times of hours or even days. This feature is so critical that cluster operators use it despite the performance overhead for small jobs [140, 143]. Hence, it is likely that in Hadoop 0.20.2, incast will be masked by the performance overhead.

#### 7.3.2.2 Workload replay

We replay a day-long Facebook 2009 workload on the default and 1ms-min-RTO versions of TCP. We synthesize this workload using the method in [44]. It captures, in a relatively short synthetic workload, the representative job submission and computation patterns for the entire six-month trace.

Our measurements confirm the hypothesis earlier. Figure 7.8 shows the distribution of job completion times. We see that the distribution for 1ms-min-RTO is 10–20 seconds right shifted compared with the distribution for default TCP. This is in line with the 10–20 seconds overhead we saw in the workload-level measurements in [44], as well as the stand-alone job measurements earlier in the dissertation. The benefits of addressing incast are completely masked by overhead from other parts of the system.

We should highlight that the distribution of job durations is *longer* for the 1ms-min-RTO TCP. This result potentially shows the performance cost of spurious RTOs. Spurious RTOs arise from setting mininum RTO too low, which cause the TCP sender

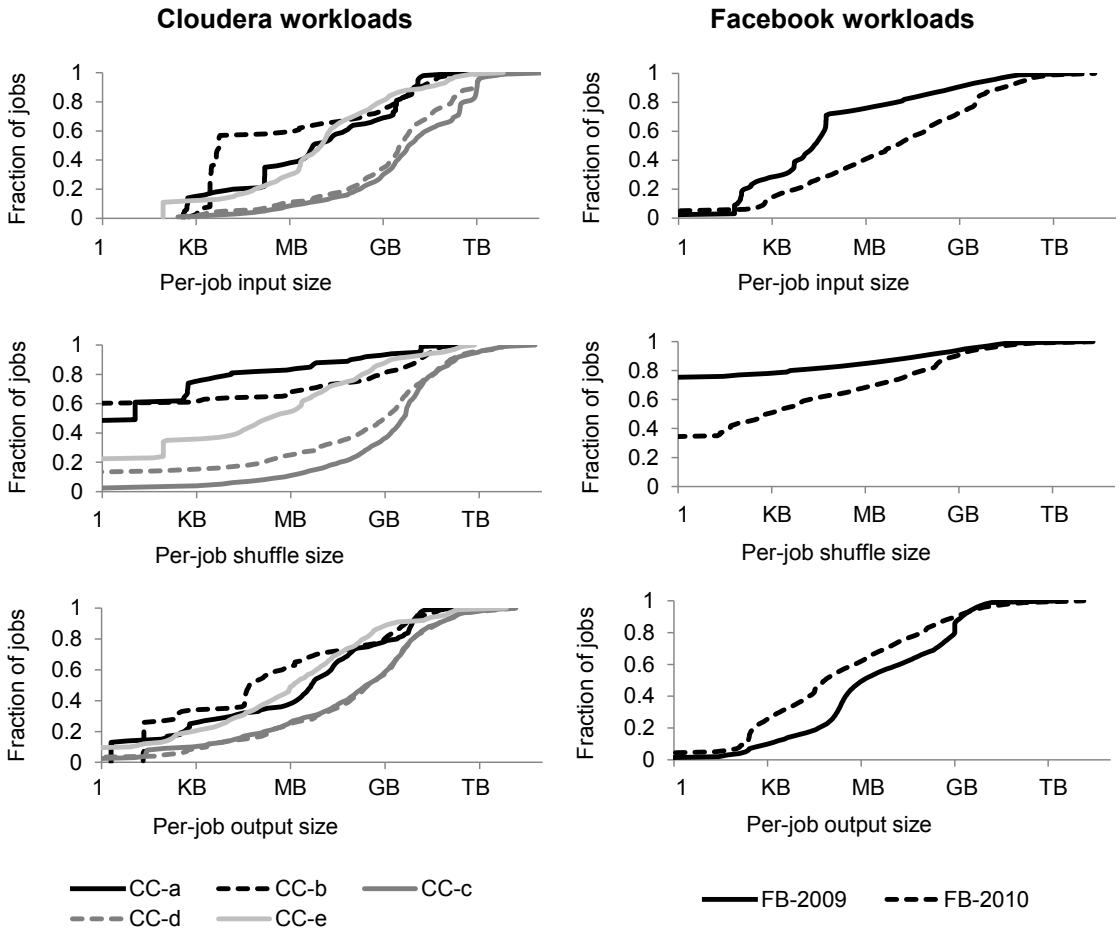



Figure 7.7. Per job input, shuffle, and output size for each workload. `FB-*` workloads come from a six-months cluster trace in 2009 and a 45-days trace in 2010. `CC-*` workoads come from traces of up to 2 months long at various customers of Cloudera, which is a vendor of enterprise Hadoop.

to retransmit a packet that is delayed but not lost. We could verify whether this is indeed the case by looking at TCP packet level traces. However, running `tcpdump` [14] on our machines interferes network performance. Better tracing infrastructure should be explored.

