Scalable Storage Systems

Thesis submitted for the degree of
"Doctor of Philosophy"

by

Elliot Jaffe

Submitted to the senate of
THE HEBREW UNIVERSITY OF JERUSALEM
in the year 2010

Scalable Storage Systems

Thesis submitted for the degree of
"Doctor of Philosophy"

by

Elliot Jaffe

Submitted to the senate of
THE HEBREW UNIVERSITY OF JERUSALEM
in the year 2010

This dissertation was compiled under the supervision of
Dr. Scott Kirkpatrick.

Abstract

Petabyte and Exabyte storage systems are a reality. These systems introduce unique challenges to the systems architect because of their size and unique requirements. In this thesis, I suggest that access patterns to very large storage systems are long-tailed distributions. I explore three live systems and show that each of them has a very strong long-tailed access distribution. Based on this finding, I prove there is no operational benefit to be gained by applying a directed placement policy allocating items to storage units. I conclude this thesis with a discussion of impacts and future research opportunities based on these findings, including large scale caches, backup techniques and distributed databases.

Contents

Abstract ii

List of Figures vi

List of Tables vii

Constants viii

Foreword 1

1 Introduction 4

2 Scalable Storage Architectures 8
- 2.1 Chapter Overview . . . 8
- 2.2 Application Programming Interface (API) . . . 8
  - 2.2.1 Hard Drives . . . 11
    - 2.2.1.1 Hard Drive Properties . . . 12
    - 2.2.1.2 Power Consumption . . . 15
- 2.3 Traditional File Systems - TFS . . . 16
  - 2.3.1 TFS Indices . . . 17
- 2.4 Over Capacity . . . 18
- 2.5 Big Storage . . . 19
  - 2.5.1 Backup . . . 20
    - 2.5.1.1 RAID Storage . . . 21

3 What's Driving Storage Research 24
- 3.1 File System Architectures . . . 24
- 3.2 Distributed file systems . . . 29
- 3.3 Application level file systems . . . 31
- 3.4 Tertiary Storage Systems . . . 34

4 System Model 36
- 4.1 System Parameters . . . 36
- 4.2 Modeling the access distributions . . . 37
- 4.3 Long tailed access distributions . . . 38
  - 4.3.1 Statistical Long-tail . . . 38

4.3.2 Anderson Long-tail . . . 39
4.3.3 Zero class . . . 40
4.3.4 Distribution of requests . . . 41
4.4 Placement policies . . . 41
4.5 Oracular Placement . . . 43
4.5.1 Random Placement . . . 45
4.6 Pareto . . . 47
4.7 Regions of operation . . . 48
4.7.1 The Head . . . 49
4.7.2 The top tail . . . 50
4.7.3 The long tail . . . 51
4.7.4 The zero class . . . 52

5 Empirical Findings **53**
5.1 SourceForge . . . 53
5.1.1 Distribution Fitting . . . 56
5.1.2 Mass Count Disparity . . . 60
5.1.3 Heat Maps . . . 62
5.1.4 Discussion . . . 63
5.2 Internet Archive Wayback Machine . . . 64
5.2.1 Distribution Fitting . . . 65
5.2.2 Mass Count Disparity . . . 65
5.2.3 Access over time . . . 67
5.3 Internet Archive Media Collection . . . 67
5.3.1 Distribution Fitting . . . 70
5.3.2 Mass Count Disparity . . . 70
5.4 Conclusions . . . 72
5.4.1 Impact of placement strategies . . . 73

6 The Architecture of the Internet Archive **75**
6.1 What is it? . . . 75
6.2 System Architecture . . . 76
6.2.1 Requirements . . . 77
6.2.2 Logical View . . . 77
6.2.3 Process View . . . 78
6.2.4 Development View . . . 79
6.2.4.1 Storage . . . 79
6.2.4.2 Import . . . 80
6.2.4.3 Index and Search . . . 80
6.2.4.4 Access . . . 82
6.2.5 Physical View . . . 82
6.2.5.1 Web Nodes . . . 83
6.2.5.2 Storage Nodes . . . 83
6.2.6 Upgrade Path . . . 84
6.3 Actual Performance . . . 85
6.4 Existing Placement Strategies . . . 86
6.5 Should ARC files be unpacked? . . . 87

