BIG DATA STORAGE WORKLOAD CHARACTERIZATION, MODELING AND SYNTHETIC GENERATION

BY

CRISTINA LUCIA ABAD

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Professor Roy H. Campbell, Chair
Professor Klara Nahrstedt
Associate Professor Indranil Gupta
Assistant Professor Yi Lu
Dr. Ludmila Cherkasova, HP Labs

Abstract

A huge increase in data storage and processing requirements has lead to *Big Data*, for which next generation storage systems are being designed and implemented. As Big Data stresses the storage layer in new ways, a better understanding of these workloads and the availability of flexible workload generators are increasingly important to facilitate the proper design and performance tuning of storage subsystems like data replication, metadata management, and caching.

Our hypothesis is that *the autonomic modeling of Big Data storage system workloads through a combination of measurement, and statistical and machine learning techniques is feasible, novel, and useful.* We consider the case of one common type of Big Data storage cluster: A cluster dedicated to supporting a mix of MapReduce jobs. We analyze 6-month traces from two large clusters at Yahoo and identify interesting properties of the workloads. We present a novel model for capturing popularity and short-term temporal correlations in *object request streams*, and show how unsupervised statistical clustering can be used to enable autonomic type-aware workload generation that is suitable for emerging workloads. We extend this model to include other relevant properties of storage systems (file creation and deletion, pre-existing namespaces and hierarchical namespaces) and use the extended model to implement MimesisBench, a realistic namespace metadata benchmark for next-generation storage systems. Finally, we demonstrate the usefulness of MimesisBench through a study of the scalability and performance of the Hadoop Distributed File System name node.

To my husband, Rafael, and my children, Rafa and Luciana,
who bravely accompanied me during this journey.

Acknowledgments

I would like to thank my adviser, Roy H. Campbell, for his guidance and support, and Professor Yi Lu, whose help and feedback was invaluable at every step of my graduate work.

I thank Klara Nahrstedt, Indranil Gupta, and Lucy Cherkasova, for their sharp insights and supportive comments before and during my preliminary and final exams.

I would like to thank all of the members of the Systems Research Group (SRG) and the Assured Cloud Computing Center (ACC), that provided constructive feedback during my many practice talks and research discussions. In particular, I thank Mirko Montanari and Abhishek Verma, who were always willing to listen to my research problems and offer useful suggestions; and, my research collaborators, Mindi Yuan, Chris Cai, and Huong Luu, for their important roles in the projects in which we worked together.

I also thank my many friends at UIUC, especially Cigdem Sengul, Rossana Castro, Geta Sampemane, Riccardo Crepaldi, Ariel Gertzestein, Apu Kapadia, Prasad Naldurg, Alejandro Gutiérrez, Faraz Fahgri, and Harshit Kharbanda, whose engaging and stimulating conversations made graduate life more interesting.

I express my gratitude towards all the members of the Hadoop Core Development team at Yahoo, in particular to Nathan Roberts and Mark Holderbaugh, for providing me with the opportunity to learn about the challenges faced by large Big Data deployments, and to Jason Lowe, Tom Graves, Daryn Sharp and Bobby Evans, for patiently answering my many questions related to the Hadoop codebase and its deployments at Yahoo.

Finally, I would like to thank my family: Rafael, Rafa and Luciana, whose unconditional love gave me strength; my parents, Ana Julia and Guillermo, who instilled in me the love of learning since an early age; and my brothers, Andrés and Eduardo, for their love and support.

Table of Contents

Chapter 1 Introduction 1
1.1 Thesis statement 2
1.2 Intellectual merit 3
1.3 Broader impact 3
1.4 Assumptions and limitations 4
1.5 Summary of contributions 5
1.6 Dissertation outline 7

Chapter 2 Background: MapReduce Clusters and Next-Generation Storage Systems 8
2.1 MapReduce clusters 8
2.2 Storage systems for petascales and beyond 9

Chapter 3 Towards a Metadata Workload Model 12
3.1 Motivation 12
3.2 Datasets and methodology 13
3.3 Limitations of existing approaches 15
3.4 Towards a metadata workload model 18
3.5 Application: Metadata cache for HDFS 26
3.6 Discussion and related work 30
3.7 Summary 31

Chapter 4 A Storage-Centric Analysis of MapReduce Workloads . . . 32
4.1 Motivation 32
4.2 Dataset description 34
4.3 Analysis of two MapReduce workloads 35
4.4 Related work 50
4.5 Discussion 51
4.6 Summary 52

Chapter 5 Generating Request Streams on Big Data using Clustered Renewal Processes 54
5.1 Motivation 54
5.2 Model 58
5.3 Reducing the model size via clustering 60

5.4 Evaluation 69
5.5 Discussion 73
5.6 Related work 77
5.7 Summary 79

Chapter 6 A Model-Based Namespace Metadata Benchmark for HDFS 81
6.1 Motivation 81
6.2 Model 83
6.3 Design 88
6.4 Using MimesisBench to evaluate a Big Data storage system 92
6.5 Related work 96
6.6 Summary 98

Chapter 7 Conclusions and Future Directions 99

References 101

Chapter 1

Introduction

Large-scale file systems in the Internet services and the high-performance computing (HPC) communities already handle Big Data: Facebook has 21PB in 200 million objects and Titan ORNL has 10PB (see Table 1.1 for other examples). Stored data is growing so fast that exascale storage systems are expected by 2018 to 2020, by which point there should be over five hundred 10PB deployments [36]. As Big Data storage systems continue to grow, a better understanding of the workloads present in these systems becomes critical for proper design and tuning.

Table 1.1: Deployments of petascale storage. M: million; MDS: metadata server.

Deployment	**Description**
Yahoo (Internet services)	15PB, 4000+ nodes (HDFS); 150 M objects [69]
Facebook (Internet services)	21PB, 2000 nodes (HDFS), 200 M objects [69]
eBay (Internet services)	16PB in 700-nodes (HDFS) [69]
Mira ANL (HPC)	35PB of storage (GPFS); 384 I/O nodes [55]
Titan ORNL (HPC)	10PB of storage; Lustre file system [78, 79]
JUQUEEN FZJ (HPC)	10PB of storage; GPFS file system [42, 43]
Stampede TACC (HPC)	14PB of storage (Lustre); 72 data servers and 4 MDSs [72]
Kraken NICS (HPC)	3.3PB of storage (Lustre); 1 MDS [47]

New schemes for data replication [2, 52, 89], namespace metadata management [57, 60, 90], and caching [11, 33, 94] have been proposed for next-generation storage systems; however, their evaluation has been insufficient due to a lack of realistic workload generators [3].

Workload generation is often used in simulations and real experiments to help reveal how a system reacts to variations in the load [14]. Such experiments can be used to validate new designs, find potential bottlenecks, evaluate performance, and do capacity planning based on observed or predicted workloads.

Workload generators can replay real traces or do model-based synthetic generation. Real traces capture observed behavior and may even include nonstandard or undiscovered (but possibly important) properties of the load [76]. However, real-trace approaches treat the workload as a "black box" [14]. Modifying a particular

workload parameter or dimension is difficult, making such approaches inappropriate for sensitivity and what-if analysis. In addition, sharing of traces can be hard due to their size and privacy concerns. Other problems include those of scaling to a different system size and describing and comparing traces in terms that can be understood by implementors [76].

Model-based synthetic workload generation can be used to facilitate testing while modifying a particular dimension of the workload, and can be used to model expected future demands. For that reason, synthetic workload generators have been used extensively to evaluate the performance of storage systems [4, 76], media streaming servers [41, 73], and Web caching systems [14, 16]. Synthetic workload generators can issue requests on a real system [14, 76] or generate synthetic traces that can be used in simulations or replayed on actual systems [73, 86].

New and realistic workload models are needed to help understand and synthetically generate emerging workloads in the Big Data storage domain. Ideally, these models should be efficient (to be able to generate Big workloads), workload-agnostic (to support emerging workloads), type-aware (to enable the modification of the workload of specific types of object behavior), and autonomic (so that user involvement is not required in the process of identifying distinct types of behavior).

1.1 Thesis statement

Our hypothesis is that *the autonomic modeling of Big Data storage system workloads, through a combination of measurement, and statistical and machine learning techniques is feasible, novel, and useful.*

We support this hypothesis as follows:

1. **Feasibility:** We propose a statistical model to generate realistic and large namespace hierarchies and experimentally show that it is feasible to issue realistic namespace metadata operations atop that namespace. We propose a model that reproduces the object popularity and short-term temporal correlations of *object request streams*, and then expand that model to make it suitable for *metadata-intensive* benchmarking in next-generation storage systems. We implement a synthetic trace generator and a benchmarking tool based on these models.

2. **Novelty:** We survey previous models related to storage workloads and object request streams, and contrast our work.

3. **Usefulness:** We use two case studies (on caching and distributed storage replication) to show that our models produce accurate results and can be used in place of the real workloads. We use MimesisBench, the benchmarking tool based on our models, to do a performance and scalability study of the Hadoop Distributed File System name node (MDS).

1.2 Intellectual merit

Our work makes the following contributions. First, we identify the limitations with current evaluation approaches in evaluating the namespace metadata management subsystems of next-generation storage systems; a proof-of-concept model for *namespace metadata traces* is proposed and we show that it can outperform more traditional yet simplistic approaches. Second, we present a detailed workload characterization of how a common Big Data workload (MapReduce jobs) interacts with the storage layer. Third, we consider the general problem of reproducing the short-term temporal correlations and system-wide object popularity of object request streams, and propose a model based on clustered delayed renewal processes in which the popularity distribution asymptotically emerges through explicit reproduction of the per-object request arrivals and active span (time during which an object is accessed); this model is workload-agnostic, type-aware, fast, and suitable for request streams with high object churn. Finally, we extend this model and use it to implement a realistic metadata-intensive benchmark for next-generation storage systems.

1.3 Broader impact

Our **workload characterization** of how typical MapReduce jobs interact with the storage layer is publicly available so that other researchers can use our findings when guiding their designs and evaluations.

Our **model for object request streams** based on clustered renewal processes was motivated by Big Data observations; however, the model is workload-agnostic, making it applicable to other types of object request streams like Web document references [35], media streaming requests [23], and disk block traces [76].

We have **publicly released our tools** so that other researchers and practitioners can use them to evaluate next-generation storage systems, or adapt them for their specific needs.

1.4 Assumptions and limitations

We have designed models that can be used to generate realistic synthetic workloads to evaluate the *namespace metadata management subsystems* of next-generation storage systems. Our models can generate realistic synthetic namespaces and metadata operations (e.g., file and directory creates and deletes, file opens, and list status operations) atop those namespaces. Modeling dimensions related to I/O behavior—like a correlation between file size and popularity or the length of data R/W operations—is out of the scope of this thesis.

Our models aim to capture the behavior of **stationary segments** of a trace; for example, we can choose to model a one-hour segment with a heavy load, and/or a one-hour segment with an average load. We discuss how to extend the models to non-stationary workloads in Section 5.5.1.

Our models represent arrivals with a sequence of **interarrival times** $X = (X_1, X_2, ...)$, where X is a sequence of *independent, identically distributed random variables*. In practice, there could be autocorrelations in the arrival process, leading to bursty behavior at different timescales. In Section 5.6, we briefly discuss how our model can be extended to capture burstiness in the arrival processes.

We assume that each arrival process—in this context, of the accesses to a file in the storage system—is **ergodic**, so its interarrival distribution can be deduced from a single sample or realization of the process (i.e., the trace of events representing the stationary segment being modeled). In addition, instead of forcing a fit to a particular interarrival distribution, we use the empirical distribution of the interarrivals inferred from the arrivals observed in the trace being modeled.

Our current models capture the temporal locality of the original workloads, but do not attempt to capture **spatial locality**. Considering this dimension of the workload is left for future work.

Finally, there may be unknown, but important, behavior in the original workloads that our models do not capture. However, as other researchers have argued in the past [76, 84], *we ultimately care about the ability of the synthetic workloads of producing the same application-level results as the original workload.*

For this reason, we validate our models not only by their **ability to reproduce the statistical parameters** captured by them (e.g., being able to reproduce the file popularity distribution), but also by their ability of **producing the equivalent domain-specific, application-level results** as the original workload. Thus, in addition to a validation with the statistical properties of the workload, we also use case-studies to show that our models produce equivalent application-level results.

1.5 Summary of contributions

We study an important problem in Big Data storage systems: enabling *realistic evaluation of metadata management schemes*. In Big Data storage systems, larger I/O bandwidth is achieved by adding storage nodes to the cluster, but improvements in metadata performance cannot be achieved by simply deploying more MDSs, as the defining performance characteristic for serving metadata is not bandwidth but rather latency and number of concurrent operations that the MDSs can handle [9, 31]. For this reason, novel and improved distributed metadata management mechanisms are being proposed [57, 60, 90]. However, as explained in Chapter 3, realistic validation and evaluation of these designs was not possible as no adequate traces, workload models, or benchmarks were available. We strengthen our case by showing how a statistical proof-of-concept model can be used to evaluate a least recently used (LRU) namespace metadata cache, outperforming earlier yet simplistic models. In Chapter 3, we also present our work on modeling large hierarchical namespaces; to the best of our knowledge, our namespace metadata generator (NGM), is the only available tool that can be used for this purpose.

Within the Big Data domain, we focus on a common type of workload and study *how MapReduce interacts with the storage layer*. We analyzed the workload of two large Hadoop clusters at Yahoo. The traces we analyzed were 6-month traces of the largest production cluster (PROD, more than 4000 nodes) and a large research and development cluster (R&D, around 2000 nodes). While both clusters run MapReduce jobs, the types of workloads they sustain are different. PROD has tight controls on job submissions and runs time-sensitive periodic and batch jobs on clickstream data. The R&D cluster is used to test possible variants of the algorithms that may in the future be moved to production, and also to support interactive, business intelligence (BI) queries.

We identify interesting properties of the workloads—like the presence of high file churn that leads to a very dynamic population and short-lived files—and highlight key insights and implications to storage system design and tuning. To the best of our knowledge, this was the first study of how MapReduce interacts with the storage layer. Our analysis and findings are presented in Chapter 4.

In the path towards modeling namespace metadata workloads, we study the more general problem of being able to reproduce the temporal access patterns and popularity in *object request streams*[1]. We present a model based on a set of delayed renewal processes that can capture the temporal access patterns (per-file interarrivals and per-file active span) as well as the system-wide file popularity distribution. However, this model is not scalable as it needs to keep track of the interarrival distributions of each object in the workload; for Big Data workloads, the number of objects can be quite big (e.g., 4.3 million distinct files in a one day trace). We propose a reduced version of the model that uses unsupervised statistical clustering to reduce the model to a tiny fraction of its original size (e.g., keep track of 350 clusters vs. 4.3 million files). We evaluate our model with two case studies and show that it can be used in place of the real traces, outperforming previous models.

This model is workload-agnostic, making it suitable for emerging workloads. The use of statistical clustering to group objects according to their behavior naturally leads to type-aware synthetic trace generation and analysis. Furthermore, these "types" of objects are identified without human intervention. Previous domain knowledge is not needed in order to identify the object types, and the process cannot be biased by preexisting misconceptions about the workload. To the best of our knowledge, we are the first to propose an autonomic type-aware model for object request streams. These models are presented in Chapter 5.

Finally, we extend the prior model to make it suitable for *namespace metadata management benchmarking*. We use this model to implement a metadata-intensive storage benchmark to issue realistic metadata workloads on HDFS. This benchmark can be used to evaluate the performance of next-generation storage systems under realistic workloads, without having to deploy a large cluster and its applications. In Chapter 6 we present this benchmark, MimesisBench, and demonstrate its usefulness through a study of the scalability and performance of the Hadoop Distributed File System (HDFS). MimesisBench is extensible, sup-

[1]The objects in this case are files, but the model can be applied to other types of objects like disk blocks, Web documents, and media objects.

ports multi-dimensional workload scaling, and its autonomic type-aware model facilitates sensitivity and 'what-if' analysis.

1.6 Dissertation outline

This dissertation is structured as follows: Chapter 2 describes the related background in MapReduce clusters and Big Data storage systems. In Chapter 3, we describe the problem of evaluating namespace metadata management schemes and identify the limitations of prior approaches; a proof-of-concept model is presented and we show how it outperforms traditional yet simplistic approaches. Chapter 4 describes our findings of a detailed workload characterization of how MapReduce clusters interact with the storage layer. Chapter 5 describes a model for object request streams based on delayed renewal processes, as well as a reduced version of this model, that uses statistical clustering to enable type-aware workload generation. Chapter 6 extends the previously proposed model and uses it in a namespace metadata benchmark for HDFS. Finally, Chapter 7 concludes the thesis.

Chapter 2

Background: MapReduce Clusters and Next-Generation Storage Systems

In general, we are interested in storage systems supporting Big Data applications. Specifically, we have studied storage clusters supporting MapReduce jobs. In this Chapter we provide a summary of how MapReduce clusters are configured and of the challenges in design and performance tuning faced by next-generation storage systems. We also analyze the issue of evaluating namespace metadata management schemes and identify limitations with existing approaches.

2.1 MapReduce clusters

MapReduce clusters [13, 30] offer a distributed computing platform suitable for data-intensive applications. MapReduce was originally proposed by Google and its most widely deployed implementation, Hadoop, is used by many companies including Facebook, Yahoo and Twitter [63].

MapReduce uses a divide-and-conquer approach in which input data are divided into fixed size units processed independently and in parallel by *map* tasks, which are executed distributedly across the nodes in the cluster. After the map tasks are executed, their output is shuffled, sorted and then processed in parallel by one or more *reduce* tasks.

To avoid the network bottlenecks due to moving data into and out of the compute nodes, a distributed file system typically co-exists with the compute nodes (GFS [37] for Google's MapReduce and HDFS [70] for Hadoop).

MapReduce clusters have a master-slave design for the compute and storage systems. In the storage system, the master node handles the metadata operations, while the slaves handle the read/writes initiated by clients. Files are divided into fixed-sized blocks, each stored at a different data node. Files are read-only, but appends may be performed in some implementations. For the sake of simplicity, in this proposal we refer to the components of the distributed file system using the HDFS terminology, where *name node* refers to the master node and *data node*

refers to the slave.

These storage systems may use replication for reliability and load balancing purposes. GFS and HDFS support a configurable number of replicas per file; by default, each block of a file is replicated three times. HDFS's default replica placement is as follows. The first replica of a block goes to the node writing the data; the second, to a random node in the same rack; and the last, to a random node. This design provides a good balance between being insensitive to correlated failures (e.g., whole rack failure) and minimizing the inter-rack data transmission. A block is read from the closest node: node local, or rack local, or remote.

2.2 Storage systems for petascales and beyond

Efficient storage system design is increasingly important as next-generation storage systems are designed for the peta and exascale era. These increasingly larger workloads have led to new storage systems being designed and deployed. Examples of storage systems designed for these emerging workloads include GFS [37], HDFS [70], Ceph [88], QFS [64] (formerly KFS [46]), Lustre [65], OrangeFS [92] (a branch of PVFS), Panasas [90], and Tachyon [1].

In the remainder of this section, we briefly discuss some of the challenges in the design and performance tuning of these systems.

Large-scale storage systems typically implement a separation between data and metadata in order to maintain high performance [9]. With this approach, the *management of the metadata* is handled by one or more namespace or metadata servers (MDSs), which implement the file system semantics; clients interact with the MDSs to obtain access capabilities and location information for the data [90]. Larger I/O bandwidth can be achieved by adding storage nodes to the cluster, but improvements in metadata performance cannot be achieved simply by deploying more MDSs, as the defining performance characteristic for serving metadata is not bandwidth but rather latency and number of concurrent operations that the MDSs can handle [9, 31]. Data-intensive applications can lead, in many instances, to metadata-intensive workloads due to too many small files being created by applications [9, 62, 67, 72], creating a bottleneck in the MDSs. For this reason, novel and improved distributed metadata management mechanisms are being proposed [57, 60, 82, 90, 91].

Figure 2.1 shows how the separation between data and metadata is often im-

plemented. Note that in some deployments, the clients may be co-located with the storage nodes; for example, in Hadoop the MapReduce job clients (or YARN containers) are co-located with the HDFS data nodes.

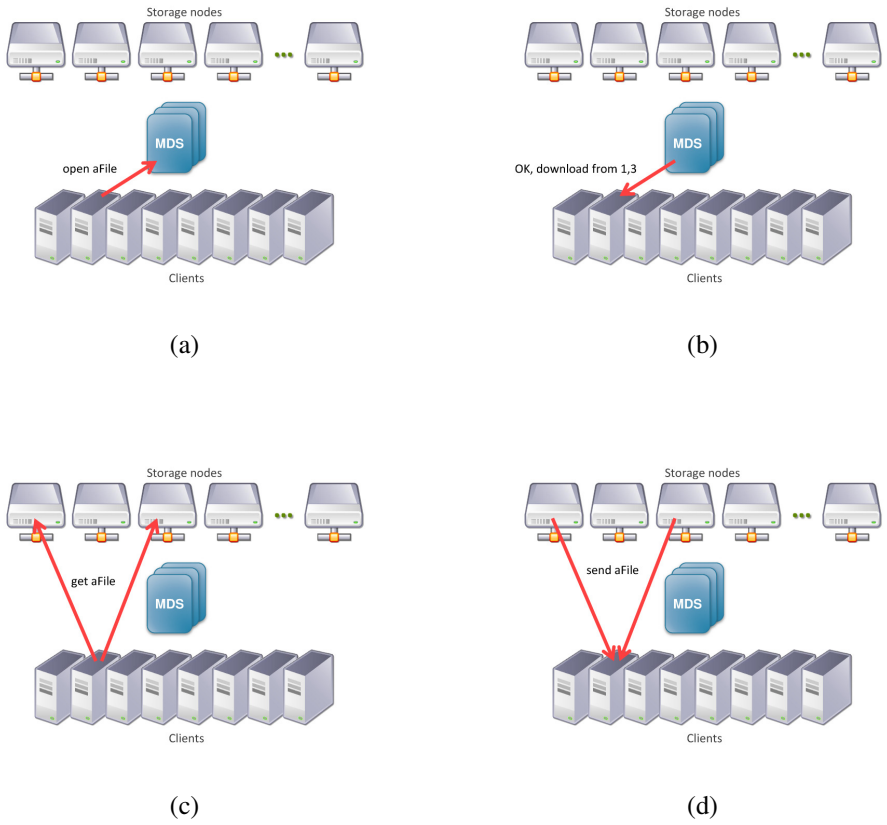


Figure 2.1: Example of a storage system that separates the management of the data and metadata operations. In some deployments the clients may be co-located with the storage nodes (e.g., in Hadoop the job clients are co-located with the HDFS data nodes).

Data replication (at the level of files or blocks/chunks) is a common approach to reduce hotspots, distribute the load among the nodes, increase I/O throughput, and increase data availability. For this reason, some research on next-generation storage systems focuses proposing new replica number calculation schemes and/or new replica placement schemes [2, 17, 51, 52, 77, 85, 87, 89].

Caching [11, 33, 94] in a storage tier faster than disk (e.g., RAM or SSDs), is frequently used to improve I/O throughput, reduce job completion times, reduce network bandwidth consumption, and minimize data latency.

Big Data leads to big storage systems, which can have thousands of nodes

per deployment, and in turn lead to high energy costs. To reduce costs, different *power management* schemes are being proposed [44, 45, 66, 94].

The effectiveness and performance of the previously described mechanisms depends on the specific workload sustained by the storage system. However, realistic validation and evaluation of these designs is hard as no adequate traces or workload models are publicly available. Our work in this thesis provides a step towards providing realistic workloads, models and benchmarks that can be leveraged for storage systems design, performance tuning, and evaluation.

Chapter 3

Towards a Metadata Workload Model

No Big Data storage system metadata trace is publicly available and existing ones are a poor replacement. We studied publicly available traces and one Big Data trace from Yahoo and note some of the differences and their implications to metadata management studies. We discuss the insufficiency of existing evaluation approaches and present a first step towards a statistical metadata workload model that can capture the relevant characteristics of a workload and is suitable for synthetic workload generation. We describe Mimesis, a proof-of-concept synthetic workload generator, and evaluate its usefulness through a case study in a *least recently used* metadata cache for the Hadoop Distributed File System.

3.1 Motivation

Large-scale file systems in the Internet services and high-performance computing communities already handle Big Data: Facebook has 21PB in 200M objects and Jaguar ORNL has 5PB [3]. Stored data is growing so fast that exascale storage systems are expected by 2018-2020, by which point there should be over five hundred 10PB deployments [36].

Large-scale storage systems frequently implement a separation between data and metadata to maintain high performance [9]. With this approach, the management of the metadata is handled by one or more namespace or metadata servers (MDSs), which implement the file system semantics; clients interact with the MDSs to obtain access capabilities and location information for the data [90]. Larger I/O bandwidth is achieved by adding storage nodes to the cluster, but improvements in metadata performance cannot be achieved by simply deploying more MDSs, as the defining performance characteristic for serving metadata is not bandwidth but rather latency and number of concurrent operations that the MDSs can handle [9, 31]. For this reason, novel and improved distributed metadata management mechanisms are being proposed [57, 60, 90]. However, realistic

validation and evaluation of these designs is not possible as no adequate traces or workload models are available (see § 3.3).

We argue for the need of Big Data traces and workload models to enable advances in the state-of-the-art in metadata management. Specifically, we consider the case of namespace metadata traces. We define a **namespace metadata trace** as a storage system trace that contains both a snapshot of the namespace (file and directory hierarchy) as well as a set of events that operate atop that namespace (e.g., open a file, list directory contents, create a file). I/O operations need not be listed, making the trace smaller. Namespace metadata traces can be used to evaluate namespace management systems, including their load balancing, partitioning, and caching components. Our methodology is presented in § 3.2.

We focus on namespace metadata management because it is an important problem for next-generation file systems. However, other types of research that need information about realistic namespaces and/or realistic workloads or data access patterns in Big Data systems could benefit significantly from access to these traces and models: job scheduling mechanisms that aim to increase data locality [93], dynamic replication [2], search in large namespaces [49], schemes that want to treat popular data differently than unpopular data [11], among others.

Publicly available storage traces do not meet our definition of a *namespace metadata trace* since they do not contain a snapshot of the namespace. Due to the heavy-tailed access patterns observed in Big Data workloads, this means that the trace-induced namespace will not contain a large portion of the namespace (i.e., that portion that was not accessed during the storage trace). More importantly, existing traces are not representative of Big Data workloads and are frequently scaled up through ad-hoc poorly documented mechanisms (see § 3.3).

In Section 3.4, we present a proof-of-concept statistical model for namespace metadata workloads and describe Mimesis, a synthetic workload generator based on our model. Results shown in Section 3.5 suggest that our model outperforms simplistic models like the Independence Reference Model (IRM). In Section 3.6, we discuss the related work; in Section 3.7 we provide a summary of this chapter.

