Workload Characterization and Modeling, and the Design and Evaluation of Cache Policies for Big Data Storage Workloads in the Cloud

Sacheendra Talluri

TU Delft

Workload Characterization and Modeling, and the Design and Evaluation of Cache Policies for Big Data Storage Workloads in the Cloud

by

Sacheendra Talluri

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday December 7, 2018 at 10:00 AM.

Student number:	4608313	
Project duration:	December 21, 2017 – December 7, 2018	
Thesis committee:	Prof. dr. ir. A. Iosup,	TU Delft and Vrije Universiteit, Amsterdam
	Dr. J. S. Rellermeyer,	TU Delft
	Dr. ir. F. A. Kuipers,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

The proliferation of big-data processing platforms has already led to radically different system designs, such as MapReduce and the newer Spark. Understanding the workloads of such systems enables tuning and could foster new designs. However, whereas MapReduce workloads have been characterized extensively, relatively little public knowledge exists about the characteristics of Spark workloads in representative environments. In this work, we focus on understanding the behavior and cache performance of the storage sub-system used for Spark workloads in the cloud. First, we statistically characterize its usage. Second, we design a generative model to tackle the scarcity of workload traces. Third, we design a cache policy putting our insight from the characterization to work. Finally, we evaluate the performance of different cache policies for big data workloads via simulation.

Preface

I started this thesis naive and bright eyed, dreaming of building cool systems, because I assumed that was research. Over the last year, my conception of research has transformed. I now see it as the process of investigation, experimentation, argumentation, and a lot more. This change was only possible with a lot of guidance. For that, I thank my supervisor, Alexandru.

I want to thank my collaborators on this work, Alicja Łuszczak and Cristina Abad, for their insight and contributions. I am grateful to Pieter Senster and everyone at Databricks Amsterdam for their help.

I am grateful to Erwin, Laurens, Ahmed, and all other members of the AtLarge team for their support. The many discussion we had were always helpful and a source of joy.

I want to thank all my friends from the TU Delft Debating Club. This gratitude also extends to everyone I call a friend in Delft. All of you have enriched my life during my stay here.

Finally, this work would not have been possible without the love and support of my parents. For that, I am eternally grateful.

Sacheendra Talluri
Delft, November 2018

Contents

1 Introduction

Big data is at the core of many different applications relevant to our society, for example, in healthcare [45] [55], finance [57], and gaming [22]. To address more diverse and sophisticated uses of big data, system designers are creating radically different systems designs. For example, Spark [62] emerged at the beginning of the 2010s, as a response to changes in the needs previously addressed by MapReduce [23] and related systems in the 2000s.

The difference between designs is significant, even radical. For example, Spark addresses the current need of many users to run their big data workloads on-demand, by building an ecosystem that runs commonly in small clusters of virtual machines (VMs), provisioned from clouds, and attached to remote-object storage systems. In contrast, the MapReduce and GFS [31] ecosystem champions statically deployed [32], tightly coupled physical clusters, with integrated storage.

The radical change in system designs is not arbitrary. Until the mid-2000s, large organizations alone could afford to operate large compute-clusters containing tens of thousands of computers, shared between the multiple organizational units; small- and medium-scale organizations could not easily access such resources. Through the advent of cloud computing, individual organizational units in both large and small organizations can lease resources on-demand from a cloud provider. Cloud computing has reduced the barrier to computation by enabling many to access significant compute resources for only a short period of time, at accessible cost. After overcoming initial performance-related [38] and technical challenges, cloud computing resources are now used in many fields, including for big data processing [36].

For big data processing, the use of clouds introduces many new parts, in contrast to traditional data processing. A key difference resulting from the transition between self-hosting and cloud computing infrastructure is the architecture used for *persistent storage*, where data gets stored and from which it is retrieved. In the cloud, a common storage medium for large amounts of data is the *object store* provided by the cloud vendor. Examples of storage sub-systems operating as object stores and available in the cloud include Amazon Web Services (AWS) S3, the Microsoft Azure Blob storage, and the Google Cloud Storage.

Given the change in system design, information on the characteristics and performance of these systems is scarce. In this work, we focus on the storage sub-system. We statistically characterize its usage. We design a generative model to tackle the scarcity of usage traces. We design a cache policy putting our insight from the characterization to work. Finally, we evaluate the performance of different cache policies for big data workloads via simulation.

1.1. System Model

We introduce in this section a system model for the operation of (Spark-based) big data workloads in the cloud. This model is inspired by deployments we have observed in practice across many organizations, in particular, for big data operations at Databricks. Figure 1.1 depicts the system model. We focus in this work on the workload of requests issued to the storage layer, between the system workers (component 4 in the figure) and the system storage (6).

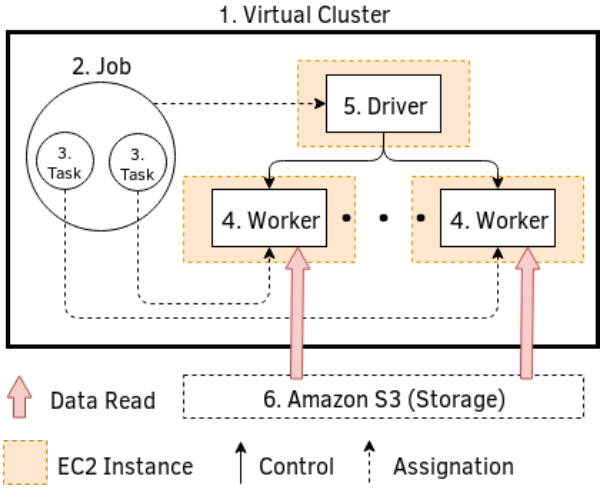


Figure 1.1: System architecture of a virtual cluster running on AWS. The workload analyzed in this thesis from virtual clusters operated by Databricks.

1.1.1. Workload Model

In our model, the workload consists of jobs arriving in the system as a *stream*. Jobs are either interactive and non-interactive. For interactive jobs, the arrival time is decided by the data analyst. For non-interactive jobs, e.g., batch or periodic, the system itself schedules when the job should run.

Each *job* (component 2 in Figure 1.1) is a unit of work that reads some input data from persistent storage, processes it, and produces an output. The output can be stored in persistent storage or directly displayed to the user.

A job is composed of at least two *tasks* (3), data generation (*read*) and data processing; other tasks, including tasks that produce data (*modification*), may also appear. Jobs have a directed acyclic graph (DAG) structure, with nodes representing tasks, and directed relationships between nodes representing dependencies (Ⓐ→Ⓑ means that task B cannot start before task A completes).

1.1.2. System Architecture

We model the class of systems analyzed in this work after a common architecture used by organizations around the world when using Spark or Hadoop in the cloud. This architecture is comprised of one or more *virtual clusters* running as sets of virtual machines (VMs) leased from a cloud provider, such as Amazon Web Services, and organized into a logical group using a virtual network. Virtual clusters get deployed on-demand—when needed, for as long as needed. Further, each virtual cluster is comprised of a set of core components, running on VMs obtained on-demand.

Figure 1.1 depicts the architecture of *one* virtual cluster. The virtual cluster corresponds to a single deployment of the data processing software, for example, the Databricks Runtime (components 4 and 5, explained later in this section). The virtual cluster is connected at runtime to a source of incoming data, from which it reads. In our model, a typical source of data is a cloud-based, persistent, *object-store* (component 6), for example, Amazon S3. If the processing results in data modifications, the virtual cluster is further connected at runtime to a data sink, for example, also Amazon S3.

A typical instance of this model appears at Databricks, whose Runtime system extends Apache Spark [62] with techniques to achieve high performance and usability. All references to Apache Spark in this work refer to the Databricks Runtime. With the Databricks Runtime, VMs are occupied by one or several Spark *workers* (component 4), and a *driver*. The single driver schedules jobs onto each worker. Workers process tasks, by performing the computation and I/O related to the task. This involves reading data from a persistent source or another worker, running computations such as maps and filters, and storing the data locally or to persistent storage.

1.1.3. Storage Workload Model

In our model, workers read data from the object store. All the requests from all jobs to the object store form the *workload* under study.

The object store is eventually consistent and has a *key-value interface*. It supports a hierarchy of *files* like a traditional file system by means of having a path as a key to an object. A directory structure can be achieved by having all the files in the directory have the directory name followed by a '/' as a prefix. A file can have several prefixes, simulating a hierarchy of directories. The files stored in an object store are much larger than the size of the key. It is possible to read only part of a file specifying a range of bytes to read. AWS S3 and Azure Blob Storage are examples of object stores. S3 is the store accessed in this workload.

Files are stored in several popular *formats*. Parquet[1], JSON, CSV, and Avro[2] are some examples. Some file types such as parquet make use of metadata to store column statistics and other index information. Databricks runtime takes advantage of the S3 ability to read specific byte ranges to read this metadata for efficient filtering and index traversal.

Files are compressed using popular compression schemes such as Snappy or Gzip. Some files, those formatted in CSV and such formats, are compressed as a whole. Other file formats, such as Parquet, have a higher granularity than whole file. For example, data in parquet is stored in row-groups in a file. In these cases, compression is applied at the row-group level. This makes row-groups self contained and readable without reading the whole file.

1.2. Problem Statement and Research Questions

It has been posited that ecosystems of complex interconnected systems exhibit behavior different than isolated applications running on individual servers. This has been labeled warehouse scale computing [10], and datacenter as a computer [54]. Understanding the behavior of such systems can uncover performance targets, bottlenecks, and opportunities. It has been found that *scheduling tasks take about 5% of all CPU cycles in a datacenter* [41]. This implies that scheduling of other tasks is considered important enough for allocation of such a large portion of resources. But, benchmarking, and profiling scheduling policies and algorithms at datacenter and ecosystem scale is hard and many times infeasible due to cost and operational concerns. Instead, *usage traces*, *their statistical analysis* and *their characteristics* help make the study of large scale scheduling systems feasible for a wide class of researchers and engineers, who might not have such systems available for research purposes. *Generative models* are useful for people with access to characteristics but not the original trace data to generate synthetic traces for experiments. Generative models also help study the evolution of workloads by modifying parameters. *Simulations* make large scale experiments affordable. Simulation results indicate performance of policies and algorithms in real-world systems, and can better direct real-world experimentation if that is at all possible.

Our systematic survey [43] of related work (details in Chapter 2) reveals that there is a dearth of existing research into characterization, modeling, and evaluation of Spark-based storage workloads. Reasons for this include: Spark-based data processing systems becoming business-critical, leading to companies being hesitant to share data; lack of organization-wide infrastructure to collect data due to proliferation of small clusters located remotely in the cloud; focus on optimizing the compute bottleneck and the increase in speed of storage devices.

We intend to provide computer systems researchers and engineers, particularly those interested in storage, the data and tools to design and experiment with systems for these use cases. Our main use cases are applications that use small Spark-based clusters in the cloud for data processing. We also intend to evaluate and extend existing research in cache policies for such systems because caching has historically proved beneficial for a large number of workloads including but not limited to web and database.

To this end, we strive to answer the research questions described below. The *impact* and our *process* of answering each question are also mentioned.

1. What are the characteristics of Spark-based big data storage workloads in the cloud?

   Impact: Enables new system designs [59], better tuning [33], and operations decisions [35].

   Process: Collection and statistical characterization of storage access trace from a prominent provider (Chapter 3).

2. How can the characteristics such workloads be modeled to generate synthetic traces?

   Impact: Enables realistic simulations and experiments when there is a lack of real world trace data [13] [15] [25].

   Process: Statistical reproduction of characteristics from original trace (Chapter 4).

[1] https://parquet.apache.org/
[2] https://avro.apache.org/

3. What is a principled approach to design an adaptive cache policy with few parameters?

   Impact: Novel estimator for the utility of an object being read.

   Process: Exploration of design space and overcoming a weakness of a policy recently proposed in a top publishing venue [11] (Chapter 5).

4. How do cache policies perform on such workloads?

   Impact: Enables the selection of better policies by system designers and operators [13].

   Process: Systematic comparison of cache policy performance on multiple workloads (Chapter 6).

1.3. Approach: Science, Engineering, and Design

In Chapter 3, we process hundreds of terabytes of data, aggregate data for trend analysis, and generate probability density functions for statistical characterization. This was a significant engineering and analytical effort. Our scientific contributions in this chapter are observations about long-term trends, statistical characteristics of reads, distribution of reads across sub-systems and filetypes, etc.

In Chapter 4, we design and engineer a system for fast concurrent generation of synthetic traces. The traces correspond to billions of reads. Apart from this, our scientific contribution includes the empirical comparison of fitting very large datasets to known probability distributions, and validation of generated synthetic traces.

In Chapter 5, we systematically design a cache policy that leverages the read density family of eviction metrics. We present a novel data structure called the approximate histogram for keeping track reads in different time intervals. We also overcoming a weakness of a recently proposed policy in a top venue.

In Chapter 6, we design and implement a system for distributed simulation of cache policies. Using this system, we compare systematically and comprehensively several recent cache policies and representative policies from the past. We also conduct an evaluation of the read density based eviction metric proposed in Chapter 5.

1.4. Chapter Structure

Chapters 3, 4, and 6 follow a similar structure for readability. First, we introduce the research question addressed in Section 1.2 which the chapter will answer. This is followed by a description of the process used to tackle it. Then, we describe the results of the process. The chapters each conclude with a discussion about threats to validity.

Chapter 5 on caching in storage systems proposes a reference architecture of a specialized cache policy, introduces a new cache policy coupled with a family of eviction metrics, and expands on implementation detail. This chapter ends with a discussion about possible threats to the validity of the reference architecture and our proposed design.

1.5. Additional Information

1. Part of this work was done while the author was an intern at Databricks bv, Amsterdam.

2. Chapter 3 has been submitted to the 10th ACM/SPEC International Conference on Performance Engineering (ICPE 2019), and is awaiting review. Authors: **Sacheendra Talluri**, Alicja Łuszczak (Databricks bv), Cristina L. Abad (ESPOL, Ecuador), and Alexandru Iosup.

3. The methods described in Chapter 3 and Chapter 4 are part of a new article, under submission at 19th Annual IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing (CCGrid 2019). Authors: Laurens Versluis (VU, Amsterdam), Roland Matha (University of Innsbruck, Austria), **Sacheendra Talluri**, Radu Prodan (University of Klagenfurt, Austria), and Alexandru Iosup.

4. The work presented in Chapter 4, and the work present in Chapters 5 and 6, is part of new articles about to be submitted to high-quality conferences.

2

Related Work

We conduct a systematic survey [43] of 18 high quality venues in large-scale systems (HPDC, SC, etc.), systems and operating systems (NSDI, OSDI, etc.), and performance (SIGMETRICS, ICPE, etc.). Because the common type of storage studied in this work, object stores hosted in the cloud, is relatively new, we include in our systematic survey only work published since 2010. For design and evaluation of cache policies, we also compare with prior work which has been cited by publications studied in our systematic survey.

In this chapter, we first describe the process of systematic survey. Then, we analyze the findings as they are related to each of the chapters.

2.1. Systematic Survey

This section details how the papers to be included in the survey were chosen. We use a list of curated keywords and the publicly available DBLP dataset to find papers. The list of keywords, more accurately key phrases, was compiled by looking at the results of a Google Scholar search for terms such as "storage tiering" and popular papers in the field. The keywords are listed in Table 2.1.

Table 2.1: List of keywords queried.

1. Storage Scheduling	2. Storage Workload	3. Parallel file system	4. Cloud Storage
5. Heterogeneous Storage	6. Storage Model	7. Distributed file system	8. Large Array
9. Tiered Storage	10. Cache	11. Parallel I/O	12. I/O
13. Hadoop Storage	14. Data Placement	15. Storage Characterization	16. Distributed Storage
17. HDFS	18. Massive Array	19. Storage Provisioning	

These keywords were searched for in the titles of all papers published in the conferences listed in Table **??** and all associated workshops between 2010 and 2017.

Table 2.2: List of conferences queried.

1. NSDI	2. OSDI	3. HPDC	4. SC
5. SOSP	6. SOCC	7. SIGMETRICS	8. VLDB
9. EuroSys	10. CCGrid	11. IEEE Cluster	12. ICPE
12. MASCOTS	13. IEEE BigData	14. EuroSys	

All the keywords excluding "I/O" were searched for in the titles of all papers published in the conferences, associated workshops and journals listed in Table 2.3 between 2010 and 2017.

Table 2.3: List of additional conferences queried.

1. FAST	2. MSST	3. TOS	4. SIGMOD

The publicly available DBLP dataset[1] was loaded into Elasticsearch[2] and was used for search. Several papers were unrelated to topic, extension of same paper published in another place, etc. These were excluded. The process resulted in around 140 relevant results.

2.2. Characterization

We compare our characterization work (Chapter 3) with previous studies on real-world, large-scale storage workloads, including big data. Overall, because of our focus on object stores, our work complements the body of work done on hardware-level storage for big data workloads, e.g., [52]. Table 2.4 summarizes the results of our systematic survey which are concerned with characterization. The table compares different studies across the following dimensions: popularity of files, time dependence, interarrival times, read sizes, levels of abstraction studied, popularity of different systems in an ecosystem and types of operations. We find that the characteristics we investigate are comparable to those investigated by recent works in the field.

Table 2.4: Comparison of our work with previous characterization studies, ordered chronologically.

	Type	Pop.	TimeDep.	Interarrival	Size	Levels	SysPop.	Ops.
Chen 2011 [17]	Enterprise						✓	✓
Carns 2011 [16]	HPC	✓			✓			
Abad 2012 [1]	MapReduce + HDFS	✓	✓	✓	✓			✓
Chen 2012 [18]	MapReduce + HDFS	✓	✓		✓			
Atikoglu 2012 [7]	Web Cache	✓		✓	✓			✓
Liu 2013 [46]	Consumer Cloud				✓		✓	✓
Harter 2014 [35]	Messaging + HDFS	✓		✓	✓			
Gunasekaran 2015 [34]	HPC	✓			✓			
Summers 2016 [58]	Video Delivery				✓		✓	
This Work	**Spark + S3**	✓	✓	✓	✓	✓	✓	✓

Closest to our work we find the previous work by Abad et al. [1] and by Chen et al. [18] on big data workloads that read from a distributed storage system. Our work complements these studies with analysis of long-term trends, time-dependence, bursts, stationarity, and distribution across clusters. Our work also extends and supports the characterization of file popularity, interarrival time, reuse time, and file formats.

2.3. Generative Modeling

We compare our modeling work (Chapter 4) with previous work on storage systems modeling. Because of our focus on storage workload, our work complements the body of work done on modeling compute workloads in big data processing, e.g., [60]. It also complements work on system performance modeling, e.g., [50]. Table 2.5 summarizes the results of our systematic survey which are concerned with modeling. The table compares different studies across the following dimensions: fitting of heavy tailed distributions, concurrent workload generation, performance analysis and comparison of different curve fitting methods. We find that the features of our work are comparable to other recent work in the field. A unique feature of our is that we compare curve fitting technique for goodness of fit, and demonstrate that maximum likelihood analysis is not suitable for very large datasets such as ours.

Table 2.5: Comparison of our work with previous modeling studies, ordered chronologically.

	Type	Curve Fitting	Concurrency	Performance	Fit Compare
Delimitrou 2011 [24]	Database		✓		
Atikoglu 2012 [7]	Web Cache	✓			
Abad 2013 [2]	MapReduce + HDFS	✓	✓	✓	
This Work	**Spark + S3**	✓	✓	✓	✓

The works closest to ours are Atikoglu et al. [7] and Abad et al. [2]. Both fit observed features of workloads

[1] https://dblp.uni-trier.de/xml/
[2] https://www.elastic.co/products/elasticsearch

to heavy tailed distributions. Abad et al. even models HDFS reads, which is a big data processing workload.

Older modeling work uses Poisson and Markov models. These are not suitable for modeling storage workloads as they assume independence between reads which doesn't hold for real workloads [63]. We also show the existence of long term time dependence between reads for our workload in Chapter 3. Modeling using heavy tailed distributions takes into account this time dependence.

2.4. Cache Policy Design

Caching is a well studied problem with a long history. We trace the lineage of concepts used in our cache policy. We also compare our work to recent studies on cache policy design.

LIRS [40] was one of the first algorithms to use reuse distance as a cache policy metric. Segmented LRU [42] was one of the first to use acandidate cache. ARC [47] also divides the cache into segments, but the reason for segmentation is to optimize the size of cache for the policy used by each segment and not collecting information and buffering rarely accessed items as done in Segmented LRU and our work. S4LRU [37] extends the concepts of LRU to multiple candidate caches. AdaptSize [13] uses an admission policy but its paramaterized on the intrinsic feature of object size rather than any extrinsic one like popularity or interaccess time. Hyperbolic caching [15] also uses a tiny candidate cache and divides popularity by time. LRU-K [51] remembers the K most recent reads of each object. The concept of using multi approximate data structures in storage related scenarios has been introduced in counter stacks [61]. Multiple approximate data structures to keep track of histograms has been recently used in databases [29].

Several policies take into account factors such as size of the object, latency and importance. Our work complements such works. These costs can be incorporated into equations 5.5 and 5.6 from Subsection 5.5.3. Our work also complements studies whose cost functions take factor such as latency and fairness into account [14] [3].

W-TinyLFU [25] and LHD [11] are the most closely related works to this one. W-TinyLFU uses a candidate cache and a main cache with the TinyLFU admission policy. It also uses approximate counters. Our eviction policy for the main cache also based on approximate counter takes into account reuse time and is novel. LHD takes into account reads at different reuse times. For each reuse time category, object are further categorized based on features such as application name and reuse time of last read. There is little reason why those reasons are good for categorization. In many caching applications, little other features than those that can be directly computed by the cache are available. Using approximate counters instead of categories sidesteps the whole decision of choosing good categories and thus has less parameters to tune.

In our policy design, we make explicit the different components of cache policies and decisions taken about each of them in a principled manner. This is similar to the approach taken to deconstruct scheduling policies in [6].

2.5. Cache Policy Evaluation

There are no recent exclusive evaluations of cache policies. They are paired with either a new policy design or a characterization of a workload. We summarize the type of workloads used in such recent studies in Table 2.6. Most evaluations seem focused on workloads related to the Internet. Web refers to caches in servers like nginx or application caches like memcached. Database refers to SQL databases used in web applications. Workstation refers to personal files of users on local disk or on a server.

Table 2.6: Workloads used in cache policy evaluations, ordered chronologically.