Figure 7.9 offers another perspective on workload level behavior. The graphs show two sequences of 100 jobs, ordered by submission time, i.e., we take snapshots of two continuous sequences of 100 jobs out of the total 6000+ jobs in a day. These graphs indicate the behavior complexity once we look at the entire workload of thousands of jobs and diverse interactions between concurrently running jobs. The 10–20 seconds performance difference on small jobs becomes insignificant noise in the baseline. The few large jobs take significantly longer than the small jobs, and stand out visibly from the

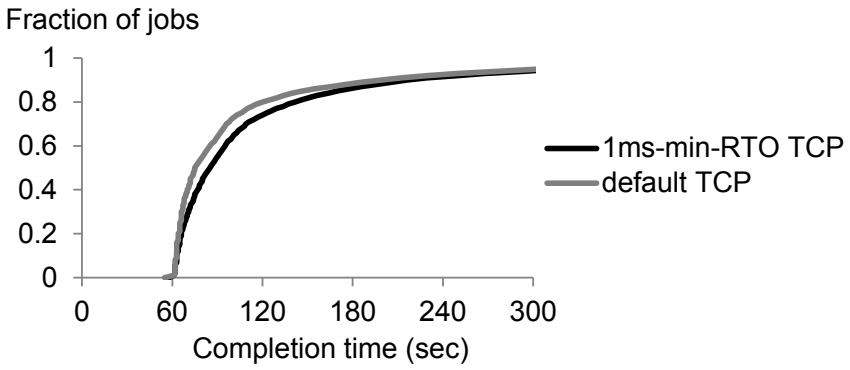



Figure 7.8. Distribution of job completion times for the `FB-2009` workload. The distribution for 1ms-min-RTO is 10–20 seconds right shifted compared with the distribution for default TCP.

baseline. For these jobs, there are no clear patterns to the performance of 1ms-min-RTO versus standard TCP.

The Hadoop community is aware of the performance overheads in Hadoop 0.20.2 for small jobs. Subsequent versions partially address these concerns [86]. It would be worthwhile to repeat these experiments once the various active Hadoop code branches merge back into the next mainline Hadoop [143].

*Insight: Small jobs dominate several production Hadoop workloads. Non-network overhead in present Hadoop versions mask incast behavior for these jobs.*

## 7.4 Incast for Future Large-Scale Data-Centric Workloads

Hadoop is an example of the rising class of large-scale data-centric computing paradigms, which almost always involve some amount of network communications. To understand how incast affects future large-scale data-centric workloads, one needs to appreciate the technology trends that drives the rising prominence of such systems, the computational demands that result, the countless design and mis-design opportunities, as well as the root causes of incast.

As discussed in Chapter 1, the top technology trends driving the prominence of large-scale data-centric include increasingly easy and economical access to large scale storage and computation infrastructure [5, 23], ubiquitous ability to generate, collect, and archive data about both technology systems and the physical world [56], and growing desire and statistical literacy across many industries to understand and derive value from large datasets [7, 37, 91, 76].

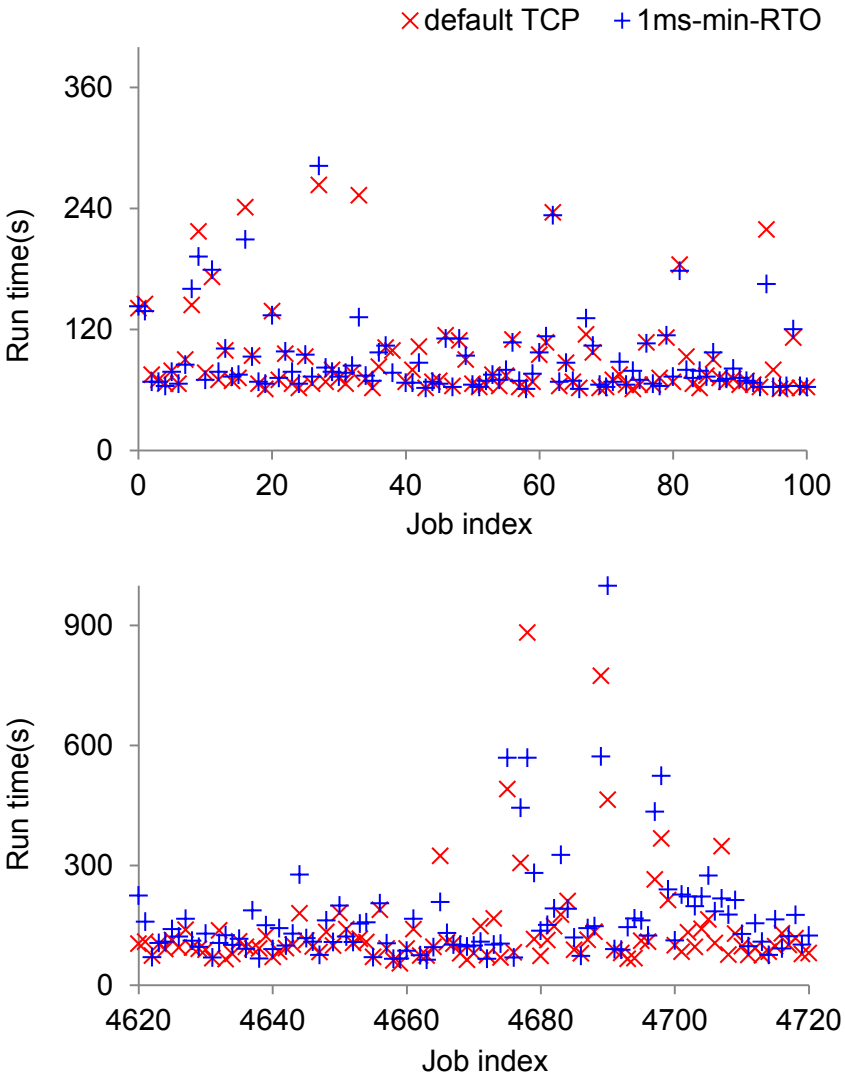



Figure 7.9. Sequences of job completion times. Showing two continuous job sequences of 100 jobs. The few large jobs have long completion times, and stand out from the baseline of continuous stream of small jobs.