3.2 Datasets and methodology

We support our arguments through an analysis of: a Big Data *namespace metadata trace* from Yahoo and two traces used in prior work (Home02 and EECS in

Table 3.1: Description of the traces used by one or more of the surveyed papers; references in [3].

Trace name	Year	Source description	Publicly available?	Scaled?
Sprite	1991	One month; multiple servers at UC Berkeley.	Yes	Yes
Coda	1991-3	CMU Coda project, 33 hosts running Mach.	Yes	Yes
AUSPEX	1993	NFS activity, 236 clients, one week in UC Berkeley.	Yes	No
HP	2000	10-days; working group, HP-UX time-sharing; 500GB.	Yes	Yes
INS (HP)	2000	HP-UX traces of 20 PCs in labs for undergraduate classes.	Yes	Yes
RES (HP)	2000	HP-UX; 13 PCs; 94.7 million requests, 0.969 million files.	Yes	Yes
Home02 (Harvard)	2001	From campus general-purpose servers; 48GB.	Yes	Yes
EECS (Harvard)	2001	NFS; `home` directory of CS department; 9.5GB.	Yes	Yes

Table 3.2: Traces analyzed in this Chapter; AOA: age at time of access. # files includes files created or deleted during the trace.

Trace	# Files	Used storage	Mean interarrival time (milliseconds)	AOA (median, in seconds)
Yahoo	150M	3.9 PB	1.04	267
Home02	$>$ 1M	48 GB	243.80	4682
EECS	$>$ 1M	9.5 GB	27.20	1228

Tables 3.1-3.2; which are the most recent public traces used in the papers we surveyed [3]). The Big Data trace comes from the largest Hadoop cluster at Yahoo (4000+ nodes, HDFS); this is a production cluster running data-intensive jobs like processing advertisement targeting information. The *namespace metadata trace* has a snapshot of the namespace (04/30/2011) obtained with Hadoop's Offline Image Viewer tool, and a 1-month trace of namespace events (05/2011) obtained by parsing the namenode (MDS) audit logs. See [5] for workload details.

Storage system workloads are multi-dimensional [22, 76] and can be defined by several characteristics like namespace size and shape, arrival patterns, and temporal locality patterns. In this Chapter, we discuss of some of these dimensions where appropriate. One of these dimensions is the temporal locality present in the workload, which we measure through the distribution of the age of a file at the time it is accessed (*AOA*). For every operation (namespace metadata event), we calculate how old the file is and use this information to build a cumulative distribution function (CDF) that represents the workload in this dimension. We chose this dimension because it is one that is very relevant to namespace metadata management, since the temporal locality of the workload has an incidence in mechanisms like load balancing, dynamic namespace partitioning/distribution, and caching.

In this Chapter, an *access* to an object refers to an access through a *namespace metadata event*. For example, getting the attributes of a file constitutes a metadata

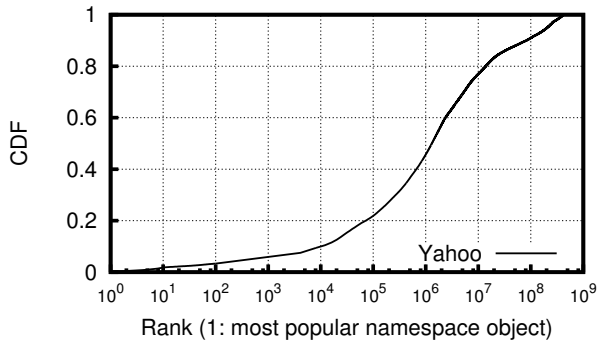


Figure 3.1: Cumulative distribution function of the popularity of namespace objects (files/directories) in Yahoo trace, for those accessed (metadata event) at least once during trace; 3.28% of objects account for 80% of accesses.

access.

3.3 Limitations of existing approaches

Table 3.3: Evaluation approaches used in prior metadata management papers; references in [3].

Evaluation mechanism	**Description**
Metadata-intensive microbenchmarks	
mdtest	Multiple processes create/stat/delete files/dirs. in shared or separate directories.
metarates	MPI program that coordinates file system accesses from multiple clients.
self-designed	Non-standard programs that perform some sequence of namespace operations.
Application benchmark	
Checkpoint	Each process writes its system states to an individual checkpoint file periodically.
	Metadata intensive and commonly found in large scale parallel applications.
SSCA	Pseudo-real applications that mimic high-performance computing workloads.
IMAP build	Metadata operations recorded during a Linux kernel build and IMAP server.
mpiBLAST	MPI program; searches multiple DB fragments and query segments in parallel with large DB size.
Trace-based	
Simulation	Simulator designed by authors; traces may be scaled up.
Replay	Replay of real traces, scaled up to simulate a larger system.

We surveyed the papers published in the last five years that propose novel namespace metadata management schemes and identified their evaluation methodology. The approaches fall into one of three categories listed in Table 3.3. We focus on approach (iii), but briefly discuss (i) and (ii) first.

(i) Metadata-intensive microbenchmarks While many I/O benchmarks exist, they do not evaluate the metadata dimension in isolation [74], making them inadequate for metadata management studies. Mdtest and metarates are metadata-

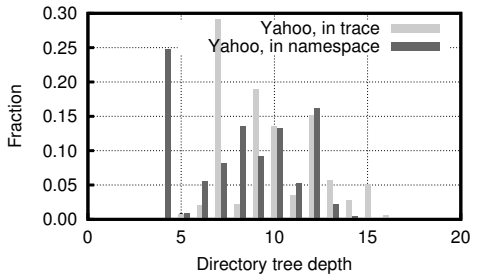


Figure 3.2: Fraction of files at each depth in hierarchy; a high percentage of files are stored at depth 4 but rarely accessed.

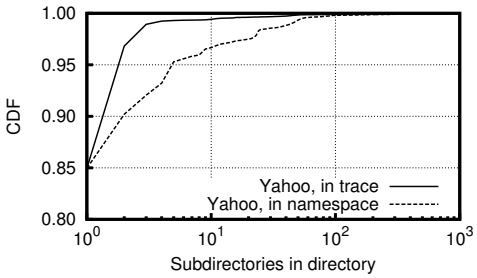


Figure 3.3: In trace-induced namespace, non-leaf subdirectories appear to have fewer children than in snapshot.

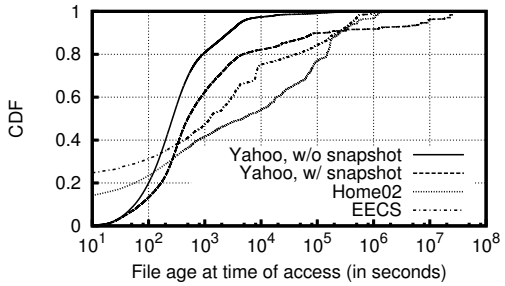


Figure 3.4: File age at time of access for a Big Data and two enterprise storage traces; there's significant difference in temporal locality between workloads.

intensive microbenchmarks; however, they do not attempt to model real workloads. For example, one can use mdtest to issue a high-rate of creates (of zero-sized files) in a random namespace, but not to do creates modeled after real workloads and atop a realistic namespace.

(ii) Application benchmarks Synthetic benchmarks exist and real non-interactive applications can be used too. However, the full workload of a system is typically a combination of different applications and client usage patterns that, as a whole, can differ from the individual application workloads.

3.3.1 Limitations of existing traces

We analyzed the traces used in prior papers (Table 3.1), as well as the storage traces available for download at the Storage Networking Industry Association (SNIA) trace repository, and identified these limitations in the context of namespace metadata management studies for next-generation storage systems:

No public petascale traces are available

Sub-petascale traces are often scaled up in some way; the modification of the traces, if done through ad-hoc poorly documented mechanisms, makes the results difficult to reproduce and raises questions about the validity of the results. Working at a smaller scale is not always adequate since inefficiencies in design/implementation may only be evident at scale.

No traces include both a namespace snapshot and a trace of operations on that namespace

If no namespace snapshot is included with trace, the researcher must rely on the trace-induced namespace[1] (i.e., that portion of the namespace that is accessed during the trace). The trace-induced namespace can be significantly smaller—due to the heavy-tailed access patterns in which some files are rarely, if ever, accessed (see Figure 3.1)—and may have a different form than the full namespace (see Figures 3.2 and 3.3). In other words, the trace-induced namespace is a bad predictor of the actual namespace, and the omission of a significant portion of the files and directories in an evaluation could affect its results.

Additionally, any study that requires knowledge only available in the snapshot (e.g., file age or size) would be limited or biased if no snapshot is available. For an example, consider the significant difference in AOA when calculated with full knowledge of the ages of the files and with partial knowledge of the ages (without a snapshot, we can only know the age of the files created during the trace of events) shown in Figure 3.4.

No Big Data traces are available[2]

Big Data workloads may differ considerably from more traditional workloads. For example, consider the age of a file at the time it is accessed, which is a measure of the temporal locality on a trace based on previous Big Data observations. Figure 3.4 shows the difference in temporal locality between two traditional traces and a Big Data trace[3]. In the Yahoo trace, most of the accesses to a file occur

[1] Or, create a namespace from a model [7], if available.

[2] Application workloads exist, such as Hadoop's Gridmix for MapReduce and the simulation traces from Sandia National Labs. While these traces are useful, they do not represent the full load observed by the *storage* system.

[3] For traces without a namespace snapshot (Home02 and EECS), it is not possible to know the age of files that were created before the trace of events. To enable comparison between traces, for

within a small window of time after the file is created; this is typical of MapReduce pipelines. In contrast, in Home02 and EECS files remain popular longer.

3.3.2 Lack of metadata workload models

When adequate traces are not available, workload models can be useful by enabling researchers to generate synthetic traces or modify existing traces in a meaningful way. While several models have been proposed to describe certain storage system properties (e.g., directory structure in namespace snapshots [7]), to the best of our knowledge no work has been published that proposes a model that combines a namespace structure and a (metadata) workload on that trace. This lack of adequate models make it hard for researchers to design their own synthetic workloads or to modify existing traces to better fit access patterns of large scale storage systems.

3.4 Towards a metadata workload model

We present a model for namespace metadata traces that *captures the relevant statistical characteristics of the workload and is suitable for synthetic workload generation*.

We begin the description of our model by formalizing our definition of a namespace metadata trace as follows.

Definition 1. *In a distributed file system that separates metadata management from data storage, a* ***namespace event*** *is a client request received by the namespace metadata management subsystem, which corresponds to a namespace request. Typical requests include those to create, open, or delete a file, creating or deleting a directory, and listing the contents of a directory. The format of each event record in a trace may vary between systems, but its simplest form is: <timestamp, operation, operation parameters>.*

Definition 2. *Let S_t be a snapshot of the namespace at time t and $E = \{e_i...e_k\}$ be the list of all the namespace events as observed by the namespace server(s). Then, a* ***namespace metadata trace****, T_{t1-t2}, is a trace that contains a snapshot S_{t1}*

the Yahoo trace we plotted the AOA with full information of the file ages (with snapshot) and after ignoring those files that were not created during the trace (without snapshot).

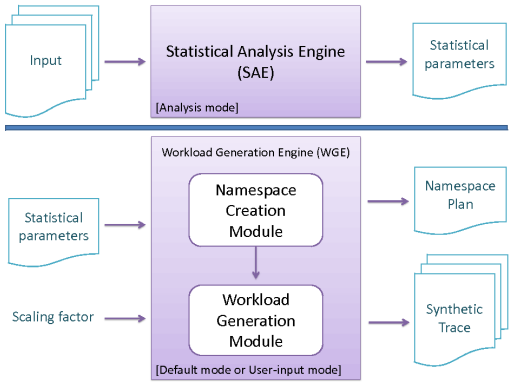


Figure 3.5: Block diagram of our synthetic workload generator, Mimesis.

of the namespace at a time t_1*, and a set* E_{t1-t2} *of events* $e_{t1}...e_{t2}$*, where* $t2 = t1 + \delta$*,* $\delta \gg 0$ *and* $e_j \in E_{t1-t2}$ *iff* $e_j \in E,\ j \geq t1$ *and* $j \leq t2$*.*

Given T_{t1-t2}, we can model its workload as a set of parameters that together define the namespace (modeled after S_{t_1}) and the workload of metadata events (modeled after E_{t1-t2}). Our current implementation contains the statistical parameters listed in Table 3.4, which is the set of probability distributions (empirical or fit to a known distribution) that describe the most relevant[4] statistical properties of the namespace, namespace as used in the workload, and of the workload itself.

We have an implementation, called Mimesis (see Figure 3.5), that contains a format in which the model can be stored, processed and shared, and two subcomponents: (1) the Statistical Analysis Engine (SAE), that takes T_{t1-t2} and generates a configuration file with the model parameters, and (2) the Workload Generation Engine (WGE), that takes the configuration file and generates a synthetic metadata trace based on the model.

3.4.1 Parameters

Two sets of parameters characterize the workload, regarding: namespace structure and workload characteristics, including access patterns and trace-induced namespace (Table 3.4).

The namespace structure is extracted from S_{t1} and describes the shape and size of the namespace hierarchy tree. The number of directories and files describe the

[4]Set of parameters chosen after a literature review in storage namespace and workload modeling; other parameters can be added in the future.

size of the namespace. The shape of the hierarchy tree is described by the following distributions: files at each depth, directories at each depth, files per directory and subdirectories per directory. The file size distribution is also extracted, which can be relevant to problems involving data block replication and placement.

The access patterns describe the relationship between the operations and the age of the files, which indirectly describe temporal locality. This is important, for instance, to namespace partitioning or metadata caching. The distribution of the file age at time of access (AOA) and age at time of deletion (AOD) are extracted and reproduced in synthetic workloads.

The workload-induced namespace describes the hierarchy of objects accessed in the trace. The shape of the hierarchy tree induced by the workload can be different from the snapshot when there is a long tail of rarely accessed files. The hierarchy of the accessed objects is relevant to metadata caching, for instance. In addition, the fraction of operation types and interarrival rate distribution are also extracted.

Two main contributors to *locality of reference* in file request streams are the *popularity distributions* and *temporal locality* present in the requests [80]. In our current implementation, we capture the temporal locality of the references, and keep track of file accesses to favor frequently accessed ones (except for *deletes*), as described later in this section.

Input Format and Parameter Extraction

The SAE takes a format shown in Table 3.5. We first convert a namespace metadata trace to this format before the SAE can process it.

Each of the distributions in Table 3.4 takes value as either a known distribution or an empirical CDF. When extracting parameters, the SAE attempts to fit a known distribution to the measured values using R (MASS package). If the values pass the goodness-of-fit test (Kolmogorov-Smirnov), the known distribution becomes the value of the parameter; otherwise, an empirical distribution is built using the CDF obtained from the input data. If multiple distributions pass the goodness-of-fit test, the one with the smallest test statistic (D) is chosen.

Table 3.4: Statistical parameters captured by our model.

Namespace characterization
Number of directories *and* number of files
Distribution of files at each depth in namespace hierarchy
Distribution of directories at each depth in namespace hierarchy
Distribution of number of files per directory
Distribution of number of subdirectories per directory
Distribution of file sizes (in MB)
Distribution of ages of the files at t_1
Workload characterization
Percentages of operation type in trace
Interarrival rate distribution
Distribution of operations observed at each depth in namespace
Distribution of files per depth in namespace, as observed in trace
Distribution of dirs. per depth in namespace, as observed in trace
Distribution of number of files per directory, as observed in trace
Distribution of number of subdirs. per dir., as observed in trace
Distribution of file age at time of access
Distribution of file age at deletion (i.e., file life span)

Table 3.5: Mimesis input and output formats.

(a) HDFS namenode log record example
2011-5-18 00:00:00,134 INFO FSNamesystem.audit: ugi=USERID ip=<IP> cmd=listStatus src=/path/to/file dst=null perm=null
(b) Input/output format: namespace
File creation time stamp, full path, file size (-1 for directories)
(c) Input format: metadata operations
Time stamp, metadata operation, source, destination (for renames)
(d) SAE output format for empirical distributions
Item (discrete) or bin (continuous), count, fraction, CDF
(e) WGE output format; stamp is relative to beginning of trace.
Time stamp, metadata operation, source, destination (for renames)

3.4.2 Generating synthetic traces

The WGE has two modules: namespace creation and workload generation. The former is used to generate a namespace hierarchy that maintains the same structure as the original. The latter is used to generate a synthetic trace that maintains the desired workload characteristics, operates on the namespace generated by the namespace module, and preserves the data access patterns extracted from the original trace.

Namespace creation module

Creates the namespace in two phases: the directory hierarchy is created in the first phase, the files in the second. The naive approach, creating files and directories in parallel, leads to a bias towards locating more files in the lower depths of the hierarchy (which would naturally be created first) [7]. Output format detailed in Table 3.5(b).

Creating the directory hierarchy The simplest approach is to iteratively create the directories starting from $depth$ 1, where $numTargetDirs \times percDirsAtDepth1$ directories are created; to decide how many directories to create in $depth+1$ we can sample from the distribution of subdirectories per directory, once per every directory at the current depth; the iterative process continues until the current depth has zero directories. Unfortunately, the output of this approach does not accurately model the input namespace. Due to the high percentage of directories with 0 or 1 subdirectories (68% and 27%)[5], this iterative process creates a small shallow hierarchy in which the distribution of directories at each depth is not maintained (except for depth 1). Alternatively, we can: (a) use two independent distributions (directories at each depth and subdirectories per directory) and try to match both constraints at the same time, or (b) model them as a joint distribution in which both random variables are defined on the same probability space. We chose approach (a) because it generates a smaller, simpler model, favored by the principle of parsimony; the accuracy of this approach is studied in § 3.4.3.

With approach (a), we need to satisfy two constraints simultaneously: directories at each depth and subdirectories per directory. We model this as a *bin packing* problem:

- There's one bin for each depth of namespace hierarchy.
- The capacity of each bin is defined by multiplying the target number of subdirectories (by default, the same number of directories as in the input namespace) by the fraction of directories at each depth.
- The items being packed is a list of subdirectory counts obtained by sampling from the distribution of subdirectories per directory until the sum of the samples is greater or equal than the target number of subdirectories. If we

[5] A 5-year study of file system metadata also found a high percentage ($> 80\%$) of directories with 0-1 subdirectories [8].

exceed the target, the last sample is adjusted so that the sum of the samples does not exceed it.

Bin packing is a classic NP-complete problem. We use a greedy approach in which we first sort (in descending order) the list of subdirectory counts per directory that we want to pack. Each of these counts is packed as a single item with some specific weight (number of subdirectories); we refer to these as a *set of sibling directories*. We then pack each set of sibling directories in the worst bin (i.e., the one with the most free space). Obtaining a solution using a greedy approach would not typically work out; however, given the high percentage of directories with 0 or 1 subdirectories/children, which we pack last, finding a solution with this approach is feasible.

Once the bins have been packed, we iteratively assign subdirectory counts at each depth starting from depth 1 (the root of the hierarchy is created at the beginning of the process). The process then proceeds as follows: at depth d we assign a parent to all the sets of sibling directories packed in the corresponding bin. For each set, a childless parent from $depth-1$ is chosen. If no childless parent exists, a random parent at $depth-1$ is chosen. Using this method, the distribution of directories per subdirectory may differ slightly from the target (see § 3.4.3).

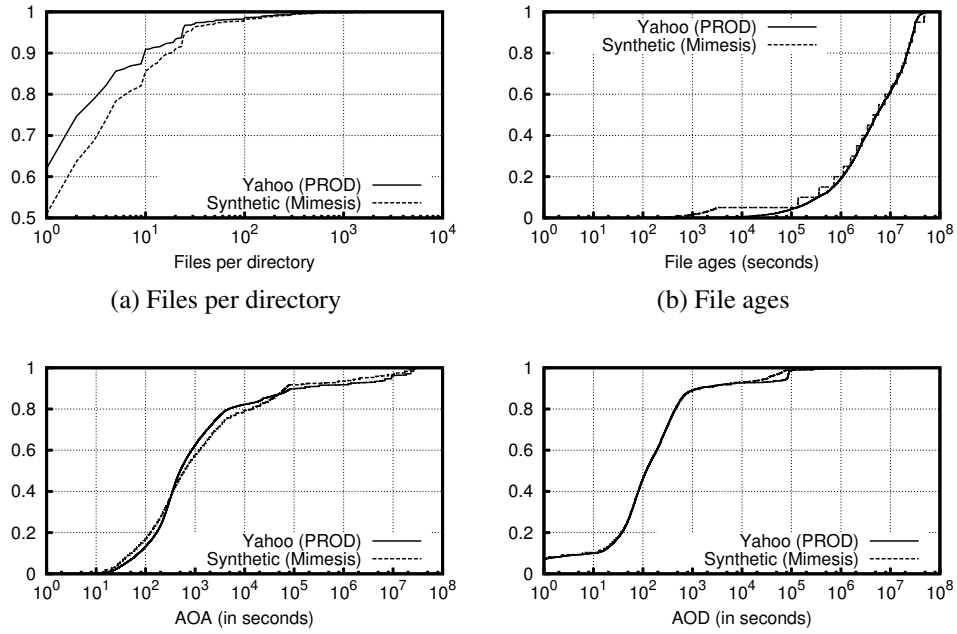
Creating the files The workload generation module chooses files to access according to their age. For this mechanism to work, every file should have a creation time stamp. When creating the namespace, each file is assigned an age randomly sampled from the distribution of file ages; the age is converted to a creation stamp at the end of the file creation process ($time\,stamp = max(sampled\,ages) - age$).

Creating the files has the same multiple constraint issue as directories: distributions of files at each depth and number of files per directory. We model this problem as a bin packing problem in the same way as before.

Finally, the file size is assigned by sampling from the distribution of file sizes.

Workload generation module

Generates the synthetic trace from the output of the namespace module and configuration parameters. It currently preserves the interarrival rates, distribution of operations at each hierarchy depth, percentages of operation types, and temporal locality (AOA and AOD); and the namespace is stressed as in the original trace.



(a) Files per directory

(b) File ages

(c) Age of file at time of access

(d) Age of file at time of deletion

Figure 3.6: Accuracy of Mimesis for some of the statistical parameters (cumulative distribution functions) captured by our model; full results in Table 3.6.

For simplicity, in this section we refer to removal operations as *deletes*, creation operations as *creates* and other operations as *regular accesses*; this classification is applied regardless of whether the operation is performed on a file or directory.

We simulate the operations arriving at the namespace server as events in a discrete event simulation. Interarrivals are sampled from the interarrival distribution defined in the configuration parameters. Upon arrival of an event we make a weighted random selection of the operation based on the table of percentages of operation types.

Next, a target depth is chosen by sampling from the distribution of operations at each depth of the hierarchy. Once the depth in the namespace hierarchy has been determined, the specific file or directory is chosen at that depth for *regular accesses* and *deletes*, or at the target $depth - 1$ for *creates*.

To preserve temporal locality of *regular accesses*, we use the age distribution of a file at the time of access (AOA) obtained by the SAE. We sample from the AOA distribution and obtain a target age. We search for the objects at the desired depth and choose the one with age closest to the target age. If more than one file

approximates the desired age within some configurable *delta* (2000 milliseconds by default), we consider this to be a tie. Mimesis uses popularity information to break the ties as follows: the total number of accesses for each file is recorded during the workload generation. When a tie occurs, the file with the highest number of accesses (i.e., the most popular file) is chosen. The reasoning is that a popular file is more likely to keep receiving accesses than an unpopular file.

Similarly, we preserve the file life span or *AOD* by sampling from this distribution in a *delete*. For the case of deletes, we break ties in the opposite manner: when a tie occurs, we choose the file with the least number of accesses (i.e., the least popular file). The reasoning is that a file that is unpopular is more likely to be deleted than a popular file.

Table 3.5 (e) shows the output format of the trace, which can be used for simulations and for testing real implementations.

3.4.3 Evaluation

We evaluate our approach's: (i) accuracy, (ii) performance, and (iii) usefulness (see § 3.5).

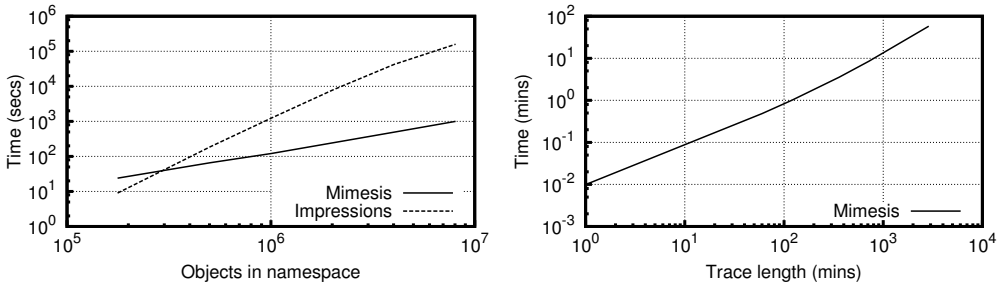
To measure the **accuracy** of the synthetic traces generated by Mimesis, we use the Kolmogorov-Smirnov test statistic (*D*) (or, the maximum difference between the two CDF curves). Table 3.6 shows the accuracy of the synthetic traces generated using our model parametrized after two traces: the Yahoo trace that has been described throughout this Chapter (PROD), and an additional 1-month trace (05/2011) from a 1900+ research and development cluster (R&D) at Yahoo. R&D is used for MapReduce batch jobs and ad-hoc data analytics/business intelligence queries (for details, see [5]). The synthetic traces Mimesis generates maintain the statistical properties of the original trace that are captured by our model with high accuracy (small *D* values).

Figure 3.6 shows the real and synthetic trace CDFs, for four distributions with high *D* values (PROD). For the files per directory CDF (highest *D* value), the error comes from the cases for which no childless parent at $depth-1$ is found and a random parent is chosen instead.

To evaluate **performance**, we generated increasingly larger traces on a two quad-core PC (Xeon E5430, 2.66 GHz) with 16 GB RAM and a 1 TB SATA 7200 RPM disk. Mimesis is currently a single threaded Java program. Figure 3.7a

Table 3.6: Accuracy of Mimesis; two different workloads, averages across 10 trials. The Kolmogorov-Smirnov test statistic (D) converges to 0 if sample comes from target distribution.