	Workloads
Huang 2013 [37]	Social Media
Berger 2017 [13]	Web
Blankstein 2017 [15]	Web, Database, Workstation
Einziger 2017 [25]	Web, Database, Workstation
Beckmann 2018 [11]	Web, Database, Workstation
This Work	**Big Data**

This is the first study to evaluate the efficacy of cache policies for big data workloads. We use 2 traces for this. One is from Databricks. This is the trace that is characterized in Chapter 3. The other are the public

Yahoo Webscope 3 traces.

3

Characterization of a Big Data Storage Workload in the Cloud

Motivated by the importance of Spark-based systems processing big data in the cloud (Described in Chapter 1), by the novelty introduced by their cloud storage sub-systems, and by the scarcity of information publicly available about Spark storage workloads, in this chapter we endeavor to answer the following research question: *What are the characteristics of Spark-based big data storage workloads in the cloud?*.

3.1. Overview

Toward answering the research question on which this chapter focuses, our contribution is four-fold:

1. We collect and process long-term workload traces from a relevant Spark deployment, at Databricks (Section 3.2). Our data spans over 6 months of operation, resulting in over 600 TB of log data. We devise a method for pre-processing and for the statistical analysis of these traces.
2. We analyze the long-term trends (Section 3.3). We focus on two key I/O operations, reads and modifications. We investigate if diurnal and weekly patterns occur, if long-term patterns emerge, and if reads and modifications occur with relatively similar frequency.
3. We analyze statistically the read operations (Section 3.4). (We focus on reads because we find that modifications are relatively rare.) We study if heavy-tails and burstiness appear in the distributions of number and size of reads.
4. We analyze statistically the popularity of clusters and file types (Sections 3.5 and 3.6). We investigate if the clusters deployed on-demand are similarly used, and if the big data file formats and compressing schemes are similarly popular.

3.2. Process for Data Collection, Processing, and Analysis

The goal of our analysis is to gain a statistical understanding of the series of accesses to the object store by various Spark-based applications deployed in the cloud. The operational data corresponding to Spark-based storage workloads presents many collection, processing, and analysis challenges. Monitoring many small virtual clusters across organizational boundaries is challenging, which raises a complex data collection challenge. Pre-processing the data needs to balance preserving meaningful information, with the need to protect the anonymity and corporate information. Analyzing the rare information raises the same challenge facing basic research—finding a balance between the breadth and depth of explored concepts. Addressing these challenges, in this section we introduce a process to collect, process, and analyze storage-workload data about cloud-based Spark operations.

3.2.1. Overview of the Process

Our process for cloud-based Spark operations consists of large-scale data monitoring, done concurrently with data pre-processing and preliminary analysis, followed by in-depth analysis.

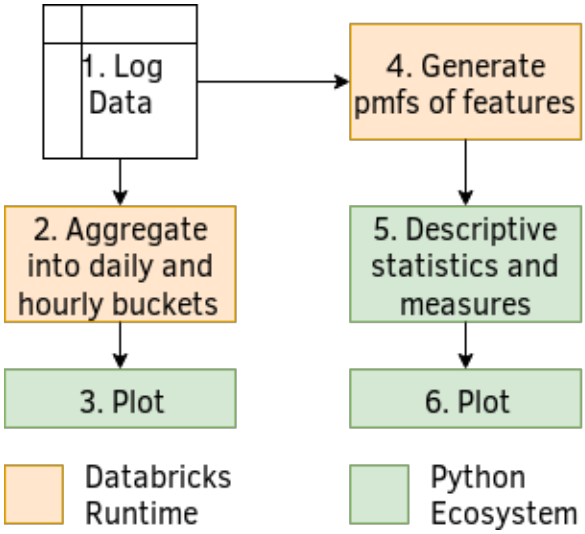


Figure 3.1: Characterization workflow.

Figure 3.1 depicts the core of our process. Log Data (component 1 in the figure) performs *log collection*. It continuously collects monitoring data and stores it into logs ready to be parsed (queried). To achieve this goal, this component must be highly reliable and redundant, stream monitoring data in and process it rapidly, use batch processing to compact and store data, and generate indexes to make querying tractable. Although for this component we rely on the Databricks Runtime system out-of-the-box, such features are provided by many of the monitoring systems based on the Apache ecosystem, e.g., using Kafka as communication pipeline, a streaming platform such as Apache Storm or Spark, and a big data platform such as MapReduce or Spark; the indexing, high availability, and redundancy features require domain-specific engineering.

For this work, the operations (*accesses*) recorded in the log are *reads* and *modifications*. Modifications are namespace modifications performed by the runtime, i.e., create, delete, bulk delete, create directory, and rename operations. Overall, the analysis focuses on *features*. A feature is a property of the access that can be computed by virtue of the item being accessed, such as popularity, or by virtue of its relative position to another (previous) access, such as interarrival time. These features can take on different values called *variates*.

Following log collection, the process branches into the analysis of long-term trends and the analysis of current features, both of which are conducted periodically in parallel with the log collection process, but with frequency under the control of the system analyst. (This assumes the system operator collects monitoring data at much higher frequency than the analyst needs updated results, which is typically the case; in practice, a re-analysis of monitoring data could occur every day for business decision.)

For the *analysis of long-term trends*, the raw monitoring data is both too large (big data) and includes sensitive information. We thus first pre-process it into a *compact format* and *apply a normalization step*, which we describe in Section 3.2.3. Then, we perform a typical long-term analysis, observing the evolution of the number of reads and modifications, and of the sizes of each of these operations (component 2 in Figure 3.1).

For the *feature analysis*, we use a diverse set of statistical tools on selected features: we compute the empirical probability density function (*EPDF*) and the empirical cumulative distribution function (*ECDF*), descriptive statistics, the Hurst parameter for long-term dependence, etc. We describe the key elements of this analysis in Section 3.2.4.

A key feature of our process is that it is primarily aimed at human analysis, for which it always concludes its operational branches with visualization (*plotting*)—components Plot (3 and 6 in Figure 3.1).

3.2.2. Log Collection

Table 3.1 summarizes the logs collected for this work. All accesses are timestamped, with timestamps were recorded at the time of logging as time since Unix Epoch, in milliseconds. Overall, the traces correspond to the combined workloads of organizations spanning healthcare, manufacturing, web services, and advertising. (We address the bias inherent to these traces in Section 3.7.)

Trace `Full` (F) collects all data accesses, both reads and modifications, over the entire 6-month period. We

Table 3.1: Format of a pre-processed log line.

hashed file path	hashed cluster id	hashed worker id	size	timestamp	operation

Table 3.2: Information about the analyzed logs.

Name	Period	Log Data Size
F	6 months	600TB
W1	1 week	15TB
W2	1 week	30TB

use F to analyze long-term trends. Collecting this data results in a massive log; the F trace exceeds 600 TB. In addition to the full workload, we collect over shorter periods of time more detailed data, useful for computing the histograms of features such as popularity and interarrival time. We thus collect one-week traces `Week 1` (W1) and `Week 2` (W2). The size of these smaller traces was of 15 TB and of 30 TB, respectively.

3.2.3. Pre-Processing

The collected logs were pre-processed into a compact format for analysis, which we depict in Table 3.1. Any data fields in the logs that are unrelated to our analysis were removed. The cluster and worker identifiers were originally strings, which both go contrary to the privacy needs of Databricks, and increased the storage, memory and computation costs. They were hashed using Murmur3 hash, and stored the most significant 64 bits of the 128-bit numbers of the hash. Thus, we had no access to any identifiable information during the analysis. Murmur3 was used for its performance and uniform distribution over the key space.

All features have been normalized by dividing the variates by a constant normalizing factor. Thus, the absolute values seen in this work are not real. This was done to keep the popularity and costs of the company a secret. However, the relative difference between variates of a feature and across features is still true and remains applicable. It is contracted by the value of the normalizing factor. The normalization does not effect the empirical probability density functions as the fraction of items that belong to a particular bin remains the same. Some descriptive statistics like the mean are reduced by the normalizing factor amount. Others like tail weight and the distances between W1 and W2 are not.

3.2.4. Statistical Analysis

For ECDF plots, in some cases, we use a *symlog* axis instead of a pure logarithmic axis. We do this when the 0 variate is important to the plot. A symlog axis is a logarithmic axis where the are regions close to 0 are on a linear scale. Thus, we avoid the issue of $log(0)$ being undefined.

In many plots, such as Figure 3.6, we depict two similarly shaped curves. Each such curve corresponds to one the two different one-week-long periods mentioned in Table 3.2. For these figures, the *purple curve represents the data from W1; the green one, the data from W2*.

We observe that even though there is a large increase in the accesses from January to May, the general distribution does not change significantly. To quantify this claim, we use two tests to measure the similarity between two empirical distributions: Kolmogorov-Smirnov (KS) test [20] and Pearson's χ^2 test [56]. The KS test measures the maximum difference between the two cumulative distribution functions. We use the two-sample KS test, which measures the maximum difference between two empirical cumulative distribution functions. We chose this because it is easy to understand and the reader can visually see the quantitative distance output by this test in the graphs. Another reason is that it is distribution independent, unlike the Anderson-Darling test. Thus, we do not require critical value tables to measure goodness of fit. This is useful as we do not know the underlying distributions of the samples we have. The result of the KS test is used as an indicator of divergence. The χ^2 test measures the difference between histograms of two empirical distributions. i.e., it measures the difference in the probability at each variate. This was also chosen to be easy to understand and gives a complete view of the distribution instead of just at the location where the difference is maximum. This helps identify minor differences.

In some cases, we use more than one trace in the same part of the analysis. Because the variates for these cases may be different, we rebin the data into 100 logarithmic bins so that the distributions can be compared. Where rebinning without interpolation is not possible, we use 10 logarithmic bins or linear bins. The KS statistic is not sensitive to the number of bins. The χ^2 statistic is very sensitive to the number of bins

chosen. Increasing the number of bins by an order of magnitude increases the χ^2 statistic by an order of magnitude, especially when it is small. However, the p-value doesn't change, making it a valid test. We use the implementation of χ^2 from the SciPy scientific computing library, version 1.1.0.

For each empirical distribution presented, we also present descriptive statistics. This includes the median, mean and standard deviation which are widely known. We also provide additional descriptive statistics. To quantify the dispersion of the data, we use the Coefficient of Variation (CV), which is the ratio of the standard deviation to the mean, and the Inter Quartile Range (IQR), which is the difference between the value at 75th percentile and 25th percentile. **Tail weight** is the sum of magnitudes of those elements which are in the 99th percentile as a fraction of the sum of magnitudes of all the elements.

For features where it is relevant and informative, the Hurst parameter is used to estimate long term dependence [27]. It specifies if the value at a certain time would be higher, lower or randomly distributed based on the previous values. A Hurst parameter below 0.5 indicates a tendency of the series to move in the opposite direction of the previous values. i.e., highs are followed by lows. Thus, appearing to have high jitter. A value of 0.5 indicates random Brownian motion and above 0.5 indicates a tendency towards well defined peaks. We use the rescaled range (R/S) method to estimate the Hurst parameter.

3.3. Analysis of Long Term Trends

We highlight the following long term trends:

MF3.3.1 The number of reads and bytes read per day have doubled over 6 months.

MF3.3.2 The number modifications per day has remained at the same level throughout the analysis period.

MF3.3.3 Both reads and modifications follow a diurnal pattern.

MF3.3.4 Large imbalance in number of reads and bytes read per hour occur on *daily* and *weekly* basis.

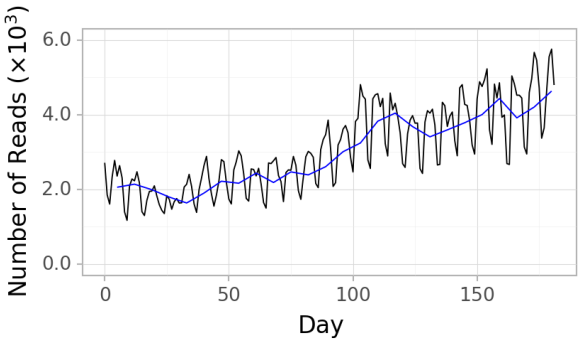
MF3.3.5 There are 2 orders of magnitude less modifications happening than reads.

MF3.3.6 Most modifications are file creations.

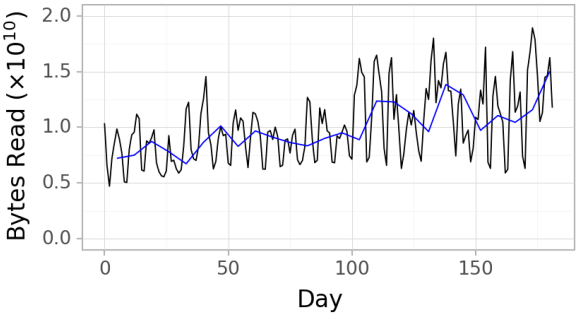
3.3.1. Reads

Table 3.3: Descriptive statistics about number of reads and data read *per day*.

	total	mean	median	std. dev.
num. reads	5.57×10^5	3.06×10^3	2.83×10^3	1.11×10^3
bytes read	1.84×10^{12}	1.00×10^{10}	9.60×10^9	3.18×10^9



(a) Number of reads per day with weekly mean as the blue trend line.



(b) Bytes read per day with weekly mean as the blue trend line.

Figure 3.2: Number of reads and total amount of bytes read *per day* over a period of 6 months.

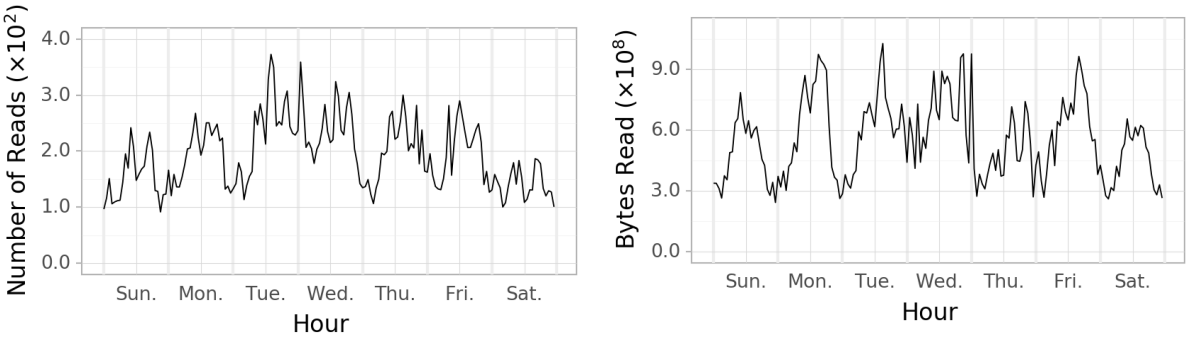
We analyze the trends in number of reads and in bytes read, and find both increase significantly over the period we analyze (**MF3.3.1**). Figure 3.2 depicts the number of reads and bytes read per day over a 6

month period. Excluding the variation between days of the same week, the number of reads (Figure 3.2a) has increased from about 2.0×10^3 to over 4.0×10^3. We observe a similar phenomenon for number of bytes read (Figure 3.2b). We conjecture that this is due to a combination of more users using the subset of ecosystem under study and existing users increasing their usage.

The descriptive statistics for the metrics "number of reads per day" and "bytes read per day" are summarized in Table 3.3. The high standard deviation indicates that days which experience significant higher or lower number of reads and bytes read occur. The similar values of mean and median indicate that the deviation from the mean is evenly distributed across low activity and high activity days.

Table 3.4: Descriptive statistics about number of reads and data read *per hour* for a selected week.

	total	mean	median	std. dev.
num. reads	3.25×10^4	193.4	193.1	60.6
bytes read	9.36×10^{10}	5.57×10^8	5.57×10^8	1.95×10^8



(a) Number of reads per hour.

(b) Bytes read per hour.

Figure 3.3: Number of reads and total amount of bytes *per hour* over a period of one week.

We analyze the number of reads per hour and bytes read per hour over a period of one week, and find that significant load imbalances occur on a *weekly* and *daily* basis (**MF3.3.4**). Figure 3.3 depicts the number of reads and bytes read over a 1 week period. The number of reads (Figure 3.3a) changes from day to day. It peaks on Tuesday and bottoms out on Saturday. This is the same pattern that is also visible as variation in Figure 3.2. The number of reads also vary on a hourly basis, with the peak occurring during noon GMT and the period of least activity around midnight GMT (**MF3.3.3**). This is a diurnal pattern. The ecosystem subset studied was composed of North American users. This implies that they were most active in the mornings. Both the aforementioned variations (weekly and diurnal) also occur in number of bytes read (Figure 3.3b). A likely hypothesis is that a lot of jobs contributing to the peak are interactive jobs users submit during working hours.

The observation that there is an imbalance of usage during the week can prompt organizations to schedule their workload to fall on less busy days. In the same day, jobs can be scheduled during less busy hours of the night. This helps organizations take advantage of spot auction markets on EC2, to lower their compute costs, derived from the assumption that the spot market costs would be lower if there are fewer users bidding for the same compute resource.

3.3.2. Modifications

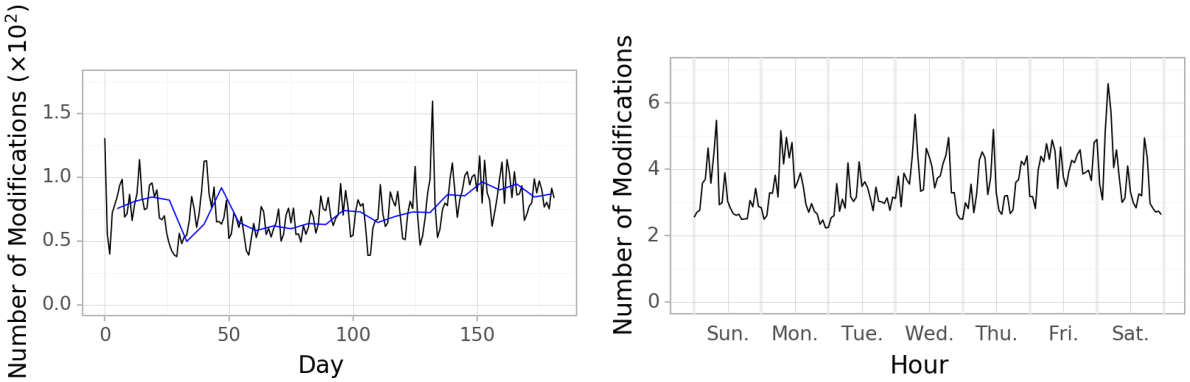
Table 3.5: Descriptive statistics about number of modifications *per day* over 6 months and *per hour* for a selected week.

	total	mean	median	std. dev.
per day	1.37×10^4	75.03	64.23	19.19
per hour	594.3	3.5	3.4	0.8

We analyze the number of modifications per day, and find that the number of modifications has remained

about the same throughout the analysis period (**MF3.3.2**). Figure 3.4a shows the number of modifications per day. Excluding the variation between days of the same week, the number of modifications remains at approximately the same level. Figure 3.4b depicts the number of modifications per hour. A diurnal pattern is readily apparent (**MF3.3.3**).

Table 3.5 presents the descriptive statistics characterizing the number of modifications per day and per hour. The total and mean number of modifications are two orders of magnitude lower than the number of reads in Tables 3.3 and 3.4 (**MF3.3.5**). Table 3.6 presents the distribution of number of modifications across operation types. The number of creation operations is much higher than any other operation (**MF3.3.6**).



(a) Number of modifications *per day* over 6 months with weekly mean as the blue trend line.

(b) Number of modifications *per hour* over 7 days.

Figure 3.4: Modifications trend.

Table 3.6: Types of modifications over the 6-month period.

Create	Delete	DeleteMultiple	MkDirs	Rename
1.15×10^4	2.13×10^3	2.28	41.89	28.47

3.4. Statistical Analysis of Reads

Reads constitute a large majority of this workload and are the type of accesses that are frequently optimized through caching, tiering or other techniques to improve system performance. It can be seen from the long term trends in Section 3.3 that reads overwhelmingly dominate the workload. This can also be inferred analytically, as big data processing (this workload) is concerned with reading large amounts of data and processing it in different ways to gain valuable insight. In this section, we focus on the statistical properties of reads.

A single read has several features which we look at: the file that is being read, popularity of the file, size of the read, interarrival time and reuse time. Our major observations are:

MF3.5.1 The number of reads and bytes read exhibit negative long range time dependence.

MF3.5.2 All features (size, popularity, etc.) have a heavy tail.

MF3.5.3 Reads occur in bursts.

MF3.5.4 The distributions of features are stationary over long time periods.

3.4.1. Count

We analyze the number of reads per time window, and find evidence of inverse long range time-dependence (**MF3.5.1**), and bursty behavior (**MF3.5.3**). We estimate the decay of this dependence using the Hurst Parameter for different window sizes in Figure 3.5. The Hurst parameter is particularly relevant to stationary time series, and we assume that the series of reads is stationary at small intervals.

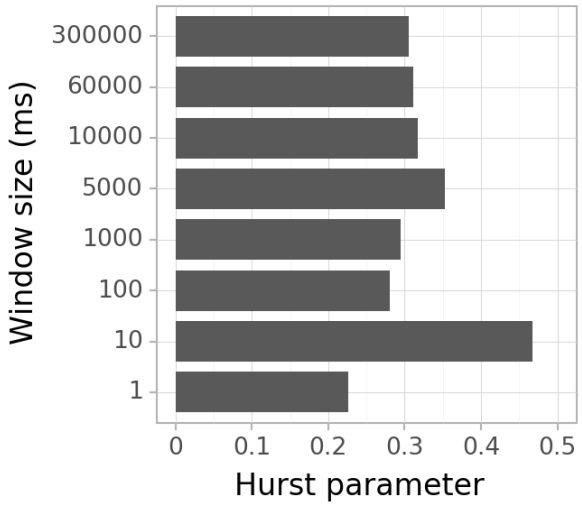


Figure 3.5: Hurst parameter estimation for number of reads in a time window.

The period we look at is one thousand times the window size for every window size; for example, if the window size is 10 milliseconds then we look at a period of 10 seconds. The Hurst parameter is lower than 0.4 for all window size, except one. This is evidence of inverse long term time-dependence. That means that over a specific period, the number of reads might increase or decrease but generally falls back towards the mean. This contrasts other studies of time dependence in big data workloads which present a positive long range time-dependence [1].

A burst is defined as a period of high number of reads (high activity) surrounded by a period of low number of reads (low activity). Negative long range time-dependence necessarily implies that periods of low activity are followed by high activity and vice versa. Thus, periods of high activity are surrounded by periods of low activity, satisfying the definition of a burst. Therefore, number reads exhibits bursty behavior.