Several data analysis trends emerge, confirmed by the cluster operators who provided the traces in Figure 7.7 and discussed in detail in Chapter 4:

1. There is increasing desire to do interactive data analysis, as well as streaming analysis. The goal is to have human with non-specialist skills explore diverse and evolving data sources, and once they discover a way to extract actionable insights, such insights should be updated based on incoming data in a timely and continuous fashion.

2. Bringing such data-analytic capability to non-specialists requires high-level computation frameworks built on top of common platforms such as MapReduce. Examples

of such frameworks in the Hadoop MapReduce ecosystem include HBase, Hive, Pig, Sqoop, Oozie, and others.

3. Data sizes grow faster than the decrease in the unit cost of storage and computation infrastructure. Hence, efficiently using storage and computational capacity are major concerns.

Incast plays into these trends as follows. The desire for interactive and streaming analysis requires highly responsive systems. The data size required for these computations are small compared with those required for computations on historical data. We know that when incast occurs, the RTO penalty is especially severe for small flows. Applications would be potentially forced to either delay the analysis response, or give answers based on partial data. Thus, incast could emerge as a barrier for high quality interactive and streaming analysis.

The desire to have non-specialists use large-scale data-centric systems suggest that functionality and usability should be the top design priorities. Incast affects performance, which can be interpreted as a kind of usability. It becomes a priority only after we have a functional system. Also, as our Hadoop experiments demonstrate, performance tuning for multi-layered software stacks would need to confront multiple layers of complexity and overhead.

The need for storage capacity efficiency entails storing compressed data, performing data deduplication, or using RAID instead of data replication to achieve fault tolerance. In such environments, memory locality becomes the top concern, and disk or network locality becomes secondary [24]. If the workload characteristics permits a high level of memory or disk locality, network traffic gets decreased, the application performance increases, and incast becomes less of a concern.

The need for computational efficiency implies that computing infrastructure needs to be more highly utilized. Network demands will thus increase. Consolidating diverse applications and workloads multiplexes many network traffic patterns. Incast will likely occur with greater frequency. Further, additional TCP pathologies may be revealed, such as the similarly phrased TCP outcast problem. This problem involves a single output port aggregating flows from multiple input ports, with a different number of flows from each input port (versus the traffic pattern for incast where the output port aggregates a single flow from each input port), and results in link share unfairness for large flows [112].

## 7.5 Recommendations and Chapter Conclusions

*Set TCP minimum RTO to 1ms.*

Future large-scale data-centric workloads likely reveal TCP pathologies other than incast. Incast and similar behavior are fundamentally transport-level problems. It is not resource effective to overhaul the entire TCP protocol, redesign switches, or replace the datacenter network to address a single problem. Setting `tcp_rto_min` is a configuration parameter

change – low overhead, immediately deployable, and, as we hope our experiments show, it does no harm inside the datacenter.

*Deploy better tracing infrastructure.*

It is not yet clear how much incast impacts future large-scale data-centric workloads. This chapter discusses several contributing factors. We need further information linking application-level performance to network packet-level behavior to determine which factors dominate under what circumstances. Better tracing helps remove the uncertainty. Where possible, such insights should be shared with the general community. We hope the workload comparisons in this chapter encourage similar, cross-organizational efforts elsewhere.

*Apply a scientific design process.*

Future large-scale data-centric systems demand a departure from design approaches that emphasize implementation over measurement and validation. The complexity, diversity, scale, and rapid evolution of such systems imply that mis-design opportunities proliferate, redesign costs increase, experiences rapidly become obsolete, and intuitions become hard to develop. Our approach in this chapter involves performing simplified experiments, developing models based on first principles, empirically validating these models, then connecting the insights to real life by introducing increasing levels of complexity. Our experiences tackling the incast problem demonstrate the value of a design process rooted in empirical measurement and evaluation.

# Chapter 8

# Closing

*Incoming wave pushes the outgoing wave. — Chinese proverb.*

To summarize, in this dissertation we identified the workload-driven design and evaluation of large-scale data-centric systems as an important research problem and made progress in addressing the associated research challenges. We posit that the complexity, diversity, scale, and rapid evolution of large-scale data-centric systems makes empirical, workload-driven design and evaluation the primary, if not the only, feasible method. The research advances proposed in the dissertation take advantage of nine production workload traces of front line, business critical large-scale data-centric systems. We developed a conceptual framework for considering the workload for such systems as composed of data patterns, compute patterns, and load arrival patterns. We developed workload synthesis, replay, and simulation methods to measure system performance subject to real life conditions. We applied the workload analysis methods to our traces and derived design insights for MapReduce and enterprise network storage, two important examples of large-scale data-centric systems. We further applied the workload-driven design and evaluation methods to improve MapReduce energy efficiency and explore the TCP incast pathology, two topics that would be very challenging to address without workload specific insights.