Parameter	Yahoo, PROD	Yahoo, R&D
Namespace characterization		
Files at each depth	0.0001	0.0001
Directories at each depth	0.0002	0.0001
Files per directory	0.1105	0.1001
Subdirectories per directory	0.0158	0.0219
File ages	0.0478	0.0457
File sizes	0.0403	0.0419
Workload characterization		
Interarrival times	0.0009	0.0008
Operations at each depth	0.0001	0.0001
Files per depth, trace-induced	0.0001	0.0001
Dirs. per depth, trace-induced	0.0001	0.0001
Files per dir., trace-induced	0.0998	0.0999
Subdirs. per dir., trace-induced	0.0106	0.0113
Age at time of access	0.0592	0.0617
Age at time of deletion	0.0444	0.0457



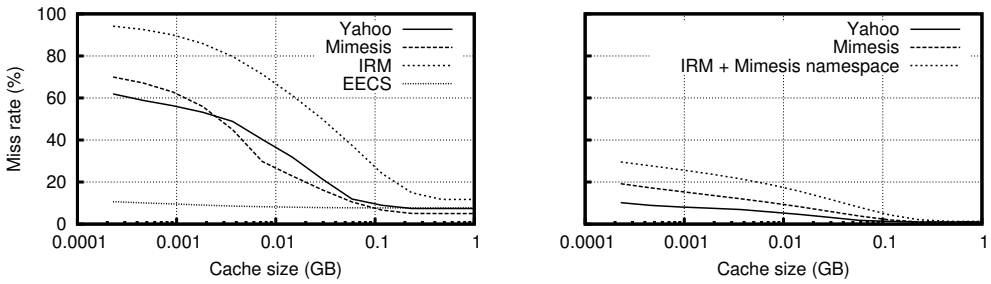
(a) Namespace creation (b) Trace generation

Figure 3.7: Performance of the *namespace creation* and *workload generation* modules; number of objects = number of directories + number of files.

shows that the namespace creation module outperforms Impressions [7] for large namespaces. Figure 3.7b shows how long the workload generation module takes to generate increasingly larger traces.

3.5 Application: Metadata cache for HDFS

We evaluate Mimesis' **usefulness** with a case study: the need for a cache for the HDFS namespace server (MDS).



	MSE	RMSE
Flat namespace		
Mimesis	439.95	5.82
IRM	8939.76	26.22
EECS	12550.90	31.07
Hierarchical namespace		
Mimesis	315.06	4.92
IRM+Mimesis	1934.05	12.20

Figure 3.8: LRU metadata cache miss rates for flat (left) and hierarchical (right) namespaces. The table shows the mean squared error (MSE) and root mean squared error (RMSE) for each model. The synthetic trace created with Mimesis produces the most accurate results (lowest MSE).

3.5.1 Background

The HDFS has reached its scalability limits in large data-intensive systems [67]. One of the areas that can be improved is the MDS metadata handling. HDFS's design was inspired by the GFS [54], and inherited its design choice of keeping all metadata in memory. However, recent studies have shown that the memory footprint of an MDS server grows faster than the physical data storage [67], due to the file-count growth problem [54] which has emerged from an (incorrect) assumption that designing a file system with a large block size would encourage applications and users to generate a small number of large files. Furthermore, this design is wasteful considering that the access patterns in HDFS can show a long tail of infrequently accessed files (see Figure 3.1).

A common approach to this problem is to cache the popular metadata in memory and keep the rest in secondary storage. We evaluate the expected effectiveness of introducing a *least recently used* (LRU) metadata cache for the HDFS.

3.5.2 LRU metadata cache

We developed a simulator that replays a metadata trace and calculates the cache miss rate[6] under different eviction policies; we implemented and evaluated a *least recently used* (LRU) policy. Figure 3.8 shows the miss rate for varying cache sizes[7], calculated after the cache warms up.

To measure the accuracy between the expected cache miss rate (i.e., the one obtained the real Yahoo trace) and the predicted miss rate (i.e., using a model) we use the following metrics: *mean squared error (MSE)*, a classical metric used in statistical modeling, and *root mean squared error (RMSE)*, which has the same units as the quantity being estimated (in this case, the cache miss rate is expressed as a percentage). Results are shown in Figure 3.8 (right) and discussed below.

Case 1 We consider a LRU cache in which each entry uniquely identifies an object (e.g., using a fully qualified path). Traces that contain flat namespaces, hierarchical namespaces, or unique file identifiers can be used to evaluate this approach.

We evaluate this cache using different 2-hour traces: Yahoo, Mimesis, IRM and EECS. The Yahoo trace constitutes the first two-hours of the original Yahoo (PROD) trace. The Mimesis trace was generated using our model, with the parameters configured using the empirical distributions that describe the Yahoo trace, obtained by the Statistical Analysis Engine.

IRM is a trace generated using interarrivals modeled after the interarrivals of the Yahoo trace, and object accesses sampled from the popularity distribution obtained from the Yahoo trace, assuming the Independent Reference Model (IRM)[8].

We scaled up EECS to match the interarrival rate of Yahoo. To scale up or intensify the trace, we used an approach used by prior studies. The trace is "folded" onto itself as follows: the trace (in this case, EECS) is divided into subtraces, then the subtraces are all aligned to time zero while namespace objects are appended with a unique subtrace identifier. This process increases the number of operations per second (*time*) and the namespace size (*space*). To match the request arrival rate of the Yahoo workload, we folded the EECS trace 18 times (i.e., divided it in

[6]The miss rate of a cache is the percentage of accesses for which the data being looked for—in this case, metadata—cannot be located in the cache.

[7]Calculated assuming 1.5 blocks per file and 160 bytes per cache entry, as documented in `https://issues.apache.org/jira/browse/HDFS-1114`.

[8]The IRM assumes that object references are statistically independent.

18 subtraces).

Figure 3.8 (left) shows that using a trace from a system with different access patterns (EECS) is a poor alternative: the EECS miss rate is significantly smaller than the Yahoo one ($RMSE = 31.07\%$) because in EECS objects remain popular longer (see Figure 3.4), thus leading to a higher hit rate in the cache. The IRM trace provides a slightly better approximation because it was modeled after the original workload ($RMSE = 26.22\%$). The file popularity observed in the Yahoo trace is heavy tailed (see Figure 3.1), so the IRM trace does have some very popular objects; however, accesses to unpopular objects appear randomly throughout the trace, whereas in Mimesis accesses to unpopular objects tend to appear close together, as given by the AOA distribution which captures the temporal locality of the original trace. As the cache size increases, the behavior of the IRM trace approaches the real trace behavior as a cache with more entries is less sensitive to temporal locality. The best approximation of the results is obtained with Mimesis, having $RMSE = 5.82\%$.

Case 2 Consider a metadata cache in which each entry of the cache uniquely identifies an object within a directory; the request to access the file `/path/to/file` requires one cache lookup for every element in the path. To evaluate this cache, we need a trace that contains information about the fully qualified path to each object (or a mechanism to associate unique object IDs to fully qualified names).

We evaluate the cache miss rate using different traces: Yahoo, Mimesis, and IRM + Mimesis namespace. Yahoo and Mimesis correspond to the same traces described before.

The IRM + Mimesis namespace was generated as follows: we first used Mimesis' Namespace Creation Module to create a namespace modeled after the original Yahoo namespace; this is the same namespace used in the Mimesis trace. We then created a random permutation of the objects in the namespace to eliminate any bias in the order in which the Namespace Creation Module outputs the list of objects in the namespace. Next, we assigned a rank to each object, according to the randomized order: the first object in the list was assigned rank 1, the second object was assigned rank 2, etc. Finally, we used the IRM model to sample objects according to the (ranked) popularity distribution of objects in the Yahoo trace, associating each rank in the sample with one file in the namespace as given by the order of the random permutation.

This cache has a lower miss rate (see Figure 3.8, middle), resulting from the

hits to the directories at the lower depths in the hierarchy tree. Mimesis outperforms the IRM + Mimesis namespace approach (*RMSE* is 4.92% vs. 12.20%) because it is able to capture the temporal locality of the original workload, while the independent reference model does not.

Our results show that our model provides a good approximation to the real workload ($RMSE < 6\%$). We are exploring further improvements, like combining our model with explicit file popularity information, which could help minimize the *MSE* at the cost of increasing the complexity of the model (and corresponding performance degradation of Mimesis) so the current, simpler, model would be still valuable.

3.6 Discussion and related work

Release of petascale traces by industry would open research opportunities in next-generation storage system design. The first step is obtaining those traces. For some systems, like HDFS, this may be simple since metadata accesses can be logged for auditing purposes and namespace snapshots can be obtained using existing tools. For other systems, unobtrusive mechanisms to obtain these traces may not be available, and should thus be implemented before the traces can be recorded.

Once obtaining the traces is possible, industry can (a) release anonymized traces, or (b) model the workloads of their traces and release these models. To enable the latter, researchers should come up with expressive metadata workload models and tools to process the traces and obtain the models. Workload generators or compilers can be built to take the models as an input and generate realistic synthetic workloads or configuration files in the languages of existing benchmarking tools. While synthetic workloads will, by definition, differ from the original ones, they are useful if they maintain the characteristics of the original workload that the researcher is interested in and, when used in evaluations, lead to results within some small margin of those that would be obtained with the original workload [76]. Selecting those features that make a workload relevant is crucial to this process [22]. Our model and tools constitute a step towards this goal.

Some tools provide a subset of the features of Mimesis. *mdtest* generates metadata intensive scenarios; however, it does not provide a way to fit the workload to real traces or realistic namespaces. Impressions [7] generates realistic file sys-

tem images; however, it is not readily coupled with a workload generator to easily reproduce workloads that operate on the generated namespace. Furthermore, the generative model used by Impressions to create the file system hierarchy is not able to reproduce the distributions observed in our analysis.

Fsstats [29] runs through a file system namespace and captures statistics on file attributes, capacity, directory size, file name length and age, etc. The output of fsstats provides empirical CDFs, but details on the shape of the hierarchy tree are limited to the directory size histogram. LANL has released fsstats reports of large namespaces (up to 0.5 PB).

MediSyn [73] captures the temporal locality of (media server) request streams in a way similar to Mimesis; however, it does not capture or reproduce the storage namespace.

ScalaIOTrace [83] compresses traces so that they can easily be shared, but preserves only minimal data access patterns.

FileBench [34] shares some similarities with Mimesis; however, it lacks a method to extract the statistics from real traces and the configurable statistical parameters on which this tool currently operates does not capture the level of detail captured by Mimesis. On the other hand, it has a mature replay implementation suitable for networked file systems.

3.7 Summary

We considered the case of evaluating namespace metadata management schemes for next-generation file systems suitable for Big Data workloads, and showed why a common evaluation approach—using old traces from traditional storage systems—is not a good alternative. Big Data storage traces and workload models should be used instead; specifically, a *namespace metadata trace* should contain information about both the namespace and the storage workload atop that namespace. We proposed the use of statistical models that can capture the relevant properties of the namespace and workload. We developed Mimesis, a proof-of-concept system that uses a statistical model to generate synthetic traces that mimic the statistical properties of the original Big Data trace.

Chapter 4

A Storage-Centric Analysis of MapReduce Workloads

A huge increase in data storage and processing requirements has lead to *Big Data*, for which next generation storage systems are being designed and implemented. However, we have a limited understanding of the workloads of Big Data storage systems. In this Thesis, we consider the case of one common type of Big Data storage cluster: a cluster dedicated to supporting a mix of MapReduce jobs. We analyze 6-month traces from two large Hadoop clusters at Yahoo and characterize the file popularity, temporal locality, and arrival patterns of the workloads. We identify several interesting properties and compare them with previous observations from web and media server workloads.

4.1 Motivation

Due to an explosive growth of data in the scientific and Internet services communities and a strong desire for storing and processing the data, next generation storage systems are being designed to handle peta and exascale storage requirements. As Big Data storage systems continue to grow, a better understanding of the workloads present in these systems becomes critical for proper design and tuning.

We analyze one type of Big Data storage cluster: clusters dedicated to supporting a mix of MapReduce jobs. Specifically, we study the file access patterns of two multi-petabyte Hadoop clusters at Yahoo across several dimensions, with a focus on popularity, temporal locality and arrival patterns. We analyze two 6-month traces, which together contain more than 940 million creates and 12 billion file open events.

We identify unique properties of the workloads and make the following key observations:

- Workloads are dominated by high file churn (high rate of creates/deletes)

which leads to $80\% - 90\%$ of files being accessed at most 10 times during a 6-month period.

- There is a small percentage of highly popular files: less than 3% of the files account for $34\% - 39\%$ of the accesses (opens).
- Young files account for a high percentage of accesses, but a small percentage of bytes stored. For example, $79\% - 85\%$ of accesses target files that are most one day old, yet add up to $1.87\% - 2.21\%$ of the bytes stored.
- The observed request interarrivals (opens, creates and deletes) are bursty and exhibit self-similar behavior.
- The files are very short-lived: 90% of the file deletions target files that are $22.27\,\text{mins} - 1.25\,\text{hours}$ old.

Derived from these key observations and knowledge of the domain and application workloads running on the clusters, we highlight the following insights and implications to storage system design and tuning:

- The peculiarities observed are mostly derived from short-lived files and high file churn.
- File churn is a result of typical MapReduce workflows: a high-level job is decomposed into multiple MapReduce jobs, arranged in a directed acyclic graph (DAG). Each of these (sub-)jobs writes its final output the storage system, but the output that interests the user is the output of the last job in the graph. The output of the (sub-)jobs is deleted soon after it is consumed.
- High rate of change in file population prompts research on appropriate storage media and tiered storage approaches.
- Caching young files or placing them on a fast storage tier could lead to performance improvement at a low cost.
- "Inactive storage" (due to data retention policies and dead projects) constitutes a significant percentage of stored bytes and files; timely recovery of files and appropriate choice of replication mechanisms and media for passive data can lead to improved storage utilization.

- Our findings call for a model of file popularity that accounts for a very dynamic population.

To the best of our knowledge, ours [5] is the first study of how MapReduce workloads interact with the storage layer.

We provide a description of our datasets in Section 4.2. In Section 4.3 we provide a characterization of the storage workloads of two MapReduce clusters. The related work is discussed in Section 4.4. Finally, in Section 4.6 we provide a summary of this Chapter.

4.2 Dataset description

We analyzed 6-month *namespace metadata traces*[1] from two Hadoop clusters at Yahoo:

- *PROD*: 4100+ nodes, using the Hadoop Distributed File System (HDFS). Production cluster running pipelines of data-intensive MapReduce jobs like processing advertisement targeting information.
- *R&D*: 1900+ HDFS nodes. Research and development cluster with a superset of search, advertising and other data-intensive pipelines.

The jobs in PROD are typically batch jobs that need to run on a regular basis (e.g., hourly jobs, daily jobs, weekly jobs). R&D is used to test some of the jobs running in PROD and jobs to be moved to PROD in the future; in R&D there is less emphasis in timeliness. Additionally, R&D is also used to run somewhat interactive, data-analytics/Business Intelligence queries. Both clusters run plain MapReduce jobs, as well as MapReduce workflows generated by Apache Pig (a dataflow language that is compiled into MapReduce) and by Apache Oozie (a workflow scheduler for Hadoop).

The *namespace metadata traces* analyzed consist of a snapshot of the namespace on June 8th, 2011 (t_0), obtained with Hadoop's Offline Image Viewer tool, and a 6-month access log trace (Jun. 9th, 2011 through Dec. 8, 2011), obtained by parsing the name node audit logs. For some of our analysis, we also processed

[1] We define a **namespace metadata trace** as a storage system trace that contains a snapshot of the namespace (file and directory hierarchy) and a set of events that operate atop that namespace (e.g., open a file, list directory contents) [4]. These traces can be used to evaluate namespace management systems, including their load balancing, partitioning, and caching components.

a snapshot of the namespace taken on Dec. 9, 2011 (t_1). For simplicity, we refer to the log with the set of events (open, create, etc.) as a **trace**. Figure 4.1 shows an example of a typical entry in the HDFS audit logs (trace). Table 4.1 provides a summary of these traces.

```
2012-5-18 00:00:00,134 INFO FSNamesystem.audit:  ugi=USERID ip=<IP-ADDRESS>
cmd=open src=/path/to/file dst=null perm=null
```

Figure 4.1: HDFS name node log record format example.

Table 4.1: Some relevant statistics of the datasets used in this Chapter; $t_0 = 2011/06/08$ and $t_1 = 2011/12/9$; 1 M: 1 million, 1 B: 1000 M.

	Cluster size	**Used storage at:**		**Files in namespace at:**		**Creates during**	**Opens during**
		t_0	t_1	t_0	t_1	$(t_0 - t_1)$	$(t_0 - t_1)$
PROD	4146 nodes	3.83 PB	3.93 PB	51.39 M	54.22 M	721.66 M	9.71 B
R&D	1958 nodes	2.95 PB	3.63 PB	38.26 M	51.37 M	227.05 M	2.93 B

Limitations of the traces: (i) millisecond granularity (a higher granularity would be desirable), and (ii) do not include I/O information. The latter precludes us from knowing the size of a file once its created. While we can obtain the size of the files in a snapshot (say, at time t_0), Yahoo only keeps record of daily snapshots making it impossible to know the size of files created and deleted in between snapshots. Issue (ii) also precludes us from knowing how many bytes are read upon an open event. While MapReduce jobs typically will read a file all at once, we cannot do any analysis that requires certainty in the knowledge of the number of bytes read.

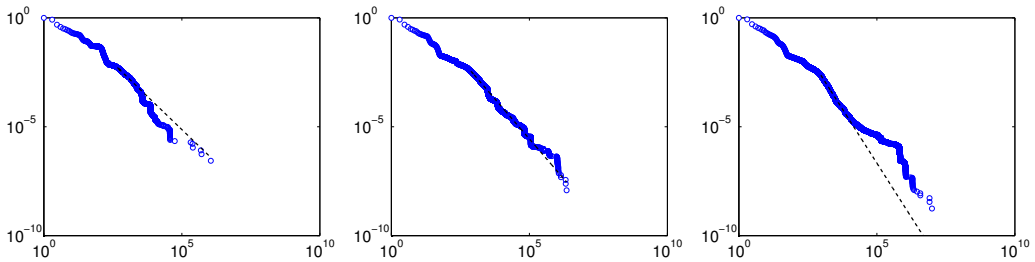
4.3 Analysis of two MapReduce workloads

We present an analysis of the data (file) access patterns present in the traces described in Section 4.2 and discuss the implications to storage design. Other characteristics of the workloads, not directly related to the access patterns, are also presented to help provide a broader characterization of the workloads and which may be of interest to other researchers.

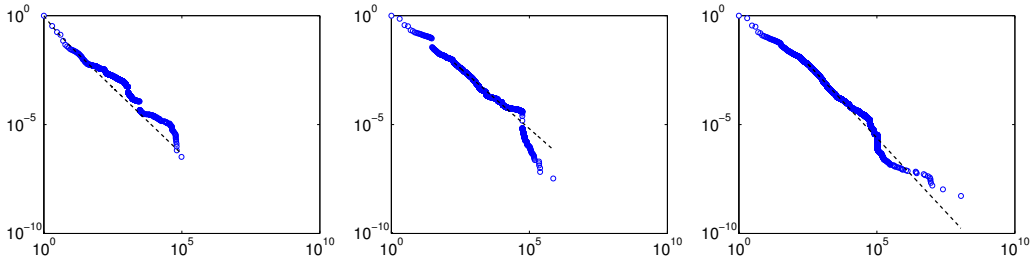
We highlight some of the most important insights using italics; for example, *I0: Insight about workload*.

4.3.1 File popularity

Figure 4.2 shows the Complementary Cumulative Distribution Function (CCDF) of the file accesses (opens), for both clusters, for different periods: first day of the trace, first month of the trace and full six-month trace. The CCDF shows $P(X \geq x)$, or the cumulative proportion of files accessed x or more times. The dashed line shows the best Power Law fit for the tail of the distribution. Files not accessed during the trace were ignored for these plots; a brief discussion on "inactive" storage is presented later in this section.



(a) 1-day trace, PROD (b) 1-month trace, PROD (c) 6-month trace, PROD



(d) 1-day trace, R&D (e) 1-month trace, R&D (f) 6-month trace, R&D

Figure 4.2: Complementary Cumulative Distribution Function (CCDF) of the frequency of file accesses (opens), for increasingly larger traces. The CCDF shows $P(X \geq x)$, or the cumulative proportion of files accessed x or more times in the trace. The dashed line shows the best Power Law fit for the tail.

Table 4.2: Best fit of file access frequency (Figure 4.2) to a Power Law. α: scaling parameter, x_{min}: lower bound of power-law behavior.

	α	x_{min}
PROD, 1-day trace	2.22	464
PROD, 1-month trace	2.47	770
PROD, 6-month trace	2.99	937
R&D, 1-day trace	2.22	1
R&D, 1-month trace	2.11	189
R&D, 6-month trace	2.36	325

Since file access patterns in other workloads exhibit Power Law behavior (or Zipf Law if ranked data is analyzed), we provide the results of the best fit of the tail of the distribution to a Power Law. To find the best fit, we apply the methodology (and toolset) described by Clauset et al. [24]. Results are shown in Figure 4.2 and Table 4.2. The latter shows the Power Law scaling parameter (α) and x_{min}, the value where the fitted tail begins. x_{min} is chosen so that the Kolmogorov-Smirnov goodness-of-fit test statistic (D)—which is the maximum difference between the two CDF curves—is minimized.

How popular are the most popular files? With Power Law tail behavior, a small percentage of the files typically account for a large percentage of the accesses. For example, for enterprise media server workloads the authors in [23] found that between $14\% - 30\%$ of the files account for 90% of the media sessions. In our analysis, we found the workloads to be less skewed towards popular files (see Figure 4.2). Specifically, for the case of the 6-month traces, $22\% - 29\%$ (R&D and PROD, respectively) of the distinct files accessed in the 6-month period account for 90% of the accesses. If we instead calculate these values as a percentage of the total number of files stored at time t_1 (see Table 4.3), the percentages increase to $88.89\% - 304.61\%$ (R&D and PROD, respectively). The percentage above 100% is an artifact of many files in that 90% that were deleted during the 6-month period. None of these two ways of calculating the percentage provide an accurate view of the popularity of the files. The second approach (dividing by the number of files stored at t_1) is obviously wrong, as it may lead to percentages above 100%. It may be less obvious, however, why dividing by the total number of distinct files in the trace is problematic too.

Table 4.3: Most popular files statistics (6-month traces). Refer to the text for an explanation of why some values are above 100%.

	Files accounting for up to 80% / 90% of the accesses
As % of distinct files in trace	
PROD	12.88% / 29%
R&D	5.39% / 22%
As % of files in namepsace (t_1)	
PROD	135.35% / 304.61%
R&D	19.88% / 88.89%

The problem with dividing by the number of distinct files that were accessed at least once during the traces is that this number of files never exists in the system

at one time. From Table 4.1, we have that the number of creates during $t_0 - t_1$ is one order of magnitude larger than the number of files at t_1; most of the files are deleted soon after they are created.

To understand why these short-lived files constitute a problem when quantifying the popularity of files, we flip the question and analyze the *unpopular* files: At the other end of the distribution, we find a high percentage of unpopular (or rarely accessed) files. For example $80-90\%$ of the files are accessed no more than 10 times during the full 6-month period (Table 4.4). An important contributor to the long tail of unpopular files is the high percentage of short lived files (details in Section 4.3.2 and 4.3.4).

Table 4.4: Infrequently accessed files statistics, as a percentage of the number of distinct files in the 6-month traces.

Trace	Files accessed 1 / up to 5 / up to 10 times
PROD	15.03% / 68.40% / 80.98%
R&D	23.66% / 84.25% / 90.08%

Another study from a different Big Data workload (6-month 2009 trace from Bing's Dryad cluster [10]) shows similar findings for unpopular files: 26% of the files were accessed at most once, $\approx 92\%$ at most 5, and $\approx 97\%$ at most 10 times. On the other hand, a study on media server workloads [23] found that $47-59\%$ of the files were requested up to 10 times.

Table 4.4 does not include information about files never accessed in the 6-month period. The files that exist t_1 and were not accessed (open) during the 6-month period constitute *inactive storage* and account for a high percentage of the files ($51\%-52\%$) and bytes stored ($42\%-46\%$) at t_1[2]. Of those files, 33% (R&D) -65% (PROD) ($15\%-26\%$ of bytes) existed at t_0; the rest were created between t_0 and t_1. There are two main reasons for having inactive storage: (i) data retention policies (i.e., minimum period during which a copy of the data needs to be stored), and (ii) dead projects, whose space is manually recovered (freed) from time to time (e.g., when the free space on the cluster falls below some limit).

Automatic dead project space recovery, intelligent replication/encoding [32] and tiered-storage mechanisms can reduce the impact of inactive storage. For

[2]To avoid a distortion in the results due to transient files at t_1, we ignored those files that do not exist at $t_1 + 1$ day. Otherwise, the inactive storage percentage would go up to $57\%-65\%$ of the files at t_1.

example, using erasure encoding instead of replication and a slower (and thus, cheaper) storage tier for passive data can lead to a significant cost reduction while still fulfilling data retention policies.

I1: Inactive storage constitutes a significant percentage of stored bytes and files; timely recovery of files and appropriate choice of replication mechanisms and media for passive data can lead to improved storage utilization and reduced costs.

We use the information gathered from our analysis on the unpopular files to go back to the question of *how popular are the popular files*. Recall that, using the approach of dividing the number of accesses (frequency) by the number of distinct files that were accessed at least one during the trace, we found that 29% of the files in PROD and 22% of the files in R&D account for 90% of the accesses. However, we also know from Table 4.4, that 81% (PROD) − 90% (R&D) of the files are accessed no more than ten times in the full 6-month period. It should now be more clear that these metrics are misleading and can lead to confusion. To be specific, consider the case of R&D: $22\% + 90\% > 100\%$; this means that these two groups are not mutually exclusive, and some files are counted in both groups. In other words, some files in the "very popular" group have been accessed at most 10 times during the 6-month period! While the term "very popular" is subjective, we believe it is unreasonable to apply the tag to a file that has been accessed, on average, less than twice a month.

I2: A model of file popularity with a static number of files is inadequate for workloads with high file churn.

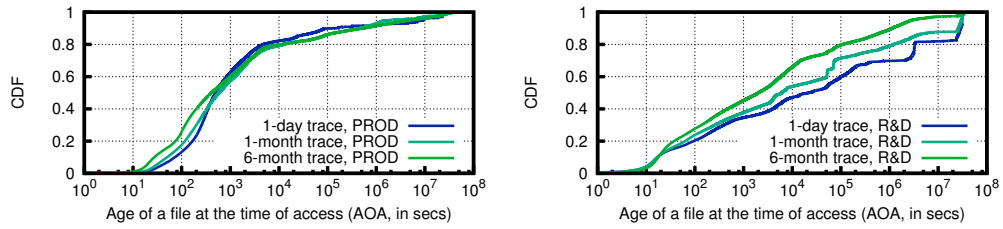
Are some files extremely popular? The answer depends on how we define *extremely*. Since the issues in the percentages discussed before come from the difficulty in deciding what 100% means in a period during which the population has changed significantly, we now use frequency counts instead of percentages. We did an analysis on the files that were accessed at least 10^2 times and those accessed at least 10^3 times (during the 6-month period). For PROD, 117505 files were accessed at least 10^3 times, and 8239081 files were accessed at least 10^2 times, constituting 2.17% and 15.2% of the files in the namespace at t_1. For R&D, 243316 files were accessed at least 10^3 times, and 3199583 files were accessed at least 10^2 times, constituting 0.47% and 6.23% of the files in the namespace at t_1. Finally, if we sum all the accesses to all the files that are in the "10^3 or more accesses" group, we have that 34% of the open events in PROD and 39% of the open events in R&D targeted the top 2.17% and top 0.47% files respectively.