3.4.2. Read Sizes

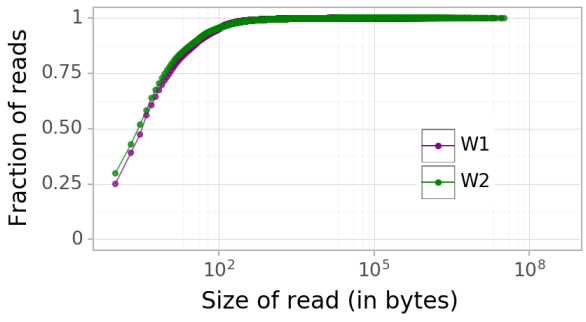


Figure 3.6: ECDF Read size vs. fraction of reads with logarithmic horizontal axis.

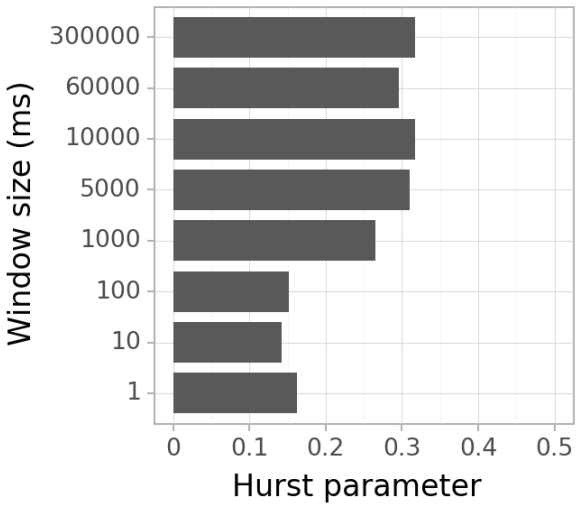


Figure 3.7: Hurst parameter estimation for bytes read in a time window.

The read size refers to the number bytes transferred from the cloud storage to the local machine per request. While checking for long term dependence, it refers to the number of bytes transferred in a time window. We analyze read sizes, and find evidence of inverse long range time-dependence (**MF3.5.1**), a heavy tail

(**MF3.5.2**), bursty behaviour (**MF3.5.3**), and stationarity (**MF3.5.4**). Figure 3.6 depicts an empirical cumulative distribution of read sizes with logarithmic horizontal axis.

Visually and also from the descriptive statistics in Table 3.7, we observe that most requests read little data with the median request reading 3 or 4 bytes. Most of the bytes transferred are due to reads of large items. This is most visible in the "tail" item in the descriptive statistics. That shows that more than 80% of the data transferred in both periods is due the 99th percentile of requests sorted by size. We conjecture that this is a result of a large number of requests being *metadata reads*. They take advantage of the S3 feature which allows one to read certain range of bytes in a file instead of the whole file. These reads can be used to read file headers and assorted metadata and make decisions, such as to read the file or not. Thus, metadata reads account for most reads while data transfer for processing accounts for most of the data transferred.

The read size is small compared other studies with 90% reads smaller than 1KB. In other studies 90% of reads are smaller than 1KB in web caches [7], 100KB in HPC [16], 1MB in HPC [34], 4MB in consumer cloud [46], 10MB in video delivery [58] and 15MB in HBase [35].

Figure 3.6 depicts that distributions of the two traces W1 and W2 are not too different. The similarity has been quantified in Table 3.8. The variates were rebinned into 10 logarithmic bins for this purpose. We know from Figure 3.2 that the number of reads and bytes read has changed. This indicates that magnitude of number of reads or total data read has minor influence on the distribution of read sizes. Thus, the distributions remain stationary over long time periods.

We use the Hurst parameter to estimate the time-dependence of the series at different time scales, as done for counts. For every time window the sum of the bytes read by all reads in that window is considered as the value of the window. Figure 3.7 indicates that for all considered window sizes the Hurst parameter was estimated to be less than 0.35. The value is low, indicating an inverse correlation between subsequent values and a tendency of the series to fall back towards the mean—more so for small windows. This leads to periods of high activity surrounded by periods of low activity. Hence, bursty behaviour.

Table 3.7: Descriptive statistics about read size.

Period	median	mean	std. dev.	CV	IQR	tail weight
W1	4	100	31,535	316	11	0.84
W2	3	136	54,191	400	9	0.89

3.4.3. Popularity

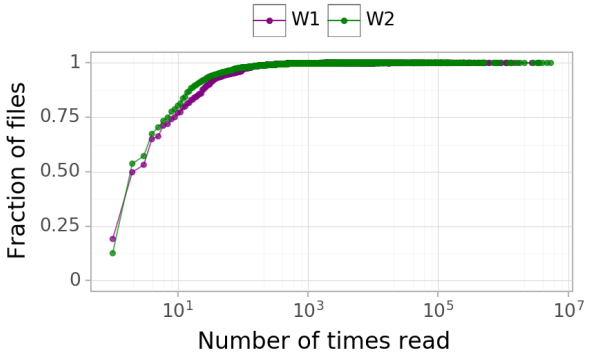


Figure 3.8: ECDF of Popularity vs. number of files with that popularity with logarithmic horizontal axis.

Table 3.8: Quantifying similarity of size distributions during two periods.

KS distance	χ^2 distance	χ^2 p-value
0.03	0.01	1.0

The popularity of files is a measure of the number of times a file has been accessed. We analyze the popularity of files, and find a heavy tail (**MF3.5.2**) and stationarity (**MF3.5.4**). We compute popularity of files from hashed file paths in the anonymized data. Figure 3.8 depicts the empirical cumulative distribution function of popularity of files with a logarithmic horizontal axis. The descriptive statistics of popularity of files are presented in Table 3.9.

Visually and also from the descriptive statistics in Table 3.9, we observe that most files are not popular with a median and mean of 3 and 18. From the tail weight statistic, 0.37%-0.44% of reads are to the 99th percentile (1%) of files. But, 90% of reads are due to 70th percentile (30%) of files. This matches the observation in [1] that 90% of reads are due to 71st to 78th (29% to 22%) percentile. The tail of popularity of files is less heavy than read size, but it is heavy nonetheless.

Figure 3.8 depicts that distributions of the two traces W1 and W2 are not too different. Their similarity is quantified in Table 3.10. The variates were rebinned into 100 logarithmic bins for this purpose. We know from Figure 3.2 that the number of reads is higher during the second period. This is evidence towards the stationary nature of this distribution of popularity across a sufficiently large period, even if the magnitude of number of reads changes.

Table 3.9: Descriptive statistics about popularity.

Period	median	mean	std. dev.	CV	IQR	tail weight
W1	3	18	355	20	7	0.37
W2	2	16	615	40	6	0.44

Table 3.10: Quantifying similarity of popularity distributions during two periods.

KS distance	χ^2 distance	χ^2 p-value
0.07	0.2	1.0

3.4.4. Interarrival Times

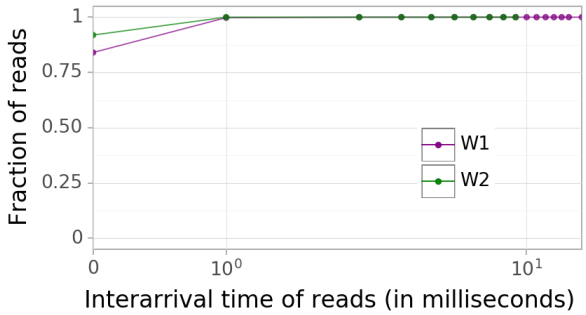


Figure 3.9: ECDF of interarrival time vs. number of reads.

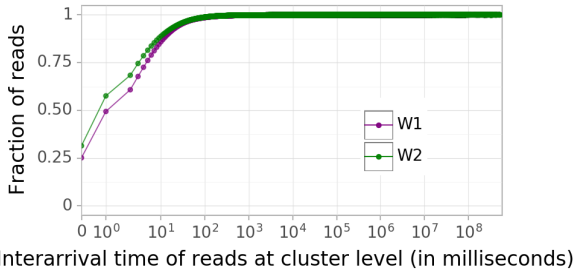


Figure 3.10: ECDF of interarrival time at cluster level of all clusters vs. number of reads with logarithmic horiz. axis.

Interarrival time of a read is the time elapsed between the read and the previous read. We analyze interarrival times, and find near 0 values as the ecosystem level.

Large ecosystems such as the one at Databricks do not operate as a single monolith; i.e., it is not just one large compute system making requests to a storage system. These ecosystems consist of numerous systems which themselves contain multiple sub-systems. As mentioned in Section 1.1, the ecosystem at Databricks consists of numerous Spark clusters, each of which is further composed of multiple workers. Storage related operations such as metadata management and caching operate in these layers and not globally. Storage accesses from different clusters and from different workers may not exhibit the same behavior as that of the whole system. Thus, we study interarrival time behavior at the whole ecosystem level and the level of individual virtual clusters.

Figure 3.9 depicts the empirical cumulative distribution function of interarrival times with linear axes. From the figure and the descriptive statistics in Table 3.11, an overwhelming number reads have a 0 interarrival time, when measured with millisecond precision. The number of reads falls precipitously with increasing interarrival time, highlighting the high frequency of reads in the system.

Figure 3.9 depicts the ECDFs for W1 and W2 having a similar distribution. This similarity has been quantified in Table 3.12. The first six categories of the histograms were considered for this quantification. The reads are more frequent (smaller mean interarrival time) in May than they were in January. This is a consequence of a higher total reads by the system as presented in Section 3.3.

Table 3.11: Descriptive statistics about interarrival of reads.

Period	median	mean	std. dev.	CV	IQR	tail weight
W1	0	0.16	0.38	2.32	0	1.0
W2	0	0.08	0.27	3.38	0	1.0

Table 3.12: Quantifying similarity of interarrival time distributions during two periods.

KS distance	χ^2 distance	χ^2 p-value
0.08	0.12	0.99

We define *interarrival time at cluster level* to be the time difference between a read and a previous read from the same cluster. We analyze interarrival times at the cluster level, and find a heavy tail (**MF3.5.2**), bursty behavior (**MF3.5.3**) and stationarity (**MF3.5.4**). Figure 3.10 depicts the empirical cumulative distribution function of this feature. Unlike interarrival times at the ecosystem level, where more than 75% reads had 0 interarrival time, the fraction of reads with 0 interarrival time is only around 25% at the cluster level.

The descriptive statistics for this feature are presented in Table 3.13. The median interarrivals of 2 and 1 indicate that reads are still frequent at this level of granularity. Nevertheless, there are long periods with no reads as evidenced by the high mean value and the extremely high standard deviation, which is over 1000 times larger than the mean. Thus, there are reads which occur very close together and there are long periods of no activity. This is again evidence of burstiness.

Figure 3.10 depicts the similarity between distributions for W1 and W2. This similarity is quantified in Table 3.14. Thus, the distributions remain stationary over long time periods. The variates were rebinned into 100 logarithmic bins for this purpose.

Table 3.13: Descriptive statistics about interarrival times at cluster level.

Period	median	mean	std. dev.	CV	IQR	tail weight
W1	2	136	127,558	936	5	0.96
W2	1	95	104,626	1101	4	0.95

3.4.5. Reuse Times

Reuse time refers to the time elapsed between two reads to the same file. We analyze the reuse times, and find a heavy tail (**MF3.5.2**), and stationarity (**MF3.5.4**). Figure 3.11 depicts the empirical cumulative distribution function of reuse times. The histogram of reuse times had over 90 million categories. This is interesting as other features such as size and popularity had histograms with multiple orders of magnitude fewer categories. This shows that reuse times are dispersed over the number line with little grouping behaviour when considered at the millisecond scale. There is also a steep increase in reads with reuse time between 10 and 100 milliseconds.

The descriptive statistics for reuse time are quantified in Table 3.15. The median reuse time is high at 2720ms for W1 and 3042ms at W2 compared to the median interarrival times of 0-4 observed. Therefore,

Table 3.14: Quantifying similarity of cluster level interarrival time distributions during two periods.

KS distance	χ^2 distance	χ^2 p-value
0.08	0.03	1.0

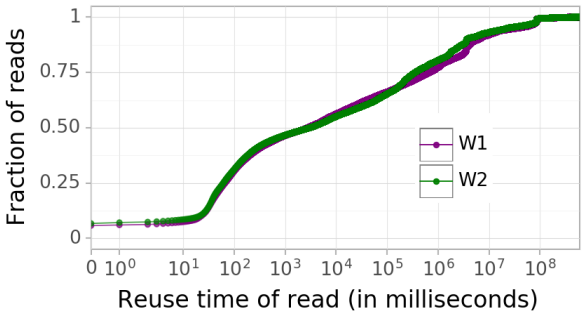


Figure 3.11: ECDF of reuse time vs. number of reads with logarithmic horizontal axis.

reads of the same file are necessarily interspersed with reads of other files most of the time. The tail weight of 0.30 corresponds to a heavy tail.

Figure 3.11 depicts the similar distribtuion of reuse times for W1 and W2. This has been quantified in Table 3.16. Thus, the distributions remain stationary over long time periods.

Table 3.15: Descriptive statistics about reuse times.

Period	median	mean	std. dev.	CV	IQR	tail weight
W1	2,720	5.77×10^6	2.46×10^7	4	6.77×10^5	0.30
W2	3,042	5.10×10^6	2.47×10^7	5	3.55×10^5	0.35

Table 3.16: Quantifying similarity of reuse time distributions during two periods.

KS distance	χ^2 distance	χ^2 p-value
0.05	0.008	1

3.5. Distribution across Clusters

The Databricks ecosystem is composed of many systems and subsystems working in concert to deliver results. The distribution of reads across clusters refers to the number of reads contributed by each cluster to the total workload. We analyze the distribution of number of reads and bytes read across clusters, and our major finding is:

MF3.6.1 The distribution of number of reads and bytes read over clusters is heavy tailed.

Figure 3.12 depicts the empirical cumulative distribution function of the number of reads by each cluster. Most clusters perform a small number of reads, and the majority of the reads are by a very small number of clusters. The tail is very heavy which is apparent from the statistics in Table 3.17. Particularly from the tail weight statistic that 70% of the reads are from the 99th percentile of the most popular clusters.

The empirical cumulative distributions in Figure 3.12 depict that the distributions of W1 and W2 are similar. There is a slight difference and this is quantified in Table 3.18. The variates were rebinned into 10 logarithmic bins for this purpose.

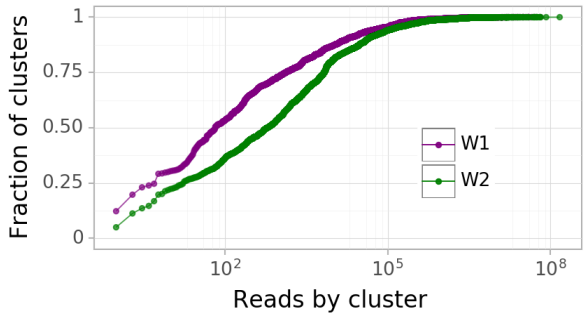


Figure 3.12: ECDF of number of reads by a cluster vs. fraction of clusters with logarithmic horizontal axis.

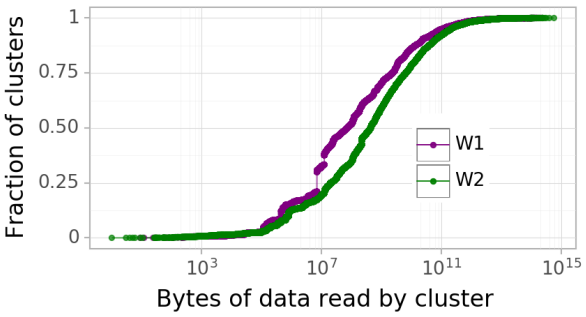


Figure 3.13: ECDF of bytes read by a cluster vs. fraction of clusters with logarithmic horizontal axis.

Table 3.17: Descriptive statistics about distribution of reads across clusters.

Period	median	mean	std. dev.	CV	IQR	tail weight
W1	67	44,694	764,354	17	1,332	0.78
W2	606	69,841	999,796	14	6,769	0.69

3.5.1. Bytes Read by Clusters

Figure 3.13 depicts the empirical cumulative distribution function of bytes read by each cluster. Unlike all other distributions in this work, this one is also flat on the lower end (closer to 0) and not just the higher end of the horizontal axis. The steep incline in the middle indicates that the majority of the clusters read a moderate amount of data, between 10MB and 10GB. We hypothesize that the prevalence of cluster little data read is due to failed or newly started clusters. It is a very heavy tailed curve from descriptive statistics in Table 3.19. Particularly the tail weight statistic that 82%-90% of reads are from 99th percentile of most popular clusters.

The empirical cumulative distributions in Figure 3.13 depicts that the distributions during W1 and W2 are close. There is a slight difference and this is quantified in Table 3.18. The variates were rebinned into 100 logarithmic bins for this purpose.

3.6. Distribution Across File Types

The distribution of reads across file types helps understand what formats and file features are prevalent. This leads to informed decision making about what formats to optimize and support. In this section, we look at two features of files: storage format and compression. The storage format is the structure according to which bytes are stored on disk; examples include JSON, CSV and Parquet. The compression algorithm is the one used to compress stored data; examples include gzip and snappy. We obtain the information of the storage format and compression algorithms from the file extensions. Our major findings are:

MF3.7.1 Parquet is the most popular file format.

MF3.7.2 Snappy and Gzip are the most popular compression schemes.

3.6.1. File Format Popularity

Figure 3.14 depicts the distribution of file formats in trace W2. We see that Parquet is the most popular format. This is mainly due to its suitability for big data analysis; in addition, it is the default format for storage in the Databricks ecosystem. A surprise to us was that this was followed by accesses to file without any file extension. This means that they did not have any string with a "." (dot) in them after the last "/" (slash). We conjecture that reads with no filetype are comprised of parquet or other files without any extension. This is followed by an almost equal fraction of accesses to JSON and files with unknown file format; JSON is a popular data format, particularly for web applications. The reads with the unknown extension are those whose paths were truncated or did not have an extension apart from the extension of the compression scheme used. The number of unknown files was a surprise; this is evidence that a significant fraction of the files are deeply

Table 3.18: Quantifying similarity of distribution of reads across clusters.

KS distance	χ^2 distance	χ^2 p-value
0.20	0.17	1.0

Table 3.19: Descriptive statistics about distribution of bytes read across clusters.

Period	median	mean	std. dev.	CoV	IQR	tail weight
W1	7.36×10^7	1.87×10^{11}	3.61×10^{12}	19	2.36×10^9	0.90
W2	4.37×10^8	2.01×10^{11}	4.17×10^{12}	21	6.54×10^9	0.82

nested, thus having long file paths, or are in compressed archives without any special file formatting such as Parquet. Finally, popular big data file formats like CSV, Avro, and ORC, also make an appearance.

3.6.2. Compression Scheme Distribution

Figure 3.15 depicts the distribution of compression schemes in trace W2. Gzip and Snappys are, by far, the most popular compression schemes, with an order of magnitude higher fraction of reads than anything else. The unknown fraction refers to files where the compression schema is not present in the extension or the file path was truncated.

3.7. Threats to Validity

Our work in this chapter has several limitations. We discuss in this section the three main limitations, the correlation between features, the bias inherent to our traces, and the lack of an example of direct use of the main findings.

Correlation Between Features

Two features are said to be positively (negatively) correlated if an increase or decrease in one necessarily causes a corresponding (inverse) increase or decrease in the other. Not checking for correlations can lead to biased characterizations, because the reader is left with the impression that the variables under study are independent. Although not presented in this work, we have conducted a preliminary analysis of the linear correlation between the measured features. We have calculated the Pearson correlation coefficient between all possible pairs of features, at different levels of magnification. As expected, most correlations were between 0.1 and -0.1, indicating low correlation. Notably, among all the correlations we calculate, a few were above 0.9 (high correlation), but all others were between -0.17 and 0.25, which indicates these features are independently varying.

Biased Traces

The bias inherent in trace use is that it is possible that our results, albeit valid for the traces used in the work, are not representative at-large. We argue that this trace is representative of remote storage used by collections of small clusters. The different clusters which are part of the trace are used by widely different organizations including healthcare, manufacturing, web services, and advertising. We also observed that the workloads from clusters from diverse organizations looked similarly across the different characteristics.

The bias might also be in time. The two weeks chosen for the analysis might not representative. We verified that this was no the case by picking the two weeks very far apart. We also verify the representativeness of the two weeks by visually comparing them to several other weeks. The inclusion of two weeks in this work is to demonstrate that characteristics of the workload didn't change over time.

Lack of Example Usage

It has become common in recent characterization studies to include in the publication an example of direct use of the main findings, for example, for tuning a component of the system under study. To the proponents of this approach, providing such an example can reduce the threat that the findings may be useless. In our view, doing so is actually more of a threat to validity than not conducting such an experiment, because (i) the

Table 3.20: Quantifying similarity of distribution of bytes read across clusters.

KS distance	χ^2 distance	χ^2 p-value
0.18	1.51	1.0

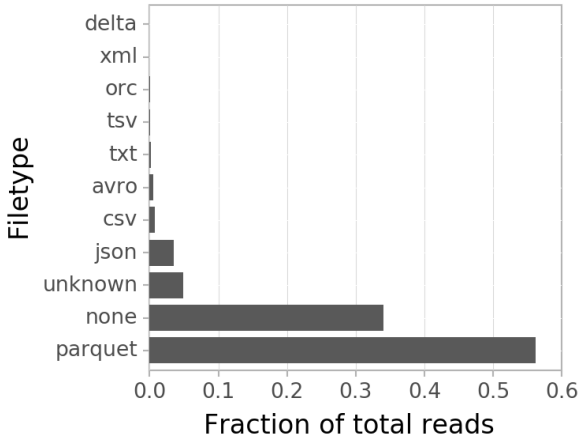


Figure 3.14: File format popularity for W2.

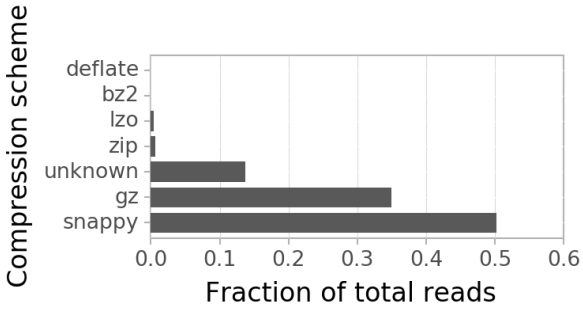


Figure 3.15: Popularity of compression schemes for W2.

analyst has a direct incentive to select specific results over others, (ii) the construction of a useful example removes resources (and, ultimately, pages) from the characterization itself, and (iii) in the long-run, implying characterizations are not enough diminishes the scientific standing of our community.