Overall, the dissertation develops a more objective and systematic understanding of an emerging and important class of computer systems. The work in this dissertation helps further accelerate the adoption of large-scale data-centric systems to solve problems relevant to business, science, and day-to-day consumers. Eventually, the broader, societal technology progression would result in another wave of disruptive innovation in the capabilities of computer systems. Just as our work builds on predecessor studies on the measurement and evaluation of existing computer systems, we hope one day this dissertation can spur further improvements in design and evaluation methods for the next generation of computer systems.

## 8.1 Future Work

Many opportunities exist for future research. For workload characterization, both MapReduce (Section 8.1.1) and enterprise network storage (Section 8.1.2) hold mirror opportunities with regard to analyzing more workload over longer time periods, additional analysis beyond that covered in the dissertation, building automated analysis and monitoring tools, and creating a workload taxonomy as we gain additional insights.

For energy efficient MapReduce (Section 8.1.3) and TCP incast (Section 8.1.4), workload insights help engineers evaluate the importance of these problems, and develop solutions specifically targeting real life behavior. Where multiple solutions are available, workload insights help identify which solutions yield the greatest benefit for the implementation/deployment costs. Further, we need to periodically re-evaluate existing solutions in light of the constant evolution of large-scale data-centric systems and the workloads they service.

### 8.1.1 MapReduce workload characterization

**Analyze more workloads over longer time periods.**

Doing so allows us to improve the quality and generality of the derived design insights. While we can never cover the entire universe of *possible* workload behavior, the goal is to collect sufficient empirical data that would allow us to build confidence that we have a good understanding of *common* behavior. So far, the empirical data analyzed in this dissertation represents a starting point for developing such an understanding. A priority should be to understand workload behavior at MapReduce clusters beyond the industry sectors we have analyzed so far.

**Additional analysis.**

It would be worthwhile to analyze the meaning and hierarchy of the HDFS input and output file paths to see if the small data sets are stand-alone or samples of larger data sets. Frequent use of data samples would suggest opportunities to pre-generate data samples that preserve various kind of statistics. This analysis requires proprietary file name information, and would be possible within each MapReduce user's organization.

The job name analysis should look beyond the first word of job names. Presently job name strings have no uniform structure, but contain pre- or postfixes such as dates, computation interactions, or steps in multi-stage processing, albeit with irregular format and ordering. Improved naming conventions, including UUIDs to identify multi-job workflows, would facilitate further analysis.

Regarding the k-means analysis, future work should seek to perform k-means analysis for multiple workloads together, instead of for each workload separately. Such combined analysis would reveal what are the common job categories across workloads. This combined analysis requires normalizing each workload by its "size". Given that each workload contains different number of jobs, and have different ranges in each of the 6 dimensions,

it is not yet clear how this "super k-means" analysis would proceed. A further innovation in analysis methodology awaits.

Other aspects of workload behavior that should be examined include the time between job submit and the first task being launched, which reflects task assignment bottlenecks; job submission patterns separated by submitters, which likely helps distinguish human versus automated analysis; task placement on machines, which likely reveals opportunities for improving data locality.

**Automate analysis and monitoring tools.**

Enterprise MapReduce monitoring tools [48] should have the following properties. They should perform workload analysis automatically and present graphical results in visually succint dashboards. This will help lower the cognitive load required to understand workload behavior. The monitoring tools should further collect anonymized metrics and aggregated statistics, an important concern for balancing the necessity to protect proprietary data while facilitating public sharing of workload traces. Tracing capabilities for Hive, Pig, HBase, and other such frameworks should be improved, so that operators can understand the workload at both the underlying MapReduce level and higher semantic levels layered on top of MapReduce.

**Create a MapReduce workload taxonomy.**

This would be possible once we have a large collection of insights from companies of different sizes and industries. Such a taxonomy would help with identifying design priorities for future MapReduce-style systems. For the workloads analyzed here, the similarities we found could define them as a common class, while the differences could help distinguish sub-classes. Such a taxonomy is likely to constantly evolve and expand with the collection of new workload data. We again invite the community to contribute additional workloads insights.

### 8.1.2 Enterprise network storage workload characterization

**Analyze more workloads over longer time periods.**

Doing so for enterprise network storage systems allows us to improve the quality and generality of the derived design insights.

**Additional analysis.**

It should be rewarding to explore the dynamics of changing working sets and access sequences, with the goal of anticipating data accesses before they happen.

Another worthwhile analysis is to look for optimization opportunities *across clients*; this requires collecting traces at different clients, instead of only at the server.

Also, we would like to explore opportunities for deduplication, compression, or data placement. Doing so requires extending our analysis from *data movement* patterns, i.e., how the data is accessed, to also include *data content* patterns, i.e., what the data is.

Furthermore, we would like to perform on-line analysis in live storage systems to enable dynamic feedback on placement and optimization decisions.

**Automate analysis and monitoring tools.**

Enterprise network storage systems should introduce appropriate monitoring points to dynamically adjust the decision thresholds of placement, caching, or consolidation policies. We need to monitor both clients and servers. Extensible tracing APIs would expedite the collection of long-term future traces. The goal is to begin with trace collection tools that record the most basic statistics, and later, as we understand the need to monitor additional statistics, extend the tracing API such that subsets of the statistics would be comparable over time. This will expedite the periodic re-evaluation of design insights, and make it easier to compare future work to this dissertation.