I3: Workloads are dominated by high file churn (high rate of creates/deletes) which leads to $80\% - 90\%$ *of files being accessed at most 10 times during a 6-month period; however, there is a small percentage of highly popular files: less than* 3% *of the files account for* $34\% - 39\%$ *of accesses.*

4.3.2 Temporal locality

Prior studies have noted that MapReduce workloads tend to concentrate most of the accesses to a file in a short period after the file is created [2, 32]. This temporal locality can be captured with the distribution of the *age of a file at the time of access* (AOA). Basically, for each access to a file, we calculate how old the file is at that moment. To do this, we need to know *when* each file was created. We obtain this information from: (a) the namespace snapshot, for those files that were created before the trace was captured, and (b) from the create events present in the trace. Since the HDFS audit logs contain the full path + name of each file instead of a unique identifier for the file, we also kept track of file renames to have an accurate record of the creation stamps.

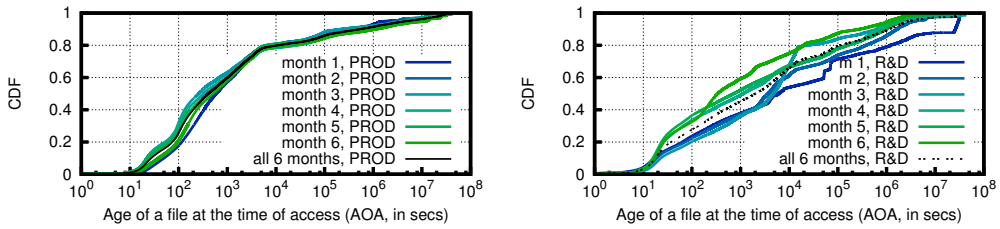
Figure 4.3 shows the AOA for traces of varying length. We observe some changes in the distribution, due to the non-stationary nature of the workload (the monthly changes are shown in Figure 4.4).



(a) PROD cluster (b) R&D cluster

Figure 4.3: Cumulative distribution function (CDF) of the age of a file at each time of access (access = open), for increasingly larger traces.

Let's consider the AOA distribution during the 6-month period (see Figures 4.3 and 4.4 and Table 4.5). In PROD, most accesses target very young files: 50% of the accesses (open events) target files that are at most 407 seconds old. In R&D, files remain popular for a longer (but still short) period: 50% of the accesses target files that are at most 33 minutes old. The difference can be explained by



(a) PROD cluster (b) R&D cluster

Figure 4.4: Cumulative distribution function (CDF) of the age of a file at each time of access (access = open), for each of the months analyzed in this Chapter.

understanding the characteristics of the workloads of these two clusters: the jobs in PROD process data recently recorded or generated (for example, the day's log of user clicks), and they are *not* interactive; on the other hand, some of the jobs on R&D tend to be of the interactive, data-analytics/Business Intelligence type. The latter type of jobs are more user-driven, which accounts for the higher latency between accesses (as opposed to the highly automated batch jobs in PROD).

Table 4.5: Age of a file at a time of access (AOA) statistics (6-month trace). Full distribution of accesses shown in Figure 4.3.

Cluster	50%	80%	90%
PROD	407.80 secs	3.06 hours	6.11 days
R&D	33.53 mins	1.25 days	13.06 days

We notice the closeness of the 90% percentile in Table 4.5 (PROD) to the 1-week mark and ask this question: *What percentage of accesses target files that are at most one week old?* The answer, is surprisingly close for both clusters: 90.31% (PROD) and 86.87% (R&D). To provide some perspective, a media server study [23] found that the first five weeks of a file's existence account for $70-80\%$ of their accesses.

Regarding accesses to very young files, $29\%-30\%$ of accesses target files that are at most 2 minutes old. We believe this is an effect of the typically short duration of the MapReduce jobs on these clusters. For example, during the same 6-month period, $34.75\%-57.46\%$ (PROD and R&D) of the successful jobs had a total running time of 1 minute or less (including the time waiting on the scheduler queue)[3].

[3]We obtained these percentages by analyzing the job tracker's (Hadoop's central scheduler) logs.

The access skew towards young files can be exploited by caching strategies and tiered-storage designs. For example, 78.91% (R&D) to 85.41% (PROD) of the accesses target files that are most 1 day old. However, at one particular point in time (say, at t_1), the files these files constitute 1.01% (R&D) to 3.67% (PROD) of the files and 1.87% (R&D) to 2.21% (PROD) of the bytes stored. Caching these files or storing them on a tier faster than disk would improve performance.

I4: Young files account for a high percentage of accesses but a small percentage of bytes stored. Caching or placing these files on a fast storage tier could improve performance at a low cost.

For a particular cluster, there could be consistent changes in the AOA distribution as time progresses. For example, the curve could slowly start moving to the left or to the right every month. This behavior can be seen in R&D but not in PROD (Figure 4.4). We believe the difference can be explained by the nature of the workloads. Jobs in PROD are repetitive across days, weeks and months. On the other hand, jobs in R&D are more dynamic and user-driven, with changes influenced by short-term user needs.

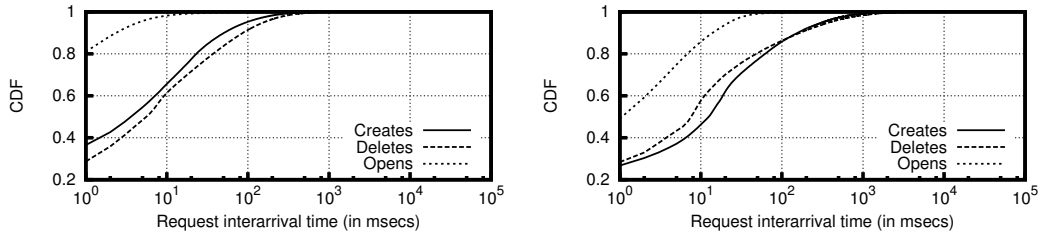
4.3.3 Request arrivals

We analyze the arrivals of the operation requests at the namespace server. Figure 4.5 shows the cumulative distribution function (CDF) of the interarrivals of the different operations (open, create, delete) in the 6-month traces[4]. As expected, the open operations are more frequent than the creates and deletes, but it is interesting to observe the high rate at which files are created (and deleted). For example, in PROD 36.5% of the create operations have an interarrival of 1 millisecond or less.

To model the interarrivals, one can fit the observed interarrivals to a known distribution and use this distribution to sample the interarrivals, or use the empirical distribution described by the CDF if no good fit is found. However, defining the interarrivals by using a CDF (empirical or fitted) implicitly assumes independence of the random process.

Interarrival times may present autocorrelations; for example, previous studies on Ethernet and Web traffic have shown that they are often bursty and even self-

[4]We did not analyze the arrivals of the other types of operations (e.g, listStatus, mkdir, etc.) because those operations are related to the namespace (and not the data), and are thus out of the scope of this Thesis.



(a) PROD cluster (b) R&D cluster

Figure 4.5: Cumulative distribution function (CDF) of the interarrival times of the open, create and delete events, during the 6-month period.

similar[5] [26, 48]. Self-similar arrivals have implications to the performance of the server queues (in this case, the namespace metadata server): unlike Poisson arrivals, self-similar arrivals exhibit burstiness that may itself be bursty and requests may backlog in the queue of the server if it is not able to serve them fast enough. In other words, the queue length distribution of a self-similar process decays more slowly than that of a Poisson process [59].

Figure 4.6 shows the burstiness of the arrivals of the open events, at different time scales; the arrivals exhibit burstiness, even at increasingly larger aggregation periods, a sign of self-similar behavior. Create and delete arrivals are also bursty; we do not include those graphs due to space constraints.

To test for self-similarity, we use two methods used by previous literature [26]: the *variance-time plot* and the *R/S plot*, and estimate the *Hurst parameter (H)*. The *Hurst parameter* provides a measure of the burstiness of a time series (in this case, the counting process of the arrivals); more formally, it expresses the speed of decay of the series' autocorrelation function [26]. The results are shown in Figure 4.7. Using the *variance-time plot* method we estimate $H = 0.937$ (PROD) and $H = 0.902$ (R&D); using the *R/S plot* method we estimate $H = 0.8136$ (PROD) and $H = 0.9355$ (R&D). These results correspond to the first hour of the 6-month trace, with an aggregation period m of 60 msecs (e.g., we counted the arrivals for each of the 60,000 non-overlapping time slots of 60 milliseconds each during that hour). In all cases, $1/2 < H < 1$, which implies that the time series is self-similar with long-range dependence (i.e., the autocorrelation function decays slower than exponentially).

[5] A *self-similar* process behaves the same when viewed at different scales. In this context, the request arrivals are bursty at different time scales.

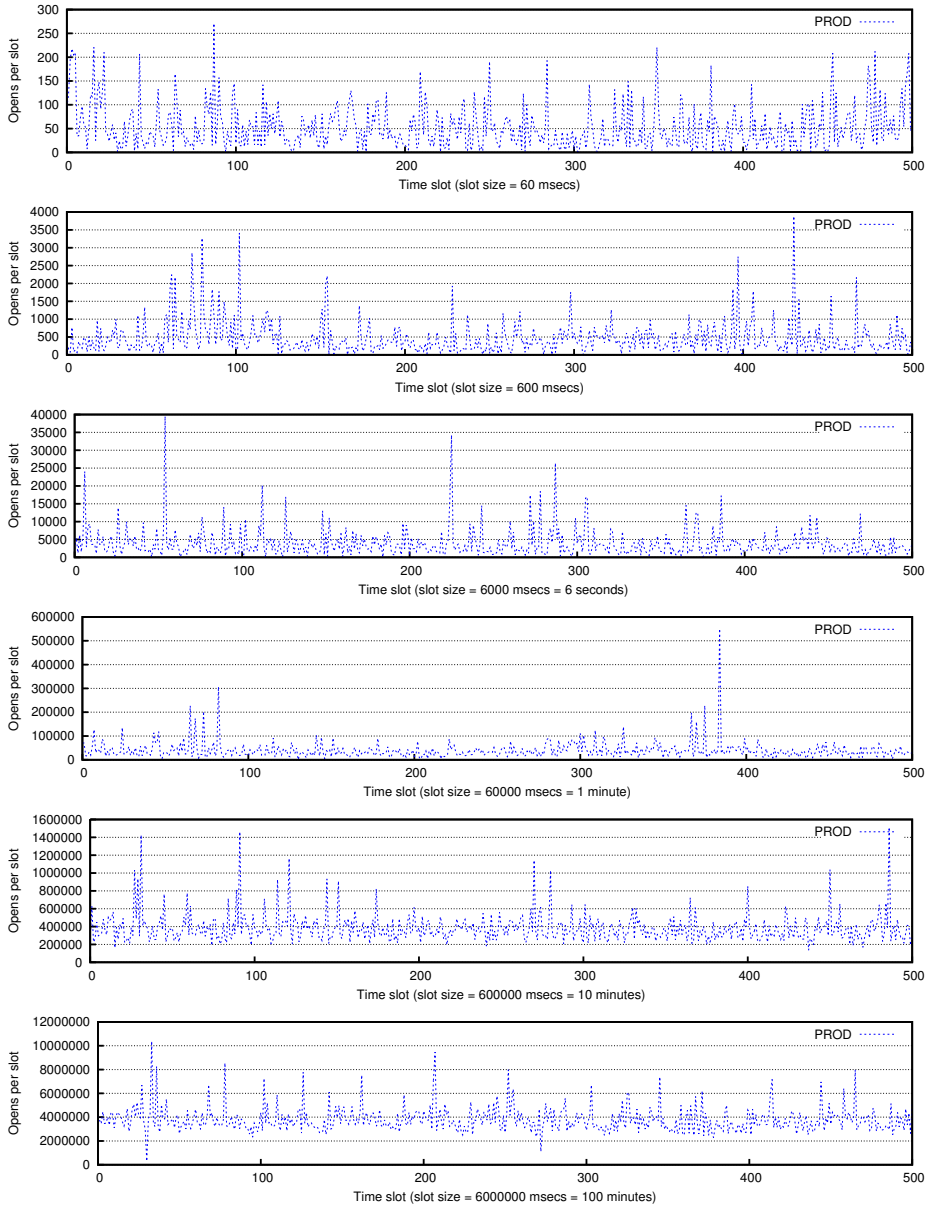
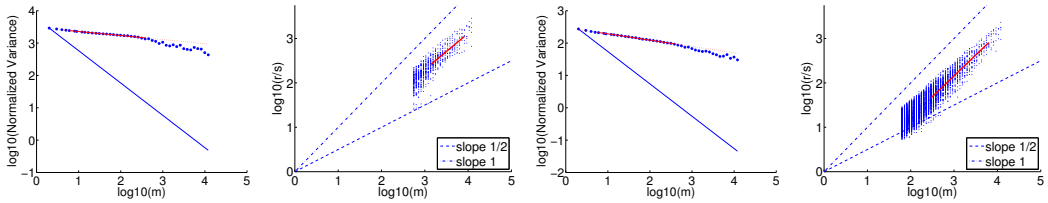


Figure 4.6: Number of opens per time slot, for the PROD cluster. The arrivals are bursty at different timescales.

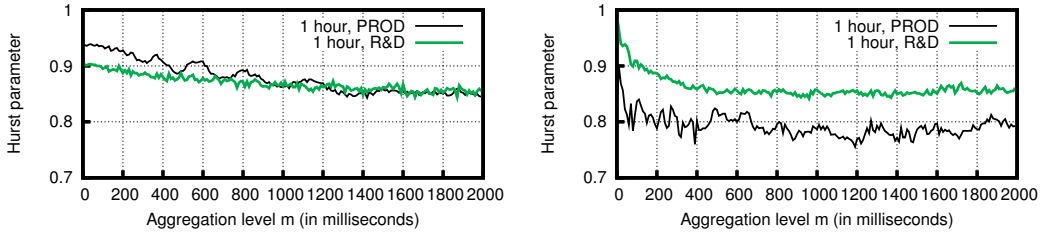
Figure 4.8 shows the estimation of H during the same hour, for varying aggregation periods (m). Figure 4.9 shows the changes in H for every hour of a 24-hour period. For all these cases, the estimation of H is consistently in the range $1/2 < H < 1$. The variability in the value of H in Figure 4.9 is due to load changes during different hours of the day, with a smaller H during less busy hours [26]. The value of H in PROD is much more stable because this cluster is not affected by user working hours. The load of R&D is user-driven and, thus,



(a) Variance-time plot, $H = 0.9370$ (b) R/S plot, $H = 0.8136$ (c) Variance-time plot, $H = 0.9020$ (d) R/S plot, $H = 0.9355$

Figure 4.7: Graphic analysis of the self-similarity in the arrivals of the open requests and Hurst parameter (H) estimate, for PROD (a–b) and R&D (c–d), during the first hour of the trace, calculated with an aggregation period $m = 60$ msecs.

more variable; the spike at the end of the day results from jobs scheduled during low usage hours.



(a) Variance-time plot method (b) R/S plot method

Figure 4.8: Estimation of H with varying aggregation period m, for both clusters (1^{st} hour of the 6-month period).

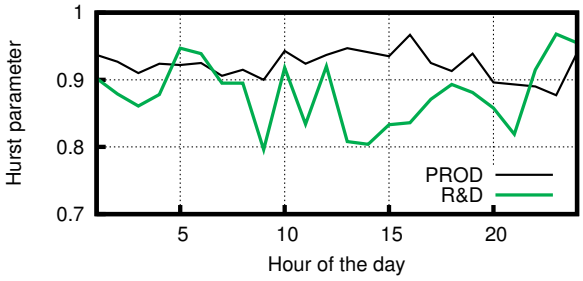


Figure 4.9: Estimate of H during a consecutive 24-hour period (1^{st} day of 6-month trace); there is some change in burstiness during the day.

To accurately model the arrivals of the requests, we should preserve the inter-arrivals *and* the burstiness; for example, by using Markovian Arrival Processes [19] which can capture the autocorrelations present in the trace (ordering) with a minor loss in accuracy in the distribution fitting.

I5: The request interarrivals are bursty and exhibit self-similar behavior.

The arrivals of create and delete operations are also bursty and self-similar. Table 4.6 shows the estimate of H for the arrival processes of creates and deletes ($m = 60$ msecs).

Table 4.6: Estimation of H for creates and deletes, during the first hour of the 6-month traces and $m = 60$ msecs.

	PROD	R&D
Creates		
Variance-time plot method	0.8840	0.9280
R/S plot method	0.9919	0.9696
Deletes		
Variance-time plot method	0.8670	0.9310
R/S plot method	0.8832	0.8716

4.3.4 Life span of files

Figure 4.10 shows the distribution of the age of a file at the time of deletion (how old are files when deleted). The files are short-lived: 90% of deletions target files that are 22.27 mins (PROD) to 1.25 hours (R&D) old (see Table 4.7). In more traditional workloads like that of media servers, files have a longer life span: a study [23] found that $37\% - 50\%$ of media files "live" (calculated as time between the first and last access) less than a month, a lower bound on the real life span.

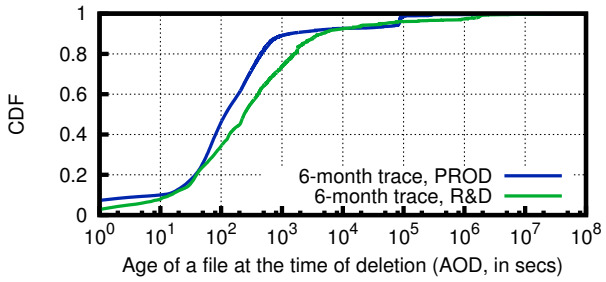


Figure 4.10: Age of file at the time of deletion (AOD), for files deleted during trace. This distribution encodes information about the life span of files.

In MapReduce workloads, many jobs are actually composed of several smaller MapReduce jobs executed in series (or as part of a Directed Acyclic Graph (DAG) workflow); the output of one job is the input of the next one. These files are

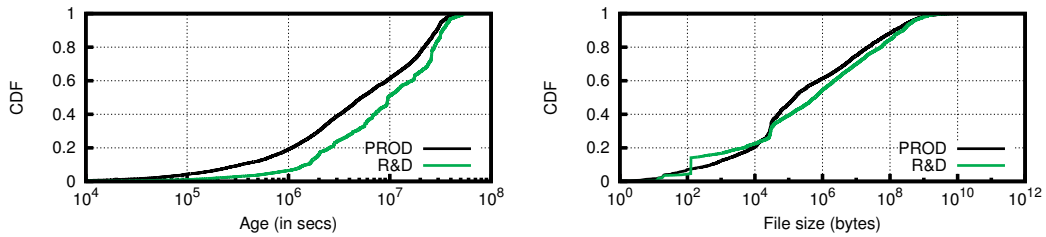
Table 4.7: Age of a file at the time of deletion (AOD) statistics (6-month trace). Full distribution of accesses shown in Figure 4.10.

Cluster	50%	80%	90%
PROD	117.1 secs	453.36 secs	22.27 mins
R&D	238.51 secs	26.61 mins	1.25 hours

not needed after the next job finishes and are thus deleted soon after consumed, leading to short-lived files and high file churn. They should be thought of as "intermediate" data that is temporarily written into the distributed storage system[6].

I6: The files are very short-lived: 90% of deletions target files that are 22.27 mins – 1.25 hours old.

For modeling and workload generation purposes it may be useful to know the age of the files that exist in the system at t_0 [4]. Figure 4.11a shows this distribution. Note that in R&D—where there is less emphasis on processing "fresh" data—the stored files tend to be older: the file age median in R&D is 111.04 days vs. 60.85 days in PROD.



(a) (b)

Figure 4.11: CDF of the ages and sizes of the files in the namespace snapshot at t_0. In (b), we binned the file sizes using a 2MB bin; zero-sized files were excluded from the total. At t_0, the zero-sized files constitute 4% (PROD) and 11% (R&D) of the files in the namespace.

4.3.5 File sizes

Figure 4.11b shows the sizes of the files in the namespace at t_0, using 2MB bins. This information is useful for modeling the initial status of the namespace [4]. We ignored the files with size 0, which constitute 4% (PROD) and 11% (R&D) of the files in the namespace, because they increase significantly the percentage of the first bin and would not allow us to know the exact percentage of very small

[6]Not to be confused with the intermediate data that is generated by the map phase and consumed by the reduce phase, which is not written to HDFS.

files. The zero-sized files would otherwise account for 99.9% (PROD) and 99.98% (R&D) of the files in the 0-2 MB bin.

Why is there such a high number of zero-sized files? Hadoop uses some zero-sized files as flags (e.g., signaling that a job has finished). Flag files constitute 35% (PROD) and 5.6% (R&D) of the zero-sized files. However, the biggest contributor to zero-sized files are empty job output files. For example, out of 100 reducers, some percentage of those may not generate any data, but will nevertheless generate an empty output file named `part-X` where X is a number that identifies the reducer (or mapper) that generated the file. These files account for 52.96% (PROD) and 93.71% (R&D) of the zero-sized files at t_0.

Is there a correlation between the size of a file and its popularity? We calculated the *Pearson's correlation coefficient* between the size of a file and its popularity, using both the number of accesses and the file rank as a metric of popularity, and using two types of binning approaches for the data: fixed-size bins of 2MB and bins with logarithmically (*ln*) larger width. We found no strong correlation between the size of a file and its popularity, although a medium correlation is found in the R&D cluster, when using the file rank as the popularity metric (Table 4.8). However, as indicated in Section 4.2, we cannot know the size of files created and deleted in between the daily snapshots, so those files were ignored in this analysis. Sampling prior to a correlation analysis yields accurate results if there is no bias in the sampling of the data. In this case, we know that there is a bias against short-lived files but cannot tell if there is a bias in the file sizes. It is possible that the correlation results would be different if all files had been considered, so this issue warrants further analysis in the future.

Table 4.8: Pearson's correlation coefficient between file size and popularity (6-month traces); $|1| \Rightarrow$ strong correlation.

	PROD	R&D
Access count – 2MB bins	0.0494	0.0299
Access count – *ln* bins	−0.0014	0.0063
File rank – 2MB bins	0.0144	−0.3593
File rank – *ln* bins	−0.2297	−0.4048

I7: There appears to be no strong correlation between the size of a file and its popularity.

Note that a correlation between size and popularity was found in a Big Data workload from Bing's Dryad [10]. On the other hand, a study on web server

workloads did not find a correlation between these dimensions [15]. A lack of correlation would have negative implications to caching; however, the previous observations on file life span and churn could be used to design effective tiered storage systems.

4.3.6 File extensions

We calculated the percentage of files that have an extension; for this purpose we used the Java RegEx "`\.([^.]*)$`" on the full path, and excluded those files for which (a) the extension had more than 5 characters, and (b) had an extension that consisted of only numbers. Using this approach, we found that $39.48\% - 54.75\%$ (R&D and PROD, respectively) of the files have an extension. Table 4.9 shows the top eight extensions and their percentages. The three most common file extensions in these clusters are bz2, gz and xml (note that bz2 and gz are compression formats supported by Hadoop). Using compression, which provides a trade-off between computation and storage/network resources, is a common pattern in Hadoop clusters. Between and $9.27\% - 31.65\%$ (PROD $-$ R&D) of the bytes stored are compressed; however, this provides only a lower bound on the percentage of the stored bytes that are compressed because Yahoo makes heavy use of Hadoop `SequenceFiles` that may not have an identifiable extension and are—by default in these clusters—compressed.

Table 4.9: Statistics of the most common file extensions, as a % of the total number of files / bytes in the namespace at t_0.

Extension	PROD	R&D
gz	19.00% / 3.80%	11.32% / 8.11%
xml	13.16% / 0.033%	3.28% / 0.003%
bz2	12.18% / 4.29%	18.95% / 15.71%
pig	2.55% / 0.02%	0.24% / 0.007%
dat	1.74% / 0.001%	1.56% / 0.0004%
jar	1.37% / 0.03%	0.24% / 0.007%
proprietary compression	1.28% / 1.42%	2.51% / 7.82%
txt	0.49% / 0.48%	0.18% / 0.017%

I8: With the exception of compressed files and xml files, no other extension is associated with a significant percentage of the stored files or stored bytes.

4.3.7 Percentage of operations

The name node handles the namespace metadata requests, amongst which we have the three operations studied in this Chapter: create, open and delete. Figure 4.12 shows the percentage of these and other operations in the 6-month traces. The most common operation is open ($55\% - 60\%$), followed by listStatus (*ls*); together, they account for the vast majority of the operations ($80\% - 90\%$). Thus, to be able to satisfy the requests in a timely fashion, the processing of these two types of requests should be handled by the name node in the most efficient manner.

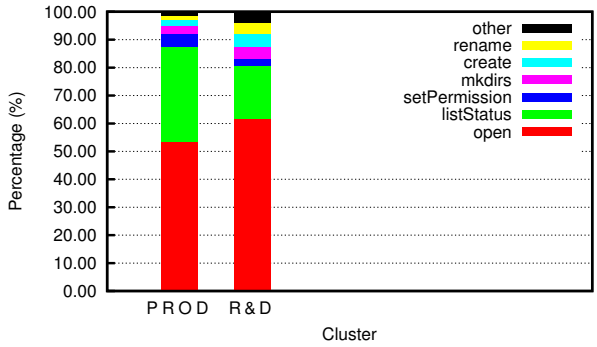


Figure 4.12: Percentage of operation types in the 6-month traces.

I9: The open events account for more than half of the operations issued to the namespace metadata server; open + listStatus, together account for the vast majority of the operations ($80\% - 90\%$*).*

4.4 Related work

The workloads of enterprise storage systems [22], web servers [15] and media server clusters [23] have been extensively studied in the past. Big Data clusters have recently been studied at the job workload level [21, 50], but not at the storage system level. A few recent studies have provided us with some limited insight on the access patterns in MapReduce scenarios [2, 10, 32] but these have been limited to those features of interest to the researchers for their specific projects, like block age at time of access [32] and file popularity [2, 10].

Parallel to this work, other researchers did a large scale characterization of MapReduce workloads, including some insights on data access patterns [20]. Their work concentrates on interactive query workloads and did not study the

batch type of workload that PROD has. Furthermore, the logs they processed were those of the Hadoop scheduler, and for this reason the authors did not have access to information like age of the files in the system, or when a file is deleted.