4

Generative Model For Storage Workloads

Motivated by the importance of traces in research, and by the scarcity of traces and models of Spark storage workloads, we endeavor to answer the following research question: *How can the characteristics of big data storage workloads in the cloud be modeled to generate synthetic traces matching the statistical properties of real-world traces?*.

4.1. Overview

We present and validate a generative model for big data storage workload in the cloud. A generative model can produce data indistinguishable from the real data in many aspects, based on just as few input parameters. We use the model to generate a trace of *reads* to an object store by an ecosystem of big-data-processing applications.

The usefulness of traces stems from real world experiments being expensive, and often even infeasible to conduct. For example, a datacenter operator often cannot shut down a significant part of the datacenter to satisfy research purposes. This would often incur a high financial and opportunity cost. Traces enable to us to replay important conditions and scenarios. Traces collected correspond to a specific configuration of the system from which they were collected. They are useful, for flexibility purposes, to generate new synthetic traces that have characteristics close to the real traces. This enables conducting representative experiments at different scales and represent "what if x?" scenarios as close to original as desired.

Real world traces are hard to come by. Companies and other institutions are hesitant to make traces public. There is a fear that trace data might compromise privacy [49], reveal operational secrets and costs to competitors, and alienate customers. Therefore, there is a need for generative models to make synthetic traces which are indistinguishable from the real ones available for research. Research is also biased towards traces which are public and popular [5]. A generative model partially addresses this problem by being able to change and tune the model to generate a wide variety of synthetic traces.

Towards answering this question on which this chapter focuses, our contribution is four-fold:

1. We present a synthetic trace generation process using interarrival times and reuse times, which can *concurrently* generate different parts of the trace.
2. We investigate 3 different methods of curve fitting to approximate the features of the original trace.
3. We validate the traces generated by the model using an emergent feature, file popularity, and a explicitly designed feature, reuse times.
4. We measure the performance of the trace generation process.

4.2. Generation Process

The trace generation process is depicted in Figure 4.1. We first fit the empirical probability density functions (EPDFs) (component 1) of the features characterized in Chapter 3 to known *heavy tailed* distributions (component 2). The curve fitting takes place once at the start of the generation process. The 2 features of the trace we fit are the interarrival time of reads at the cluster level and the reuse time of reads. Interarrival

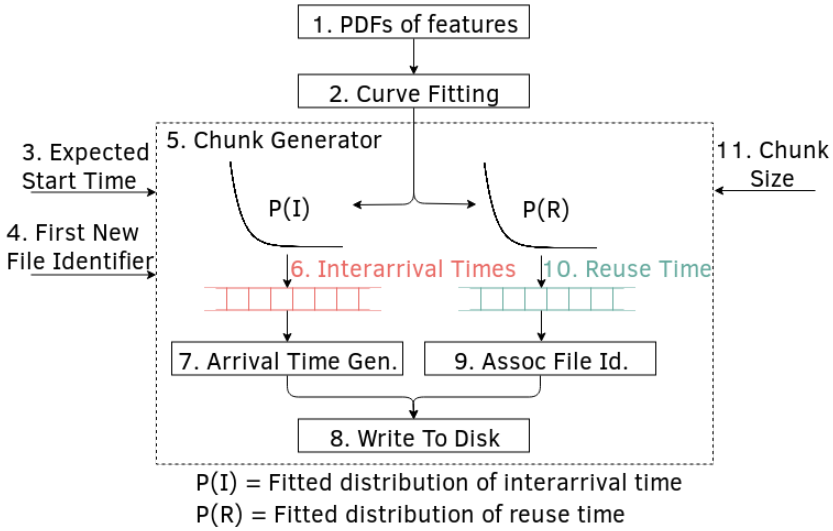


Figure 4.1: Trace generation process for one chunk.

time of a read refers to the time elapsed between a read and the previous read. Reuse time of a read refers to the time elapsed between the time when a file is read and the time when the same file was previously read. The output of the trace generation process are a list of tuples with two properties: the arrival time of a read, and the identifier of the file which was read at that time.

The trace to be generated is divided into *resource-clusters* and *data-chunks*. As introduced in Chapter 1, we are modeling the storage workload of an ecosystem composed of multiple virtual clusters. Hence, trace generation also happens independently per cluster. The trace corresponding to each cluster is further divided into chunks. Each chunk contains all reads between two specific times in the trace of a cluster. Important for the scalability of the process, the trace for each cluster can be generated in parallel, independently from the traces of other clusters. The chunks of the trace from the same cluster can also be generated in parallel.

A chunk is generated by a *chunk generator* (component 5). The chunk generator takes the *chunk size* (component 11), an *expected starting time* (component 3), and an *identifier to assign to the first new file* (component 4) it encounters as input. It generates a chunk with a specified number of reads. The number of reads in a chunk is the chunk size. Chunks are ordered. Each chunk is assigned a position in trace.

Parallel generation of chunks of the same cluster is possible because we use probability theory to estimate the start time of each chunk. Thus, a previous chunk doesn't need to be generated to start generation process of any chunk. The estimate passed as input to the generator is the *estimates start time*. The expected start time assignment process is depicted in Figure 4.2. The expected start time is computed by multiplying the expected value of the interarrival time distribution and the number of reads that will be generated by all the previous chunks. Due to this process, chunk boundaries are not clearly demarcated in time and there might be some overlap between chunks.

Another reason parallel generation of chunks is possible is that the new file identifiers assigned in every chunk are independent. First new file identifiers are chosen such that there is no possibility of overlap between new files assigned by each chunk generator. The subsequent ones are arrived at by incrementing the identifier of the previous new file by one. The first new file identifier to assign to every chunk is determined empirically by running the generator a few times and determining an optimal gap between the first new file identifiers of new chunks such that there is no overlap. For example, empirically we find that a chunk of size 100 million reads generates less than 10 thousand new files. The first new file identifiers of chunks 0, 1 and 2 could be 0, 10,000 and 20,000 respectively. Each subsequent new file in a chunk is identified by incrementing the new file identifier. For example, the second new file of chunk 1 will be 10,001 if the chunk started identifying new files from 10,000.

The generation procedure in Listing 4.1 takes place in each chunk generator. The distribution of interarrival time of reads at the cluster level, obtained by curve fitting, is sampled to generate the interarrival times for N reads (component 6 in Figure 4.1). N is the length of the trace to generate. The arrival times are computed by computing a cumulative sum of the interarrival times (component 7 of Figure 4.1). The arrival time is the first property in the output tuples of the generation process.

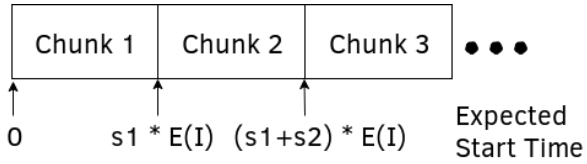


Figure 4.2: Assigning start times to chunks.

The distribution of reuse time of reads at the cluster level, obtained by curve fitting, is sampled to generate the reuse times for N reads (component 10). The reuse times are used to associate reads with a file identifier they are reading. The process of *associating reads with files* (component 9 in Figure 4.1) is depicted in Figure 4.3. A list of sequential numbers of size N, called list of files to read, is generated. This corresponds to the file read by each read. At this point, each read is associated with a unique file. The reuse times are then subtracted from the list of files to read. This produces a list of indices for reuse where each entry is an index to the list of files to read. Each entry, i, in the list of indices for reuse satisfies the property i is less than position of i and is called a *reuse index* (component 1 in Figure 4.3). The list of *files to read* (component 2) is sequentially traversed and each entry is set to the value present in the reuse index position of the same list. This generates dependencies across elements of the list and leads to a file being read multiple times. If the reuse index is out of bounds, a new file is generated and the read corresponds to the new file. The list of files to read at the end of association process becomes the second property of the output tuples of the whole generation process.

The association of reads to file identifiers can happen independent of the generation of file arrivals, as the file interarrival times, and other features such as file popularity or reuse time aren't correlated as we found in Chapter 3.

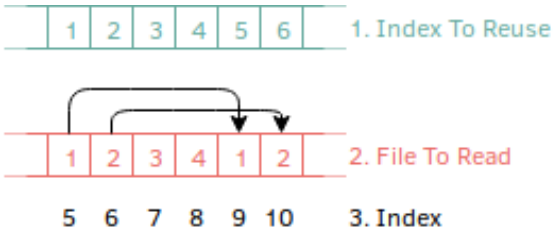


Figure 4.3: Dependency generation using reuse times.

Listing 4.1: Trace Generation Alorithm for one Cluster

```
1  N = number of reads to generate
2  expected_start_time = from generator
3  first_new_file_identifier = from generator
4
5  interarrival_times_vector = sample of size N from distribution of interarrival times
6  arrival_times_vector = expected_start_time + interarrival_times_vector.cumumative_sum()
7  reuse_times_vector = sample of size N from distribution of reuse times
8
9  list_of_files_to_read = np.arange(0, len(arrival_times), dtype=np.int64)
10
11 index_for_reuse = list_of_files_to_read - reuse_times_vector - 1
12
13 new_file_id = first_new_file_identifier
14 for i in range(N):
15 reuse_index = index_for_reuse[i]
16 if reuse_index > 0:
17 list_of_files_to_read[i] = file_of_files_to_read[reuse_index]
18 else:
19 list_of_files_to_read[i] = new_file_id
20 new_file_id = new_file_id + 1
21
22 return arrival_times_vector, list_of_file_to_read
```

The number files read between two successive reads of the same file is the *reuse distance*. The time elapsed between two successive reads of the same file is the *reuse time*. In this algorithm, we consider the reuse time

to be a good proxy for reuse distance. In cases where the orders of magnitude of reuse time and reuse distance are different, a scaling factor can be used to scale the reuse time. This only works if the reuse distance and reuse time are linearly correlated. In our case, they were linearly correlated and the scaling factor was not necessary.

4.3. Curve Fitting

Curve fitting is important because known probability distributes can be configured to imitate a different workload by just changing a few parameters increasing flexibility. Furthermore, curve fitting is important for the big data setting we target because it reduces the size of input data to the model. Some EPDFs, such as the one for reuse times, are over sized over 1GB making the slowing the sampling and generation process.

We select, out of the many results our curve-fitting process produces, two main findings:

MF4.3.1 Among the methods we have used to fit heavy tailed distributions observed in computer systems. The unweighted least-squares regression of their survival functions gives the best result.

MF4.3.2 We confirm the heavy tailed nature of the distributions, by observing that their survival function is above that of the exponential distribution. This is a sufficient condition for a distribution to be heavy tailed [44].

The trace we use to extract empirical probability distribution functions (EPDFs) for regression is W2 from Chapter 3.

4.3.1. Process for Curve Fitting

We fit the empirical distribution of features of the original data to several standard heavy tailed distributions using least-squares regression. The distributions are Weibull, generalized Pareto (subsumes Pareto), Gamma (subsumes Erlang and other exponential based distributions), Log-normal, and Levy. These are distributions commonly used to model computer systems related phenomena [28]. Following the suggestions in [48], we fit *both* the probability density function ($P(X) = Pr(X = x)$) and the survival function ($\hat{F}(X) = Pr(X > x)$) computed from empirical data. Here, x is the value a feature can, a *variate*. The survival function (SF) is the inverse of the cumulative distribution function ($F(X) = Pr(X \leq x)$), and can be derived from the CDF. $\hat{F}(X) = Pr(X > x) = 1 - Pr(X \leq x) = 1 - F(X)$. Heavy tailed distributions originally arose out of survival analysis [44]. Even though not all features we model are survival related, we thought taking heavy tailed functions back to their root and modeling their survival would give good results.

We also try weighted least-squares regression where the weight of each variate is the inverse of *bin size*. The bin size of a variate is the difference between a variate and the previous variate that is being fitted. The weight of a bin is $\frac{1}{x_i - x_{i-1}}$. The reason for trying weighted least-squares is because it approximates to maximum likelihood estimation (MLE) for large samples, and MLE is a popular method of fitting probability distributions [39].

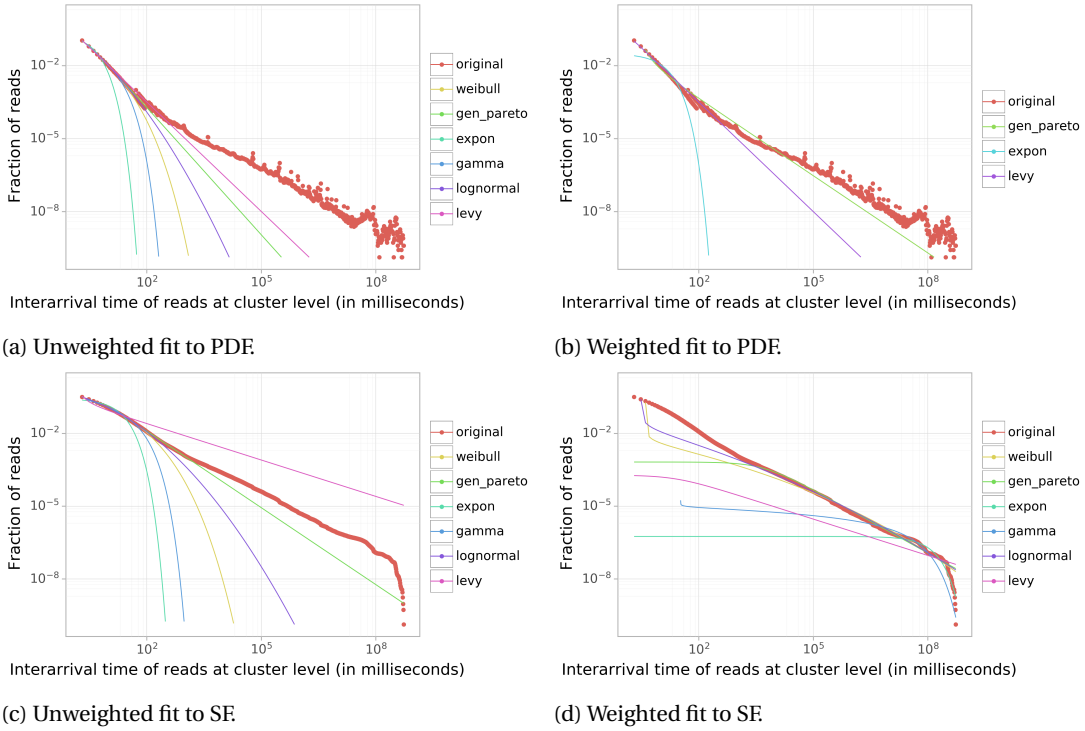
Heavy tailed functions are non-linear, so we use non-linear least-squares regression for fitting the PDFs and SFs. We use Levenberg-Marquardt (LM) algorithm for least-squares regression, we the implementation of the algorithm provided by MINPACK through the SciPy library, version 1.1.0. We use LM because of its ubiquity and generally higher performance, relative to other methods [30]. We verify goodness of fit using the Kolmogrov-Smirnov (KS) test and the χ^2 test. The distance statistic of the KS-test (d-statistic) has been recommended for measuring the goodness of fit of heavy tailed distributions by prior work [19]. We also evaluate he goodness of fit is also evaluated visually using PDFs and SFs. The original distribution is labeled "original". In each statistic comparison table e.g., Table 4.1, we label the column with weighted or unweighted fit "Wt.".

We fit four features of the workload to know probability distributions in this section. The PDF and SF of each of interarrival time of reads at cluster level, the reuse time of files, the popularity of files, and the distribution of request sizes. We fit the popularity of files and the distribution of request sizes because the are features of broad interest to the general community. The distributions of the interarrival times of reads, and the reuse time of files are use in our model.

4.3.2. Fitting the Interarrival of Reads at Cluster Level

We present in this section the results of fitting the PDF and SF of the distribution of interarrival time of reads at cluster level. We use the process described in Subsection 4.3.1. Figure 4.4 depicts the weighted and unweighted fit of PDFs and SFs of several know distributions to the distribution of interarrival time of reads

at cluster level. The heavy tailed nature of the distribution is confirmed by the position of the empirical SF, above the SF of exponential (**MF4.3.2**) as depicted in Figures 4.4c and 4.4d.



(a) Unweighted fit to PDF.

(b) Weighted fit to PDF.

(c) Unweighted fit to SF.

(d) Weighted fit to SF.

Figure 4.4: Distribution fitting to interarrival time of reads at cluster level.

Table 4.1 quantifies the goodness of fit. The Log-normal distribution seems to fit best closely followed by Generalized Pareto distribution, and further by Weibull distribution. All three when survival functions were fitted without considering weights (**MF4.3.1**). The best fit found in the Log-normal distribution has the shape, location, and scale parameters with values 2.254, 0.048, and 0.638, respectively.

4.3.3. Fitting the Reuse Time

We present in this section the results of fitting the PDF and SF of the distribution of reuse time of reads. We use the process described in Subsection 4.3.1. Figure 4.5 depicts the weighted and unweighted fit of PDFs and SFs of several know distribution to the distribution of reuse time of reads. The heavy tailed nature of the EPDF is confirmed by the position of the empirical SF, above the SF of exponential (**MF4.3.2**) as depicted in Figures 4.5c and 4.5d.

Table 4.2 quantifies the goodness of it. The Weibull distribution seems to fit best closely followed by Log-normal distribution, and further by Generalized Pareto distribution. All three when survival functions were fitted without considering weights (**MF4.3.1**). The best fit found in the Weibull distribution has shape, location, and scale parameters as values 0.152, 42.951, and 34,981.631, respectively.

4.3.4. Fitting the File Popularity

We present in this section the results of fitting the PDF and SF of the distribution of popularity of files. We use the process described in Subsection 4.3.1. Popularity of a file refers to the fraction of all reads that were of the file. Figure 4.6 depicts the weighted and unweighted fit of PDFs and SFs of several know distributions to the distribution of popularity of files. The heavy tailed nature of the distribution is confirmed by the position of the empirical SF, above the SF of exponential (**MF4.3.2**) in Figures 4.6c and 4.6d.

Table 4.1: Goodness of fit of distributions for interarrival time of reads at cluster level ordered by the Kolmogrov-Smirnov distance.

Distribution	Fit to	Wt.	KS dist.	χ^2 dist.
Log-normal	SF	no	0.001	1.99×10^{-02}
Gen. Pareto	SF	no	0.004	3.04×10^{-02}
Weibull	SF	no	0.006	1.55×10^{-02}
Gamma	SF	no	0.020	9.52×10^{-02}
Weibull	PDF	no	0.031	1.64×10^{-02}
Gamma	SF	yes	0.038	4.90×10^{-02}
Gamma	PDF	no	0.040	3.27×10^{-02}
Log-normal	PDF	no	0.047	9.50×10^{-03}
Levy	SF	no	0.055	4.19×10^{-01}
Gen. Pareto	PDF	no	0.070	5.50×10^{-03}
Exponential	PDF	yes	0.073	1.33×10^{-01}
Exponential	SF	no	0.079	1.78×10^{-01}
Exponential	PDF	no	0.109	1.03×10^{-01}
Levy	PDF	yes	0.118	1.24×10^{-02}
Levy	PDF	no	0.133	1.27×10^{-02}
Weibull	SF	yes	0.178	$6.99 \times 10^{+12}$
Log-normal	SF	yes	0.186	$9.04 \times 10^{+04}$
Gen. Pareto	SF	yes	0.316	4.24×10^{-01}
Levy	SF	yes	0.316	4.24×10^{-01}
Exponential	SF	yes	0.316	4.24×10^{-01}
Gen. Pareto	PDF	yes	0.798	3.16×10^{-02}

Table 4.2: Goodness of fit of distributions for reuse time ordered by the Kolmogrov-Smirnov distance.

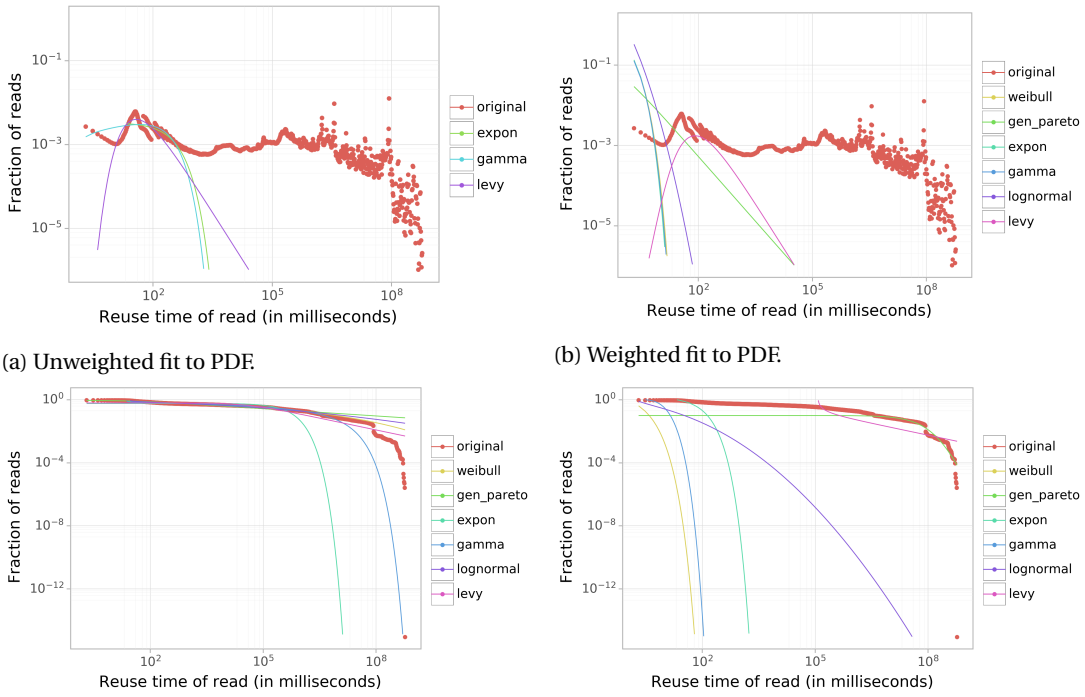
Distribution	Fit to	Wt.	KS dist.	χ^2 dist.
Weibull	SF	no	0.039	8.80×10^{-01}
Log-normal	SF	no	0.056	8.39×10^{-01}
Gen. Pareto	SF	no	0.088	7.18×10^{-01}
Gamma	SF	no	0.143	$1.18 \times 10^{+06}$
Gen. Pareto	PDF	yes	0.259	$1.58 \times 10^{+00}$
Levy	SF	no	0.304	9.23×10^{-01}
Levy	PDF	yes	0.332	6.63×10^{-01}
Exponential	SF	no	0.351	9.25×10^{-01}
Levy	PDF	no	0.363	5.94×10^{-01}
Exponential	PDF	no	0.512	2.81×10^{-01}
Gamma	PDF	no	0.522	2.75×10^{-01}
Levy	SF	yes	0.572	3.37×10^{-01}
Exponential	SF	yes	0.589	$1.62 \times 10^{+00}$
Log-normal	SF	yes	0.762	$6.50 \times 10^{+01}$
Gen. Pareto	SF	yes	0.827	9.29×10^{-01}
Gamma	SF	yes	0.887	$6.23 \times 10^{+01}$
Log-normal	PDF	yes	0.905	$5.14 \times 10^{+01}$
Weibull	SF	yes	0.913	$5.40 \times 10^{+01}$
Weibull	PDF	yes	0.916	$7.88 \times 10^{+00}$
Gamma	PDF	yes	0.917	$7.31 \times 10^{+00}$
Exponential	PDF	yes	0.917	$8.18 \times 10^{+00}$

Table 4.3: Goodness of fit of distributions for popularity of files ordered by the Kolmogrov-Smirnov distance.