7 Implications and Opportunities **89**
7.1 Energy . . . 89
7.2 Managing the head of the access distribution . . . 90
7.3 Caching . . . 91
7.3.1 Empirical Requirements . . . 91
7.3.2 Sizing the Cache . . . 94
7.3.3 I/Os per Second . . . 95
7.3.4 Implementation options . . . 97

8 Epilogue **99**
8.1 Summary . . . 99
8.2 Future Research . . . 100
8.2.1 Databases . . . 101

Bibliography **103**

Acknowledgements **113**

List of Figures

2.1 Hard Drive Components 11
2.2 IBM Hard drive capacity 1960 – 2000 [1] 13
2.3 Hard drive capacity [2] 14
2.4 A secure file system [3] 17

3.1 Hard drive capacity [2] 25

4.1 Statistical Long tail from SourceForge downloads 39
4.2 Anderson Long Tail 40
4.3 Access Distribution and Requests to the system 42
4.4 Access Regions 48
4.5 Reverse proxy 49

5.1 SourceForge Activity 1999 to 2009 55
5.2 SourceForge downloads as a percentage of the total number of active files 55
5.3 SourceForge downloads by count and rank 56
5.4 Log-log plot of a normal distribution 57
5.5 Log-log plot of a pareto distribution 57
5.6 SourceForge Downloads vs. Pareto Distribution 59
5.7 Mass Count disparity for SourceForge downloads 60
5.8 Access to ICEWM project files 62
5.9 Internet Archive Wayback ARC file access vs. Pareto Distribution 66
5.10 Internet Archive ARC File Mass/Count distribution 66
5.11 Internet Archive Wayback Cumulative ARC file access over time 67
5.12 Internet Archive Media files by size 68
5.13 Internet Archive Media downloads by size 2008-2009 70
5.14 Internet Archive Media files access vs. Pareto Distribution 71
5.15 Internet Archive Media Mass/Count distribution 71

6.1 Logical View 78
6.2 Process View 78
6.3 Physical View 83
6.4 Network load from Jan 2008 through August 2008 85
6.5 Downloads from July 2008 through early Dec 2008 85
6.6 Bytes Downloaded from July 2008 through early Dec 2008 86

7.1 Cache Rates vs. Cache Size from Nov 1, 2008 to Nov 7, 2008 93
7.2 Estimated IOPS over time from Nov 1, 2008 to Nov 7, 2008 96

List of Tables

2.1 Sample Disk Properties 13
2.2 Sample Disk Power Properties 16
2.3 Average IOPS and disk RPM speeds [4] 18

4.1 System Parameters 37

5.1 Empirical analysis summary 72
5.2 x_{tail} values 73

7.1 Coverage achieved as function of cache size. 94

Constants

Constant Name	Symbol	=	Constant Value (with units)	Note
bit	b	=	one binary value	(0 or 1)
byte	B	=	8 bits	one or two characters
kilobyte	K	=	one thousand bytes (10^3)	
megabyte	MB	=	one million bytes (10^6)	the text of one book
gigabyte	GB	=	one billion bytes (10^9)	one full length movie
terabyte	TB	=	one trillion bytes (10^{12})	
petabyte	PB	=	one quadrillion bytes (10^{15})	
exabyte	EB	=	one quintillion bytes (10^{18})	
zettabyte	ZB	=	one sextillion bytes (10^{21})	
yottabyte	YB	=	one septillion bytes (10^{24})	

[I]f a man declares to you that he has found facts that he has observed and confirmed with his own experience - even if you consider this man to be most trustworthy and highly authoritative - be cautious in accepting what he says to you. If he attempts to persuade you to accept his opinion, which is his viewpoint, or any doctrine that he believes in, then you should think critically and understand what he means when he declares that he has observed it, and your thoughts should not become confused by these "novel ideals."

Rather, investigate and weigh this opinion or that hypothesis according to the requirements of pure logic, without paying attention to this contention that he affirms empirically. This is so irrespective of whether this assertion is advanced by a single person or by many who adhere to that particular viewpoint.