Perhaps the work most similar to ours (in approach) is that of Cherkasova and Gupta [23], who characterized enterprise media server workloads. An analysis of the influence of new files and file life span was made, but they did not possess file creation and deletion time stamps, so a file is considered to be "new" the first time it is accessed, and its lifetime "ends" the last time it is accessed. In addition, the burstiness of the requests was not analyzed. Their results have been cited in this Chapter where appropriate, to enable us to contrast MapReduce workloads with a more traditional workload.

Our work complements prior research by providing a better understanding of one type of Big Data workload, filling gaps at the storage level. The workload characterization, key observations and implications to storage system design are important contributions. More studies of Big Data storage workloads and their implications should be encouraged so that storage system designers can validate their designs and deployed systems can be properly tuned.

4.5 Discussion

Existing **file popularity** models and metrics tend to assume (to simplify the model) a somewhat static population. While those models are in practice "wrong" (web sites, words in the English language, etc., appear and disappear in time too), they have proven to be useful when the rate of change of the population is not significant and most population members exist during the time-frame being analyzed.

For the case of the workloads studied in this Chapter, the analysis presented in Section 4.3.1 showed how traditional popularity metrics (e.g., percentage of population that accounts for 90% of the frequency counts–in this case, accesses) can be misleading and make it harder to understand what those numbers imply about the popularity of the population (files). In the analysis presented, the problem arose from the high percentage of short-lived (and thus, infrequently accessed) files. New or adapted models and metrics are needed to be able to better express popularity under these conditions.

The **high rate of change in file population** has some interesting implications in the design of the storage systems: does it make sense to handle the short-lived

files in the same way as longer-lived files? Tiered storage systems that combine different types of storage media for different types of files can be tailored to these workloads for improved performance.

While the **burstiness** and autocorrelations in the request arrivals may be a result of typical MapReduce workloads in which multiple tasks are launched within some small time window (all operating on different parts of the same large file or set of related files), a characterization of the autocorrelations is relevant independently of the MapReduce workload that produced them, for the following reasons:

- It allows researchers to reproduce the workload in simulation or real tests without having to use an application workload generator (e.g., Apache GridMix or SWIM [21] for MapReduce). This is useful because current MapReduce workload generators execute MapReduce jobs on a real cluster, which would thus preclude researchers without a large cluster to perform large-scale studies that could otherwise be performed at the simulation level.

- Current MapReduce workload generators (and published models) have overlooked the data access patterns, so their use to evaluate a storage system would be limited.

- Some of the autocorrelations present may also be evident in other Big Data workloads, for example *bag-of-tasks* parallel jobs in High Performance Computing (HPC). If that's the case, our characterization (and future models that could be proposed) could be useful for designers of storage systems targeted at the HPC community[7].

4.6 Summary

In this Chapter, we presented a study of how two large MapReduce clusters interact with the storage layer. These workloads, from two large Hadoop (MapReduce) clusters at Yahoo, have some unique properties that set them apart from previously studied workloads. Their high file churn and skewed access towards young files, among others, should be further studied and modeled to enable designers of next

[7] A discussion on whether it is a good idea to have different storage systems for the Internet services community and for the HPC community is out of the scope of this Thesis. For one particular view on this subject, see [61].

generation file systems to optimize their designs to best meet the requirements of these emerging workloads.

Chapter 5

Generating Request Streams on Big Data using Clustered Renewal Processes

The performance evaluation of large file systems, such as storage and media streaming, motivates scalable generation of representative traces. We focus on two key characteristics of traces, *popularity* and *temporal locality*. The common practice of using a system-wide distribution obscures per-object behavior, which is important for system evaluation. We propose a model based on *delayed renewal processes* which, by sampling interarrival times for each object, accurately reproduces the original popularity and temporal locality. A lightweight version reduces the dimension of the model with statistical clustering. It is workload-agnostic and object type-aware, suitable for testing emerging workloads and 'what-if' scenarios. We implemented a synthetic trace generator and validated it using: (1) a Big Data storage (HDFS) workload from Yahoo, (2) a trace from a feature animation company, and (3) a streaming media workload. Two case studies in caching and replicated distributed storage systems show that our traces produce application-level results similar to the real workload. The trace generator is fast and readily scales to a system of 4.3 million files. It outperforms existing models in terms of accurately reproducing the characteristics of the real trace.

5.1 Motivation

Workload generation is often used in simulations and real experiments to help reveal how a system reacts to variations in the load [14]. Such experiments can be used to validate new designs, find potential bottlenecks, evaluate performance, and do capacity planning based on observed or predicted workloads.

Workload generators can replay real traces or do model-based synthetic workload generation. Real traces capture observed behavior and may even include nonstandard or undiscovered (but possibly important) properties of the load [76]. However, real trace-based approaches treat the workload as a "black box" [14]. Modifying a particular workload parameter or dimension is difficult, making such

approaches inappropriate for sensitivity and what-if analysis. Sharing of traces can be hard because of their size and privacy concerns. Other problems include those of scaling to a different system size and describing and comparing traces in terms that can be understood by implementors [76].

Model-based synthetic workload generation can be used to facilitate testing while modifying a particular dimension of the workload, and can model expected future demands. For that reason, synthetic workload generators have been used extensively to evaluate the performance of storage systems [4, 76], media streaming servers [41, 73], and Web caching systems [14, 16]. Synthetic workload generators can issue requests on a real system [14, 76] or generate synthetic traces that can be used in simulations or replayed on actual systems [73, 86].

In this work, we focus on synthetic generation of *object request streams*[1], which may refer to different object types depending on context, like files [4], disk blocks [76], Web documents [16], and media sessions [73].

Two important characteristics of object request streams are popularity (access counts) and temporal reference locality (a recently accessed object is likely to be accessed in the near future) [73]. While highly popular objects are likely to be accessed again soon, temporal locality can also arise when the interarrival times are highly skewed, even if the object is unpopular [35].

For the purpose of synthetic workload generation, it is desirable to simultaneously reproduce the access counts and the request interarrivals of each individual object, as both of these dimensions can affect system performance. However, single-distribution approaches—which summarize the behavior of different types of objects with a single distribution per dimension—cannot accurately reproduce both at the same time. In particular, the common practice of collapsing the per-object interarrival distributions into a single system-wide distribution (instead of individual per-object distributions) obscures the identity of the object being accessed, thus homogenizing the otherwise distinct per-object behavior [86].

As Big Data applications lead to emerging workloads and these workloads keep growing in scale, the need for workload generators that can scale up the workload and/or facilitate its modification based on predicted behavior is increasingly important.

Motivated by previous observations about Big Data file request streams [5, 20, 32], we set the following goals for our model and synthetic generation process:

[1] Also called *object reference streams*.

- *Support for dynamic object populations*: Most previous models consider static object populations. Several workloads including storage systems supporting MapReduce jobs [5] and media server sessions [73] have dynamic populations with high object churn.

- *Fast generation*: Traces in the Big Data domain can be large (e.g., 1.6 GB for a 1-day trace with millions of objects). A single machine should be able to generate a synthetic trace modeled after the original one without suffering from memory or performance constraints.

- *Type-awareness*: Request streams are composed of accesses to different objects, each of which may have distinct access patterns. We want to reproduce these access patterns.

- *Workload-agnostic*: The Big Data community is creating new workloads (e.g, key-value stores [25], batch and interactive MapReduce jobs [5], etc.). Our model should not make workload-dependent assumptions that may render it unsuitable for emerging workloads.

In this Thesis, we consider a stationary segment[2] of the workload and describe a model based on a set of delayed renewal processes (one per object in the stream) in which the system-wide popularity distribution asymptotically emerges through explicit reproduction of the per-object request arrivals and active span (time during which an object is accessed). However, this model is unscalable, as it is heavy on resources (needs to keep track of millions of objects).

We propose a lightweight version of the model that uses unsupervised statistical clustering to identify groups of objects with similar behavior and significantly reduce the model space by modeling "types of objects" instead of individual objects. As a result, the clustered model is suitable for synthetic generation.

We implemented a synthetic trace generator based on our model, and evaluate it across several dimensions. Using a Big Data storage (HDFS [70]) workload from Yahoo, we validate our approach by demonstrating its ability to approximate the original request interarrivals and popularity distributions (supremum distance between real and synthetic cumulative distribution function, CDF, under 2%). Workloads from other domains were also modeled successfully ($1.3 - 2.6\%$ distance between real and synthetic CDFs). Through a case study in Web caching

[2]Workloads consisting of a few stationary segments can be divided using the approach in [76]; for more details, see Section 5.5.1.

and a case study in the Big Data domain (load in a replicated distributed storage system), we show how our synthetic traces can be used in place of the real traces (results within 5.5 percentage points of the expected or real results), outperforming previous models.

Our model can accommodate for objects appearing and disappearing at any time during the request stream (making it appropriate for workloads with high object churn) and is suitable for synthetic workload generation; experiments show that we can generate a 1-day trace with more than 60 million object requests in under 3 minutes. Furthermore, our assumptions are minimal, since the renewal process theory does not require that the model be fit to a particular interarrival distribution, or to a particular popularity distribution.

Additionally, the use of unsupervised statistical clustering leads to autonomic "type-awareness" that does not depend on expert domain knowledge or introduce human biases. The statistical clustering finds objects with similar behavior, enabling type-aware trace generation, scaling, and "what-if" analysis (e.g., in a storage system, what if the short-lived files were to increase in proportion to the other types of files?)

Concretely, the technical contributions of this Chapter are the following: (1) We present a model based on a set of delayed renewal processes in which the system-wide popularity distribution asymptotically emerges through explicit reproduction of the per-object request interarrivals and active span; (2) we use clustering to build a lightweight clustered variant of the model, suitable for synthetic workload generation; and (3) we show that clustering enables workload-agnostic type-awareness, which can be exploited during scaling, what-if and sensitivity analysis.

In Section 5.2 we describe our system model, and explain why it can approximate an object's access count based on the object's temporal behavior. In Section 5.3, we show how we can use statistical clustering to reduce the model size and still achieve high accuracy in synthetic trace generation. In Section 5.4, we evaluate the effectiveness of our model in producing the same results as the real trace using two case studies in the Web caching and Big Data domains. In Section 5.5 we discuss some benefits and limitations of our model. We discuss related work in Section 5.6. Finally, we provide a summary of this Chapter in Section 5.7.

5.2 Model

Consider an object request stream that accesses n distinct objects $\{O_1, O_2, ..., O_n\}$ during $[0, t_{end}]$. We model the object request stream as a set of *renewal processes* in which each object (or file, in the context of this Thesis) has an independent interarrival distribution. The file population may not be static, and not all files may exist (or be active) at time 0. Thus, we consider the case of *delayed renewal processes*, in which the first arrival is allowed to have a different interarrival time distribution. For a brief summary of renewal processes, see the Appendix.

Our model is defined by $\{F_1, F_2, ..., F_n\}$, $\{G_1, G_2, ..., G_n\}$, and $\{t_1, t_2, ..., t_n\}$[3]. F_i is the interarrival distribution of the accesses to object i or O_i. G_i is the interarrival time distribution of the first access to O_i. t_i is the observed *active span* for O_i, or the difference or period between the first and last accesses to O_i.

Each renewal process is modeled after the behavior of a single object in the request stream. For this reason, the interarrival time for the first access to O_i, given by G_i, has only one possible outcome: the time when the first access to O_i was observed in the original request stream, or T_{i_1}. The model is summarized in Figure 5.1.

Thus, we have one arrival time process T_i for every object O_i in the system. T_i is the partial sum process associated with the independent, identically distributed sequence of interarrival times; F_i is the common distribution function of the interarrival times. A particular O_i's popularity is given by the counting process N_i. The random counting process is the inverse of the arrival time process.

We use an O_i's corresponding t_i to determine when to stop generating arrivals (sampling from F_i), at which point the number of accesses (N_{t_i}) is evaluated.

Figure 5.2 shows the synthetic trace generation algorithm based on our model.

5.2.1 Convergence to real distributions

In this section, we refer to the distributions obtained from the real trace as the *real distributions*. Mainly, we are concerned with reproducing the per-object interarrival distribution and the popularity distribution. In this section, we explain why

[3] In this work, we use real time (in milliseconds). The choice of real time versus virtual time, which is discrete and advances by one unit with each object reference, is typically determined by project goals. Workload generators tend to prefer real time [41, 73], while workload characterization projects favor virtual time [35, 40].

$$M = \{\{F_1, F_2, ..., F_n\}, \{T_{1_1}, T_{2_1}, ..., T_{n_1}\}, \{t_1, t_2, ..., t_n\}, t_{end}\}$$

n: number of objects in request stream
O_i: object i, where $i \in \{1, ..., n\}$
F_i: interarrival distribution for O_i
T_{i_1}: time at which O_i becomes active; i.e., time of 1st access to O_i
t_i: time (since T_{i_1}) at which O_i becomes inactive; i.e., active span
t_{end}: duration of trace (in milliseconds)

Figure 5.1: Statistical parameters that describe our system model.

---

for $i \leftarrow 1$ **to** n **do**
  $t \leftarrow T_{i_1}$
  $span \leftarrow t_i$
  $last \leftarrow t + span$
  while $t \leq t_{end}$ **and** $t \leq last$ **do**
    print t, O_i
    $interarrival \leftarrow$ sample from F_i
    $t \leftarrow t + interarrival$
  end while
end for
sort trace

} a renewal process

Figure 5.2: Synthetic trace generation algorithm.

these distributions asymptotically converge to the real ones.

Per-object interarrival distribution

The real per-object interarrival distribution is obtained by calculating the time between an access to an object and the previous access to that same object, when the number of accesses to the object is greater than 1.

Let $\hat{F}_n(t)$ be the estimator, or empirical distribution function, obtained by sampling from the real distribution $F(t)$:

$$\hat{F}_n(t) = \frac{\#\text{ elements in sample} \leq t}{n} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\{x_i \leq t\} \tag{5.1}$$

By the strong law of large numbers, we have that the estimator $\hat{F}_n(t)$ converges to $F(t)$ with probability one as $n \to \infty$.

Object popularity distribution

From the renewal theorem, we know that the expected number of renewals or arrivals in an interval is asymptotically proportional to the length of the interval: $m(t,t+h] \to h/\mu$ as $t \to \infty$, where μ is the mean interarrival time. Since the distribution of interarrivals comes from the observed empirical distribution, $\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$, where $\sum_{i=1}^{n} x_i$ is the sum of the observed interarrivals, or the observed active span; since we sample only during the observed active span, $\sum_{i=1}^{n} x_i = h$. Thus, we have $m(t,t+h] \to h/(1/n \times h)$ as $t \to \infty$ or $m(t,t+h] \to n$ as $t \to \infty$, where n is the observed number of renewals.

5.2.2 Discussion

The asymptotic behavior of the interarrivals and counting process does not imply that a single run of our algorithm will generate a trace in which each object's behavior is statistically reproduced. A way to reach the asymptotic behavior is to perform a Monte Carlo simulation to ensure that the sequence of interarrivals of an O_i contains the expected number of renewals or requests (h/μ).

However, our experimental results (Section 5.3.5) show that for a large trace with a large number of files, the synthetic distributions can approximate the real ones in a single run.

On the other hand, a problem with the model presented in this section is that it is not scalable, as it needs to keep track of millions of distributions. Next, we propose an approach that uses statistical clustering to reduce the model size, making it suitable for synthetic workload generation.

5.3 Reducing the model size via clustering

In this section, we describe how we can use unsupervised statistical clustering to reduce the state space of our model to a tiny fraction of its original size.

The basic idea is to cluster objects with "similar" behavior so that we only need to keep track of the temporal access patterns of the *cluster*, not those of each object; thus, we reduce the state space, as shown in Figure 5.3. The reduction in size that this approach entails is quantified in Section 5.3.5.

$$M = \{\{F_1, F_2, ..., F_k\}, \{G_1, G_2, ..., G_k\}, \{H_1, H_2, ..., H_k\}, n, \{w_1, w_2, ..., w_k\}, t_{end}\}$$

n: number of objects in request stream
O_i: object i, where $i \in \{1, ..., n\}$
k: number of clusters (object types)
K_j: cluster j, where $j \in \{1, ..., k\}$
F_j: interarrival distribution for accesses to an object in cluster K_j
G_j: interarrival distribution for first access to objects in cluster K_j
H_j: active span distribution $\forall O_i \in K_j$
w_j: percentage of objects in K_j ; $\sum_{i=1}^{k} w_i = 1$
t_{end}: duration of trace (in milliseconds)

Figure 5.3: Statistical parameters that describe the reduced, clustered model; $k \ll n$.

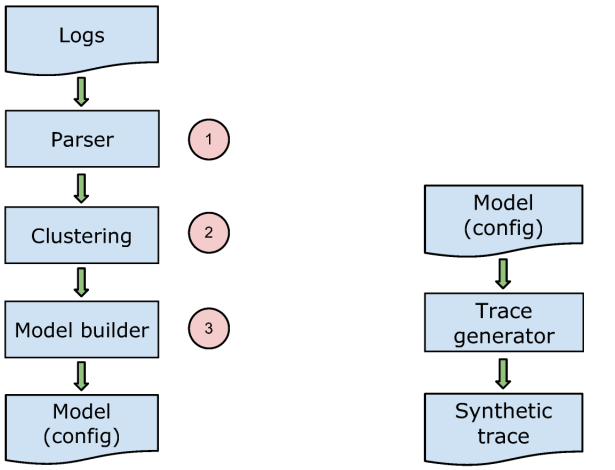
The workflow we describe in this section is as follows (Figure 5.4): (a) Build model, once per source workload; (a.1) Parser: Extracts features for clustering; (a.2) Clustering: Use k-means to find similar objects; (a.3) Model builder: Use data-processing tools, like SQL, and standard techniques to get per-cluster distributions; (b) Generate synthetic trace: Multiple traces can be generated, based on a particular model.

5.3.1 Dataset description

We analyzed a 1-day (Dec. 1, 2011) *namespace metadata trace*[4] from an Apache Hadoop cluster at Yahoo. The trace came from a 4100+ node production cluster running the Hadoop Distributed File System (HDFS) [70][5]. We obtained the trace of namespace events by parsing the metadata server audit logs. For the purpose of this Chapter we analyzed only the open events, which constitute the file request stream. As it came from a Big Data cluster, the 1-day trace is quite big, containing 60.9 million object requests (opens) that target 4.3 million distinct files, and 4 PB of used storage space. Apache Hadoop is an open source implementation of Google's MapReduce [30] framework, used for data-intensive jobs. In prior work, we presented a detailed workload characterization of how the MapReduce

[4]We define a **namespace metadata trace** as a storage system trace that contains a snapshot of the namespace (file and directory hierarchy) and a set of events that operate atop that namespace (e.g., open a file, list directory contents) [4]. These traces can be used to evaluate namespace management systems, including their load balancing, partitioning, and caching components.

[5]For a brief description of the design of HDFS, see Section 5.4.2.



(a) Model building (b) Trace generation

Figure 5.4: Process of how the model is built (left), and how the synthetic traces are generated (right).

jobs operating in this cluster interact with the storage layer [5] (see Chapter 4).

5.3.2 Clustering similar objects

We begin by explaining what "similar behavior" means in the context of this work. The goal is to cluster objects with the same interarrival distribution, so that sampling from one common (per-cluster) distribution reproduces the per-object interarrivals of the objects in the cluster.

We used k-means for the unsupervised statistical clustering. K-means is a well-known clustering algorithm that partitions a dataset into k clusters [53]. Each observation in the dataset is described by several features or dimensions. The output of the k-means algorithm is the center of each cluster, and a list with the cluster identifier to which each of the original observations belongs (the closest center).

We propose the use of two per-object features as input to k-means: skewness and average interarrival time (span/accessCount). We next describe the reasoning behind those choices.

It is not possible to know a priori if the interarrival of an object can be represented by a known distribution. Thus, comparing two interarrival distributions is not as simple as comparing two distribution parameters, but would rather entail comparing the two empirical distributions in some other way. So that k-means can perform that comparison efficiently, it is desirable to summarize each distribution

with as few numbers as possible. We choose **skewness** as a metric that can help describe the shape of the per-object interarrival distribution. We use the *Bowley skewness*, or quartile skewness, as the metric: $(Q_1 - 2Q_2 + Q_3)/(Q_3 - Q_1)$, where Q_i is the i^{th} quartile.

Other ways of describing a statistical distribution, such as quartiles or the five-number summary[6], could be used instead. We did not explore those options, since our experiments show that skewness (combined with the average interarrival time) works well in practice. Furthermore, using multi-value representations of each distribution would require designing a way to find the "distance" between two different multi-value summaries; a task that is nontrivial. In our current implementation, we are using the Euclidean distance on z-score normalized values; this is the default distance metric used by many k-means implementations.

The choice of **average interarrival time** as a feature ensures a strong correlation between the active span or period during which an object is accessed, and the number of accesses in that period.

5.3.3 Synthetic trace generation

The trace generation algorithm is shown in Figure 5.5.

```
for j ← 1 to k do
  for o ← 1 to w_j × n do
    t ← sample first arrival from G_j
    span ← sample active span from H_j
    last ← t + span
    while t ≤ t_end and t ≤ last do
      print t, id(o, j)
      interarrival ← sample from F_j
      t ← t + interarrival
    end while
  end for
end for
sort trace
```

a renewal process

Figure 5.5: Synthetic trace generation algorithm for the clustered model.

[6]Sample minimum, first quartile, median, third quartile, and sample maximum.

The number of accesses generated for O_i, belonging to cluster j, depends on the per-object interarrival distribution for cluster j (F_j) and the active span distribution for the objects in K_j (H_j). Thus, a good clustering should lead to a strong correlation between the span and access count of the objects in the cluster, which is achieved through the choice of average interarrival time (span/accessCount) as one of the features used in the clustering step.

The number of renewal processes used during trace generation remains the same as in the full model: n, or the number of distinct objects accessed in the request stream. Clustering does not reduce the number of renewal processes used during workload generation; it reduces the number of distributions that we need to keep track of, and thus reduces the memory requirements of the workload generation process. It also reduces the storage space used by the model or configuration file.

Scaling

We can increase the number of objects in the synthetic workload by increasing n; the proportion of each type of object (given by w_i) is maintained. Alternatively, we can increase the number of only one particular type of object by increasing n and doing a transformation on the w_i weights. For example, to double the number of objects of type 1 while keeping the number of objects of types 2 to k unchanged, we can obtain the values of n_{new} and $\{w_{1_{\text{new}}}, w_{2_{\text{new}}}, ..., w_{k_{\text{new}}}\}$ as follows:

$$
\begin{aligned}
n_{\text{new}} &= n + w_1 \times n \\
w_{1_{\text{new}}} \times n_{\text{new}} &= 2(w_1 \times n) \\
w_{i_{\text{new}}} \times n_{\text{new}} &= w_i \times n, \quad i \in \{2, ..., k\}
\end{aligned}
$$

5.3.4 Optimization: Fitting the tail of the popularity distribution

The model described so far is successful at approximating the popularity distribution (see Section 5.3.5). However, it is not able to reproduce the tail of the distribution with the most popular objects.

A common characteristic of popularity distributions is a small percentage of highly popular objects (e.g., Zipfian popularity distributions) [5, 15, 23, 28].

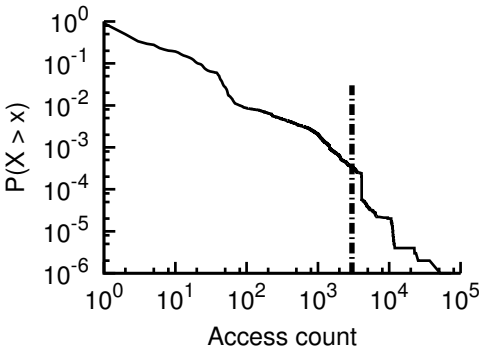


Figure 5.6: Complementary CDF of the popularity distribution. The CCDF highlights the tail of very popular objects. The vertical dashed line shows the (heuristical) beginning of the tail.

These highly popular objects are too few, and their access counts too different from each other, for their asymptotic popularity to be reproduced in one single run of our generation process (see ragged tail in Figure 5.6). While some systems may not be sensitive to the tail of highly popular objects (e.g., Web caching, discussed in Section 5.4.1), other systems are (e.g., replicated storage, discussed in Section 5.4.2). To address this problem, we define the following approach and heuristic.

Optimization

As an optimization, we propose using the full model instead of the clustered model for the highly popular objects located at the tail of the popularity distribution. Before generating the interarrivals for an object in the tail, determine the expected number of arrivals for the object during its span (see Section 5.2.1). Then, sample sets of arrivals until we find one set whose number of arrivals is within a small %, δ, of the expected number of arrivals. Use that set as the sampled interarrivals for that object, thus simultaneously reproducing the object's access counts, interarrivals, and span.

In our current implementation we keep sampling sets of interarrivals until we find one within $\delta = 0.5\%$ of the expected number of arrivals for that object.

Heuristic

We define the beginning of the tail of highly popular objects as the position where: (1) access counts are held by single files, and (2) distribution has sparse access counts. For our trace, we defined the tail to begin at the 1478^{th} to last file in

our trace. Only one file had 3000 accesses; one file had 3001 accesses; and one file had 3010 accesses. No file was accessed 3002 to 3009 times. This is quite noticeable in Figure 5.6, where the tail of the complementary CDF is ragged and not smooth.

5.3.5 Experimental validation

We implemented the components described in Figure 5.4 as follows: (a.1) Apache Pig and bash scripts; (a.2) R using the fpc package [38]; (a.3) Apache Pig and bash scripts; and, (b.1) Java.

We ran experiments for increasingly larger numbers of clusters and discuss the results in this section. Unless otherwise noted, the optimization to fit the tail was not used in the results shown in this section.

Figure 5.7a shows the Pearson's correlation coefficient between the span and the access count of the objects in the cluster, averaged across all clusters. We can observe that increasing the number of clusters leads to a higher correlation.

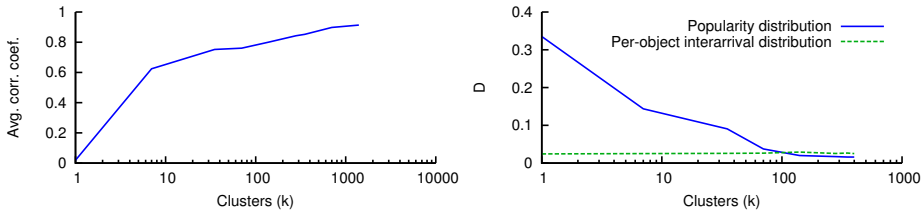
Figure 5.7b shows how the popularity and per-object interarrival distributions approximate the real distributions. We evaluate the closeness of the approximation using the Kolmogorov-Smirnov distance, D, which is the greatest (supremum) distance between the two CDFs.

By comparing Figures 5.7a and 5.7b, we can appreciate the usefulness of the correlation coefficient between span and access count as a metric to evaluate whether the clustering results will be useful for synthetic trace generation. In our experiments, an average correlation coefficient of 0.76 or higher ($k > 70$) enables us to approximate the popularity distribution within 3% of the real one, and a correlation of 0.8 or higher ($k = 140$) leads to an approximation within 2% of the real CDF.