Distribution	Fit to	Wt.	KS dist.	χ^2 dist.
Gen. Pareto	SF	no	0.027	4.73×10^{-01}
Log-normal	SF	no	0.032	4.60×10^{-01}
Weibull	SF	no	0.042	4.36×10^{-01}
Exponential	PDF	yes	0.053	5.33×10^{-01}
Gamma	SF	no	0.061	4.00×10^{-01}
Gamma	PDF	yes	0.080	3.15×10^{-01}
Exponential	SF	no	0.091	7.52×10^{-01}
Log-normal	SF	yes	0.102	4.69×10^{-01}
Levy	SF	no	0.109	2.78×10^{-01}
Weibull	SF	yes	0.118	$2.03 \times 10^{+00}$
Weibull	PDF	yes	0.124	3.60×10^{-01}
Log-normal	PDF	yes	0.163	3.84×10^{-01}
Gamma	SF	yes	0.167	$1.58 \times 10^{+09}$
Levy	PDF	no	0.168	2.49×10^{-01}
Gamma	PDF	no	0.238	$1.12 \times 10^{+00}$
Gen. Pareto	PDF	yes	0.276	4.52×10^{-01}
Levy	PDF	yes	0.284	5.42×10^{-01}
Gen. Pareto	SF	yes	0.335	7.74×10^{-01}
Exponential	PDF	no	0.392	4.28×10^{-01}
Exponential	SF	yes	0.427	8.45×10^{-01}
Levy	SF	yes	0.428	$1.00 \times 10^{+12}$
Weibull	PDF	no	0.468	$2.18 \times 10^{+00}$
Gen. Pareto	PDF	no	0.538	4.92×10^{-01}

Table 4.4: Goodness of fit of distributions for size of read ordered by the Kolmogrov-Smirnov distance.

Distribution	Fit to	Wt.	KS dist.	χ^2 dist.
Gen. Pareto	SF	no	0.009	5.16×10^{-02}
Log-normal	SF	no	0.017	5.14×10^{-02}
Weibull	SF	no	0.033	1.75×10^{-01}
Gen. Pareto	PDF	no	0.049	5.19×10^{-02}
Levy	SF	no	0.066	3.50×10^{-01}
Gamma	SF	no	0.068	$2.40 \times 10^{+00}$
Log-normal	PDF	no	0.075	6.64×10^{-02}
Weibull	PDF	no	0.104	9.31×10^{-02}
Gamma	PDF	no	0.124	1.13×10^{-01}
Log-normal	SF	yes	0.140	$4.13 \times 10^{+01}$
Weibull	SF	yes	0.167	$8.18 \times 10^{+03}$
Exponential	PDF	no	0.183	1.69×10^{-01}
Exponential	SF	no	0.196	3.75×10^{-01}
Levy	PDF	no	0.208	3.81×10^{-02}
Gamma	SF	yes	0.214	$2.85 \times 10^{+10}$
Levy	SF	yes	0.357	$7.10 \times 10^{+07}$
Gen. Pareto	SF	yes	0.570	7.02×10^{-01}
Exponential	SF	yes	0.570	7.02×10^{-01}
Gen. Pareto	PDF	yes	0.759	1.86×10^{-02}
Gamma	PDF	yes	0.997	7.01×10^{-01}
Exponential	PDF	yes	1.000	1.23×10^{-05}
Levy	PDF	yes	1.000	2.18×10^{-05}



(a) Unweighted fit to PDF.

(b) Weighted fit to PDF.

(c) Unweighted fit to SF.

(d) Weighted fit to SF.

Figure 4.5: Distribution fitting to reuse time of read.

Table 4.3 quantifies the goodness of fit. The Generalized Pareto distribution seems to fit best closely followed by Log-normal distribution, and further by Weibull distribution. All three when survival functions were fitted without considering weights (**MF4.3.1**). The best fit found in the Generalized Pareto distribution has the shape, location, and scale parameters with values 1.063, 0.622, and 1.783, respectively.

4.3.5. Fitting the Read Size

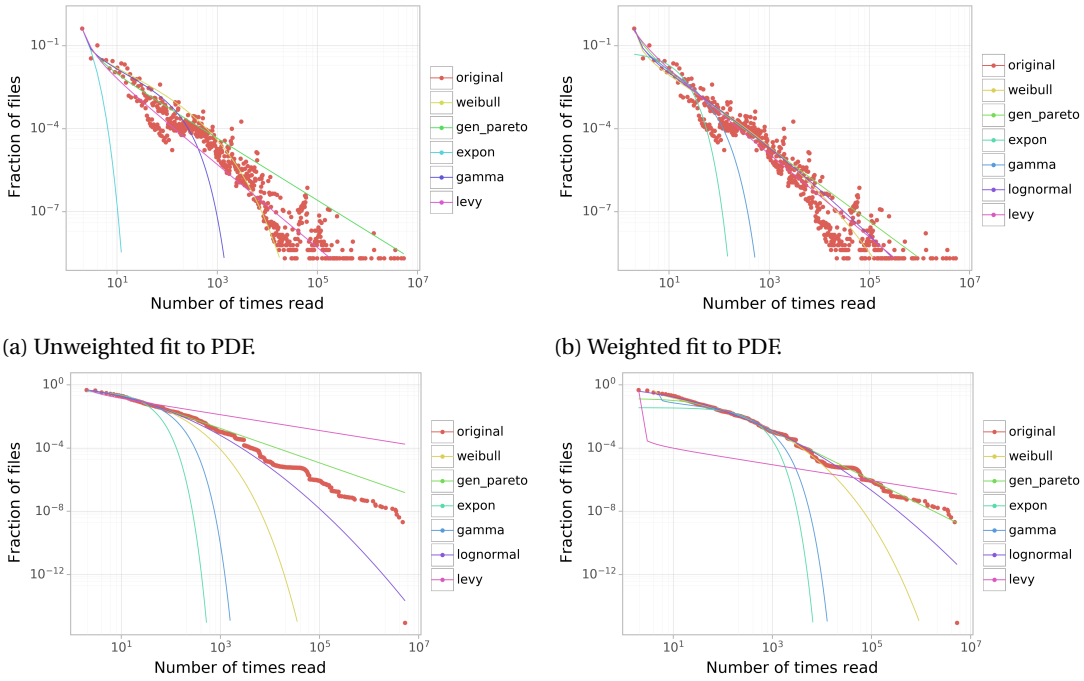
We present in this section the results of fitting the PDF and SF of the distribution of size of reads. We use the process described in Subsection 4.3.1. The size of a read is defined as the number of bytes transferred from the storage system in one read. Figure 4.7 depicts the weighted and unweighted fit of PDFs and SFs of several know distribution to the distribution of size of reads. The heavy tailed nature of the distribution is confirmed by the position of the empirical SF, above the SF of exponential (**MF4.3.1**) in Figures 4.7c and 4.7d.

Table 4.4 quantifies the goodness of fit. The Generalized Pareto distribution seems to fit best closely followed by Log-normal distribution and further by Weibull distribution. All three when survival functions were fitted without considering weights (**MF4.3.1**). The best fit found in the Generalized Pareto distribution has shape, location, and scale parameters as values 1.185, -0.307, and 2.842, respectively.

4.4. Model Validation

There are two major schools of model validation: similar performance and similar statistical properties. In similar performance validation, a generative model is said to be be similar to the original system if the traces it produces lead to performance similar to the original system, for a wide range of algorithms and policies. In similar statistical properties validation, a generative model is said to be similar to the original system if the traces it produces have similar observed statistical properties to the original system. Validation using statistical properties requires that the original data is first characterized to identify them. We characterize the original data used for this validation in Chapter 3. Validation more statistical properties requires more features to be characterized.

A trace of storage reads can be uniquely identified by two properties: its spatial distribution and its temporal distribution. The spatial distribution is defined as the distribution of reads across different elements that constitute the trace. In our context, this is the probability distribution of the popularity of files. The temporal distribution is defined as the distribution of reads over time, and also by the relation between reads



(a) Unweighted fit to PDF.

(b) Weighted fit to PDF.

(c) Unweighted fit to SF.

(d) Weighted fit to SF.

Figure 4.6: Distribution fitting to popularity of file.

over time. In our context, this is approximated by the distribution of interarrival times and of reuse times.

We use two features, interarrival time of reads, and reuse time of reads, to generate the workload trace, using our model. We validate the model by observing another feature of the workload, the popularity of files, which is an emergent feature describing the spatial distribution. It is emergent because it is not an input to our model and occurs organically due to the interplay of other modeled features. It arises as a result of the synthetic trace having the same temporal distribution as the original trace. The property which describes the temporal distribution, reuse time, is explicitly modeled by us. We compare the distribution of the new feature in our generated data to that we observe in the original data. Similar distributions for both traces would indicate similarity of spatial distribution. We also compare the reuse time distribution we observe in the generated trace to the original trace. Similarity between the two distributions would validate our hypothesis of linear correlation between reuse distance and reuse time. It would also validate the generated trace because the distribution of reuse time remains the same after using reuse time as reuse distance when generating the synthetic trace. We quantify the similarity of two distributions using the Kolmogrov-Smirnov test.

The results we obtain lead to three main findings:

MF4.4.1 The synthetic trace has a file popularity distribution similar to that we observe in the original trace.

MF4.4.2 The popularity distribution is similar only if the chunks are sufficiently large for a file to be read a large number of times.

MF4.4.3 The reuse-time distribution of the synthetic trace is similar to that we observe in the original trace.

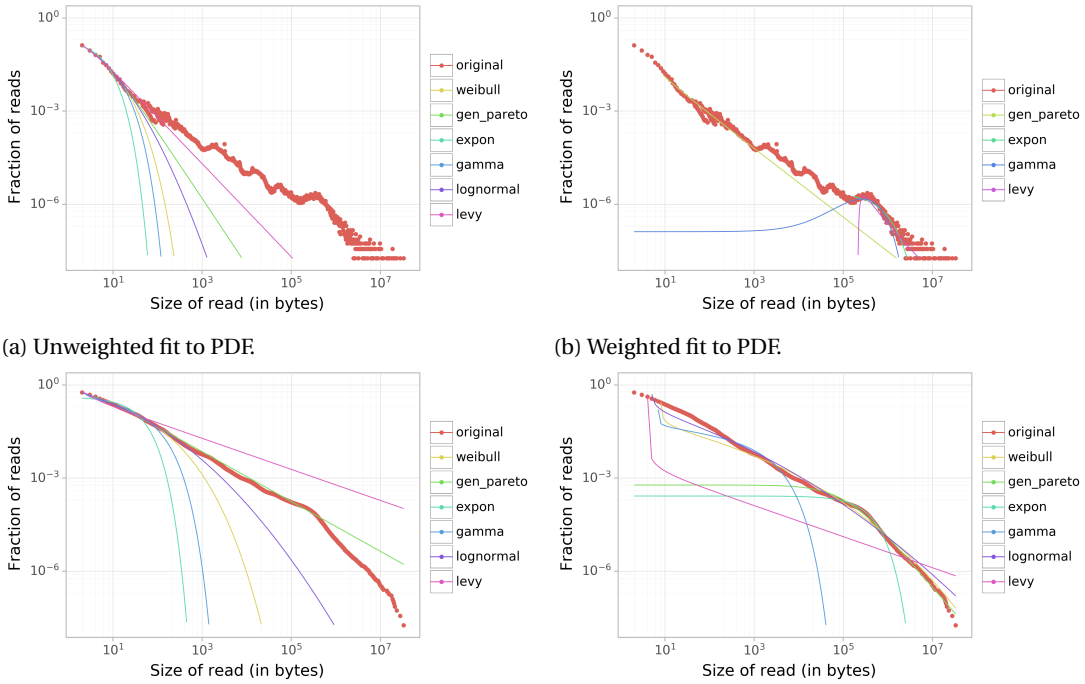
4.4.1. Experimental Setup

The original trace we use as input to the generative model for all experiments in this section is trace W2 from Chapter 3.

We generate two synthetic traces:

1. 1 billion reads in 10 chunks of 100 million reads each.
2. 100 million reads in 10 chunks of 10 million reads each.

The distributions are smoothed by binning data into 10,000 bins for fair comparison. Unbinned data has noise which influences the KS distance measurement.



(a) Unweighted fit to PDF.

(b) Weighted fit to PDF.

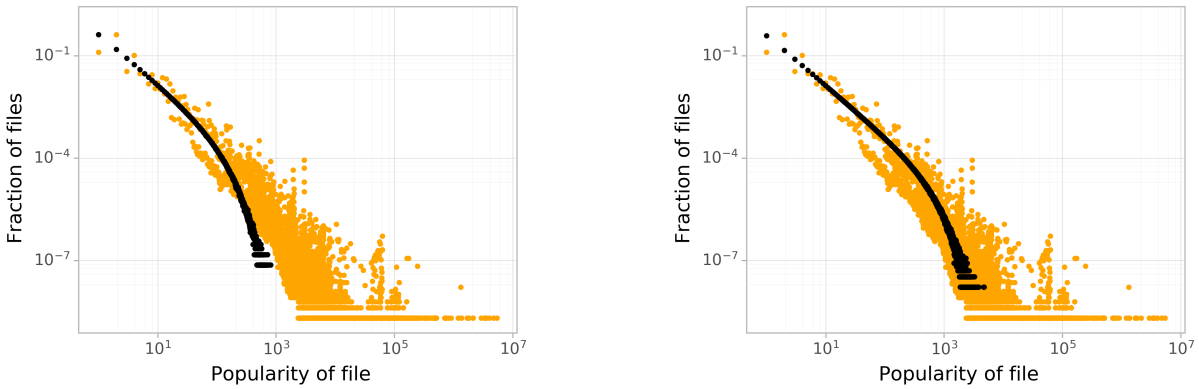
(c) Unweighted fit to SF.

(d) Weighted fit to SF.

Figure 4.7: Distribution fitting to size of read.

4.4.2. Popularity of Files

The distribution of popularity of files of the two synthetic traces are compared to that of the original trace. This is depicted in Figure 4.8. The EPDF of the synthetic trace appears slightly to the left of the original trace when the chunk size is 10 million reads (**MF4.4.2**). But, it lines up well when the chunk size is 100 million reads (**MF4.4.1**). This is caused because files read in a chunk are unrelated to files read in another chunk. Therefore, the chunks need to be of a certain minimum size for a file to be read enough number of times for the distribution to be comparable to the original trace at higher popularities. The KS distance is 0.29 for chunk sizes of 10 million and 0.25 for 100 million. If we only consider the tail elements, the KS distance of 0.014 for chunk size 10 million is higher than 0.001 for chunk size 100 million. With chunk sizes large enough that all uses of the same file can fit in a single chunk, the emergent popularity distribution of the synthetic trace matches that of the original.



(a) Chunk size: 100 million reads.

(b) Chunk size: 10 million reads.

Figure 4.8: Distributions of popularity of files of synthetic traces (black) compared to original (orange).

4.4.3. Reuse Time of Reads

We compare reuse times for a single chunk of length 10 million. We use chunks of length 10 million and not 100 million for two reasons. Reuse time is expensive to compute and thus the computation for a smaller chunk is faster. The second reason is that this would demonstrate that reuse time distribution is not as effected by chunk length as popularity distribution. We use length The comparison is depicted in Figure 4.9. The reuse time distribution of the synthetic trace is below that of the original trace. But, both have a similar shape and are at similar orders of magnitude at different variates (**MF4.4.3**).

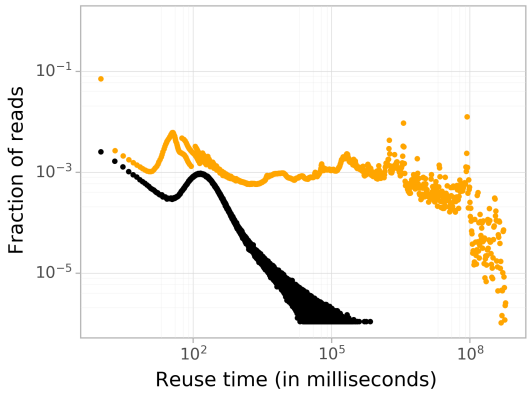


Figure 4.9: Comparison of distribution of reuse times of 1 chunk of synthetic trace (black) to that of original trace (orange).

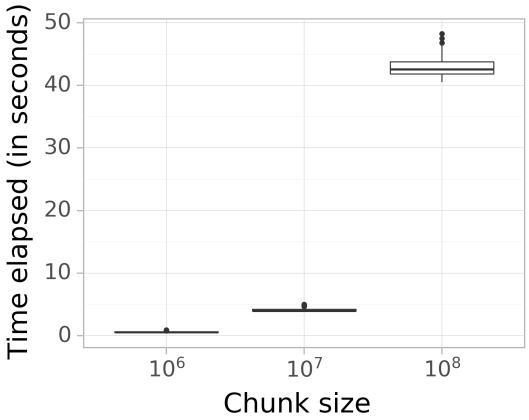


Figure 4.10: Time elapsed to generate chunks of different sizes.

4.5. Performance of the Trace Generator

We measure the performance of the trace generator to analyze its suitability for generating long traces and real world usage. We generate traces with different chunk sizes and measure the time it takes to generate a chunk. The time required to write to disk is not measured. The experiment is repeated ten times. The results are depicted as a boxplot in Figure 4.10. The upper, middle and lower notches represent the 75th, 50th and 25th percentile, respectively. The computers used for trace generation have two Intel Xeon E5-2630-v3 CPUs and 64GB of RAM. We see a linear increase in the time required for trace generation. An order of magnitude higher size requires an order of magnitude more time.

We generate ten chunks of the same size, and belonging to the same cluster, in parallel and observe that the number chunks has no impact on the generation time as long as each chunk generator can use one full CPU core for its generation. Similarly, chunks belonging to multiple clusters can also be generated in concurrently as there is no dependence between chunks of different clusters.

4.6. Threats to Validity

We consider and discuss in this section three major threats to validity of the work presented in this chapter.

Usage of Least Squares Regression

Maximum Likelihood Estimation (MLE) is commonly used technique to fit data to know probability distributions. We weren't able to use MLE due to the massive size of the dataset under consideration. After all the pre-processing describe in (ref characterization chapter), the dataset was still sized in multiple terabytes. MLE on such a large dataset was not feasible. Next, we tried systematic sampling of the dataset. A reasonably sized sample of a few gigabytes was not representative of the original data. The difference between the PDF of the sampled data and the PDF of the original data is depicted in Figure 4.11. To prevent this loss of information due to sampling, we use Least Squares Regression to fit the EPDFs and ESFs to probability distributions.

Weighted least squares regression with large samples approximates MLE. We see that it doesn't give a good fit. We believe this is because of the assumption by MLE that all noise Gaussian. This might not be the case in our data.

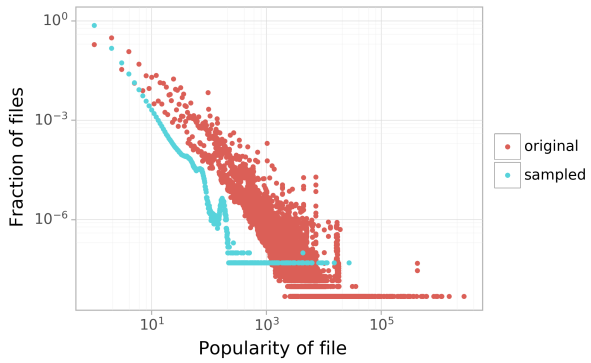


Figure 4.11: PDF os sampled data vs. PDF of original data.

Using Reuse Time instead of Reuse Distance

We estimate the reuse distance of a read from the reuse time distribution. This works in cases where reuse time and reuse distance are linearly correlated. The most a user would need to do is tune the scaling factor to get reuse times to the same order of magnitude as the reuse distance. We expect linear correlation to be the case for most workloads. In case of a more complex relation between the 2 features, this model is not valid.

File Arrival and End

We don't model the file arrival and end times of the workload. We expect the model to still be useful for modeling read dominated workload like Big Data and for use cases such as benchmarking cache policies. The model is not applicable to modification heavy workloads.

5

Design of the *Approximate Read Density* Cache Policy

The cache policy is an important yet sensitive part of any data processing system. The trade-off, size-performance, requires careful understanding of the workload, yet the caching sub-system must be transparent to the user. Implementation and tuning can be challenging, especially for modern policies such as LHD [11], which try to reuse information about prior usage patterns using PDFs. In contrast to these policies, explicitly to LHD, we attempt to design policies starting from a few stated principles and using only few parameters. We endeavor to answer the research question: *What is a principled approach to design an adaptive cache policy with few parameters?*.

5.1. Overview

Caching is widely used technology. We use it over the normal course of operating a computer in CPUs, in storage hardware and while accessing said storage hardware (page cache). It is used between a database and application that use it to reduce reads from the database[7]. Furthermore, most browsers accessing the Internet fetch part of the content from geo-distributed cache locations that act as a Content Delivery Network (CDN), such as Akamai's. Given its widespread deployment, it is an important technology worth studying.

In general a cache is beneficially employed whenever an object is expensive to compute or retrieve and doesn't change over a reasonably long period of time. A cache system is comprised of fast storage and algorithms (*policies*) that manage cached data. A cache is beneficial when, on *average*, the cached objects can be retrieved from the cache much faster than computing them or retrieving them from the storage system used before caching was added.

Towards answering the research question on which this chapter focuses, designing a cache policy based on few principles and parameters, our contribution is three-fold:

1. We propose a reference architecture for a cache policy (Section 5.2).
2. We design a family of eviction metrics, based on the concept of read density (Section 5.5).
3. We design a novel approximate data structure to compute the number of reads of an object in different time intervals (Section 5.6).