Rosner's translation of *Maimonides' Commentary on the Aphorisms of Hippocrates*, 4.

Dedicated to my wife, children and parents, without whom this work would not have been possible.

Foreword

I have been working in the field of computer storage for more than 20 years. My first experiences were at Carnegie Mellon where Prof. Alfred Spector gave me an opportunity to work on the Camelot Distributed Transaction Processing system. There I learned the costs and benefits of asymmetric components such as CPUs, hard disks and networks.

I was one of the first employees of Transarc, an IBM company that productized Camelot as well as a distributed file system called AFS. We built a new distributed file system called DFS for the DCE environment supported by the OSF, IBM, HP and others. It was at Transarc that I learned how hard it is to get a distributed file system to work as transparently and as robustly as a local file system. The additional complexities of network and nodes failures greatly increase the risk of incorrect operations.

After I left Transarc, I started a company called PictureVision in 1995. Our business was in scanning and merchandising consumer images. I was responsible for the distributed infrastructure of our new online effort. PictureVision developed a simplistic distributed file system where consumer's images could be stored at any one of hundreds of development labs. Users were automatically directed to the appropriate storage and server node to see their images.

In 2002, I started back in academia on the road to a Masters in Computer Science. Prof. Danny Dolev believed in my abilities and worked toward my acceptance into the Hebrew Universities School of Computer Science. Storage and File systems were my main interests, but the course work and projects during that phase of my education were more theoretical and formal. I completed my Masters thesis with Prof. Dahlia Malki, focusing on attacks to distributed systems that use multiple identities [5].

It was at that time that I met Prof. Scott Kirkpatrick, who had also spent a number of years in industry. We immediately found common ground based on experiences and shared interests. Dr. Kirkpatrick is interested in large scale systems and the future of the Internet, while my focus was on large scale Internet systems. Together we embarked on this thesis.

At some point, we discussed the possibility for idling disks in large storage systems. Dr. Kirkpatrick suggested that I talk to his friend Jim Gray at Microsoft, believing that Jim would be a good external member for my thesis committee. Jim graciously met me at his offices in San Francisco and proceeded to pour cold water on all of my suggestions. Jim believed that all file systems were in fact databases. That the main bottleneck was the number of operations per second supported by each disk. His years of experience in transaction systems and databases had proven to him that there would never be systems on which you could idle some of the disks. I left that meeting in despair, but I was not convinced. Less than a year later, Jim Gray was lost at sea, never to be seen again.

On that same trip, I met Dr. Bruce Baumgart who at the time worked at the Internet Archive. Bruce is a veteran systems person, having studied at the Stanford A.I. Lab in the 1970s. Bruce spent the 1980s as an entrepreneur, founded, ran and sold Softix, Inc. which built computerized ticketing systems around the world, including BASS San Francisco and Ticketek Australia. His career includes work at institutions such as Xerox PARC, Foonly Inc., Yaskawa Robotics, and IBM Research at Almaden. Bruce agreed that there is something to looking at concerning how files in very large systems are accessed. He agreed to be on my thesis committee and arranged for access to the monitoring information of the Internet Archives petabyte storage system. This thesis would not have been possible without Bruce's help.

The last cog in this machine is Prof. Dror Feitelson. Prof. Feitelson works on empirical computer science. He teaches courses on analyzing data sets and carefully presenting the results. Prof. Feitelson has provided many hours of help and direction. The many graphs and charts in this thesis are due to his direction and support.

This thesis is a response to Jim Gray's position. I agree that many small scale file systems are in fact databases and should learn and borrow from database techniques. Yet, there is at least one class of scalable storage system for which there are many cold

items that are infrequently accessed. As you will see in this thesis, these systems have unique properties that diverge from the traditional database world.

I am sorry that Jim could not read this thesis. He was right about many systems, but the world is full of exceptional cases and environments.

Chapter 1

Introduction

The United States Library of Congress has collected more than 21 million books [6]. These books fill 21 terabytes of storage assuming that each book requires 1 megabyte of storage [7]. The total estimated size of the Library of Congress, including movies, web archives and other media is estimated to be at least 10 petabytes [8]. Each year, there are approximately 1.7 million physical visitors to the libraries buildings and more than 680 million web page hits.