Figure 5.8 provides a visual confirmation of the effectiveness of our approach in approximating the real popularity distribution as the number of clusters increases.

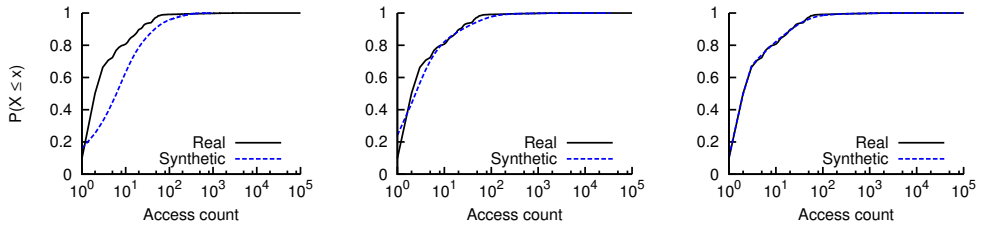
There is a trade-off between approximating the popularity distribution and the size of the model. The extreme cases are when $k = 1$ and when no clustering is performed ($k = n$): we need to keep track of three distributions ($k = 1$)[7] vs.

[7] F_1, G_1, and H_1.



(a) Correlation coefficient (b) D

Figure 5.7: *Left*: Pearson's correlation coefficient between object popularity (access count) and span, for all objects in a cluster, averaged across clusters. *Right*: Kolmogorov-Smirnov distance (D), comparing the real distribution to the synthetic one. D converges to 0 if the sample comes from the target distribution.



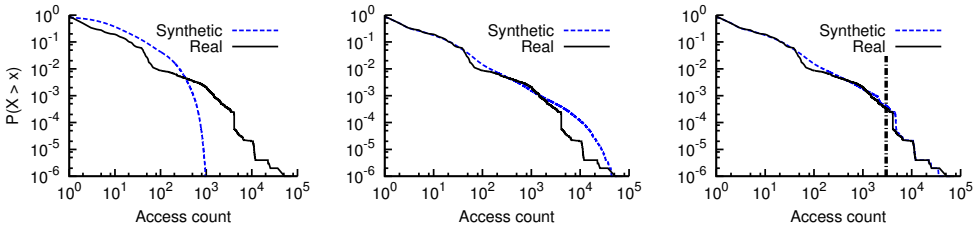
(a) $k = 1$ (b) $k = 7$ (c) $k = 400$

Figure 5.8: Cumulative distribution functions (CDFs) of the real and synthetic popularity distributions, when using 1, 7, and 400 clusters. More clusters leads to a better approximation of the popularity distribution.

millions of distributions ($k = n$). In our experiments, using 70 to 400 clusters led to results that approximate the popularity distribution well (with a small D value and overlapping CDFs). For our dataset, when $k = 400$, the configuration file can be as small as 24KB if the distributions are fitted to known distributions and only their parameters are stored, or as big as 190MB if the distributions are stored as uncompressed empirical histograms (0.0001 precision) or 20MB if compressed.

We now evaluate the effectiveness of the technique to fit the tail of the popularity distribution. Figures 5.9a and 5.9b show how the basic clustering approach is not able to match the tail of the distribution, even as k increases. Figure 5.9c shows that we can approximate the tail of the popularity distribution by modeling the top 1478 files using the full model (described in Section 5.2) instead of the clustered model (described in Section 5.3). Our simulation results, described in Section 5.4, show that the current heuristic provides a good approximation of the real workload.

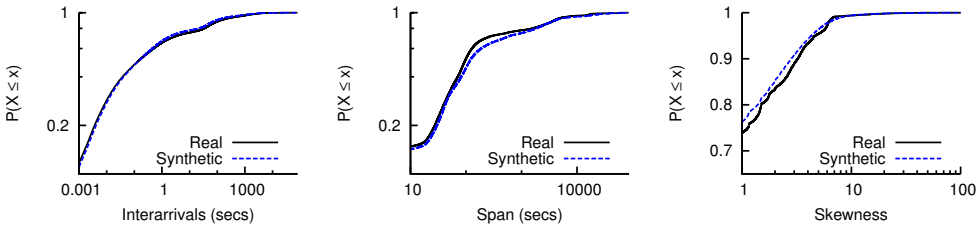
Finally, in Figure 5.10, we present a visual confirmation of how we match the



(a) $k = 1$ (b) $k = 400$ (c) $k = 350$ w/ TF

Figure 5.9: Complementary CDF of the popularity distribution, when using 1 cluster, 400 clusters, and 350 clusters with the tail-fitting (TF) technique. The CCDF highlights the tail of very popular objects. The vertical dashed line in (c) shows the beginning of the fitted tail.

interarrivals, span, and skewness (of the per-object interarrival distributions). The results shown are for the case of $k = 400$. As expected, the distributions closely match the real ones.



(a) Interarrivals (b) Span (c) Skewness

Figure 5.10: Real and synthetic ($k = 400$) per-object interarrival, span and skewness CDFs.

In addition to the detailed validation performed with the HDFS trace, we used two other traces to validate our model: ANIM and MULTIMEDIA. ANIM is a 24-hour NFS trace from a feature animation company supporting rendering objects, obtained in 2007 (Set 0, available for download at `iotta.snia.org`) [12]. MULTIMEDIA is a one-month trace generated using the Medisyn streaming media service workload generator from HP Labs [73][8]. These experiments also produced a close approximation of the popularity distribution: $D = 0.0263$ (ANIM, $k = 500$) and $D = 0.0134$ ($k = 140$).

[8]We used the default configuration shipped with Medisyn, with the following changes to generate a large trace: PopularityMaxFreq=50000, PopularityNumOfFiles=1000000, PopularityZipfAlpha=1.61, LifespanclassTrivialClassDistParamUnifb=30

Running time

On our test system with 12GB RAM and 2.33GHz per core (8 cores) running Java code, the synthetic trace generator takes an average of 113 seconds (std. dev. = 22.01) to generate an unsorted trace (without tail fitting) and an average of 14 minutes with tail fitting. Note that this implementation can be further improved by using threads so that all the cores are utilized. The current implementation uses only two threads: one for introducing new files and one for generating the accesses to the files. The step of sorting the sub-traces with a merge sort takes 6.5 minutes in our testing environment.

5.3.6 Choosing k

Our results show that increasing the number of clusters, k, leads to a better approximation of the popularity distribution. Our current approach to choose k is as follows:

Step 1 Find a small k for which k-means yields good clustering results. The quality of the clustering can be evaluated using standard techniques, such as cluster silhouette widths or the Calinski and Harabasz index (both of which are included with the fpc package used in our implementation [38]).

Step 2 Use increasingly larger values of k until we find a clustering that is good for the purpose of synthetic workload generation. To determine whether the clustering is useful, we use Pearson's correlation coefficient between the span and the access count of the objects in the cluster. Based on our experimental results, we use the following heuristic: an average correlation value of 0.8 or higher is good for workload generation.

5.4 Evaluation

In this section, we use two case studies to evaluate how well the synthetic workloads produced by our model can be used in place of the real workload. We first present a case study in Web caching, because it is a well understood and studied problem. Both the file popularity and the short-term temporal correlations have an impact on the performance of a Web cache. We then present a case study from the Big Data domain: an analysis of unbalanced accesses in a replicated distributed storage system supporting MapReduce jobs.

5.4.1 Case study: Web caching

In this section, we compare the cache miss rates of a server-side Web cache. We use trace-based simulations with the following 24-hour traces:

- *Real*: The original trace, which we want to approximate as closely as possible.
- *Independent Reference Model (IRM)*: Assumes that object references are statistically independent. We obtained this trace by creating a random permutation of the real trace. It is equivalent to a sampling from the popularity distribution.
- *H-IRM*: Obtained by permutating the requests within hourly chunks. Active span of a file can be off by at most one hour (vs 24 hours in IRM case).
- *Interarrivals*: Obtained by sampling from a global interarrival distribution, modeled after the per-file interarrivals of the real trace. This is akin to using our model when $k = 1$.
- *Synthetic*: Obtained using the process described in Figure 5.5, $k = 400$; the tail was not fitted.
- *Synthetic TF*: Generated using $k = 350$ and the tail-fitting technique.

The accesses and popularity come from the trace studied earlier in this Chapter. File sizes are based on the January 2013 page view statistics for Wikimedia projects [56], and matched to the trace based on popularity (i.e., any correlation between size and popularity was preserved). The file sizes are used only to compute the byte miss rate of the cache (not the file miss rates).

We used the simulator developed by P. Cao [18] to simulate a server-side Web cache and evaluated it with the following cache replacement policies: *Least recently used (LRU)*: Evicts the file whose access was furthest in the past; *Size*: Evicts the largest file; and, *Lowest relative value (LRV)*: Calculates the utility of keeping a document in the cache using locality, cost, and size information; evicts the document with the lowest value.

Figure 5.11 shows that the results obtained using our synthetic trace are almost indistinguishable from the miss rates obtained using the real trace. We quantify the difference or error between the synthetic and expected (real) results using the root

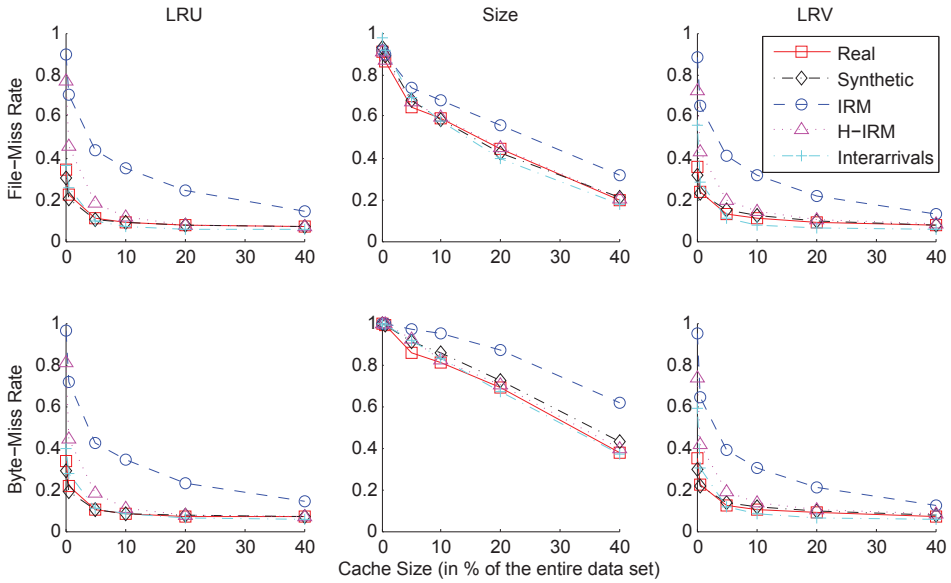


Figure 5.11: File and byte cache miss rates for different cache replacement strategies. The synthetic trace generated with our model yields results that deviate very little from the ones using the real trace.

Table 5.1: Root mean squared error (RMSE) for Figure 5.11. The synthetic traces generated by our model (labeled *Synthetic* and *Synthetic TF*) produce a very small RMSE, thus approximating the real results well. Bolded values indicate results very close to the real workload (RMSE < 0.05).

		Synthetic	Synthetic TF	IRM	H-IRM	Interarrival
LRU	File miss rate	**0.018**	**0.019**	0.352	0.202	**0.021**
	Byte miss rate	**0.021**	**0.018**	0.375	0.216	**0.037**
Size	File miss rate	**0.023**	**0.041**	0.086	**0.013**	**0.045**
	Byte miss rate	**0.039**	**0.035**	0.145	**0.030**	**0.027**
LRV	File miss rate	**0.018**	**0.019**	0.312	0.172	0.085
	Byte miss rate	**0.022**	**0.020**	0.335	0.180	0.106

mean squared error (RMSE), shown in Table 5.1. Note that the results obtained using the *Synthetic TF* trace are not plotted in Figure 5.11, because they are very close to the *Synthetic* results, and including them in the graph made it hard to differentiate the lines; the corresponding RMSE is shown in Table 5.1. The small difference between the results of the *Synthetic* and *Synthetic TF* workloads shows that this experiment is not too sensitive to the tail of popular objects.

Our results show that using our synthetic trace yields results comparable to those of the real trace. Furthermore, the low RMSE achieved for all caching strategies outperformed other alternatives. For one experiment (Size eviction),

the results using *H-IRM* led to an RMSE smaller than the one resulting from our synthetic trace. However, that same trace (*H-IRM*) performed poorly for the LRU and LRV eviction policies.

Note that, just reproducing per-object interarrivals (*Interarrival* trace) led to significantly better results than the independent reference model did (i.e., reproducing only object popularity). As we explained earlier, the *Interarrival* trace was obtained with our model when $k = 1$, and produced a popularity distribution significantly different from the real distribution (see Figure 5.9a). Clearly, concentrating on reproducing the popularity distribution of object request streams while leaving the per-object interarrival distribution as a lower priority is not acceptable if we want to use synthetically generated workloads in place of real ones.

5.4.2 Case study: Replicated storage system

We present a case study on the type of replicated clustered storage systems that are commonly used in MapReduce clusters (e.g., the Google File System [37] or the Hadoop Distributed File System (HDFS) [70]). In this section, we'll use specific details related to HDFS, but the general design is present in several systems supporting Big Data applications.

HDFS was designed to run on commodity machines (datanodes), whose disks may fail frequently. To prevent from data unavailability due to hardware failures, these clusters replicate each block of data across several datanodes. Files are divided into fixed-size blocks (of 128 MB by default), and each block is replicated thrice by default. The block sizes and replication factors are configurable per file.

HDFS's default replica placement is as follows. The first replica of a block goes to the node writing the data; the second, to a random node in the same rack; and the last, to a random node. This design provides a good balance between being insensitive to correlated failures (e.g., whole rack failure) and minimizing the inter-rack data transmission. A block is read from the closest node: node local, or rack local, or remote.

However, this replica placement policy combined with the fact that the files have nonuniform popularity, can lead to hotspots, and some nodes can become overwhelmed by data read requests. In another project [17], we are exploring the replica number computation problem and have implemented an HDFS replica placement simulator for that purpose. We use that simulator in this section.

Our trace-based simulator currently implements the default replica placement

policy described above. Our goal in this case study is to see if our synthetic traces produce the same unbalanced accesses across nodes as the real trace. We keep track of the number of requests that each node in the cluster receives during the simulation, and use the coefficient of variation ($c_v = \sigma/\mu$, often expressed as a percentage) as a measure of dispersion to quantify the imbalance in the per-node requests. A low coefficient of variation means that the per-node requests are balanced across the nodes in the cluster.

We ran experiments using the *Real*, *IRM*, *Interarrivals*, *Synthetic*, and *Synthetic TF* traces, with one change: we made the simplification of using single-block files. From our traces, we know that 90% of the files have only one block, and we found no correlation between file size and popularity.

Note that HDFS deployments run a daily rebalancing process that redistributes the blocks between the nodes to balance the used storage space. The traces used in this experiment were 24-hour traces, so we make the assumption that blocks do not migrate between nodes during the simulation. Additionally, nodes do not fail during the simulation. The simulated cluster has 4096 nodes.

Figure 5.12 shows the results of our experiments, averaged across 5 runs. This experiment is sensitive to popularity and not temporal correlations; thus, *IRM* is able to replicate the results of the *Real* trace, while the *Interarrivals* trace differs significantly from the *Real* results. The *Synthetic* trace produced a better approximation; however, the results still differ significantly from the *Real* ones due to the limitations in reproducing the tail of the popularity distribution. The synthetic trace with the tail-fitting technique produces a closer approximation (difference of 5.5 percentage points), thus showing that it can be used to assess the large imbalance in the number of requests received per node.

5.5 Discussion

In this section, we discuss some benefits and limitations of our model.

5.5.1 Non-stationary workloads

In this Chapter, we made no attempt to capture workloads with time-varying behavior. Object request streams typically exhibit non-stationary workloads [76, 86]. The implicit assumption in our model is that the workload in the trace is sta-

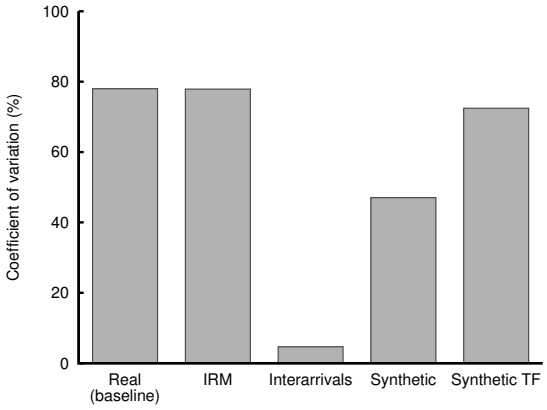


Figure 5.12: Coefficient of variation for the number of requests received per node, averaged across 5 runs. Standard error not shown because values are very small (between $0.1 - 0.25$ percentage points for all traces). The *Synthetic TF* trace produces an approximation within 5.5 percentage points of the real results.

tionary during the length of the trace. Our model can thus be used to capture the behavior of sample periods in a longer and non-stationary workload (e.g., during busiest periods [86]). Alternatively, it could be combined with the model proposed by Tarasov et al. [76], which deals with non-stationary workloads by dividing a time-varying workload into smaller stationary subtraces and extracts the statistical parameters of each subtrace, applying compression techniques to reduce the model size.

5.5.2 Object type-awareness

Our approach of using clustering to group objects according to their statistical behavior naturally leads to type-aware synthetic workload generation and analysis. Furthermore, these "types" of objects are identified without human intervention. Users do not need previous domain knowledge in order to identify the object types, nor can the process be biased by preexisting misconceptions about the workload.

To illustrate this issue, we provide the results of the clustering of the files in our trace into 7 clusters. Table 5.2 shows our results. For each cluster, we show the number of files in that cluster, the average span, access count, average interarrival time, skewness (of the per-file interarrival distributions), and a brief description of the "type" of files in the cluster, including real examples of files that belong to those clusters. We can observe that each cluster's averages are clearly distinct from the averages of the other clusters.

Most of the files (69%) belonged to cluster 6. They were very short lived[9] and unpopular, and had around one access per minute. The skewness of these files was the closest to zero, which means that the per-file interarrival distribution is uniformly distributed. A common type of file observed was MapReduce output files; which is to be expected to be the most common type of file in a MapReduce cluster.

Cluster 3 had the fewest files, but these files were very popular and frequently accessed, with a highly skewed interarrival distribution. Files commonly observed in this cluster included data schemas and Apache Oozie libraries[10].

In the second smallest cluster (cluster 5), the files were very long-lived and very unpopular. Each had fewer than 3 accesses during its long lifetime (28606 seconds, on average). MapReduce job instance configuration files were commonly observed in this cluster. Each file was accessed at the beginning of the job and again later on, to archive them or generate cluster-wide job statistics.

The clusters were all quite different; this suggests that the types of files are limited, though the original trace is huge. If files are properly clustered, we can reduce the model space without losing much of the accuracy of the per-file characteristics.

5.5.3 What-if analysis and scaling

The object type-awareness facilitates "what-if" analysis. By scaling up or down the workload of one type of object (cluster) while leaving the other clusters unaltered, we could experimentally find the answers to questions like "How would the performance of my load balancer be affected if the number of short-lived files were to increase in proportion to the other types of files?"

For a more concrete example, consider the workload studied in this Chapter: a storage cluster (HDFS) supporting MapReduce jobs. Consider a company X that is currently evaluating the possibility of changing its policies regarding the recommended number of reduce tasks per job, so that jobs will now, on average, have more reduce tasks and can process more data in parallel. However, the new policy would lead to an increase in the number of MapReduce output files (since each reducer task writes its output to an individual file). From Table 5.2, we

[9] In the context of this Chapter, the life of a file is determined by its span, not by its actual creation and deletion stamps.

[10] Apache Oozie is a workflow manager for MapReduce jobs.

Table 5.2: Clustering results when $k = 7$. Each cluster exhibits distinct characteristics. Bolded labels indicate the features used during the clustering process. Time units are in seconds. The values for Span, Access count, Average Interarrival Time (AIT) and Skew are averages for all objects in cluster.

Cluster	# files	Span	Access count	**AIT**	**Skew**
1	197378	488.31	84.26	8.16	6.54
2	548739	8975.14	9.74	875.77	0.96
3	12725	6736.17	1645.84	4.36	26.12
4	517853	4147.81	19.85	234.65	3.08
5	27783	28606.10	2.66	10863.38	0.21
6	2968676	182.64	2.58	59.92	0.07
7	53649	26861.18	5.60	5190.79	0.98

Cluster	Description and sample files
1	Short-lived, popular, frequently accessed, skewed Sample: 5-minute data feeds; MapReduce job.jar
2	Unpopular, rarely accessed Sample: Newly loaded data files
3	Very popular, frequently accessed, highly skewed Sample: Data schemas; Apache Oozie libraries
4	Median Sample: MapReduce job tokens
5	Long-lived, very unpopular, very rarely accessed Sample: MapReduce job instance configuration
6	Most common, short-lived, unpopular, unskewed Sample: MapReduce output
7	Long-lived, unpopular, rarely accessed Sample: Archived data

know that these files typically belong to cluster 6, which constitutes 69% of the files requested at least once during the 1-day trace. While increasing the per-job number of reducers could lead to jobs finishing faster, it could potentially slow things down if the namespace metadata server (called the name node in HDFS) were to become overloaded with metadata requests. A trace-based simulation or real experiment could be performed using a modified type-aware workload to answer that question.

5.5.4 Other dimensions of request streams

Our current model does not include other dimensions of request streams that may be domain-specific, such as, file size, session duration, and file encoding characteristics. We believe our model can be extended to include some of those dimensions.

As a simple example, consider the issue of file size. Previous studies have

found file sizes were uncorrelated with access patterns [73], whereas other workloads exhibit an inverse correlation between size and popularity [27]. For the case of uncorrelated size and popularity, including the size dimension is as simple as randomly sampling from the size distribution to assign a size to a file. For the case when a correlation is found, we believe the feature can be added as one more dimension to the clustering step, so that natural object types are found and their behavior reproduced. Alternatively, the problem can be treated as a matching problem and solved using existing approaches [14]. Exploration of this idea is left for future work.

5.5.5 Real-time workload generation

To validate our model, we implemented a synthetic trace generator and used the traces to do trace-based simulations. In addition to simulation-based experiments, it may be desirable to do testing on actual systems. A possible way to do this, is to replay a synthetic trace on a real system by having a client machine issue the requests in the trace at the time indicated in the trace.

However, replaying large synthetic traces in real time is not trivial. It is possible that a single client computer cannot issue the requests in the trace fast enough, and not stress the system in the desired way.

Our algorithm can be adapted to do real-time workload generation by dividing the task of generating the workload (and issuing the corresponding object requests) between multiple clients, and assigning a set of O_is to each client. A workload generating client can have one thread per each renewal process that it is in charge of. In Chapter 6 we show how it is possible to use our model as the basis of a benchmarking tool that issues operations on a real storage system.

5.6 Related work

Several synthetic workload generators have been proposed for other types of object request streams, like Web request streams and media server sessions.

Web request streams [14, 16]. SURGE [14] generates references to Web documents, matching empirical references of popularity and temporal locality, while reproducing burstiness using an ON-OFF process. The temporal locality of requests is modeled using a *stack distance* model of references, which assumes that each file is introduced at the start of the trace. That approach is suitable for static

file populations, but inadequate for populations with high file churn [73]. Our approach considers files with delayed introduction.

ProWGen [16] was developed to enable investigation of the sensitivity of Web proxy cache replacement policies to three workload characteristics: the slope of the Zipf-like document popularity distribution, the degree of temporal locality in the document request stream, and the correlation (if any) between document size and popularity. Instead of attempting to accurately reproduce real workloads, ProWGen's goal is to allow the generation of workloads that differ in one chosen characteristic at a time, thus enabling sensitivity analysis of the differing characteristic. Additionally, through domain knowledge on Web request streams, the authors note that a commonly observed workload is that of "one-timers," or files accessed only once in the request stream. One-timers are singled out as a special type of file whose numbers the user should be able to increase or decrease in relation to other files. In contrast, we were able to approximate the percentage of one-timers in the HDFS workload without explicitly modeling them (9.79% of the real workload vs. 10.69% of our synthetic trace, when $k = 400$).

GISMO [41] and MediSyn [73] model and reproduce media server sessions, including their arrival patterns and per-session characteristics. For session arrivals, both generators have the primary goal of reproducing the file popularity, and distributing the accesses throughout the day based on observed diurnal or seasonal patterns (e.g., percentage of accesses to a file that occur during a specific time slot). Additionally, MediSyn [73] uses a file introduction process to model accesses to new files, and explicitly considers two types of files that differ in their access patterns: regular files and news-like files. Our work allows the synthetic workload generation of objects with different types of behavior without prior domain knowledge.

In earlier work, we developed Mimesis [4], a synthetic workload generator for namespace metadata traces. While Mimesis is able to generate traces that mimic the original workload with respect to the statistical parameters included with it (arrivals, file creations and deletions, and age at time of access), reproducing the file popularity was left for future work.

Chen et al. [22] proposed the use of multi-dimensional statistical correlation (k-means) to obtain storage system access patterns and design insights in user, application, file, and directory levels. However, the clustering was not leveraged for synthetic workload generation.

Hong et al. [39] used clustering to identify representative trace segments to be

used for synthetic trace reconstruction, thus achieving trace compression ratios of 75% to 90%. However, the process of fitting trace segments, instead of individual files based on their behavior, does not facilitate deeper understanding of the behavior of the objects in the workload, nor does it enable what-if or sensitivity analysis.

Ware et al. [86] proposed the use of two-level arrival processes to model bursty accesses in file system workloads. In their implementation, objects are files, and accesses are any system calls issued on that file (e.g., read, write, lookup, or create). Their model uses three independent per-file distributions: interarrivals to bursts of accesses, intra-burst interarrival times, and distribution of burst lengths. A two-level synthetic generation process (in which burst arrivals are the first level, and intra-burst accesses to an object are the second level) is used to reproduce bursts of accesses to a single file. However, the authors do not distinguish between the access to the first burst and the accesses to subsequent bursts, so their model cannot capture file churn. Additionally, the authors use one-dimensional hierarchical clustering to identify bursts of accesses in a trace of per-file accesses. The trace generation process is similar to ours: one arrival process per file. However, the size of the systems they modeled (top $\sim$567 files out 8000 total) did not require a mechanism to reduce the model size. We are considering systems two orders of magnitude larger, so a mechanism to reduce the model size was necessary. The approach of modeling intra-burst arrivals independently of inter-burst arrivals can be combined with our delayed first arrival plus clustering of similar objects approach to capture per-file burstiness.