5.2. Cache Reference Architecture

We use in this chapter the system model introduced in Section 1.1, in particular, the storage subsystem. The storage device used for caching is a byte/word-addressable storage device, such as RAM, NVMe flash, or SSD. In Internet or network based applications, a layer of software such as *memcached* or *caffeine* provides a *key-value store* interface atop the storage device. In this storage architecture, caching can be added as a multi-level abstraction, where each order of magnitude of added latency is addressed by a new cache level. At any level of abstraction, the size of installed cache is typically of a much smaller size than the whole dataset. Thus, the cache needs to keep changing its content. The elements stored in the cache need to keep changing

in response to the access pattern of the application for the cache to be useful. The set f algorithms which decide how the content of the cache changes is the *cache policy*.

Orthogonally, caching comes in variants, whole-trace, online, offline, and whole-trace. *Whole-trace caching* happens when the cache policy has access to all future accesses to make fully informed decisions. This ideal case rarely happens in practice. *Online caching* decisions have to be taken as access requests stream in, in a short amount of time. Online caching policies are traditionally used for caching in CPUs, OS page cache, databases, CDNs etc. *Offline* caches are closely related to a dual of caching, *tiering*, which is a technique where parts of the data are stored on storage devices with different performance. Offline algorithms analyze past data accesses for a significant period of time, then make decisions about the placement of data for the next (also long) period of time [4] or place suggestions for the online algorithm to take into account. For example, a two tier storage system which consists of 128GB RAM, 1TB NVMe, and 4TB HDD could use auto-tiering to periodically move large datasets between in-memory location and on NVMe location.

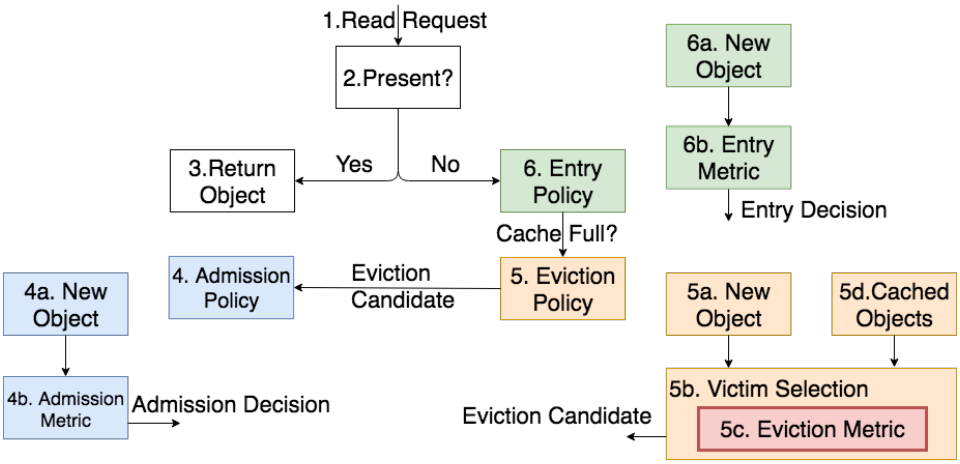


Figure 5.1: Structure of an Online Cache.

The reference architecture of an online cache is in Figure 5.1. A cache read has two results: *hit* or *miss*. In case of a hit, the object already exists in the cache and can be returned to accessor (component 3). In case of a miss, the object doesn't exist in the cache and there are two options: consider for *entry* or *reject*. The *entry policy* makes this decision (component 6). Granting an object entry means it is eligible to placed in the cache. If there is space in cache, the object is placed. It might be the case that cache is already full. It has reached its capacity in storage or the number of objects stored. In that case, an existing object has to be *evicted* from cache. The *eviction policy* (component 5) decides which object will be evicted. It picks one or multiple objects to be evicted from the those already present in the cache. This object is the *eviction candidate*. The cost-benefit analysis of retaining the eviction candidate versus admitting the new object is done by the **admission policy** and a decision is made.

A cache policy is the process and its set of algorithms (admission, eviction, etc.) that decide which objects get to stay in the cache and which are evicted. Evicting can just be deleting a computed result or a copy from the cache while it stays in slower storage. Cache policies have components which are common across many of them. In this section, we outline the structure of a generic cache policy and the components it might have. The architecture is not exhaustive, but we think it is a good representation. We provide examples of existing policies to emphasize the applicability of this generic architecture.

5.3. Anatomy of a Cache Policy

A cache policy is considered to be successfully fulfilling its role if it has the right element in cache at the right time. The "right" element and time are challenging concepts to quantify and exact optimal solutions do not exist in practice for most caching problems. Some solutions exist for problems with strict unrealistic assumptions. For example, for caches where all objects are equally sized and equally costly to fetch, Belady's OPT [12], a whole-trace caching policy, specifies that the optimal object to evict at any time is the one whose next access is furthest in the future. For all online and offline policies commonly used for systems, where all objects aren't equally sized or costly and other reasons, the rightness or suitability of an object's presence in the cache at a certain time is made from assumptions and intuitions. To be of use in an algorithm, these

simplifying assumptions and rule-of-thumb intuitions need quantification by a **metric**. A good caching metric imposes a total ordering on cacheable elements in the system. The ordering of object according to some metric is called ranking in some literature [15].

The **Entry Metric** (component 6b) is the algorithm whose result determines whether an object is eligible for entry into the cache. This can be based on the nature of the object, such as metadata or data, or some other property such as object size. For example, AdaptSize [13] restricts entry probabilistically based on object size. Admit everything is the most commonly used entry metric: every object is admitted irrespective of its size and other properties.

Eviction Metric (component 5c) is the algorithm whose result determines the item to evict when the cache is full. Examples of this are Least Recently Used (LRU) and Least Frequently Used (LFU) eviction policies. In LRU, the object which was used least recently is evicted. In LFU, the object which has been used the least number of times is evicted. These conceptions of least recency and frequency are what constitute the eviction policy. The metric chosen depends on constraints related to the environment and the objectives of the caching system (such as maximizing the object hit rate or the byte hit rate.)

Admission Metric (component 4b) is the algorithm whose result determines whether an item should be admitted into the cache, most often at the expense of the item chosen for eviction by the eviction policy. "Admit everything" is the most commonly used admission metric. The popularity of an object can be used as an admission metric, as done in TinyLFU [25].

Victim Selection is the process of selecting objects for eviction metric algorithm to run on. Cache policies don't touch all elements in the cache to make either admission or eviction decisions. There are two types of victim selection processes: pre-calculation and post-calculation.

Pre-calculation victim selection is the process of picking object from the cache for metric calculation. For policies like LRU and LFU, it happens as a direct consequence of the data structure used. Here, the element at the tail or head of the linked list is picked for eviction, respectively. In more complex policies or function-based policies, a way of selecting a subset of elements in the cache so that the ranking function (the function which computes the metric) is not executed for every element during eviction or admission. A popular way of doing that, used in memcached, is using size slabs. Elements are checked for eviction in the same size slab as an incoming element so as to avoid unnecessary evictions of several small object (several elements are evicted so that their combines size makes room for a large object). Another way to select victims before calculation is sampling. A sample of elements is taken from the cache and their metrics are computed pending eviction.

Post-calculation victim selection is the process of applying the result of ranking obtained in pre-calculation. A common thing to do is to evict the object with the minimum or the maximum value of the metric.For example, in LRU, the object with least recency of use is evicted. Another option is to use the computed metric as a probability of staying or eviction of objects [13].

Cache policies also have **meta characteristics** which are outside the purview of the admission and eviction decisions. This includes the decision to *partition* the cache into multiple segments, such as in SLRU or S4LRU. Meta characteristics also include mechanisms used to combine simple policies into *adaptive policies*. A meta characteristic might be an *expert system* that allows the policy with most success get control over the cache for a period. Different policies can also be made to compete for resources, as in ARC. This technique operates at a higher level compared to all previously presented components of a cache policy which we explain.

5.4. Caching for Big Data Workloads

Our policy is designed based on our observation of the Databricks ecosystem (The characteristics of storage accesses for this system are in Chapter 3.) From our observation, the number of reads that need to be service per second is not high on average. Moreover, in Section 3.4, we show that that both the number of reads and the amount of data read exhibit negative long range time dependence. This means there are periods of high read activity followed by periods of low read activity. We focus on regions of high read activity, which we take as the upper limit for the throughput we need to serve. This is in the order of several thousand requests per second. however we remain concerned that any compute and memory we use for the cache is unavailable for actual processing. So, we need to keep memory and compute usage for the cahce reasonably low. Therefore, we cannot use any computationally heavy algorithms, such as those that involve solving constraint-optimization problems at runtime. This necessitates online caching policies, and not offline or whole-trace ones. We will not consider meta characteristics further in this work.

Two types of objects that are cached: (1) very tiny *metadata* objects, typical of metadata, ranging in size

from a few bytes to a few KB, and (2) *large* objects which range in size from a few KB to 100s of MB. This distinction is not apparent from the logs but we notice there are reads of size 1 byte and many reads of size less than 10 bytes. At these sizes files can story only one data item. thus, we speculate that this could be metadata. This metadata could be files stored by Spark and other associated Databricks services for bookkeeping and optimization. An example of such metadata is data skipping indices. For these, for each chunk of data to be read, the system checks whether the chunk might contain anything of interest to the data processing task being executed. Another example is metadata files used to implement locks or some form of version control so that a file doesn't get modified while another program is using it. File formats, especially parquet (the most popular one), also have metadata which is essential for fast operation of the system.

We analyzed the reuse times of files and noticed that more than 75% of reads have a reuse time of less than 100 seconds and 90% less than 3 hours. The reuse times were evenly distributed between zero and three hours. This inspired us to look at LHD [11] to keep track os the reuse times of objects over time. We theorized that objects whose reuse time distribution was biased towards lower reuse times were beneficial compared to those whose distribution was biased towards higher reuse times.

We are acting with incomplete information. AWS S3 provides access to parts of a single file by specifying a byte range. Spark takes advantage of that reads parts of a file. However, due to logging and log-access restrictions, in practice we do not know which part of the file to cache. Thus, while designing our policy and experiments, we considered caching of whole files. If a file has already been accessed and is in the cache there would be no cache miss. We think the design decisions and performance results from our setup are applicable to the system under consideration, if caching is done at the whole file level. We conjecture that they are applicable even when parts of the file are cached and leave proving this for future work.

5.5. Policy Design

We partition the available cache into two parts: the *candidate cache* and the *main cache*. The candidate cache has an admit policy of admitting everything and an LRU based eviction policy. It can be anything from 1% of the total cache size to 10%. The exact value requires tuning which we leave for future work. It has also been called a buffer cache, window cache or the first cache segment in other literature. The main cache occupies the remaining space. We propose an eviction policy inspired by LHD for the main cache.

5.5.1. Candidate Cache

Before elaborating on the need for a candidate cache, we need to present the concepts of intrinsic and extrinsic features of a read. A feature is a property associated with a read which is used in making cache related decisions. All the properties we statistically characterize in Chapter 3 are features. An *intrinsic feature* is a property of the read that is independent of the workload. Examples of intrinsic features include the size of file, column in the database which this file represents, the owner of a file and the file name. These properties are intrinsic to the file and would remain the same irrespective of the workload accessing the file. An *extrinsic feature* is a property of the read that only exists due to the workload. The most intuitive example of this is the *popularity* of the file. How many times a file is accessed depends exclusively on the workload and nothing else. Another example is the reuse time: time since the file was last accessed. The features only pop into existence due to an external influence and are thus called *extrinsic features*.

A candidate cache with a simple admission and eviction policy gives the cache metadata to capture the extrinsic properties of an item. Cache metadata includes information such as number of times a file is accessed and the last time it was accessed that can help make admission and eviction decisions. One example of a system which uses a candidate cache to allow an extrinsic feature, number of accesses, to be captured before using the main eviction algorithm is Hyperbolic caching [15]. Capture means that the item has been accessed a recent number of times and the system has enough metadata about it. This capture time is helpful for policies like W-TinyLFU which depend on extrinsic features such as number of reads. It is not necessary for policies like AdaptSize which use an intrinsic feature such as file size. The size of candidate cache and the number of candidate caches are hyper-parameters of the caching policy and are areas of future work. We consider the case with one candidate cache.

For the candidate cache, we admit everything and use the LRU policy. When a burst of reads of a file occurs, it is accessed repeatedly and not evicted by the LRU policy. When the burst stops and there is a lull, it is at some point going to be evicted by the LRU policy. Otherwise it is still in the cache leading to hits. The LRU cache is implemented using a Linked List data structure whereby newly accessed elements move to the head of the cache. The element at the tail of the list is evicted.

Mapping to our reference architecture of a cache policy, the candidate cache has the entry metric of allow all, admission metric of allow all, eviction metric based on recency of use, and an ordered licked list for victim selection.

5.5.2. Entry to Main Cache

We explained the general structure of the cache with two parts: candidate cache and main cache. In Subsection 5.5.1, we explained the mechanism of entry into the candidate cache, admit everything, and the mechanism of eviction from the candidate cache, LRU. In this section, we describe the movement mechanism of an object from the candidate cache to the main cache. There are multiple techniques for this. They can broadly divided into two categories: move on hit and move on miss. SLRU uses a move on hit technique where an object is moved to the main cache after a certain number of hits. This is checked after every hit. TinyLFU uses a move on miss technique which compares the popularity of object to the least popular object in the main cache once and admits the object if it is more popular.

We chose a policy where an object is checked for inclusion into the main cache once it has been evicted from the candidate cache. This comes with the advantage that hits to the candidate cache are cheap. There is no computation to check for inclusion into main cache on every hit. Therefore, hits are truly lightweight and don't take resources from actual jobs running on the computer. An item eligible to be in the main cache early on should also be eligible when its evicted from the candidate cache given that it a simple LRU cache. For example, a popular item should remain popular even if it is being evicted due to a time constraint. If it is not, then it did not have any long term value and was ineligible to be in the main cache anyway. Elements in the main cache are those which provide long term value.

We experiment with a few admission metrics. These metrics will use extrinsic features such as popularity (LFU) as opposed to intrinsic features such as size (AdaptSize). The policy thus remains flexible as intrinsic metrics such as application name, type and user are not available to the cache policy in a lot of environments. These can be combined with techniques that use intrinsic features but that is outside the scope of this work. The techniques we will try are: TinyLFU and ReadDensity.

A **TinyLFU** admission metric keeps track of the popularities of all elements ever accessed instead of only those present in the cache. This helps in those cases where items are unable to gain entry into the main cache but return again some time later. Information from their previous attempt is preserved instead of starting their popularity counter from start every time they enter the candidate cache. A sampling based victim selection is done to determine the least popular element.

A **ReadDensity** admission metric is the one detailed in Subsection 5.5.3 for use during eviction. We test its effectiveness for admission.

5.5.3. Eviction from Main Cache: The Read Density Metric

This section describes the eviction metric for object which are in the main cache. Objects in the main cache need removal to make space for new objects. This is required because the popularity of objects changes over time and they become unused. We propose a method for tracking popularity of objects at different reuse times. This is based on the intuition that an object which is read in the near future is more valuable than an object which is read in the distant future. We also consider the case that an object which is accessed many times in the distant future might be more valuable than an object accessed few times in the near future. This ability to quantify the reward of objects at different time scales makes the cache adaptive.

We are able to do this using a data structure called *approximate histogram*. It is described later in Section 5.6. For now, just consider a normal histogram with reuse time on the horizontal axis and number of times read (popularity) on the vertical axis. As objects enter the main cache only after they spend time in the candidate cache, there is already some information about which time after a read the next read is likely to occur. The histograms are also updated as long as an object is in the main cache. Dividing the popularity of an object at a particular reuse time by the total number of reads gives the empirical probability that the object is read in that time.

$$P(R = o | I = t) = \frac{N(R = o | I = t)}{\text{Total Number of Reads}} \tag{5.1}$$

where R is the random variable of reads of objects and I is the random variable of reuse times. o is a particular object and t is a particular reuse time.

From this, we use the normal definition of expectation to obtain the expectation of a read for a given

object.

$$E(R = o) = \sum_{i=0}^{n} 1 \times P(R = o|I = i) \tag{5.2}$$

where 1 is the value of each read and the others have their previous meanings. We already have some information about the object which we can incorporate into this which is the time since the object was last accessed, henceforth called age.

$$E(R = o) = \sum_{i=a}^{n} 1 \times P(R = o|I = i) \tag{5.3}$$

where a is the age of the object.

Notice that equations 5.2 and 5.3 do not penalize reads with high reuse time nor do they reward reads with low reuse time. We need to include these as earlier reads are more valuable than later reads. Consider each object in the cache as using a resource. We call that resource a slot. An object in the main cache uses a slot. The longer an object uses a slot, the more of the slot resource is being consumed. We are trying to maximize the number of hits. An object which maximizes hits while using a slot for the least amount of time is desirable and is said to have the most *read density*.

The penalty can be applied in two ways. One is to divide the expectation of a read by the expected reuse time. This is the approach LHD takes. The other is to divide the probability of a read at a particular reuse time by that reuse time.

Using Expected Reuse Time

The expected reuse time of a particular object is given by

$$EI(R = o) = \sum_{i=a}^{n} i \times P(R = o|I = i) \tag{5.4}$$

This is similar to equation 5.3, but the value of a hit is reuse time itself instead of one.

Using equations 5.3 and 5.4, we present the value function of an object in the main cache. It can be thought of as the reward for letting an item stay in the cache and called read density using expected reuse time, $D1$.

$$D1(R = o) = \frac{\sum_{i=a}^{n} 1 \times P(R = o|I = i)}{\sum_{i=a}^{n} i \times P(R = o|I = i)} \tag{5.5}$$

which is the expected probability of hit divided by the expected reuse time.

Using Division by Reuse Time

Instead of calculating the expected reuse time, we can penalize the probability of a read directly during the expected probability calculation. This can also be thought of as a reward for letting an object stay in the cache. It is called read density using explicit penalties, $D2$.

$$D2(R = o) = \sum_{i=a}^{n} 11 \times \frac{P(R = o|I = i)}{i} \tag{5.6}$$

5.5.4. Victim Selection

In a lot of caching policies, some kind of ordered data structure, be it a tree, heap or linked list is used. Ordering objects when the order depends on time is hard. The priority (computed metric) of every object has to be computed and updated. The eviction metric we chose is based on time. If it is computed for every object on every eviction decision, this would be a very expensive policy with a $O(n)$ runtime. We use an partial order instead of a total order. Certain number of elements (s) are sampled from the list of all objects. This is an $O(s)$ operation and $s << n$. The priorities for these elements are computed using the eviction metric.

Systematic sampling is used to sample the items in $O(s)$ time. The probability of an item getting picked in systematic sampling is equal to that in uniform sampling. Systematic sampling is more efficient. The process is as follows. First, the stride length is computed by dividing the size of cache by the sample size. A random integer between 0 and the size of cache is chosen. The object at that index in cache is the first element of the sample. Index of next element is chosen by incremented the first index by the stride length and rounding it to the nearest integer. Index of third element is chosen by incrementing the first index by twice the stride length and so on.

Mapping to our reference architecture of a cache policy, the entry metric is allow all, the admission metric is TinyLFU or ReadDensity, the eviction metric is ReadDensity and the victim selection mechanism is sampling.

5.6. Approximate Histogram

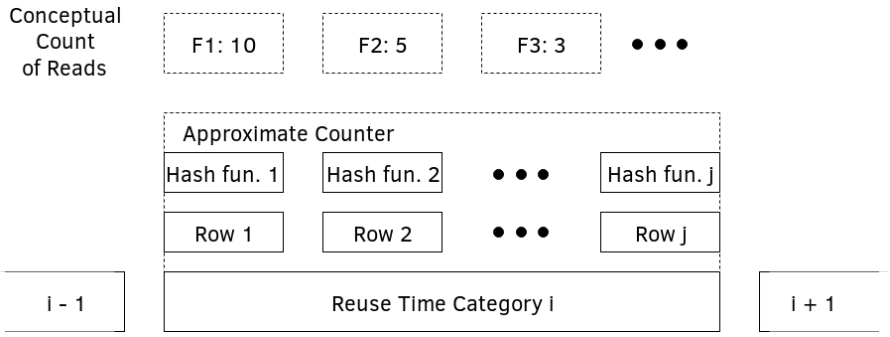


Figure 5.2: An Approximate Histogram.

In Subsection 5.5.3, we used histograms of reuse time vs number of reads to calculate reward functions to rank objects for eviction. Keeping complete histograms for each object is not feasible due to space constraints. LHD solves this problem by categorizing items using features such as application type and reuse time of last read. This puts the onus of deciding the optimal categories on the system designer. In a lot of cases, a lot of useful features of reads aren't available to the cache policy. For example, a filesystem cache might not know the application type. We propose approximate histograms as a mitigation to this problem. Using them, we keep track of the histogram for every object without the space requirements of storing whole histograms. Using approximate histograms instead requires less parameter tuning than using hand picked categories. Hand picking categories requires selection of categories and the granularity at which each category is tracked. In this section, we describe how the approximate histogram functions.

An approximate histogram is an array of approximate counters coupled with a decay function. One is depicted in Figure 5.2. Each counter in the array represents counts of objects accessed in a reuse time category. Reuse time needs to be divided into coarse grained categories to avoid blowing up memory usage and to make the histogram immune to noise. A very long time range is taken and linearly or logarithmically broken into a reasonable number of adjacent categories. For example, say the time range under consideration is 0 to 1000 milliseconds. This can be divided into adjacent categories of 100 milliseconds each such as 0-99, 100-199 and so on.

Each category has an approximate counter. Examples of approximate counting data structures include count-min sketch [21], cuckoo filter [26] and counting quotient filter [53]. They have different storage and duplication resistance properties. An approximate counter is also known as a sketch. When an object is read, the counter in the corresponding reuse time category is incremented for that object. In this work, we use the count-min sketch. But, any approximate counting data structure can be used based on performance and maintenance requirements.

We briefly explain the operating principle of a count min sketch. A count-min sketch consists of several arrays each associated with its own hash function, each called a row. The hash function takes the key of an object and returns a position in the array. Whenever an object is read, the positions with the minimum value across all rows for this object are incremented by one. When queried for the number of times an object has been read, the count-min sketch can give the minimum number of times an object has been read. This is because only the positions with minimum values were updated during insertion. If some hash functions clash with keys of other objects, they might update some positions corresponding to the current object but not all of them. Thus, the minimum value remains immune to clashes.

Caches in an application can run for a long time. During the runtime, the counters can get saturated. To prevent saturation, the counters can be reset to zero periodically. They can also be divided by a constant periodically (called decay). The reset or decay is done for counters across the whole histogram. Each position in each row in each reuse time category is reset or decayed at the same time. Approximate counters are small in number and located together in memory. Decaying all of them is not very expensive. This operation is only done occasionally, whenever a counter is saturated, amortizing its cost over a large number of reads.