**Create a enterprise network storage workload taxonomy.**

Enterprise network storage systems have enjoyed considerably longer deployment and operating experience compared with MapReduce. Thus, there is already some knowledge about the different kind of workloads that exists. However, due to the limits of prior tracing tools, many of the predecessor workload insights are at the byte and block level [101, 134, 30]. Further, as explained earlier in the dissertation, those insights are also single-dimensional. Future studies should seek to build a new enterprise storage workload taxonomy using higher level, multi-layer semantics, as well as truly multi-dimensional analysis.

**Refresh of enterprise network storage workload replay tools.**

There are already some benchmarking and workload replay tools for enterprise network storage systems. Most of the benchmarks reflect system behavior before the recent explosion in data scale, diversity, and complexity stored by enterprise network storage systems. Additionally, the workload replay often takes place at the byte and block stream level, and thus insufficient for evaluating optimizations at higher semantic layers. Thus, the field can benefit from improved tools to synthesize and replay the workload, so that designers can evaluate the optimizations proposed in this dissertation.

### 8.1.3 Energy efficient MapReduce

**Automatic configuration for cluster energy efficient policy.**

The BEEMR policy space is large. It would be desirable to automatically detect good values for the policy parameters in Table 6.3. Doing so requires constructing tools to automatically do workload analysis to narrow the range of parameters values, and tools that scan the narrowed-down range of parameter values and identify the settings for best performance, either by workload-driven simulations or by replaying synthetic workloads on actual systems.

**Interrupt and resume jobs.**

BEEMR relies on this ability. It is being prototyped under Next Generation Hadoop

for fast resume from failures [97] and will appear in future enterprise versions of Hadoop [143]. Energy efficiency would be another motivation for this feature.

**Improve cluster coordination protocols for energy efficiency.**

The chatty HDFS/Hadoop messaging protocols limits the dynamic power of machines to a narrow range. There is an opportunity to re-think such protocols for distributed systems to improve power proportionality.

**Investigate cluster zoning versus operating separate clusters.**

The disjoint interactive and batch zones in BEEMR can be further segregated into disjoint interactive and batch clusters. Segregated versus combined cluster operations need to balance a variety of policy, logistical, economic, and engineering concerns. Segregated clusters gives performance isolation and allows customized configurations, but introduces operational complexities such as the need to overprovision multiple clusters and maintain liveliness for multiple copies of data. Also, computations that require data from across clusters would experience performance penalties. Having a single, combined cluster avoids the inefficiencies of overprovisioning and maintaining multiple data copies, but risks the workloads from one organization affecting the performance of another. Further, cluster fair-share policies would become complex if the joint cluster is being provisioned from different budgets. More systematic understanding of energy costs helps inform the discussion.

**Improve MapReduce simulators.**

The BEEMR simulator trades accuracy for speed and scale (Section 6.5). How to make this tradeoff well represents an important topic, especially as it is already logistically infeasible to replay MIA workloads on real systems at duration and at scale.

**Explore fine-grained power management mechanisms.**

The gap between ideal and BEEMR energy savings increases with cluster size (Section 6.6.7). It is worth exploring whether more fine-grained power management schemes would close the gap and allow operators to provision for peak while conserving energy costs.

### 8.1.4 TCP incast

**Improve TCP incast analytical models.**

The TCP incast models developed in this dissertation covers only the simple, single-switch TCP incast setup. Even then, some features of the incast goodput collapse remain unexplained. Future work on TCP incast should seek to expand analytical modeling to more complex topologies, and to more fully explain the behavior.

**Construct TCP benchmark.**

We need an easily run benchmark to specifically measure the impacts of TCP incast

and other TCP pathologies. This benchmark would allow us to regularly test TCP behavior as large-scale data-centric systems evolve.

**Measure TCP incast for the next mainline Hadoop MapReduce distribution.**

Several Hadoop code branches are currently active. Some of these branches attempt to fix the various performance overheads that mask TCP incast. It would be worthwhile to verify whether those changes actually result in performance improvements.

**Improve network tracing infrastructure.**

Current datacenter network tracing tools record behavior at the flow and packet level. The resulting traces become very detailed and cognitively difficult to analyze. However, the complexity of problems such as TCP incast likely requires data at that level. It is an important challenge to both have more thorough network tracing tools, and to have the tools record and present data in a useful fashion.

**Increase visibility in switch behavior.**

The current black-box understanding of switch behavior hinders addressing network behavior pathologies such as incast. Our work in this dissertation shows that successful analytical models often involve detailed knowledge of switch capabilities, such as buffer management strategies. We encourage ongoing efforts to make switches more transparent [11].

## 8.2 Broader Impact of the Dissertation

Over the life time of computer systems, measurement and analysis assumes a high priority after the systems involved have sufficiently matured to enjoy widespread deployment, but not yet so established that a newer generation of systems bring about disruptive innovations. Based on the workload traces we have and our interactions with the system vendors who provided the traces, we posit that large-scale data-centric systems are entering that phase. We close the dissertation with some thoughts on the long term relevance of workload-drive design and evaluation.

Our work here shows that it is challenging to develop rigorous, empirical design and evaluation methods for large-scale data-centric systems. Such an effort is worthwhile when technology trends dictate that the speed of system evolution and improvement is bottlenecked on the speed of developing rigorous insights about such systems. This is currently the case for large-scale data-centric systems, and we expect this would continue to be the case. However, we should remain open to the possibility that major disruptions in how computer systems are designed and used will drastically simplify the engineering process. Examples of such disruptions include the move up increasing layers of programming language abstractions, the development of relational data models, the emergence of paradigms such as MapReduce that hides the details of coordinating distributed parallel computations, etc. Whenever such a disruption occurs, the emphasis returns to rapid, almost chaotic engineering and re-engineering of new features. Rigorous

performance science represents an essential step to control the chaos, and ensures that the desirable features are not hindered by poor implementation, and the systems involved can successfully mature, and ultimately catalyze the emergence of the next generation of sytems.