5.7 Summary

We presented a model for analyzing and synthetically generating object request streams. The model is based on a set of delayed renewal processes, where each process represents one object in the original request stream. Each process in the model has its own request interarrival distribution, which combined with the time of the first access to the object plus the period during which requests to the object are issued, can be used to approximate the number of arrivals or renewals observed in the original trace.

We also proposed a lightweight version of the model that uses unsupervised statistical clustering to significantly reduce the number of interarrival distributions

that we need to keep track of, thus making the model suitable for synthetic trace generation.

We showed how our model is able to produce synthetic traces that approximate the original interarrival, popularity and span distributions within 2% of the original CDFs. Through two case studies, we showed that the synthetic traces generated by our model can be used in place of the original workload and produce results that approximate the expected (real) results.

Our model is suitable for request streams with a large number of objects and a dynamic object population. Furthermore, the statistical clustering enables autonomic type-aware trace generation, thus facilitating sensitivity and "what-if" analysis.

Chapter 6

A Model-Based Namespace Metadata Benchmark for HDFS

Efficient namespace metadata management is increasingly important as next-generation storage systems are designed for peta and exascales. New schemes have been proposed; however, their evaluation has been insufficient due to a lack of an appropriate namespace metadata benchmark. In this Chapter, we describe MimesisBench, a novel namespace metadata benchmark for next-generation storage systems, and demonstrate its usefulness through a study of the scalability and performance of the Hadoop Distributed File System (HDFS).

6.1 Motivation

Currently, there are no metadata-heavy benchmarks based on realistic workloads for next-generation storage systems. A few existing tools, like DFSIO [81], S-live [68], and NNBench [58] are useful as microbenchmarks and for stress testing, but do not reproduce realistic workloads.

The storage community has long-acknowledged the need for benchmarks based on realistic workloads, and systems like SPECsfs2008 [71] and FileBench [34] are extensively used for this purpose in traditional enterprise storage systems. However, emerging Big Data workloads like MapReduce [5] have not been properly synthesized yet.

We present MimesisBench, a novel metadata-intensive storage benchmark suitable for Big Data workloads. MimesisBench consists of a workload-generating software and a set of workloads from two Yahoo Big Data clusters, plus a modeling tool so that future workloads can be easily added in the future. Yahoo is currently using this benchmark to undertake performance and scalability tests on the HDFS name node.

MimesisBench is extensible and more workloads can be added in the future. The model on which MimesisBench is based [6] allows it to generate type-aware workloads, in which specific type of file behavior can be isolated or modified for

'what-if' and sensitivity analysis.

In addition, MimesisBench's current Hadoop-based implementation allows it to be used in any storage system that is compatible with Hadoop (e.g., HDFS, Ceph, CassandraFS, Lustre). We have released the benchmark and workloads as open source so that other researchers can benefit from it[1].

In this Chapter we describe MimesisBench, and report the results of a scalability and performance study of HDFS. The salient features of MimesisBench are:

- Model-based approach allows realistic workload generation.
- MapReduce implementation allows it to be used with any storage system that is able to interface with Hadoop.
- Multiple workloads available in current release.
- Support for multi-dimensional workload scaling.
- Autonomic type-aware model allows sensitivity and 'what-if' analysis.

This Chapter makes two contributions. First, we extend a model for temporal locality and popularity in object request streams [6] to include other operations (create, delete, and listStatus) in addition to regular accesses to objects (opens), to support pre-existing files (created before the benchmark), and to support a realistic hierarchical namespace. Second, we use this model to implement a metadata-intensive storage benchmark to issue realistic workloads on distributed storage systems. This benchmark can be used to evaluate the performance of storage systems without having to deploy a large cluster and its applications.

This Chapter is structured as follows. In Section 6.2 we describe the statistical model used by MimesisBench. In Section 6.3 we describe the design of MimesisBench, including details on how the multi-dimensional workload scaling of MimesisBench can be used to test specific scenarios. We present the results of a performance and scalability study of the HDFS name node in Section 6.4. The related work is discussed in Section 6.5. Finally, in Section 6.6 we provide a summary of the Chapter.

[1] Available at `http://sites.google.com/site/cristinaabad`

6.2 Model

We extend a model we proposed for generating accesses to objects (e.g., opens to files) [6] (see Chapter 5), which is able to reproduce temporal correlations in object request streams that arise from the long-term popularity of the objects and their short-term temporal correlations. This model is suitable for Big Data workloads because it supports highly dynamic populations, it is fast and scalable to millions of objects, and it is workload-agnostic so it can be used to model emerging workloads.

Objects in a stationary segment of a request stream[2] are modeled as a set of delayed renewal processes (one per object in the stream). Each object in the stream is characterized by its time of first access, its access interarrival distribution, and its active span (time during which an object is accessed). With this approach, the system-wide popularity distribution asymptotically emerges through explicit reproduction of the per-object request arrivals and active span [6]. However, this model is unscalable, as it is heavy on resources (needs to keep track of millions of objects).

To reduce the model size, a lightweight version uses unsupervised statistical clustering (k-means) to identify groups of objects with similar behavior and significantly reduce the model space by modeling "types of objects" instead of individual objects. As a result, the clustered model is suitable for synthetic workload generation. Table 6.1 lists the parameters that are required and reproduced by this model.

[2] *Object request streams*, also called *object reference streams*, may refer to different object types depending on context, like files [4], disk blocks [76], Web documents [16], and media sessions [73]. For example, a trace of accesses to files in a storage system. See Section 6.2.2 for a brief discussion on the implications of considering a stationary segment of the workload.

Table 6.1: Parameters used by the model described in Chapter 6.

Symbol	Description
n	number of files in the trace or request stream
O_i	object/file i, where $i \in \{1,...,n\}$
k	number of clusters or types of files
K_j	cluster j, where $j \in \{1,...,k\}$
F_j	interarrival distribution for accesses to an object in cluster K_j
G_j	interarrival distribution for first access to objects in cluster K_j
H_j	active span distribution $\forall O_i \in K_j$
w_j	percentage of objects in K_j ; $\sum_{i=1}^{k} w_i = 1$
t_{end}	duration of request stream (in milliseconds)

6.2.1 Extensions to the model

The model described in the previous subsection cannot be used directly to test a storage system since it only reproduces accesses (e.g., opens) to an object (i.e., file) and not other operations that are also critical in a storage system like creates and deletes. We propose the following extensions to the original model to make it suitable for namespace metadata benchmarking: (1) support for additional storage system operations, (2) support for pre-existing files, and (3) support for realistic hierarchical namespaces.

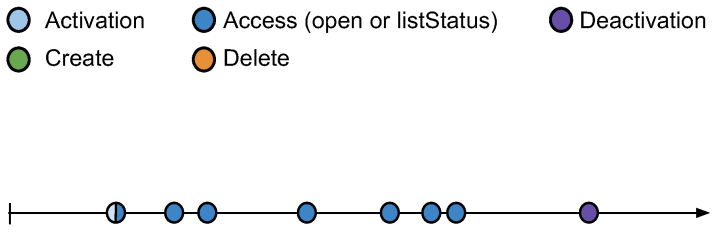
Extension 1: Support for additional storage system operations

We first extend our model to include file creations, deletions and listStatus operations in addition to regular opens. We focus in this set of operations, because together they constitute more than 95% of the operations in MapReduce clusters [5], thus accounting for the vast majority of the workload.

The **listStatus** operations are like opens and only *read* or access the namespace. Thus, we model both listStatus and opens together as *accesses* to files. An additional configuration parameter is added to keep track of the percentage of read operations that constitute opens and those that are listStatus.

On the other hand, **creates** and **deletes** are operations that *write* to (or modify) the namespace. To model these operations, we add additional per-cluster statistical distributions for the creates and deletes.

Figure 6.1 shows how the original and extended models generate operations on a file of a particular type (cluster). The extended model requires the following



(a) Original model

(b) Extended model

Figure 6.1: Operations on a single file, with the original and the extended model. The file's activation time is given by its creation time plus the delay to its first access; the deactivation time is given by its activation time plus its active span.

additional per-cluster statistical information: distribution of create interarrivals, distribution of delay to deletion (relative to the object's last access in the stream, which is given by the time of first access + active span), and percentage of accesses that are open operations (listStatus are the remaining percentage). In addition, the activation time is relative to the creation stamp of a file (or to the beginning of the test, for files that already exist at t_0).

Table 6.2 lists the parameters required and reproduced by the extended model to generate events to files.

The extension to add more operations to the model does not affect the access patterns preserved by the original model, which are characterized by the per-cluster access interarrival distribution and the per-cluster distribution of active spans.

Extension 2: Support for pre-existing files

When benchmarking storage systems, we must consider that the performance of such systems depends on characteristics of the underlying file system [4, 7], namely the pre-existing files and the structure of the hierarchical namespace (the

Table 6.2: Parameters used by the extended model to generate events to files. Note that O_i is the object or file i, where $i \in \{1,...,n\}$; K_j is the cluster j, where $j \in \{1,...,k\}$.

Symbol	Description
n	number of files in the trace or request stream
k	number of clusters or types of files
F_j	interarrival distribution for accesses to an object in cluster K_j
C_j	interarrival distribution for creation events for objects in cluster K_j
G_j	distribution of delay to first access to objects in K_j (relative to object's creation or t_0); $C_j + G_j = \text{Activation}_j$
H_j	active span distribution $\forall O_i \in K_j$; $\text{Activation}_j + H_j = \text{Deactivation}_j$
D_j	distribution of delay to delete event of objects in K_j (relative its last access in stream) $\text{Deactivation}_j + D_j = \text{Deletion}_j$
p	percentage of accesses that are open operations ($1-p$ are listStatus)
w_j	percentage of objects in K_j ; $\sum_{i=1}^{k} w_i = 1$
t_{end}	duration of request stream (in milliseconds)

latter is discussed in the next subsection).

We extend our model to consider pre-existing files by keeping track of the number of files (within each file type) that have been created sometime before the beginning of the modeled trace (i.e., before t_0). The information of whether a file exists prior to the captured trace of namespace events could be inferred from the trace itself by making the assumption that any file accessed in the trace, but not created during it, is a pre-existing file. This approach would lead to an inaccurate model if the trace contains many operations on files that do not exist (for example, due to users incorrectly entering the name of a file).

To avoid this problem we can use a *namespace metadata trace* that contains a snapshot of the namespace (file and directory hierarchy), in addition to the set of events that operate atop that namespace (e.g., open a file, list directory contents) [4]. For the case of the HDFS, the namespace metadata traces we've analyzed consist of a snapshot of the namespace obtained with Hadoop's Offline Image Viewer tool, and access log traces obtained by parsing the name node audit logs.

Extension 3: Support for realistic hierarchical namespaces

In earlier work [4] (see Chapter 3), we proposed a statistical model for generating realistic namespace hierarchies, and implemented a *namespace generation module* (NGM) based on this model. To the best of our knowledge, this is the only available tool that can generate large realistic namespaces[3].

[3]We tested the only other alternative system, the *Impressions* framework [7], and were not able to generate the large namespaces observed in Big Data storage deployments since it was designed

Prior to issuing the workload, the NGM is used to generate a realistic directory structure, which preserves the following statistical properties of the original directory hierarchy: number of directories, distribution of directories at each depth in namespace hierarchy, and distribution of number of subdirectories per directory.

By default, only that part of the namespace that was accessed on the original trace from which the model is based will be recreated, though the full directory structure can be reproduced if desired. The reason for this, is that the trace-induced directory structure is typically much smaller than the full directory structure [4], making a run of the benchmark finish faster (less `mkdir` operations need to be issued). However, the full directory structure (modeled after the snapshot of the namespace at t_0) can be reproduced if desired.

To integrate the files to this directory structure, we add another parameter to the model, the *percentage of files at each depth* of the hierarchy, and proportionally assign files to each depth in the hierarchy according to this parameter.

6.2.2 Assumptions and limitations

Our model can be used to generate realistic synthetic workloads to evaluate the *namespace metadata management subsystems* of next-generation storage systems; for this reason, dimensions related to data input/output behavior—like a correlation between file size and popularity or the length of data read/write operations—are out of the scope of our model.

We model **stationary segments** of object request streams. Workloads consisting of a few stationary segments can be divided using the approach in [76]; for more details, see Section 5.5.1. However, in practice, benchmark runs tend to issue the workload of a period of one hour or less, so the need to consider non-stationary segments is not critical.

We represent arrivals with a sequence of **interarrival times** $X = (X_1, X_2, ...)$, where X is a sequence of *independent, identically distributed random variables*. In practice, there could be autocorrelations in the arrival process, leading to bursty behavior at different timescales. We refer the reader to [6] (and Section 5.6) for a discussion on how the model can be extended to capture burstiness in the arrival processes.

We assume that each arrival process—in this context, of the accesses to a file

to model smaller (more traditional) namespaces. Furthermore, at the time of this writing, the Impressions framework is no longer available for download.

in the storage system—is **ergodic**, so its interarrival distribution can be deduced from a single sample or realization of the process (i.e., the trace of events representing the stationary segment being modeled). In addition, instead of forcing a fit to a particular interarrival distribution, we use the empirical distribution of the interarrivals inferred from the arrivals observed in the trace being modeled.

Finally, there may be unknown, but important, behavior in the original workloads that our model does not capture. Some of these dimensions, like spatial locality, may be added to our model in the future. However, there is a trade-off of increased complexity due to adding these dimensions. For this reason, smaller and simpler models are favored by the principle of parsimony.

6.3 Design

Similar to Hadoop's DFSIO [81] and S-live [68] stress tests, MimesisBench is currently implemented on top of MapReduce so that a set of mappers can simultaneously issue requests to the storage layer. Each mapper is in charge of issuing the operations on files of a particular type (as encapsulated by the model).

A run of MimesisBench has two phases (see Figure 6.2). In the first phase, the pre-existing files (those that exist before the beginning of the workload generation process) are created. In the second phase, the workload is issued.

In each phase, the *job coordinator* parses the user parameters and generates configuration files for each worker. In addition, the job coordinator of the first phase creates the hierarchical namespace based on an input parameter file that has been pre-generated with the Namespace Generation Module (NGM).

In the **first phase**, the workers create the target number of files for each type. A configuration parameter tells the workers to use a flat namespace (create all files in one single, configurable path) or to use a hierarchical namespace. When a hierarchical namespace is used, files of each type are created at different levels of the namespace hierarchy, proportionally to the configuration parameter of *files at each depth* (which indicates what percentage of the total files are located at each depth). The subdirectories at each depth are assigned to a file/cluster type, proportionally to the weight of the cluster (w_j) to which the file belongs.

In the **second phase**, the workers issue the load. Each *load generator* worker reads the configuration for the specific file type that it has been assigned and waits

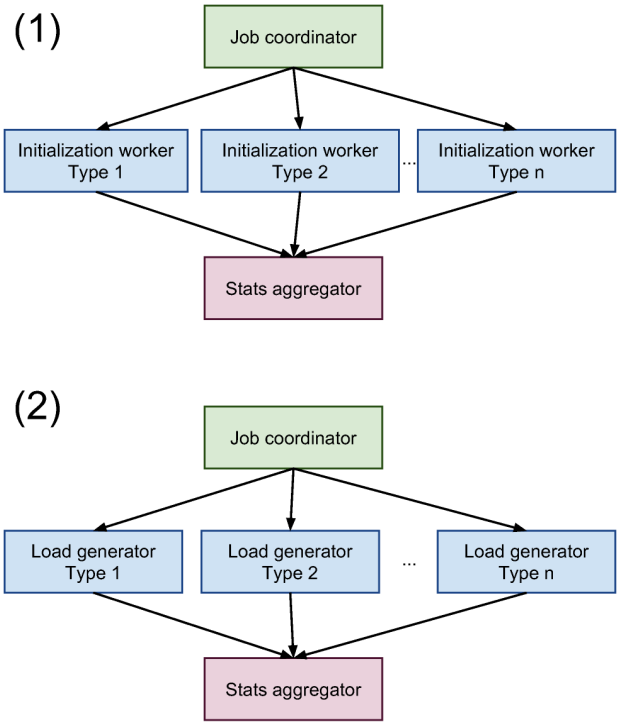


Figure 6.2: A run of MimesisBench has two phases. First, the pre-existing directory structure and files are created. Next, the workload is issued.

in a time-based barrier[4] to start issuing the load corresponding to the files that belong to the type it is in charge of. Two data structures are used to keep track of the files and events: a `PriorityBlockingQueue` of files (sorted by the timestamp of the next event—create, open, etc.—of that file) and a FIFO `BlockingQueue` of events to be issued. Three threads coordinate access to these data structures: a file introduction thread adds files to the priority queue (using the cluster's interarrival distribution)[5], another thread continuously polls the priority queue and adds details of the next event to be issued to the back of the FIFO queue. Finally, a consumer thread pulls the information of the next event to be issued from the FIFO queue and schedules it to be issued at the proper time, using Java's `ScheduledExecutorService`. A configurable maximum allowed drift is used to abort a run of the benchmark if the events are falling behind too much from their original schedule (maximum drift is 2 seconds by default). In that case, more mappers would be needed to issue the workload.

Finally, for each phase a single stats aggregator task collects the stats from the

[4]The clocks of the nodes in a Hadoop cluster must be synchronized to be able to support Kerberos authentication, used in Hadoop's security implementation.

[5]The pre-existing files of that particular file type are added to the queue when the worker loads.

workers and generates aggregate results (see Table 6.3). We chose these metrics because operation latency is the most important user metric, while throughput (of the metadata server) is the most important system metric.

Table 6.3: Statistics reported by MimesisBench.

Description	Unit
Throughput	ops/sec
Active time (workers)	*msecs*
Operations issued	*ops*
Successful creates	*ops*
Successful deletes	*ops*
Successful opens	*ops*
Successful listStatus	*ops*
Average latency: Create	*msecs*
Average latency: Delete	*msecs*
Average latency: Open	*msecs*
Average latency: listStatus	*msecs*

6.3.1 Scaling workloads

MimesisBench facilitates scaling a workload across several dimensions, as explained next. Refer to Table 6.1 for details on the symbols used in this subsection.

Number of files The number of files, n, is configurable. Each workload profile available with MimesisBench comes with a configured number of files as observed in the original namespace metadata trace on which the model is based. The user can increase (or decrease) the number of files to emulate a larger cluster; the proportion of each type of file is maintained.

Number of files of a particular type The user can increase the number of files of only one particular type by increasing n and doing a transformation on the w_i weights[6]. For example, to double the number of objects of type 1 while keeping the number of objects of types 2 to k unchanged, we can obtain the values of n_{new} and $\{w_{1_{\text{new}}}, w_{2_{\text{new}}}, ..., w_{k_{\text{new}}}\}$ as follows:

$$n_{\text{new}} = n + w_1 \times n$$
$$w_{1_{\text{new}}} \times n_{\text{new}} = 2(w_1 \times n)$$
$$w_{i_{\text{new}}} \times n_{\text{new}} = w_i \times n, \quad i \in \{2, ..., k\}$$

[6]Recall that w_j is the percentage (out of the total number of files, n) of files in cluster K_j.

Solving these equations gives us the values of n_{new} and $\{w_{1_{\text{new}}}, ..., w_{k_{\text{new}}}\}$ to use in the synthetic workload generation process.

Time Interarrivals can be *accelerated* by multiplying the random variable by a scaling factor between 0 and 1. A scaling factor of 1 reproduces the original workload, while a scaling factor of 0 provides maximum stress on the system by issuing all the operations as fast as possible (0 millisecond wait between issuing operations). Interarrivals can also be slowed down or *deaccelerated* by multiplying the random variable by a constant (scaling factor) greater than 1.[7]

Active span The active span random variable can also be multiplied by a user-defined constant greater or equal than zero. Modifying the active span has the effect of modifying the number of accesses of the files, thus modifying their popularity.

6.3.2 Isolating workloads

To facilitate an understanding on how a particular type of file affects the performance of the system, the user can instruct MimesisBench to issue only the workload of the files of that particular type or cluster (i.e., turn-off the other types of files). Any combination of dimension scaling described in the previous subsection can also be applied to the isolated workload if desired.

6.3.3 Summary of MimesisBench features

To conclude this Section, we include a comparison of the features of MimesisBench to other related tools as shown in Table 6.4. The tools listed in this table are described in Section 6.5.

[7]Note that, multiplying a random variable, X, by a constant, c, modifies the mean and the variance of the random variable. The expected value of X is multiplied by c, while the variance is multiplied by c^2 (thus, the standard deviation is also multiplied by c).

Table 6.4: Features of MimesisBench and other related tools.

Feature	NNBench	DFSIO	S-live	FileBench	SPECsfs	mdtest	MimesisBench
For next-generation storage	✓	✓	✓				✓
Metadata-intensive	✓					✓	✓
Realistic workloads				✓	✓		✓
Type-aware workload				✓			✓
Autonomic type-awareness							✓
Hierarchical namespace	Semi-flat^{+}		Semi-flat^{+}	Limited*	Fixed		✓

$^{+}$ Only multiple directories at depth 1 are supported.

* FileBench can generate directory hierarchies with a configurable depth and width; however, characteristics like subdirectories per directory and number of directories (width) per depth, are not supported.

6.4 Using MimesisBench to evaluate a Big Data storage system

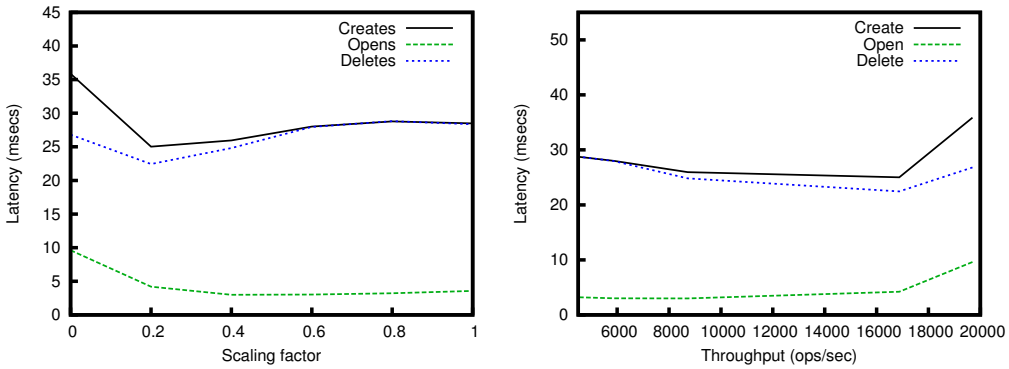
In this Section, we demonstrate the usefulness of MimesisBench by using it to evaluate the performance and scalability of the HDFS name node across several dimensions.

We modeled a 1-day (Dec. 1, 2011) *namespace metadata trace* from an Apache Hadoop cluster at Yahoo. The trace came from a 4100-node production cluster running the Hadoop Distributed File System (HDFS) [70]. The 1-day trace we analyzed contains 60.9 million opens events that target 4.3 million distinct files, and 4 PB of used storage space. For a detailed workload characterization of how the MapReduce jobs operating in this cluster interact with the storage layer see Chapter 4 and [5].

We modeled the trace using 30 file types or clusters (i.e., $k = 30$ for the k-means clustering algorithm). We chose this value of k because it was the smallest for which we could obtain a close approximation of the file popularity distribution (Kolmogorov-Smirnov distance between the real and synthetic CDFs $< 4\%$).

6.4.1 Latency of opens, creates and deletes

Figure 6.3a shows the effect on the latency of the operations as interarrivals are accelerated (i.e., operations are issued faster). In general, read operations like opens, are faster than write operations (creates and deletes) because acquiring an



(a) Operation latency

(b) Throughput vs. latency

Figure 6.3: *Left:* Latency of creates, deletes and opens, with different interarrival scaling factors, averaged across five runs. In each run, the interarrival random variable, X, was multiplied by a constant, c, shown in the x-axis. When $c = 0$, the operations are issued as fast as possible; when $c = 1$ the operation interarrival mimics the interarrivals of the original trace. For all data points, the standard deviation is very small (< 0.7 msecs). *Right:* Throughput vs. operation (open) latency.

exclusive lock is not necessary to read the namespace[8].

The same information is shown by plotting the throughput vs. the latency (see Figure 6.3b), as increasing the scaling factor leads to a higher number of operations being issued per second.

We can observe, that the latency of the operations is not initially affected by issuing them faster. This shows, that the performance of the name node is not degraded as more operations are issued per second. However, when the clients issue more than 15,000 operations per seconds ($c \geq 0.2$), the name node starts showing a rapidly degrading performance. This information can be used to determine whether the name node can properly support an increased workload (in terms of speed at which the operations are issued) in the future.

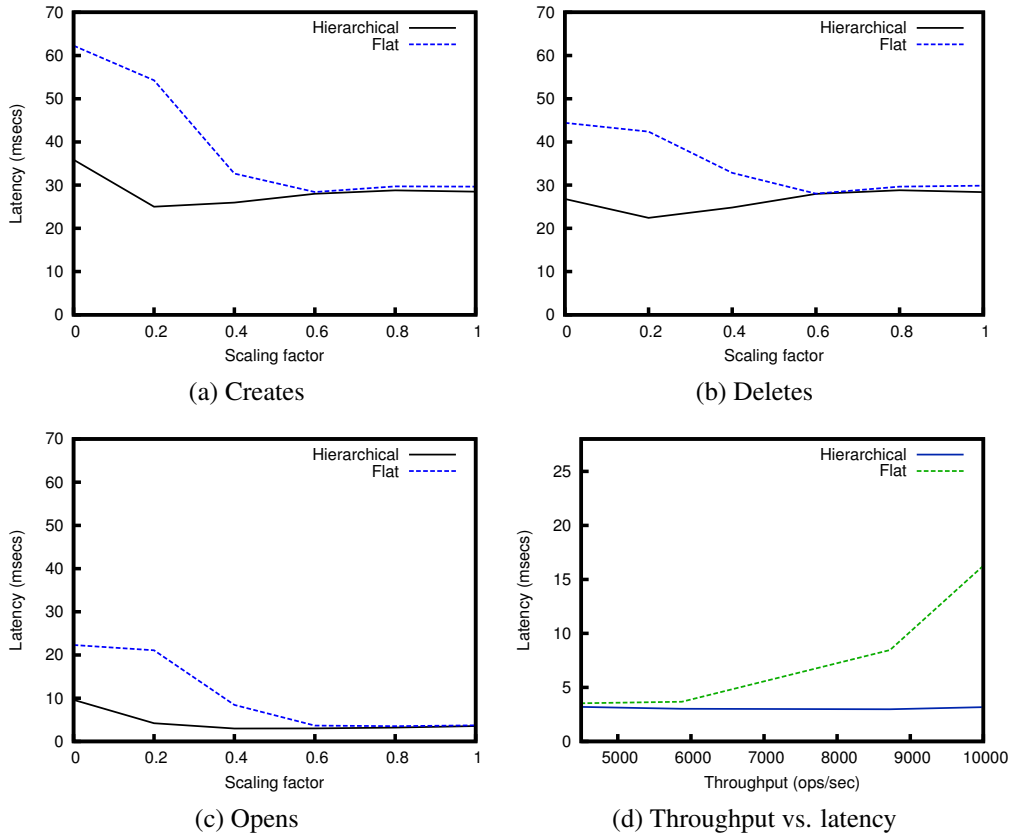
6.4.2 Flat versus hierarchical namespaces

Figure 6.4 shows the impact that using a hierarchical namespace (versus a flat namespace) has on the performance of the name node. We can observe that performance of the system degrades significantly faster on a hierarchical namespace than on a flat one (see the significant increase in latency as the load is issued faster). The throughput is also affected accordingly: the hierarchical namespace

[8]In the context of this Chapter, read/write (R/W) operations refer to operations that access (read) or modify (write) the namespace of the storage system; input/output (I/O) operations are out of the scope of this Thesis.

can serve up to 19,696 versus 10,284 ops/sec for the flat namespace; in other words, the name node can serve roughly half of the operations per second when a flat namespace is used instead of a hierarchical namespace.