We use the technique presented in TinyLFU. Briefly, the different positions in arrays which constitute and approximate counter are allowed to saturate. All counters are reset periodically. This period is determined from a scalar time which is incremented by one for every increment of the approximate counter. For example, the period can be set such that the counters are decayed every 10000 reads.

5.7. Implementation

We implement our policy in the simulation framework in the popular cache library Caffeine. An eviction policy has just one public function, record. Locations hashed to 64-bit integers are passed to this function and the policy reports a hit or miss and an eviction in case of a miss. An admission policy has two public functions, record and admit. All reads are passed to the record function so that the admission policy metadata can be updated. The admit function is queries with two arguments, the candidate to be entered into cache and the chosen eviction victim to make place for this candidate. The admit function is called before every decision to actually evict by the eviction policy.

The approximate histogram is composed of a pre-configured number of approximate counters. Each approximate counter is a modified count-min sketch. Each count-min sketch is a collection of arrays. The number and length of the arrays is decided from the values configured for epsilon and confidence. Epsilon and confidence carry the same connotation as in the original count-min sketch paper. The main modification is to synchronize the reset of all sketches in the approximate histogram. This is carried out by way of a proxy object which counts reads and initiates a reset when the reset interval is reached. The reset interval is set to a multiple of the total cache size. It conceptually corresponds to the size of the working set. We assume that the working set is an order of magnitude larger than the total cache size. The maximum value of a counter in count-min sketch is capped.

For measuring reuse time, we use a counter incremented at every read as scalar time.

5.8. Limitations

We discuss three main limitation of our design.

Past Not Representative of Future Performance

The read density eviction metric assumes that past data about reads can directly be used to predict future reads. This assumption allows us to use histograms which keep track of past patterns to predict future patterns. This might not be true and might not be the best way to go forward for all workloads.

Linear Relationship Between Reads and Time

To compute read density, the number of reads are divided by time. We consider this the utility of presence of an object in the cache. This assumption that they are linearly related might not be true. The utility can be a function where these values are exponential parameters. The utility can also be a function where these values have different exponential or logarithmic powers.

CPU utilization

We sample object from eviction from the cache and compute the read density of each object. Each read density computation involves 32 additions (number of reuse time categories: 32), 5 comparisons per category (pick minimum value from approximate histogram). When creating a sample of 64 objects, that results in $64 \times 32 \times 5 \times 2 = 20,480$ operations. This makes it one of the more expensive cache policies. Similar to LHD, the cache can be partitioned and the estimates for different partitions computed concurrently. Big data workloads, unlike web caches, don't experience high throughput in number of reads. Hence, the high cost per read is acceptable.

6

Evaluation of Cache Policies for Big Data Workloads

To respond to the increasing diversity and complexity of big data workloads, the community needs comparative studies on performance of cache policies focused on big data workloads. Motivated by the lack of such comparative studies and to provide evidence supporting claims we made in Chapter 5 about the operation and performance of our cache policy. In this section, we answer the research question: *How do cache policies perform on Spark-based big data storage workloads in the cloud?*.

6.1. Overview

Towards answering the research question this chapter focuses on, our contribution is two-fold:

1. We perform a parameter sweep on the our policy (ARD) proposed in Chapter 5 to compare different eviction metrics from the read density family and to show its lack of sensitivity to parameter variations.
2. We evaluate several cache policies on two big data workload traces from Databricks and Yahoo. This allows us to compare ARD with many other important policies, in realistic scenarios.

6.2. Evaluation Process

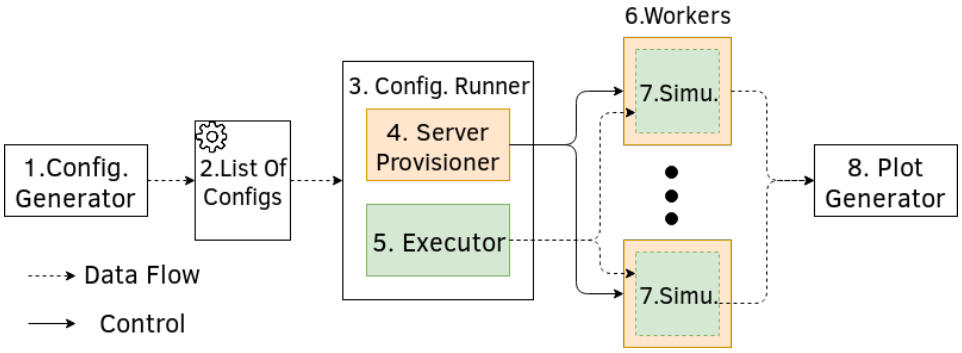


Figure 6.1: Cache policy evaluation system.

We evaluate the performance of several policies with different configurations using trace-based simulation. This required running over 10,000 simulations. We designed a system to run them in parallel using the DAS-5 [8] supercomputer. This system is depicted in Figure 6.1.

The **configuration generator** (component 1) generates a **list of configurations** (component 2). The configuration generator takes possible values of a configuration option as the input. For example, cache size can be 1,000 objects, 10,000, and so on. All possible permutations of the passed in configuration options are generated and written to the list of configurations.

The list of configurations is parsed by the **configuration runner**. A new **worker** (component 6) server is reserved by the **server provisioner** (component 4) using the DAS-5 scheduler for every 16 configurations. Each server in the DAS-5 has 16 cores and we run one simulation on every core. The **configuration executor** (component 5) starts the 16 simulations.

Simulation is performed by the **simulator** (component 7) component of Caffeine [1], which is popular in-memory cache library for Java. This simulator has been previously used in at least one other high-quality peer-reviewed study [25] published in Transaction on Storage. The simulation results are accumulated and processed to generate performance plots for human analysis.

6.3. Experimental Setup

All the experiment we conduct in this chapter are listed in Table 6.1. Cache size 5,000 to 625,000 refers to experiments with caches of size 5,000, 25,000, 125,000, and 625,000 objects. These sizes are in line with real world usage of caches for big data processing in our experience. The ratio of working set size to cache size for these cache sizes are of the order 10,000, 1000, 100, and 10 respectively. The cache sizes are large compared to other evaluation studies. Later, we also present the progression of hate as the cache size increases in small caches. The small cache sizes (100 - 5,000) correspond to experiments with caches of size 100, 300, 500, 700, 900, 1,000, 3,000, 5,000, 25,000, and 125,000 objects.

The Databricks workload trace is a subset of trace W2 analyzed in Chapter 3. The Yahoo trace is a subset of the publicly available Yahoo Webscope 3 dataset [2] of accesses to HDFS.

The performance metric that is measured in all experiments is the *hit rate*, which refers to the total fraction of reads that were hits. All experiments were run ten times and the mean hit rate for each configuration was used. The standard deviation of the results was less that 1%.

We proposed two metrics under belonging to the Read Density family in Chapter 5 Section 5.5.3: 1. Using expected reuse time, and 2. Using division by reuse time. We compare the hit rate performance the these two metrics along with just a metric based on expectation of read. We also compare the efficacy of these metrics when used as admission metrics.

Table 6.1: List of experiments and their configurations.

Subsection	Workload	Name	Policy	Parameters Varied	Cache Sizes
6.5.1	Databricks	Comparison of Read Density base eviction metrics	ARD	eviction metric	5,000 - 625,000
6.5.2	Databricks	Sensitivity to parameters of approximate histogram	ARD	max count, num. time categories	5,000 - 625,000
6.5.3	Databricks	Comparison of admission metric and eviction metric pairs	ARD	admission metric, eviction metric	5,000 - 625,000
6.6.1	Databricks	Compare eviction policies for Databricks workload	All	None	5,000 - 625,000
6.6.2	Yahoo	Compare eviction policies for Yahoo workload	All	None	5,000 - 625,000
6.6.3	Databricks	Compare eviction policies for Databricks workload for small caches	All	None	100 - 5,000
6.6.3	Yahoo	Compare eviction policies for Yahoo workload for small caches	All	None	100 - 5,000

6.4. List of Evaluated Policies

We describe briefly each policy we evaluate. The policies are chosen to represent the design approaches available in the cache-policy design space (See Chapter 5). LRU, LFU, FIFO and Random are commonly used simple to implement policies. CLOCK keeps object in cache longer than an LRU to accommodate long range reuse. Segmented LRU and S4LRU keep objects in cache longer by maintaining different partitions for objects read few times and those read many times. LIRS makes use of reuse time. ARC and CART change cache space

[1] https://github.com/ben-manes/caffeine
[2] https://webscope.sandbox.yahoo.com/catalog.php?datatype=s&did=57

allocation to different policies based on the workload. Hyperbolic caching, W-TinyLFU, and LHD are all recent algorithms per our systematic study. Approximate Read Density is the policy proposed in Chapter 5.

All policies were tuned using a naive grid search of the parameters. We believe this is representative of their usage in the real world, as opposed to being finely tuned to optimal performance. Cache policies are generally part of a larger system. Therefore, limited time and engineering effort are allocated its implementation. The best policy which fits within the time constraint is chosen. This is unlike research studies where researchers sometimes spend months optimizing a policy.

In the following, we detail the 13 policies we evaluate:

P1 Least Recently Used (LRU) evicts the least recently used object using a linked list to keep track of recency of use.

P2 Least Frequently Used (LFU) evicts the object which has been read the least number of times. It uses a linked list to track the number of times an object has been read.

P3 First In First Out (FIFO) evicts objects in the order of admission.

P4 Random evicts a random object.

P5 CLOCK is similar to FIFO. But, an object is not evicted the first time reaches he head of the linked list. It is only evicted the second time.

P6 Segmented LRU (SLRU) divides the cache into two partitions. Both partitions implement the LRU eviction policy. An object enters the second partition from the first if it is read while in the second partition. The fraction occupied by the second partition was set to 90%.

P7 S4LRU [37] is similar to SLRU but uses four equally sized partitions.

P8 LIRS [40] evicts items based the reuse distance of their last read.

P9 Adaptive Replacement Cache (ARC) [47] partitions the cache into two parts. One uses a LRU eviction policy and other a LFU eviction policy. Both partitions continuously compete for space based on their hit rate.

P10 CLOCK with Adaptive Replacement and Temporal Filtering(CART) [9] uses different LRU and LFU partition similar to ARC. Items are evicted on second eviction instead of first similar to the CLOCK policy. Items are promoted from the LRU partition to the LFU partition if an item is evicted from LRU within a specified time window.

P11 Hyperbolic [15] computes the utility of objects by dividing number of reads of an object by the time it spent in the cache. The cache is sampled and the object with the least utility is evicted.

P12 W-TinyLFU [25] divides the cache into two partitions. The first partition uses a simple LRU cache. On eviction from the first partition, an object attempts to enter the second partition. The second partition implements an SLRU eviction policy and TinyLFU admission policy. We experiment with two variants of W-TinyLFU. One using SLRU eviction policy and the other using LRU. The fraction of main cache as a percentage of total cache is set at 80%.

P13 Least Hit Density (LHD) [11] Classifies items into categories and estimates the number of future reads and time to eviction based on the number of past reads. The hit density metric refers to the estimated number of reads divided by the estimated time to eviction. The object with the least hit density gets evicted.

We also evaluate the efficacy of using the TinyLFU and Read Density (from Chapter 5) as admission policies with each of the aforementioned eviction policies, except W-TinyLFU where it is already present.

6.5. Parameter Sweep of Approximate Read Density Policy

We claim that the Approximate Read Density (ARD) policy has less parameters than LHD. While that is true, it still has some parameters. We demonstrate the insensitivity of hit rate to the remaining parameters in reasonable ranges. A policy is said to be sensitive to its parameters if the exhibited performance is changes based on the values of the parameters. A policy is insensitive to its parameters means that its performance doesn't change when the parameters change. Reasonable ranges means that the parameters should not be set to absurd values such as zero or extremely large values.

Our main finding in this section is:

MF6.3.1 The ARD cache policy is insensitive to parameters in reasonable ranges.

6.5.1. Eviction Metric Comparison

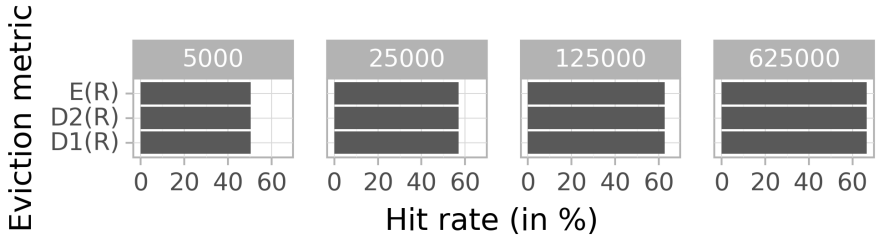


Figure 6.2: Comparison of eviction metrics. Each panel corresponds to a different cache size (number of objects).

Three eviction metrics were describe in Chapter 5: Expectation of read (E(R)), Expectation of read divided by expected reuse time (D1(R)), and Expectation of read penalized by division (D2(R)). We compare their performance on the Databricks trace using a cache with entry and admission metrics set to allow all. The results are depicted in Figure 6.2. We observe that all three eviction policies perform equally well at all measured cache sizes.

6.5.2. Insensitivity to Parameters of Eviction Policy

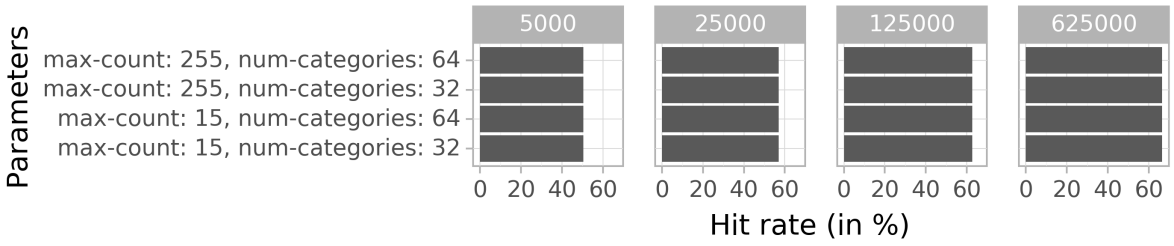


Figure 6.3: Comparison of different parameter sets. Each panel corresponds to a different cache size (number of objects).

Our eviction policy takes two parameters: maximum count and number of categories. Maximum count is the maximum count of the number of reads of an object stored per time category in the approximate histogram. Any reads higher than that are not counted till the histogram is reset. The number of categories refers to the number of categories of reuse time.

The results are depicted in Figure 6.3. We observe that the parameters don't have any effect on the performance of the cache policy as long as they are reasonable. Too few categories, 1 for example, and the algorithm degenerates to LFU. Too many, 1,000 for example, and the policy starves for data. The parameters should be chosen in relation to a reset interval explained in Chapter 5. The reset interval is the number of reads after which the counters in the histogram are reset. This generally corresponds to the size of the working set. We use a value of 10,000. We find that a reasonable interval functions well. Too low, and the policy doesn't have enough time to gather information. Too high, and all the counters are saturated leading to loss of information.

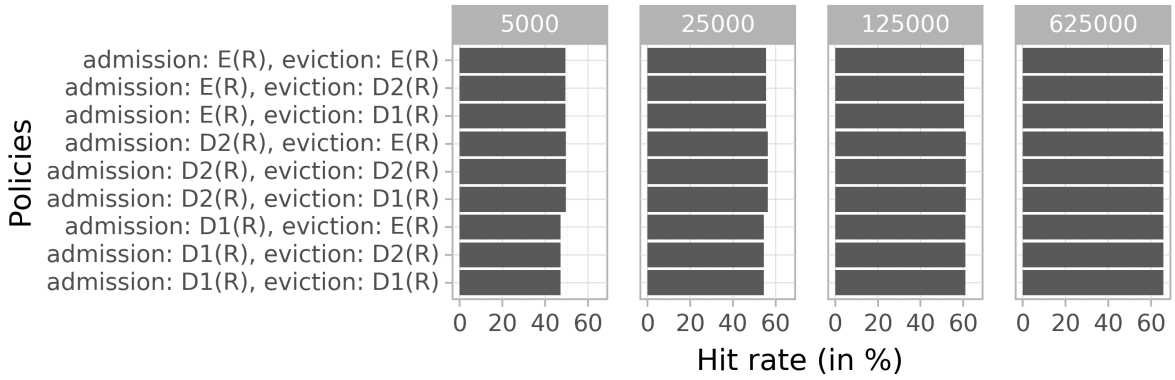


Figure 6.4: Comparison of different parameter sets for ARD with admission control. Each panel corresponds to a different cache size (number of objects).

6.5.3. Admission Metric Comparison

We simulate the ARD policy with admission control. The cahe is divided into two partitions. The first has entry and admission policies of allow all, and eviction policy LRU. The second had entry policy allow all, and a range of admission and eviction policies. 20% of the total cache space was allocated to the second partition. The results of the simulation are depicted in Figure 6.4. We observe that the difference between the various policy combinationsare less than 1%.

6.6. Cache Policy Comparison

We compare the policies described in Section 6.4 using two traces: Databricks and Yahoo. The policies with randomization elements were run ten times and the results were averaged. The standard deviation was less than 1%. The policies are first compared at large cache sizes, 5,000 to 625,000.

The main findings of this section are:

MF6.5.1 For large caches (working set to cache ratio less than 100,000), most temporal locality based cache policies perform similarly. Therefore, it is recommended to go with a simple policy like LRU.

MF6.5.2 A small cache renders significant benefit compared to no cache at all.

6.6.1. Databricks Trace

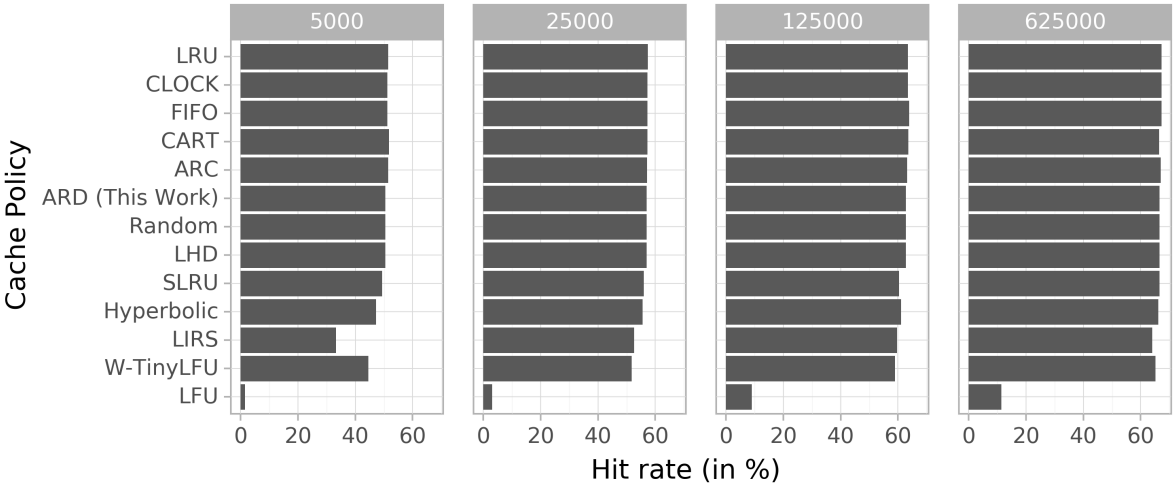


Figure 6.5: Comparison using Databricks trace. Each panel corresponds to a different cache size (number of objects).

The results of simulating the Databricks trace are depicted in Figure 6.5. The only policy that performs significantly worse at all cache sizes is LFU. All policies which focus on temporal locality perform or have a

component focused on it perform well (**MF6.5.1**). We conjecture that this is due to the peculiar usage pattern of big data workloads. A file is read many times in a short interval because it is being processed. Then, it is not read again for long period of time till it is required again. So, unless an object remains in the cache for days, it is unlikely that algorithms based solely on popularity perform well.

6.6.2. Yahoo Trace

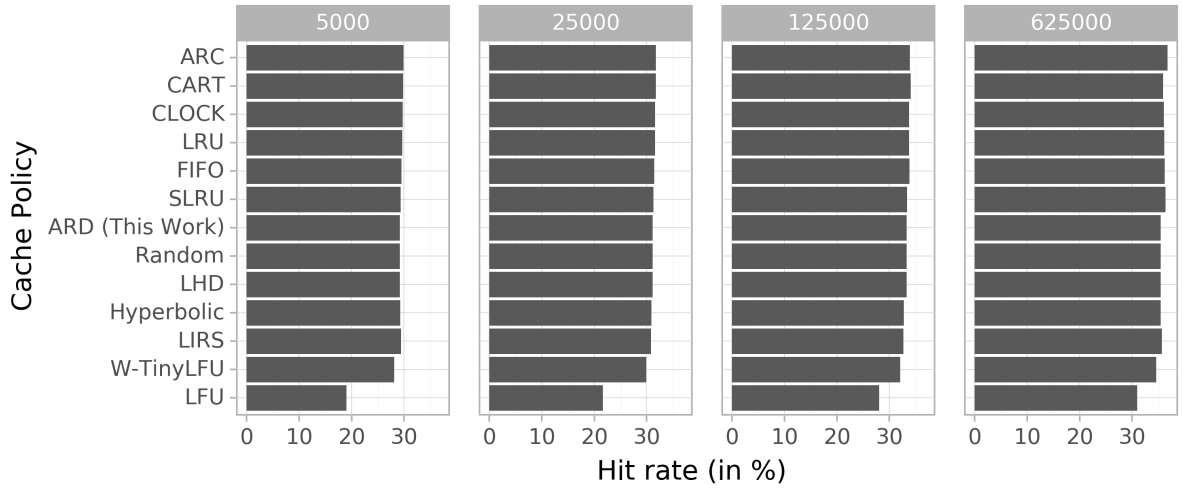


Figure 6.6: Comparison using Yahoo trace. Each panel corresponds to a different cache size (number of objects).

The results of simulating the Databricks trace are depicted in Figure 6.5. The results are similar the Databricks trace. All policies which take temporal locality into account perform well, but the difference isn't significant (**MF6.5.1**). It seems that there is more opportunity for algorithms which take popularity into account to perform well here. This is evidenced by the success of ARC and CART. It is also evidenced by LFU perform better that it did with the Databricks trace.