The Library of Congress is not unique. Companies and organizations around the world routinely store more than a petabyte of data, with some storing and managing more than one exabyte of data [9]. The National Security Agency (NSA) of the United States is reported to be building a yottabyte storage facility [10] holding more than a million petabytes of data.

Consider the problem of filing billions of documents in file cabinets. A person with a single file cabinet can be relatively sloppy, putting documents in folders anywhere in that cabinet. Once there are multiple cabinets, the filers want to optimize the placement so that they can retrieve common documents from close cabinets. The filers might group documents according to how often they were accessed in addition to the standard classifications by title and type. If there were thousands or hundreds of thousands of cabinets, the filers would be looking for better ways to organize the documents and the cabinets, trying to minimize the time to retrieve most documents.

This thesis addresses the organization and management of billions of objects in very large storage systems. The term scalable storage systems is used because these systems

are not static. The cost of storing any given object is so small that objects are never deleted. As new objects are added, the storage system expands. Because the system is not static, it is insufficient to manage only the current objects and storage units in the system. The designers and architects must also plan ahead so that the storage system will efficiently scale and grow over time.

As storage systems grow, their operational costs have become significant. It is currently possible to purchase two terabyte disks. More than 1000 such disks are needed for a petabyte storage system, and that does not include the devices that would be needed for a backup. Studies [11] have shown that electrical costs account for 20% of data center overhead, a fraction of which are spent on storage systems. 25% of a data centers cost is spent on power distribution and cooling, which includes the heat generated by spinning disks.

An industry group [12] reported on data center costs from 2000 to 2006. They noted that site infrastructure, lighting, power delivery, and cooling can account for more than 50% of the total energy consumed by a data center. This report recorded the cost of Enterprise Storage components as less than 10% of the total cost and the remaining 40% as servers. At the time of the report, Enterprise Storage systems implied products such as storage area networks produced by companies such as IBM, EMC and others. As will be seen in this thesis, modern storage systems incorporate processing units and integrated backup systems, blurring the lines between storage and servers. Regardless of the specific breakdown, energy efficiency is critical for large scale systems. One of the impacts of this thesis are mechanisms to utilize lower cost, more efficient storage units and hence reduce the overall data center energy usage.

In order to understand how objects should be stored and managed, we must look at how they are used. For a storage system, usage is equivalent to access. An object is used when it is written or read. It is idle when there are no accesses. The focus of this thesis is a study of access patterns to very large storage systems. The statistical distributions of the observed access patterns fit a *long tailed* statistical distribution. Distributions with a long tail are found in many biological and social settings. They underly the business model for companies such as Amazon and Ebay. Fitting observations to a long-tailed distribution is challenging because while the distribution is defined out to infinity, the observations typically cover only a small portion of the total range. Three

empirical studies are described, with details on how closely they fit a proposed long-tailed distribution.

This thesis explores the implications of access distributions and in particular long-tailed distributions on storage system architectures, covering both analytical models and practical implications and results. A model is presented that covers the entire range of files, from most active down to files which are almost never read. The files can be segmented into distinct regions, for each of which an architect might be tempted to design unique storage systems. A unified architecture is presented that supports all of the files using a simple base storage system with a large scale cache to manage frequently accessed items.

Backing up very large storage systems requires at least twice the amount of the base storage, so that there is a copy of each item in case of failure. Ideally, the copies should be stored on secondary backup disks to keep a single failure from taking out both versions. The proposed architecture can idle these backup disks, saving up to 50% of the operational costs as well as significantly extending the lifetime of the capital equipment by turning off all backup disks. The new architecture also addresses issues of maintenance and natural growth of the storage system.

This document begins with storage. Storage system research has a long tradition in Computer Science. The ideal Turing Machine has an infinite tape for storing instructions and data. All practical computing systems have some form of storage with clear limits on the amount of data that can retained. Chapter 2 reviews the technical components of storage systems, and the properties and measures used in their design and assessment. Most of chapter 2 is a restatement of information available from textbooks on Operating Systems.