For large-scale data-centric systems, we speculate that such a disruption will not happen for some time, but there is little doubt that we will eventually witness some kind of disruption. Good workload analysis methods could help us anticipate the precise timing of the impending disruption. Doing so would require applying another level of human expertise to translate from observed behavior (what the system does) to anticipate future user needs (what the users are actually trying to do). We only tangentially touch on this idea in the dissertation. It is clear that anticipating future user needs requires the human designer to ground her speculations in empirical evidence, while making the mental leap from "what the system is" to "what a better kind of systme could be".

Viewed in this way, the ultimate success of workload-driven design and evaluation of large-scale data-centric systems would be to clear the path for the next disruptive wave in the design of computer systems.

The workload-driven design and evaluation of large-scale data-centric systems involve organizing diverse, high volume, and potentially disorganized data sources to distill timely and actionable insights. The problems identified in this dissertation ultimately require large-scale data-centric systems to solve. Also, the complex, diverse, and rapidly changing nature of large-scale data-centric systems mirrors the complex, diverse, and rapidly changing nature of society that originally created the need for such systems. The proliferation of these systems to non-technology industries reflects the increasing society-wide desire to derive knowledge and make decisions based on objective empirical evidence as well as subjective intuition and experience. This dissertation develops some statistics and empirical techniques to make sense of data about technology systems. Our work is symbiotic with companion efforts to make sense of data about non-technology systems. Both efforts fall within the emerging field of "data science", and innovations for making sense of one kind of data translates to mirror innovations in making sense of other kinds of data.

Viewed in this way, the ultimate success of workload-driven design and evaluation of large-scale data-centric systems would be to improve such systems to the point where performance is no longer a concern, and we can focus our intellectual resources on using these systems to solve data problems relevant to business, science, and day-to-day consumers.

# Bibliography


[1] Apache Hadoop Documentation. http://hadoop.apache.org/common/docs/r0.20.2/cluster_setup.html#Configuring+the+Hadoop+Daemons.

[2] Apache Oozie(TM) Workflow Scheduler for Hadoop. http://incubator.apache.org/oozie/.

[3] Gridmix. HADOOP-HOME/src/benchmarks/gridmix in all recent Hadoop distributions.

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

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

[6] Hadoop Power-By Page. http://wiki.apache.org/hadoop/PoweredBy.

[7] Hadoop World 2011 Speakers. http://www.hadoopworld.com/speakers/.

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

[9] IEEE 802.1Qau Standard - Congestion Notification. http://www.ieee802.org/1/pages/802.1au.html.

[10] Mumak. http://issues.apache.org/jira/browse/MAPREDUCE-728, last retrieved Nov. 2009.

[11] Open Flow. http://www.openflow.org/.

[12] Personal communications with Cloudera engineering and support teams.

[13] Sort benchmark home page. http://sortbenchmark.org/.

[14] tcpdump. http://www.tcpdump.org/.

[15] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A Five-Year Study of File-System Metadata. In *FAST 2007*.

[16] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM 2008*.

[17] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *SIGCOMM 2010*.

[18] M. Alizadeh et al. Data center transport mechanisms: Congestion control theory and IEEE standardization. In *Annual Allerton Conference 2008*.

[19] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit. http://www.ietf.org/rfc/rfc3042.txt, 2001.

[20] M. Allman, V. Paxson, and W. Stevens. Request for Comments: 2581 - TCP Congestion Control. http://www.ietf.org/rfc/rfc2581.txt, 1999.

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

[22] Amazon Web Services. Amazon EC2 Instance Types. http://aws.amazon.com/ec2/instance-types/.

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

[24] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Disk-locality in datacenter computing considered irrelevant. In *HotOS 2011*.

[25] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated Memory Caching for Parallel Jobs. In *NSDI 2012*.

[26] 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*.

[27] G. Ananthanarayanan et al. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Eurosys 2011*.

[28] R. H. Arpaci et al. The interaction of parallel and sequential workloads on a network of workstations. In *SIGMETRICS 1995*.

[29] I. Ashok and J. Zahorjan. Scheduling a mixed interactive and batch workload on a parallel, shared memory supercomputer. In *Supercomputing 1992*.

[30] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder. An analysis of data corruption in the storage stack. In *FAST 2008*.

[31] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *SOSP 1991*.

[32] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. In *ISCA 2011*.

[33] C. Belady. In the data center, power and cooling costs more than the IT equipment it supports. *Electronics Cooling Magazine*, Feb. 2007.

[34] R. Bianchini and R. Rajamony. Power and energy management for server systems. *Computer*, Nov. 2004.

[35] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen. Fingerprinting the datacenter: automated classification of performance crises. In *EuroSys 2010*.

[36] D. Borthakur. Looking at the code behind our three uses of Apache Hadoop. Facebook Engineering Notes, December 10, 2010.

[37] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache Hadoop goes realtime at Facebook. In *SIGMOD 2011*.

[38] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *INFOCOM 1999*.