6.6.3. Comparison At Small Cache Sizes

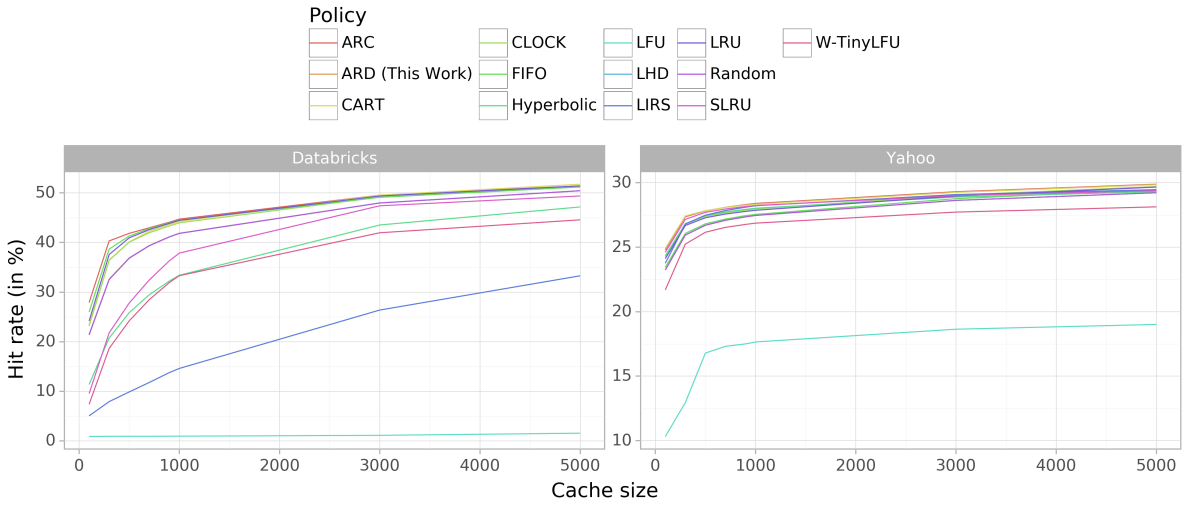


Figure 6.7: Comparison of policies at small cache sizes (number of objects). Each panel corresponds to a separate trace.

The results of simulating both traces as small caches sizes (< 5000 objects) are depicted in Figure 6.7. We observe that there is no significant difference in the performance of top performers at small cache sizes. We also observe that even small caches (< 1000 objects) can have a 40% hit rate and are thus better than having no cache at all (**MF6.5.2**).

6.6.4. ARD and LHD Compared

We observe that while ARD improves over LHD, the policy it intended to improve upon, the improvement is marginal. We conjecture that the reason behind for this is the difference between reuse time and the file active span distributions. Reuse times are typically short with 75% shorter than 100 milliseconds. File active span is the time elapsed between the first read of a file and the last. We observed that file active spans are typically quite long, with 75% taking longer than a day. This implies that a file is read a lot of times in less than 100 seconds and then is not read again for a few hours, maybe even a day. In the mean time, other files are read and enter the cache because old information about reads is decayed by our reset function. Therefore, ARD is not able to use information about reuse time distribution of file to its advantage the next time it is read after a long gap.

7

Conclusion

In this chapter, we summarize our main findings and contributions. We associate them with the research questions they answer. Finally, we discuss future work.

7.1. Summary of Answers to Main Research Questions

1. What are the characteristics of Spark-based big data storage workloads in the cloud?

   Big data storage workloads have received much attention at both the hardware level and at the level of massive clusters. There was a lack of publicly available information about behavior of big data workloads in small virtual clusters in the cloud. We tackle this problem and characterize the behavior of a large deployment of small clusters at Databricks.

   Main Findings in Chapter 3:

   MF3.3.1 The number of reads and bytes read per day have doubled over 6 months.

   MF3.3.2 The number modifications per day has remained at the same level throughout the analysis period.

   MF3.3.3 Both reads and modifications follow a diurnal pattern.

   MF3.3.4 Large imbalance in number of reads and bytes read per hour occur on *daily* and *weekly* basis.

   MF3.3.5 There are 2 orders of magnitude less modifications happening than reads.

   MF3.3.6 Most modifications are file creations.

   MF3.5.1 The number of reads and bytes read exhibit negative long range time dependence.

   MF3.5.2 All features (size, popularity, etc.) have a heavy tail.

   MF3.5.3 Reads occur in bursts.

   MF3.5.4 The distributions of features are stationary over long time periods.

   MF3.6.1 The distribution of number of reads and bytes read over clusters is heavy tailed.

   MF3.7.1 Parquet is the most popular file format we observed in practice.

   MF3.7.2 Snappy and Gzip are the most popular compression schemes we observed in practice.

2. How can the characteristics such workloads be modeled to generate synthetic traces?

   Real world traces of big data workloads are rare. This problem is stark for big data workloads deployed in the cloud. Limited availability of traces leads to researchers focusing on problems uncovered by benchmarks or the few traces available. These problems might not be representative of those in the wider ecosystem. Hence, we present a model to generate synthetic traces for bi data workloads in the cloud.

   Main Contributions of Chapter 4:

   (a) A generative model to reproduce interarrival times, temporal and spatial locality distributions of original traces.

(b) Validation of the generative model using emergent feature, popularity, and modeled feature, reuse time.

(c) Performance evaluation of the proposed generative model.

Main Findings in Chapter 4:

MF4.3.1 Among the methods we have used to fit heavy tailed distributions observed in computer systems. The unweighted least-squares regression of their survival functions gives the best result.

MF4.3.2 We confirm the heavy tailed nature of the distributions, by observing that their survival function is above that of the exponential distribution.

MF4.4.1 The synthetic trace has a file popularity distribution similar to the original trace.

MF4.4.2 The popularity distribution is only similar if the chunks are sufficiently long for a file to be read a large number of times.

MF4.4.3 The reuse time distribution of the synthetic trace is similar to that of the original trace.

3. What is a principled approach to design an adaptive cache policy with few parameters?

   Implementation and tuning of complex cache policies is hard and error prone. We make several contributions towards making cache policies easy to understand and implement.

   Main Contributions of Chapter 5:

   (a) A reference architecture for online cache policies.

   (b) The Read Density family of eviction metrics.

   (c) The Approximate Read Density cache policy using the Read Density metric.

   (d) The Approximate Histogram data structure to measure distribution of reads across time.

4. How do cache policies perform on such workloads?

   Evaluations of cache policies are typically dominated by web workloads. We present the first evaluation of cache policies on big data workloads.

   Main Findings in Chapter 6:

   MF6.3.1 The ARD cache policy is insensitive to parameters in reasonable ranges.

   MF6.5.1 For large caches (working set to cache ratio less than 100,000), most temporal locality based cache policies perform similarly. Therefore, it is recommended to go with a simple policy like LRU.

   MF6.5.2 A small cache renders significant benefit compared to no cache at all.

7.2. Future Work

The use of large-scale computers (warehouse scale computing) is growing. We, and the community, expect the growth to continue into the future. For such scales, it becomes difficult to research and design systems for large scale computing by way of simple observation. Thus, characterization becomes important as it distills the complexity of key designs and systems into essential characteristics. We present in this work a preliminary characterization of a storage workload. We identify critical features and characteristics in these complex systems. We also observe that *the analysis necessarily becomes more complex and sophisticated.* We identify two directions for future work on more sophisticated analysis: (1) In our work, we use elements of univariate statistics to analyze features. While we investigate linear correlation, complex non-linear correlations might exist between different features. Multivariate analysis and unsupervised learning are disciplines which deal with such interactions. In the future, we expect to apply tools from these disciplines to the data collected from this analysis. A way to do this is to use simple k-means clustering on a representative sample of data read operations. (2) A large ecosystem can consist of many subsystems. Methods to investigate and represent the diversity of sub-systems need to be investigated. For example, an ecosystem has 10,000 virtual clusters each with different properties. A principled technique to study this diversity is required.

Another challenge that comes with the proliferation of massive-scale computing is the *unavailability of trace data for simulations.* Thus, generative models become important. We use techniques from univariate statistics for modeling features of distributions in traces for our generative model. We propose two future

research directions: (1) Multivariate statistics and machine learning have many more tools to offer for more accurate modeling. The disadvantage of using them is that it is difficult to synthesize what if scenarios. What if the popularity distribution is more skewed? What if the interarrival time distribution is tighter? To better model what if scenarios, research is being actively conducted on causal reasoning, interpretation, and human in the loop models in machine learning. That research could be used to generate more representative traces. (2) An investigation of the relationship between goodness of fit and variance in performance results of simulations using synthetic traces is necessary. This would answer questions such as: Is fitting the body or the tail of the distribution more important to get representative results in experiments?

Historically, cache policies have been designed using analytical reasoning and with specific use cases in mind. LHD [11] introduced a principled approach to cache policy design using probability theory. We extend the principled approach by proposing a reference architecture and family of eviction metrics. We propose two future research directions: (1) There is much work done on time series prediction in mathematics, finance, economics and physics. This work could be leveraged to design cache policies based on strong conceptual foundations. (2) Cache policies at multiple sub-system levels such as cluster and worker levels and their interplay is a rich area of study. Caches designed for shared and heterogeneous resource usage, as found in the cloud, also need to be investigated.

Scheduling takes about 5% of datacenter compute usage [41]. This might grow, as ecosystems become larger. Usage of approximate computations and data structures allow scheduling techniques to scale linearly or sub-linearly with the scale of the datacenter. There is much room for research into using approximation techniques to achieve acceptable scheduling performance with low resource cost in massive computer systems.

Lastly, we intend to test the performance of the ARD policy on other workloads apart from Big Data. We believe other workloads might exhibit more favorable reuse time and file active span distributions for ARD to make use of accumulated knowledge before it decays away.

Bibliography

[1] 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 2012 IEEE International Symposium on Workload Characterization, IISWC 2012, La Jolla, CA, USA, November 4-6, 2012*, pages 100–109, 2012. doi: 10.1109/IISWC.2012.6402909. URL https://doi.org/10.1109/IISWC.2012.6402909.

[2] Cristina L. Abad, Mindi Yuan, Chris X. Cai, Yi Lu, Nathan Roberts, and Roy H. Campbell. Generating request streams on big data using clustered renewal processes. *Perform. Eval.*, 70(10):704–719, 2013. doi: 10.1016/j.peva.2013.08.006. URL https://doi.org/10.1016/j.peva.2013.08.006.

[3] Cristina L. Abad, Andres G. Abad, and Luis E. Lucio. Dynamic memory partitioning for cloud caches with heterogeneous backends. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 87–90, 2017. doi: 10.1145/3030207.3030237. URL https://doi.org/10.1145/3030207.3030237.

[4] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, François Labelle, Nate Coehlo, Xudong Shi, and Eric Schrock. Janus: Optimal flash provisioning for cloud storage workloads. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 91–102, 2013. URL https://www.usenix.org/conference/atc13/technical-sessions/presentation/albrecht.

[5] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, pages 533–546, 2018. URL https://www.usenix.org/conference/atc18/presentation/amvrosiadis.

[6] Georgios Andreadis, Laurens Versluis, Fabian Mastenbroek, and Alexandru Iosup. A reference architecture for datacenter scheduling: design, validation, and experiments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*, pages 37:1–37:15, 2018. URL http://dl.acm.org/citation.cfm?id=3291706.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, pages 53–64, 2012. doi: 10.1145/2254756.2254766. URL http://doi.acm.org/10.1145/2254756.2254766.

[8] Henri E. Bal, Dick H. J. Epema, Cees de Laat, Rob van Nieuwpoort, John W. Romein, Frank J. Seinstra, Cees Snoek, and Harry A. G. Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *IEEE Computer*, 49(5):54–63, 2016. doi: 10.1109/MC.2016.127. URL https://doi.org/10.1109/MC.2016.127.

[9] Sorav Bansal and Dharmendra S. Modha. CAR: clock with adaptive replacement. In *Proceedings of the FAST '04 Conference on File and Storage Technologies, March 31 - April 2, 2004, Grand Hyatt Hotel, San Francisco, California, USA*, pages 187–200, 2004. URL http://www.usenix.org/events/fast04/tech/bansal.html.

[10] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013. ISBN 9781627050098. doi: 10.2200/S00516ED2V01Y201306CAC024. URL https://doi.org/10.2200/S00516ED2V01Y201306CAC024.

[11] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 389–403, 2018. URL https://www.usenix.org/conference/nsdi18/presentation/beckmann.

[12] Laszlo A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. doi: 10.1147/sj.52.0078. URL https://doi.org/10.1147/sj.52.0078.

[13] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 483–498, 2017. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/berger.

[14] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. Robinhood: Tail latency aware caching - dynamic reallocation from cache-rich to cache-poor. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.*, pages 195–212, 2018. URL https://www.usenix.org/conference/osdi18/presentation/berger.

[15] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pages 499–511, 2017. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein.

[16] Philip H. Carns, Kevin Harms, William E. Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert B. Ross. Understanding and improving computational science storage access through continuous characterization. *TOS*, 7(3):8:1–8:26, 2011. doi: 10.1145/2027066.2027068. URL http://doi.acm.org/10.1145/2027066.2027068.

[17] Yanpei Chen, Kiran Srinivasan, Garth R. Goodson, and Randy H. Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 43–56, 2011. doi: 10.1145/2043556.2043562. URL http://doi.acm.org/10.1145/2043556.2043562.

[18] Yanpei Chen, Sara Alspaugh, and Randy H. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *PVLDB*, 5(12):1802–1813, 2012. doi: 10.14778/2367502.2367519. URL http://vldb.org/pvldb/vol5/p1802_yanpeichen_vldb2012.pdf.

[19] Aaron Clauset, Cosma Rohilla Shalizi, and Mark E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51(4):661–703, 2009. doi: 10.1137/070710111. URL https://doi.org/10.1137/070710111.

[20] Conover, William Jay. *Practical Nonparametric Statistics, Chapter 6*. Wiley New York, 1980.

[21] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005. doi: 10.1016/j.jalgor.2003.12.001. URL https://doi.org/10.1016/j.jalgor.2003.12.001.

[22] Microsoft Corp. 343 Industries Gets New User Insights from Big Data in the Cloud, 2013. URL https://azure.microsoft.com/en-us/case-studies/customer-stories-343industries/.

[23] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150, 2004. URL http://www.usenix.org/events/osdi04/tech/dean.html.

[24] Christina Delimitrou, Sriram Sankar, Kushagra Vaid, and Christos Kozyrakis. Decoupling datacenter studies from access to large-scale applications: A modeling approach for storage workloads. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization, IISWC 2011, Austin, TX, USA, November 6-8, 2011*, pages 51–60, 2011. doi: 10.1109/IISWC.2011.6114196. URL https://doi.org/10.1109/IISWC.2011.6114196.

[25] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *TOS*, 13 (4):35:1–35:31, 2017. doi: 10.1145/3149371. URL https://doi.org/10.1145/3149371.

[26] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014*, pages 75–88, 2014. doi: 10.1145/2674005.2674994. URL https://doi.org/10.1145/2674005.2674994.

[27] Jens Feder. *Fractals, Chapter 8*. Springer Science & Business Media, 2013.

[28] Dror G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press, 2015. ISBN 978-1-107-07823-9. URL http://www.cambridge.org/de/academic/subjects/computer-science/computer-hardware-architecture-and-distributed-computing/workload-modeling-computer-systems-performance-evaluation.

[29] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. Moment-based quantile sketches for efficient high cardinality aggregation queries. *PVLDB*, 11(11):1647–1660, 2018. URL http://www.vldb.org/pvldb/vol11/p1647-gan.pdf.

[30] Henri Gavin. The levenberg-marquardt method for nonlinear least squares curve-fitting problems. *Department of Civil and Environmental Engineering, Duke University*, pages 1–15, 2011.

[31] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 29–43, 2003. doi: 10.1145/945445.945450. URL http://doi.acm.org/10.1145/945445.945450.

[32] Bogdan Ghit, Nezih Yigitbasi, Alexandru Iosup, and Dick H. J. Epema. Balanced resource allocations across multiple dynamic mapreduce clusters. In Sujay Sanghavi, Sanjay Shakkottai, Marc Lelarge, and Bianca Schroeder, editors, *SIGMETRICS*, pages 329–341, 2014.

[33] Pedram Ghodsnia, Ivan T. Bowman, and Anisoara Nica. Parallel I/O aware query optimization. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 349–360, 2014. doi: 10.1145/2588555.2595635. URL http://doi.acm.org/10.1145/2588555.2595635.

[34] Raghul Gunasekaran, Sarp Oral, Jason Hill, Ross Miller, Feiyi Wang, and Dustin Leverman. Comparative I/O workload characterization of two leadership class storage clusters. In *Proceedings of the 10th Parallel Data Storage Workshop, PDSW 2015, Austin, Texas, USA, November 15, 2015*, pages 31–36, 2015. doi: 10.1145/2834976.2834985. URL http://doi.acm.org/10.1145/2834976.2834985.

[35] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand S. Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis of HDFS under hbase: a facebook messages case study. In *Proceedings of the 12th USENIX conference on File and Storage Technologies, FAST 2014, Santa Clara, CA, USA, February 17-20, 2014*, pages 199–212, 2014. URL https://www.usenix.org/conference/fast14/technical-sessions/presentation/harter.

[36] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of "big data" on cloud computing: Review and open research issues. *Inf. Syst.*, 47:98–115, 2015. doi: 10.1016/j.is.2014.07.006. URL https://doi.org/10.1016/j.is.2014.07.006.

[37] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of facebook photo caching. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 167–181, 2013. doi: 10.1145/2517349.2522722. URL https://doi.org/10.1145/2517349.2522722.

[38] Alexandru Iosup, Simon Ostermann, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick H. J. Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *TPDS*, 22(6):931–945, 2011.

[39] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[40] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2002, June 15-19, 2002, Marina Del Rey, California, USA*, pages 31–42, 2002. doi: 10.1145/511334.511340. URL https://doi.org/10.1145/511334.511340.

[41] Svilen Kanev, Juan Pablo Darago, Kim M. Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David M. Brooks. Profiling a warehouse-scale computer. *IEEE Micro*, 36(3):54–59, 2016. doi: 10.1109/MM.2016.38. URL https://doi.org/10.1109/MM.2016.38.

[42] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching strategies to improve disk system performance. *IEEE Computer*, 27(3):38–46, 1994. doi: 10.1109/2.268884. URL https://doi.org/10.1109/2.268884.

[43] Barbara Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. EBSE Technical Report EBSE-2007-01, 2007. updated, version 2.3.

[44] David G Kleinbaum and Mitchel Klein. *Survival analysis*, volume 3. Springer, 2010.

[45] Eva K. Lee. Innovation in big data analytics: Applications of mathematical programming in medicine and healthcare. In *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, pages 3586–3595, 2017. doi: 10.1109/BigData.2017.8258352. URL https://doi.org/10.1109/BigData.2017.8258352.

[46] Songbin Liu, Xiaomeng Huang, Haohuan Fu, and Guangwen Yang. Understanding data characteristics and access patterns in a cloud storage system. In *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13-16, 2013*, pages 327–334, 2013. doi: 10.1109/CCGrid.2013.11. URL https://doi.org/10.1109/CCGrid.2013.11.

[47] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*, 2003. URL http://www.usenix.org/events/fast03/tech/megiddo.html.

[48] Jayakrishnan Nair, Adam Wierman, and Bert Zwart. The fundamentals of heavy-tails: properties, emergence, and identification. In *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '13, Pittsburgh, PA, USA, June 17-21, 2013*, pages 387–388, 2013. doi: 10.1145/2465529.2466587. URL https://doi.org/10.1145/2465529.2466587.

[49] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*, pages 111–125, 2008. doi: 10.1109/SP.2008.33. URL https://doi.org/10.1109/SP.2008.33.

[50] Qais Noorshams, Kiana Rostami, Samuel Kounev, and Ralf H. Reussner. Modeling of I/O performance interference in virtualized environments with queueing petri nets. In *IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems, MASCOTS 2014, Paris, France, September 9-11, 2014*, pages 331–336, 2014. doi: 10.1109/MASCOTS.2014.48. URL https://doi.org/10.1109/MASCOTS.2014.48.

[51] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993.*, pages 297–306, 1993. doi: 10.1145/170035.170081. URL https://doi.org/10.1145/170035.170081.

[52] Fengfeng Pan, Yinliang Yue, Jin Xiong, and Daxiang Hao. I/O characterization of big data workloads in data centers. In *Big Data Benchmarks, Performance Optimization, and Emerging Hardware - 4th and 5th Workshops, BPOE 2014, Salt Lake City, USA, March 1, 2014 and Hangzhou, China, September 5, 2014, Revised Selected Papers*, pages 85–97, 2014. doi: 10.1007/978-3-319-13021-7_7. URL https://doi.org/10.1007/978-3-319-13021-7_7.

[53] Prashant Pandey, Michael A. Bender, Rob Johnson, and Robert Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 775–787, 2017. doi: 10.1145/3035918.3035963. URL https://doi.org/10.1145/3035918.3035963.

[54] David A. Patterson. Technical perspective: the data center is the computer. *Commun. ACM*, 51(1):105, 2008. doi: 10.1145/1327452.1327491. URL https://doi.org/10.1145/1327452.1327491.

[55] Wullianallur Raghupathi and Viju Raghupathi. Big data analytics in healthcare: promise and potential. *Health information science and systems*, 2(1):3, 2014.

[56] John A Rice. *Mathematical statistics and data analysis, Chapter 13*. China machine press Beijing, 2003.

[57] Amazon Web Services. FINRA Adopts AWS to Perform 500 Billion Validation Checks Daily. URL https://aws.amazon.com/solutions/case-studies/finra-data-validation/.

[58] Jim Summers, Tim Brecht, Derek L. Eager, and Alex Gutarin. Characterizing the workload of a netflix streaming video server. In *2016 IEEE International Symposium on Workload Characterization, IISWC 2016, Providence, RI, USA, September 25-27, 2016*, pages 43–54, 2016. doi: 10.1109/IISWC.2016.7581265. URL https://doi.org/10.1109/IISWC.2016.7581265.

[59] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schüpbach, and Bernard Metzler. Albis: High-performance file format for big data systems. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, pages 615–630, 2018. URL https://www.usenix.org/conference/atc18/presentation/trivedi.

[60] Guanying Wang, Ali Raza Butt, Henry M. Monti, and Karan Gupta. Towards synthesizing realistic workload traces for studying the hadoop ecosystem. In *MASCOTS 2011, 19th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Singapore, 25-27 July, 2011*, pages 400–408, 2011. doi: 10.1109/MASCOTS.2011.59. URL https://doi.org/10.1109/MASCOTS.2011.59.

[61] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 335–349, 2014. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wires.

[62] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, 2010. URL https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets.

[63] Qiang Zoll, Yifeng Zhu, and Dan Feng. A study of self-similarity in parallel I/O workloads. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–6, 2010. doi: 10.1109/MSST.2010.5496978. URL https://doi.org/10.1109/MSST.2010.5496978.