These results show that using benchmarks that create files in a flat namespace, like NNBench and S-live (see Table 6.4), is not desirable as they place a heavier and unrealistic burden on the locking mechanisms of the metadata server.



(a) Creates (b) Deletes

(c) Opens (d) Throughput vs. latency

Figure 6.4: Latency of creates, deletes and opens, with different interarrival scaling factors, averaged across five runs, in a hierarchical and a flat namespace. For all data points, the standard deviation is very small (< 1.5 msecs). *Bottom right*: Throughput vs operation (open) latency for both types of namespaces.

6.4.3 Isolating workloads

We isolated the workload of two clusters with very distinct data access patterns and observe the effects on the latency of open events, and on the name node throughput.

Table 6.5 shows the summary statistics of the two isolated clusters or file

types. We chose these two clusters because they represent two extreme read-mostly (cluster 29) and write-heavy (cluster 17), yet realistic, workloads.

We ran several tests with events issued at normal speed (interarrival scaling factor = 1) and at full speed (interarrival scaling factor = 0). To ensure a fair comparison, we adjusted the number of mappers issuing the workload so that the total number of operations issued in both tests was roughly the same. At 30 mappers for cluster 17, and 12 mappers for cluster 29, the number of operations issued in the tests was $3\,559\,101 \pm 0.9\%$.

Figure 6.5 shows the latency of opens and name node throughput, averaged across five runs[9]. We can observe that the latency of the open events degrades significantly more in a write-heavy workload: When operations are issued at full speed, the latency of opens in the write-heavy workload increases 3.9x in cluster 17 versus 1.4x in cluster 29. In addition, at maximum issuing speed (interarrival scaling = 0) the name node can serve 8 times more operations when the workload constitutes only reads: 53,233 vs. 6,453 ops/sec for clusters 29 and 17, respectively.

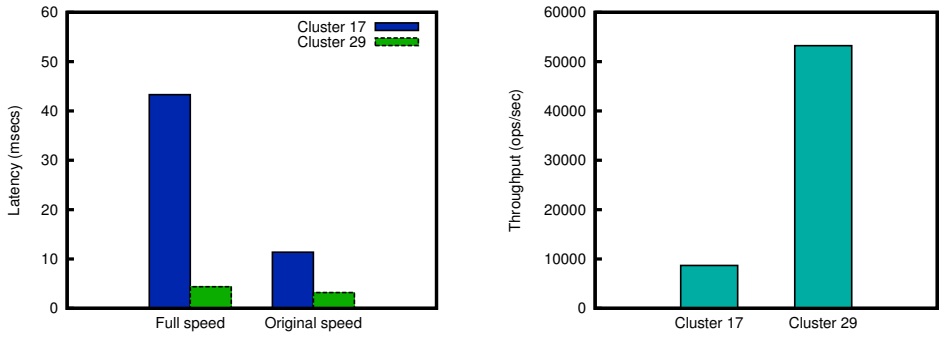
Table 6.5: Characteristics of the two clusters or file types whose workload was isolated for testing purposes. We chose these two clusters because they represent two extreme read-mostly (cluster 29) and write-heavy (cluster 17), yet realistic, workloads.

	Cluster 17	**Cluster 29**
Mean interarrivals (regular accesses)	179.61 msecs (0.18 secs)	4.92 msecs
Mean creates interarrivals	40.64 msecs	87,355.00 msecs
Mean active span	3.33 mins	8.33 mins
Percentage of read operations	69%	$\approx 100\%$
Percentage of write operations	31%	$\approx 0\%$

6.4.4 Stability of the results

Benchmarks are often used to evaluate new designs against an old design, or against another competing new design. An improvement of 10% in performance may be desirable to push in a large scale system, as long as we can trust the results of the benchmark. For this reason, it is important to have a small variability in the results produced by a benchmark. For example, it is not reasonable to trust a 10% performance improvement if the coefficient of variation, c_v, of the results is 8%.

[9]We do not compare the performance of write (i.e., create and delete) operations because cluster 29 had too few write operations to enable a fair comparison (see Table 6.5).



(a) Opens latency

(b) Name node throughput

Figure 6.5: *Left:* Latency of open events, with two interarrival scaling factors (0 or operations at full speed, and 1 or operations issued at their original speed), averaged across five runs, for two clusters with different read/write ratios (see Table 6.5). For all data points, the standard deviation was very small (< 0.5 msecs). *Right*: Name node throughput for both clusters, when operations are issued at full speed; std. dev. < 0.1 ops/sec.

In out experiments, MimesisBench yielded very small standard deviation and corresponding coefficient of variation (or the ratio of the standard deviation to the mean). For all the sets of experiments we ran, the greatest observed coefficient of variation was 2.38%, with an average coefficient of variation of 1.08%. These results suggest that MimesisBench is suitable as a tool to evaluate expected performance improvements of new designs.

6.5 Related work

Some prior tools provide a subset of the features of MimesisBench, as summarized in Table 6.4.

The *mdtest* microbenchmark generates metadata intensive scenarios for traditional storage systems; however, it does not provide a way to fit the workload to real traces or realistic namespaces. In addition, mdtest interfaces with the storage system via system calls and has not been ported to the interfaces of next-generation storage systems, nor does it allow distributed workload generation, so it cannot scale to Big Data workloads.

FileBench [34] is an expressible benchmarking tool designed for enterprise storage systems. It includes traditional workloads via loadable "personalities" like web server, file server, and database server. However, it works only on POSIX-compliant storage system interfaces, making it unsuitable for next-generation storage systems with proprietary APIs like HDFS. Furthermore, emerging Big Data

workloads have not been included as a pre-defined personality.

SPECsfs2008 [71] is a benchmark suite intended to be used as a standard to enable comparison of file server throughput and response times across different vendors and configurations. It supports NFSv3 and CIFS APIs and comes with a pre-defined workload based on enterprise NFS and CIFS workloads.

For the specific case of the Hadoop Distributed File System (HDFS), the Hadoop developer community has designed two benchmarks that can test I/O operations and do simple stress-tests on storage layer: *DFSIO* [81] and *S-live* [68]. However, these benchmarks do not reproduce realistic workloads and can only be used as microbenchmarks. Another Apache tool, *NNBench* [58] was created to benchmark the HDFS name node; however, it can only issue one type of operation at a time, and does not work atop a realistic namespace.

Tarasov et al. [75] also pointed out that workloads provided with traditional benchmarks like FileBench and SPECsfs2008 are a poor replacement for non-traditional workload models. Their work provides benchmarks for an emerging workload in cloud computing: virtual machine workloads. Our work complements this work by providing benchmarks for another type of emerging workload: MapReduce clusters.

Impressions [7] generates realistic file system images; however, it is not readily coupled with a workload generator to easily reproduce workloads that operate on the generated namespace. Furthermore, the generative model used by Impressions to create the file system hierarchy is not able to reproduce the distributions observed in our analysis, nor is it able scale to the large hierarchies observed in the Big Data systems we've studied.

Chen et al. [22] proposed the use of multi-dimensional statistical correlation (k-means) to obtain storage system access patterns and design insights in user, application, file, and directory levels. However, the clustering was not leveraged for synthetic workload generation or benchmarking.

In earlier work, we developed Mimesis [4], a synthetic trace generator for namespace metadata traces. While Mimesis is able to generate traces that mimic the original workload with respect to the statistical parameters included with it (arrivals, file creations and deletions, and age at time of access), the model it is based on is too CPU-intensive to enable real-time issuing of operations and as such is inapplicable for benchmarking real systems (though its synthetic traces could be used in simulation-based evaluations). In addition, in the proof-of-concept model used in [4], reproducing the file popularity was left for future work.

6.6 Summary

In this Chapter, we presented MimesisBench, a metadata-intensive storage benchmark suitable for Big Data workloads. MimesisBench consists of a workload-generating software and a set of workloads from two Yahoo Big Data clusters and other traditional domains.

MimesisBench is extensible and more workloads can be added in the future. It is based on a novel model that allows it to generate type-aware workloads, in which specific type of file behavior can be isolated or modified for 'what-if' and sensitivity analysis. In addition, MimesisBench support multi-dimensional workload scaling.

MimesisBench is currently implemented on top of the Apache Hadoop framework, which allows it to be used in any storage system that is compatible with Hadoop (e.g., HDFS, Ceph, CassandraFS, Lustre). We have released the benchmark and workloads as open source[10].

A study of the performance and scalability of the Hadoop Distributed File System was presented to show the usefulness of metadata-intensive benchmarking using MimesisBench.

[10] Available at `http://sites.google.com/site/cristinaabad`

Chapter 7

Conclusions and Future Directions

In this thesis, we studied the problem of workload characterization and modeling for Big Data storage systems. Specifically, we studied how MapReduce interacts with the storage layer and created models suitable for the evaluation of namespace metadata management subsystems.

We began by arguing for the need of *namespace metadata* traces and workloads that contain information about the namespace (directory hierarchy and files), as well as a stream of operations or events atop that namespace.

Our model and tool for generating synthetic hierarchical namespaces is the only available tool for this purpose, and can generate large and realistic namespaces suitable for Big Data storage evaluation.

To deepen our understanding of these emerging workloads, we analyzed two 6-month namespace metadata traces from two large clusters at Yahoo. Our workload characterization provided good insight on how to properly design and tune systems for MapReduce workloads. To the best of our knowledge, this is the only comprehensive study of how MapReduce interacts with the storage layer.

We then studied the general problem of modeling object request streams to be able to synthetically reproduce their short-term temporal correlations and their long-term popularity. The model we proposed is general, autonomic, type-aware, and supports dynamic object populations, thus making it suitable for emerging workloads. Our analysis of the model and results from two case studies show that it can be used successfully in place of the real workload.

Finally, we extended our model for object request streams so that it can support: (1) other metadata operations in addition to open events, (2) pre-existing namespaces, and (3) realistic hierarchical namespaces. Based on this model, we implemented MimesisBench, a benchmarking tool for HDFS (and other next-generation storage systems that can interface with Hadoop) and demonstrated its usefulness by performing an evaluation of the performance and scalability of the HDFS name node or metadata server. MimesisBench is extensible, supports

multi-dimensional workload scaling, and its autonomic type-aware model facilitates sensitivity and 'what-if' analysis.

Several future directions of research arise out of this work. First, our models can be extended to reproduce other dimensions of the original workload, like spatial locality. However, the cost of increased complexity due to adding these dimensions should be carefully considered.

Second, our model can be used to create workload generators for other types of systems, like accesses to key-value stores, web servers, etc. In the short term, we intend to extend existing key-value store benchmarking tools so that they can benefit from our models.

Finally, extensive studies of multiple storage systems should be performed using MimesisBench so that we can deepen our understanding of the advantages and disadvantages of diverse design decisions and possibly suggest improvements to these designs.

References

[1] Tachyon. tachyon-project.org, March 2013. Last accessed: April 18, 2013.

[2] Cristina L. Abad, Yi Lu, and Roy H. Campbell. DARE: Adaptive data replication for efficient cluster scheduling. In *Proceedings of the International Conference on Cluster Computing (CLUSTER)*, 2011.

[3] Cristina L. Abad, Huong Luu, Yi Lu, and Roy H. Campbell. Metadata workloads for testing Big storage systems. Technical report, UIUC, 2012. hdl.handle.net/2142/30013.

[4] Cristina L. Abad, Huong Luu, Nathan Roberts, Kihwal Lee, Yi Lu, and Roy H. Campbell. Metadata traces and workload models for evaluating Big storage systems. In *Proceedings of the IEEE/ACM Utility and Cloud Computing Conference (UCC)*, 2012.

[5] Cristina L. Abad, Nathan Roberts, Yi Lu, and Roy H. Campbell. A storage-centric analysis of MapReduce workloads: File popularity, temporal locality and arrival patterns. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2012.

[6] Cristina L. Abad, Mindi Yuan, Chris Cai, Nathan Roberts, Yi Lu, and Roy H. Campbell. Generating request streams on Big Data using clustered renewal processes. *Performance Evaluation*, 70(10), October 2013.

[7] Agrawal, Arpaci-Dusseau, and Arpaci-Dusseau. Generating realistic Impressions for file-system benchmarking. *ACM Transactions on Storage (TOS)*, 5, 2009.

[8] Nitin Agrawal, William Bolosky, John Douceur, and Jacob Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)*, 3, 2007.

[9] Sadaf Alam, Hussein El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni. Parallel I/O and the metadata wall. In *Proceedings of the Petascale Data Storage Workshop (PDSW)*, 2011.

[10] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: Coping with skewed popularity content in MapReduce clusters. In *Proceedings of the European Conference on Computing Systems (EuroSys)*, 2011.

[11] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Wang, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. PACMan: Coordinated memory caching for parallel jobs. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[12] Eric Anderson. Capture, conversion, and analysis of an intense NFS workload. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[13] Apache Hadoop. `hadoop.apache.org`, April 2013. Last accessed: April 15, 2013.

[14] Paul Barford and Mark Crovella. Generating representative Web workloads for network and server performance evaluation. *SIGMETRICS Performance Evaluation Review*, 26(1), June 1998.

[15] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the IEEE International Conference on Computing Communications (INFOCOM)*, 1999.

[16] Mudashiru Busari and Carey Williamson. ProWGen: A synthetic workload generation tool for simulation evaluation of Web proxy caches. *Computer Networks*, 38(6), 2002.

[17] Chris Cai, Cristina L. Abad, and Roy H. Campbell. Storage efficient data replica number computation for multi-level priority data in distributed storage systems. In *Proceedings of the Workshop Reliability and Security Data Analysis (RSDA), held in conjunction with DSN*, 2013.

[18] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the Usenix Symposium Internet Technologies and Systems (USITS)*, 1997.

[19] Giuliano Casale, Eddy Z. Zhang, and Evgenia Smirni. Trace data characterization and fitting for Markov modeling. *Performance Evaluation*, 67(2), 2010.

[20] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive query processing in Big Data systems: A cross-industry study of MapReduce workloads. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2012.

[21] Yanpei Chen, Archana Ganapathi, Rean Griffith, and Randy Katz. The case for evaluating MapReduce performance using workload suites. In *Proceedings of the IEEE International Symposium on Modeling, Analysis and Simulation of Computing Telecommunication Systems (MASCOTS)*, 2011.

[22] Yanpei Chen, Kiran Srinivasan, Garth Goodson, and Randy Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proceedings of the ACM Symposium Operating Systems Principles (SOSP)*, 2011.

[23] Ludmila Cherkasova and Minaxi Gupta. Analysis of enterprise media server workloads: Access patterns, locality, content evolution, and rates of change. *IEEE/ACM Transactions on Networks*, 12(5), 2004.

[24] Aaron Clauset, Cosma Shalizi, and M. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4), 2009.

[25] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010.

[26] Mark Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networks*, 5(6), 1997.

[27] Carlos Cunha, Azer Bestavros, and Mark Crovella. Characteristics of WWW client-based traces. Technical Report BU-CS-95-010, 1995. Boston University.

[28] György Dán and Niklas Carlsson. Power-law revisited: Large scale measurement study of P2P content popularity. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2010.

[29] Shobhit Dayal. Characterizing HEC storage systems at rest. Technical Report CMU-PDL-08-109, CMU, 2008. Data available at: `institutes.lanl.gov/data/fsstats-data`. Last accessed: April 15, 2013.

[30] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[31] Ananth Devulapalli and Pete Wyckoff. File creation strategies in a distributed metadata file system. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[32] Bin Fan, Wittawat Tantisiriroj, Lin Xiao, and Garth Gibson. DiskReduce: RAID for data-intensive scalable computing. In *Proceedings of the Petascale Data Storage Workshop (PDSW)*, pages 6–10, 2009.

[33] Ribel Fares, Brian Romoser, Ziliang Zong, Mais Nijim, and Xiao Qin. Performance evaluation of traditional caching policies on a large system with petabytes of data. In *Proceedings of the IEEE International Conference on Network, Architectures and Storage (NAS)*, 2012.

[34] FileBench. `sourceforge.net/projects/filebench`, March 2013. Last accessed: April 15, 2013.

[35] Rodrigo C. Fonseca, Virgilio Almeida, Mark Crovella, and Bruno D. Abrahao. On the intrinsic locality properties of Web reference streams. In *Proceedings of the IEEE International Conference on Computing Communications (INFOCOM)*, 2003.

[36] Al Geist. Paving the road to exascale. *SciDAC Review*, Special Issue(16), 2010.

[37] Sanjay Ghemawat, Howard Gobioff, and Shun Leung. The Google File System. In *Proceedings of the ACM Symposium Operating Systems Principles (SOSP)*, 2003.

[38] Christian Hennig. fpc: Flexible procedures for clustering. `cran.r-project.org/web/packages/fpc`, January 2013. Last accessed: March 23, 2013.

[39] Bo Hong, Tara M. Madhyastha, and Bing Zhang. Cluster-based input/output trace synthesis. In *Proceedings of the IEEE International Performance Computing and Communications Conference (IPCCC)*, 2005.

[40] Shudong Jin and Azer Bestavros. Sources and characteristics of Web temporal locality. In *Proceedings of the IEEE International Symposium on Modeling, Analysis and Simulation of Computing Telecommunication Systems (MASCOTS)*, 2000.

[41] Shudong Jin and Azer Bestavros. GISMO: A generator of Internet streaming media objects and workloads. *SIGMETRICS Performance Evaluation Review*, 29(3), 2001.

[42] JUQUEEN — Julich Blue Gene/Q. `www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/JUQUEEN_node.html`, February 2013. Last accessed: April 20, 2013.

[43] Juelich storage cluster (JUST). `www.fz-juelich.de/ias/jsc/EN/Expertise/Datamanagement/OnlineStorage/JUST/Configuration/Configuration_node.html`, December 2012. Last accessed: April 20, 2013.

[44] Rini Kaushik, Ludmila Cherkasova, Roy H. Campbell, and Klara Nahrstedt. Lightning: Self-adaptive, energy-conserving, multi-zoned, commodity green cloud storage system. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2010.

[45] Rini Kaushik and Klara Nahrstedt. T: A data-centric cooling energy costs reduction approach for Big Data analytics cloud. In *Proceedings of the ACM/IEEE Conference on High Performance Computing Networks, Storage and Analysis (SC)*, 2012.

[46] Kosmos file system (KFS). `code.google.com/p/kosmosfs`. Last accessed: April 18, 2013.

[47] Kraken overview. `www.nics.tennessee.edu/computing-resources/kraken`. Last accessed: April 20, 2013.

[48] Will Leland, Murad Taqqu, Walter Willinger, and Daniel Wilson. On the self-similar nature of Ethernet traffic. *IEEE/ACM Transactions Networks*, 2(1), 1994.

[49] Andrew Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[50] Hui Li and Lex Wolters. Towards a better understanding of workload dynamics on data-intensive clusters and grids. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[51] Thanasis Loukopoulos and Ishfaq Ahmad. Static and adaptive data replication algorithms for fast information access in large distributed systems. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2000.

[52] John MacCormick, Nicholas Murphy, Venugopalan Ramasubramanian, Udi Wieder, Junfeng Yang, and Lidong Zhou. Kinesis: A new approach to replica placement in distributed storage systems. *ACM Transactions on Storage (TOS)*, 4(4), February 2009.

[53] James MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium Mathematical Statistics and Probability*, volume 1, 1967.

[54] Kirk McKusick and Sean Quinlan. GFS: Evolution on fast-forward. *Commununications of the ACM*, 53, 2010.

[55] Argonne leadership computing facility resources and expertise. `www.alcf.anl.gov/mira`, 2013. Last accessed: April 20, 2013.

[56] Domas Mituzas. Page view statistics for Wikimedia projects. dumps.wikimedia.org/other/pagecounts-raw, March 2013. Last accessed: April 8, 2013.

[57] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Enabling high data throughput in desktop grids through decentralized data and metadata management: The BlobSeer approach. In *Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par)*, 2009.

[58] Michael G. Noll. Benchmarking and stress testing an Hadoop cluster with TeraSort, TestDFSIO & co. www.michael-noll.com/blog/2011/04/09/benchmarking-and-stress-testing-an-hadoop-cluster-with-terasort-testdfsio-nnbench-mrbench, April 2011. Last accessed: January 18, 2014.

[59] Kihong Park, Gitae Kim, and Mark Crovella. On the relationship between file sizes, transport protocols, and self-similar network traffic. In *Proceedings of the International Conference on Network Protocols (ICNP)*, 1996.

[60] Swapnil Patil and Garth Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[61] Swapnil Patil, Garth Gibson, Gregory Ganger, Julio Lopez, Milo Polte, Wittawat Tantisiroj, and Lin Xiao. In search of an API for scalable file systems: Under the table or above it? In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[62] Swapnil Patil, Kai Ren, and Garth Gibson. A case for scaling HPC metadata performance through de-specialization. In *Proceedings of the Petascale Data Storage Workshop (PDSW)*, 2012.

[63] Powered-by – Hadoop Wiki. wiki.apache.org/hadoop/PoweredBy, January 2013. Last accessed: April 15, 2013.

[64] Quantcast file system (QFS). github.com/quantcast/qfs/wiki/Introduction-To-QFS. Last accessed: April 18, 2013.

[65] Phil Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the Linux Symposium*, 2003.

[66] Navin Sharma, Sean Barker, David Irwin, and Prashant Shenoy. Blink: Managing server clusters on intermittent power. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[67] Konstantin Shvachko. HDFS scalability: The limits to growth. *;login: USENIX Magazine*, 35(2), 2010.

[68] Konstantin Shvachko. A stress-test tool for HDFS (Apache JIRA HDFS-708). issues.apache.org/jira/browse/HDFS-708, May 2010. Last accessed: December 30, 2013.

[69] Konstantin Shvachko. Apache Hadoop: The scalability update. *;login: USENIX Magazine*, 2011.

[70] Konstantin Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[71] SPECsfs2008. http://www.spec.org/sfs2008/, April 2013. Last accessed: December 30, 2013.

[72] Stampede user guide. www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide, 2013. Last accessed: April 20, 2013.

[73] Wenting Tang, Yun Fu, Ludmila Cherkasova, and Amin Vahdat. MediSyn: A synthetic streaming media service workload generator. In *Proceedings of the ACM Network Operating Systems Support for Digital Audio and Video Conference (NOSSDAV)*, 2003.

[74] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking file system benchmarking. In *HotOS*, 2011.

[75] Vasily Tarasov, Dean Hildebrand, Geoff Kuenning, and Erez Zadok. Virtual machine workloads: The case for new benchmarks for NAS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[76] Vasily Tarasov, Santhosh Kumar, Jack Ma, Dean Hildebrand, Anna Povzner, Geoff Kuenning, and Erez Zadok. Extracting flexible, replayable models from large block traces. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[77] Jeff Terrace and Michael Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2009.

[78] ORNL debuts Titan supercomputer. www.olcf.ornl.gov/2012/10/29/ornl-debuts-titan-supercomputer, October 2012. Last accessed: April 20, 2013.

[79] Spider — the center-wide Lustre file system. www.olcf.ornl.gov/kb_articles/spider-the-center-wide-lustre-file-system, February 2013. Last accessed: April 20, 2013.

[80] Sarut Vanichpun and Armand M. Makowski. Comparing strength of locality of reference – Popularity, majorization, and some folk theorems. In *Proceedings of the IEEE International Conference on Computing Communications (INFOCOM)*, 2004.

[81] Vinod Kumar Vavilapalli. Delivering on Hadoop .Next: Benchmarking performance. `hortonworks.com/blog/delivering-on-hadoop-next-benchmarking-performance`, February 2012. Last accessed: December 30, 2013.

[82] Abhishek Verma, Shivaram Venkataraman, Matthew Caesar, and Roy H. Campbell. Scalable storage for data-intensive computing. In *Handbook of Data Intensive Computing*. Springer, 2011.

[83] Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip Roth. Scalable I/O tracing and analysis. In *Proceedings of the Petascale Data Storage Workshop (PDSW)*, 2009.

[84] Mengzhi Wang, Tara Madhyastha, Ngai Hang Chang, Spiros Papadimitriou, and Christos Faloutsos. Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2002.

[85] Yu Wang, Jing Xing, Jin Xiong, and Dan Meng. A load-aware data placement policy on cluster file system. In *Proceedings of the IFIP International Conference on Network and Parallel Computing (NPC)*, 2011.

[86] Peter Ware, Thomas Page, Jr., and Barry Nelson. Automatic modeling of file system workloads using two-level arrival processes. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(3), July 1998.

[87] Qingsong Wei, Bharadwaj Veeravalli, Bozhao Gong, Lingfang Zeng, and Dan Feng. CDRM: A cost-effective dynamic replication management scheme for cloud storage cluster. In *Proceedings of the International Conference on Cluster Computing (CLUSTER)*, 2010.

[88] Sage Weil, Scott Brandt, Ethan Miller, Darrell Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[89] Sage Weil, Scott Brandt, Ethan Miller, and Carlos Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the ACM/IEEE Conference on High Performance Computing Networks, Storage and Analysis (SC)*, 2006.

[90] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the Panasas parallel file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[91] Jing Xing, Jin Xiong, Ninghui Sun, and Jie Ma. Adaptive and scalable metadata management to support a trillion files. In *Proceedings of the ACM/IEEE Conference on High Performance Computing Networks, Storage and Analysis (SC)*, 2009.

[92] Shuangyang Yang, Walter Ligon, and Elaine Quarles. Scalable distributed directory implementation on Orange File System. In *Proceedings of the IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2011. Project available at: `www.orangefs.org`. Last Accessed: Apr 18, 2013.

[93] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the European Conference on Computing Systems (EuroSys)*, 2010.

[94] Ziliang Zong, Xiao Qin, Xiaojun Ruan, and M. Nijim. Heat-based dynamic data caching: A load balancing strategy for energy-efficient parallel storage systems with buffer disks. In *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2011.