[39] L. Breslau et al. Web Caching and Zipf-like Distributions: Evidence and Implications. In *INFOCOM 1999*.

[40] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz. Energy Efficiency for Large-Scale MapReduce Workloads with Significant Interactive Analysis. In *EuroSys 2012*.

[41] Y. Chen, S. Alspaugh, A. Ganapathi, R. Griffith, and R. Katz. Statistical Workload Injector for MapReduce. http://www.eecs.berkeley.edu/~ychen2/SWIM.html.

[42] Y. Chen, L. Keys, and R. H. Katz. Towards Energy Efficient MapReduce. Technical Report UCB/EECS-2009-109, EECS Department, University of California, Berkeley, Aug 2009.

[43] Y. Chen, K. Srinivasan, G. Goodson, and R. Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *SOSP 2011*.

[44] Y. Chen et al. The Case for Evaluating MapReduce Performance Using Workload Suites. In *MASCOTS 2011*.

[45] Y. Chen et al. Understanding TCP incast throughput collapse in datacenter networks. In *WREN 2009*.

[46] Y. Chen et al. Statistical Workloads for Energy Efficient MapReduce. Technical Report UCB/EECS-2010-6, EECS Department, University of California, Berkeley, Jan 2010.

[47] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *SIGCOMM 2011*.

[48] Cloudera, Inc. Cloudera Manager Datasheet.

[49] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *NSDI 2010*.

[50] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. Technical Report UCB/EECS-2009-136, EECS Department, University of California, Berkeley, Oct 2009.

[51] IDC Whitepaper: The economics of Virtualization. www.vmware.com/files/pdf/Virtualization-application-based-cost-model-WP-EN.pdf.

[52] J. Corbet. LWN.net 2009 Kernel Summit coverage: How Google uses Linux. 2009.

[53] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[54] Q. Deng et al. Memscale: active low-power modes for main memory. In *ASPLOS 2011*.

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

[56] EMC and IDC iView. Digital Universe. http://www.emc.com/leadership/programs/digital-universe.htm.

[57] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *Micro, IEEE*, 28(3):42 –53, May-June 2008.

[58] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *ISCA 2007*.

[59] N. Feamster and H. Balakrishnan. Detecting bgp configuration faults with static analysis. In *NSDI'05*.

[60] D. Ferrari. *Computer Systems Performance Evaluation*. Prentice-Hall, 1978.

[61] D. Ferrari. Characterizing a workload for the comparison of interactive services. *Managing Requirements Knowledge, International Workshop on*, 0, 1979.

[62] S. Floyd and V. Paxson. Difficulties in simulating the internet. *IEEE/ACM Trans. Netw.*, 9(4):392–403, Aug. 2001.

[63] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven workload modeling for the cloud. In *ICDE Workshops 2010*.

[64] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[65] J. Gray et al. Quickly generating billion-record synthetic databases. In *SIGMOD 1994*.

[66] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *SIGCOMM 2009*.

[67] The gzip algorithm. http://www.gzip.org/algorithm.txt.

[68] J. Hamilton. Overall Data Center Costs. http://perspectives.mvdirona.com/2010/09/18/OverallDataCenterCosts.aspx, 2010.

[69] J. Hellerstein. Quantitative data cleaning for large databases. Technical report, United Nations Economic Commission for Europe, February 2008.

[70] Hewlett-Packard Corp., Intel Corp., Microsoft Corp., Phoenix Technologies Ltd., Toshiba Corp. Advanced Configuration and Power Interface 5.0. http://www.acpi.info/.

[71] 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*.

[72] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, S. Shenker, and I. Stoica. Nexus: A common substrate for cluster computing. HotCloud 2009: Workshop on Hot Topics in Cloud Computing.

[73] HP Networking. HP 5400 zl Switch Series. http://h17007.www1.hp.com/us/en/products/switches/HP_5400_zl_Switch_Series/, 2012.

[74] 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*.

[75] IDC Report: Worldwide File-Based Storage 2010-2014 Forecast Update. http://www.idc.com/getdoc.jsp?containerId=226267.

[76] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP 2009*.

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

[78] R. T. Kaushik et al. Evaluation and Analysis of GreenHDFS: A Self-Adaptive, Energy-Conserving Variant of the Hadoop Distributed File System. In *IEEE CloudCom 2010*.

[79] Open Source Clustering Software - C Clustering Library. http://bonsai.hgc.jp/~mdehoon/software/cluster/software.htm, 2010.

[80] E. Krevat et al. On application-level approaches to avoiding tcp throughput collapse in cluster-based storage systems. In *PDSW 2007*.

[81] W. Lang and J. Patel. Energy management for mapreduce clusters. In *VLDB 2010*.

[82] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of ethernet traffic. In *SIGCOMM 1993*.

[83] A. Leung, S. Pasupathy, G. Goodson, and E. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX ATC 2008*.

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

[85] P. Lieberman. White paper: Wake on lan technology, June 2006.

[86] T. Lipcon and Y. Chen. Hadoop and performance. Hadoop World 2011. http://www.hadoopworld.com/session/hadoop-and-performance/.

[87] B. Liu et al. A study of networks simulation efficiency: Fluid simulation vs. packet-level simulation. In *Infocom 2001*.

[88] M. Mathis et al. Request for Comments: 2018 - TCP Selective Acknowledgment Options. http://tools.ietf.org/html/rfc2018, 1996.

[89] D. Meisner et al. Power management of online data-intensive services. In *ISCA 2011*.