Researchers and Industry leaders have staked out positions on how storage systems should operate. Using studies and real-world experience, they design their systems to perform efficiently for pre-defined tasks. Chapter 3 reviews the common tasks and describes published research that has been used to define the task requirements. Existing architectures and designs are referenced along with studies of their management, operations and scalability. The final section of 3 reviews published reports of access patterns and explores how they have been used to design storage systems.

Chapter 4 builds a model of access to archival storage systems based on the underlying distributions. This model defines three or four segments of files based on their unique access patterns. The unique system requirements are derived for each of these segments and a unified architecture is proposed to simplify the final architecture. The new approach provides a number of operational savings in terms of simplicity, capacity and longevity.

Chapter 4 depends on the existence of long tailed access distributions for files in archival storage systems. Chapter 5 studies the access patterns of three long term, large scale storage systems. The access patterns for each one of these systems exhibit the long tailed distributions assumed in this thesis. A short description of each system is provided along with the data collection methods and analysis protocols used in this research.

The design used in this thesis is based on the Internet Archive, an existing system that has been operational since 1999, and now supports more than 1 petabyte of data. Chapter 6 presents an in-depth architectural review of the Internet Archives systems. The strength and weaknesses of the architecture are presented with respect to the scalability and longevity issues raised in this thesis.

With all the background completed, Chapter 7 describes practical implications of the new architecture. It explores the challenge of building a cache sufficient for storage systems at the petabyte scale. Such a cache is a necessary component of the new architecture and is itself a contribution of this thesis.

Chapter 8 concludes this thesis, summarizing the results and laying the groundwork for future research on storage systems and on the related field of database systems.

Chapter 2

Scalable Storage Architectures

2.1 Chapter Overview

The main focus of this document is on storage architectures that can scale to almost unimaginable capacities. In order to understand these systems, the reader must begin with a solid operating systems and file systems background. Much of the material in this section is a recap of a good Operating Systems textbook, extended to very large storage systems.

2.2 Application Programming Interface (API)

File systems have simple interfaces. There are only four basic operations:

Create Create a new file.

Read Read from an existing file. A read operation may be read all of the file, or a segment of a file.

Write Write into an existing file. A write operation may write all of the file, or a segment of the file. Some systems support only appending to existing files, while other allow a write operation to overwrite existing file segments.

Delete Delete an existing file.

A file system must reliably support these interfaces. Data, once written, should be immediately readable. That same data should continue to be available unless the file is deleted, or the storage media is removed.

The basic object in a file system is a file. Files have some unique identifier that can be used to reference the file during its entire life-time. The file contains the data that was written into it by the API. In principle, the data is a stream of bytes from the beginning of the file until the last byte in the file.

In addition to the file identifier, the file system usually maintains additional informational fields about the file. These fields may include the file size, the date the file was create, the date the file was updated, and the last time that the file was read. Some file systems maintain accounting information for each file, recording the owner of the file and the access rights allocated to the owner and to other users. Access rights tell the file system who is allowed to perform each of the basic API operations. All of these items make up the file's metadata; information about the file or its contents. Metadata occupies a small fraction of the files total size and is not necessarily stored contiguously with the files content.

File systems store file data on durable media. For example, files might be stored on a magnetic tape, optical disk, or nonvolatile RAM (NVRAM). File systems optimize the placement of files and their metadata on storage media. In the tape example, a file system might put the metadata at the front of the tape so that it could be retrieved quickly. Files would then be placed later on the tape.

File system placement on a finite storage unit is in principal an NP-hard problem, as it can be reduced the NP-hard bin-packing [13] problem. The goal is to fit the different size files onto one or more storage units, using the minimal number of storage units. In practice, file system placement is more of an accounting problem. It suffices to record files on the storage media. The exact perfect packing is unnecessary as long as there is some free space. Fortunately, heuristics exist which provide non-optimal solutions in polynomial time [14].

As a simplification, file systems split files into fixed size blocks. Blocks may hold anywhere from 512 bytes to more than a gigabyte. The block size is usually set as a fixed value, depending on the particular physical media and on the expected file sizes. Files smaller than the block size are allocated a full block, even though such an allocation will leave some or most of the block empty.