[90] D. Meisner et al. Powernap: eliminating server idle power. In *ASPLOS 2009*.

[91] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. In *VLDB 2010*.

[92] M. Mesnier, E. Thereska, G. R. Ganger, and D. Ellard. File classification in self-* storage systems. In *ICAC 2004*.

[93] D. T. Meyer and W. J. Bolosky. A Study of Practical Deduplication. In *FAST 2010*.

[94] A. K. Mishra et al. Towards characterizing cloud backend workloads: insights from Google compute clusters. *SIGMETRICS Perform. Eval. Rev.*, 37:34–41, March 2010.

[95] J. C. Mogul. The case for persistent-connection http. In *SIGCOMM 1995*.

[96] K. Morton et al. ParaTimer: a progress indicator for MapReduce DAGs. In *SIGMOD 2010*.

[97] A. Murthy. Next Generation Hadoop Map-Reduce. Apache Hadoop Summit 2011.

[98] Nortel. Nortel 5500 Switch Manual, 2007.

[99] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[100] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the Unix 4.2 BSD file system. In *SOSP 1985*.

[101] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the unix 4.2 bsd file system. In *SOSP 1985*.

[102] D. Patterson. Energy-Efficient Computing: the State of the Art. Microsoft Research Faculty Summit 2009.

[103] 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*.

[104] V. Paxson. End-to-end internet packet dynamics. In *SIGCOMM 1997*.

[105] V. Paxson. Empirically-Derived Analytic Models of Wide-Area TCP Connections. *IEEE/ACM Transactions on Networking*, 2(4):316–336, August 1994.

[106] V. Paxson and M. Allman. Request for Comments: 2988 - Computing TCP's Retransmission Timer. http://www.ietf.org/rfc/rfc2988.txt, 2000.

[107] V. Paxson and S. Floyd. Wide area traffic: the failure of poisson modeling. *Networking, IEEE/ACM Transactions on*, 3(3):226 –244, jun 1995.

[108] Personal email. Communication regarding release of Google production cluster data.

[109] A. Phanishayee et al. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *FAST 2008*.

[110] Pig Wiki. Pig Mix benchmark. http://wiki.apache.org/pig/PigMix.

[111] E. Pinheiro et al. Failure trends in a large disk drive population. In *FAST 2007*.

[112] P. Prakash et al. The TCP Outcast Problem: Exposing Throughput Unfairness in Data Center Networks. In *NSDI 2012*.

[113] G. F. Riley, T. M. Jaafar, and R. M. Fujimoto. Integrated fluid and packet network simulations. In *MASCOTS 2002*.

[114] S. Rivoire et al. Joulesort: a balanced energy-efficiency benchmark. In *SIGMOD 2007*.

[115] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *USENIX 2000*.

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

[117] A. Ryan. Next-Generation Hadoop Operations. Bay Area Hadoop User Group, February 2010.

[118] J. H. Saltzer. A simple linear model of demand paging performance. *Commun. ACM*, 17:181–186, April 1974.

[119] K. Shvachko. HDFS Scalability: the limits to growth. *Login*, 35(2):6–16, April 2010.

[120] D. C. Snowdon et al. Accurate on-line prediction of processor and memory energy usage under voltage scaling. In *EMSOFT 2007*.

[121] SPEC. SPECpower 2008. http://www.spec.org/power_ssj2008/.

[122] K. Srinivasan, M. A. Kazandjieva, S. Agarwal, and P. Levis. The $\beta$-factor: measuring wireless link burstiness. In *SenSys 2008*.

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

[124] The DETER Project. DETER Lab. https://www.isi.deterlab.net/.

[125] The Green Grid. The Green Grid Data Center Power Efficiency Metrics: PUE and DCiE, 2007.

[126] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *EuroSys 2011*.

[127] A. Thusoo et al. Data warehousing and analytics infrastructure at Facebook. In *SIGMOD 2010*.

[128] A. Thusoo et al. Hive - a warehousing solution over a mapreduce framework. In *Proceedings of the 35th International Conference on Very Large Data Bases*, New York, NY, USA, 2009. ACM.

[129] U.S. Environmental Protection Agency. Report to Congress on Server and Data Center Energy Efficiency, Public Law 109-431, 2007.

[130] V. Vasudevan et al. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *SIGCOMM 2009*.

[131] R. Villars. The Migration to Converged IT: What it Means for Infrastructure, Applications, and the IT Organization. IDC Directions Conference 2011.

[132] VMware Whitepaper: Server Consolidation and Containment. www.vmware.com/pdf/server_consolidation.pdf.

[133] W. Vogels. File system usage in Windows NT 4.0. In *SOSP 1999*.

[134] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with cart models. In *MASCOTS 2004*.

[135] G. Wang et al. A simulation approach to evaluating design decisions in MapReduce setups. In *MASCOTS 2009*.

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

[137] R. Wolff. Poisson arrivals see time averages. *Operations Research*, 30(2):223–231, March 1982.

[138] H. Wu et al. ICTCP: Incast Congestion Control for TCP in data center networks. In *Co-NEXT 2010*.

[139] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Mining console logs for large-scale system problem detection. In *SysML 2009*.

[140] 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*.

[141] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Apr 2009.

[142] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *8th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 2008.

[143] C. Zedlewski. An update on Apache Hadoop 1.0. http://www.cloudera.com/blog/2012/01/an-update-on-apache-hadoop-1-0/.