The files in many file systems change over time. They are created and deleted over time. A file's size may change as more data is appended or removed, creating empty spaces on the storage media where the file used to reside. These empty blocks should be coalesced into large empty spaces or filled with new blocks to make later allocations more efficient.

A file's metadata is typically stored in a block that is called an *inode*. The inode includes implementation dependent references to the data blocks for file on this or other storage media. Originally, an inode was an actual data block. Modern implementations consider an inode as an abstract data structure. Its implementation can be ignore at higher levels of the system's architecture.

In order to find a specific file amidst the blocks of the storage units, all file systems maintain some form of index. The index contains an identifier for each file and the location of the file's blocks. In order to improve efficiency, most file systems also maintain an index of free blocks, so that empty space can be found quickly when a file is created or extended.

File system implementations are designed to recover an existing file system when it is re-attached to the operating system after an intentional or accidental disconnect. This occurs each time the computer is initialized and may also occur with removable media such as CDROM, DVD or Flash disks. To aid in this effort, file systems store the main on-media data structure in well known locations on the raw media. The location of this structure is called the *superblock*, since it is the only block that must be available in order to begin recovering a file system. If this block is destroyed, the rest of the file system may be lost and unrecoverable.

Most file systems support simultaneous access to the file system and to specific files. The particular semantics of simultaneous access vary from system to system. Some systems implement a first-in, first-out (FIFO) approach, leaving the file as seen by the

last writer. Other file systems support locking mechanisms so that the users can block access to other readers or writers until the original user's operations are completed.

Storage devices that hold file system data may be characterized by their physical block size, the transfer time for each block, and the seek time necessary to ready a particular block for read or write access.

Because they are ubiquitous and relevant to this thesis, we will digress for a detailed discussion of storage devices, focusing on hard drives and their properties. While other storage devices such as solid state disks (SSDs) are becoming more common, their operational properties remain similar enough to hard drives and will as such be excluded from this general discussion. SSDs will be revisited in Chapter 7, where their operational differences will be leveraged to provide a high performance cache. At this time, SSDs are significantly more expensive than hard drives and hence are unlikely to form the basis of a very large scalable storage system.

2.2.1 Hard Drives

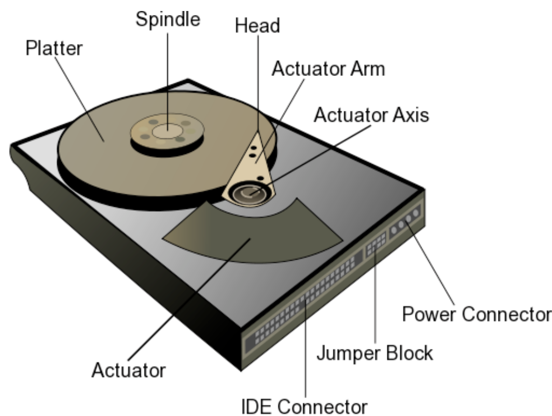

FIGURE 2.1: Hard Drive Components

Hard drives contain one or more rigid platters coated with a granular magnetic media. The magnetic surface of each platter is divided into small sub-micrometer-sized magnetic regions, each of which is used to represent a single binary unit of information. These regions are grouped into tracks, which circle the platter at a fixed distance from the center spindle. The tracks are divided into sectors, usually contained 512 bytes. A magnetic reader called a disk head is mounted on a swinging arm that can position the

head at any place on the platter. The disk is then spun up to a constant speed so that each sector will pass under the head within a reasonable short time period.

While this description is generally factual, in practice, the physical layout of each disk model is proprietary to its manufacturer. The basic concept remains sound and continues to serve as a close approximation of the actual layout. As such, we will ignore and manufacturer optimizations and focus on the generic layout.

Disk drives support one or more access protocols that allow connected devices to request or update the contents of one or more blocks. The disk controller implements these requests and translates between the public interface and the internal disk features.

2.2.1.1 Hard Drive Properties

A disk drive can be described using on a small number of basic properties. These properties are commonly presented on the drive's marketing materials.

Capacity The number of bytes that can be stored on the disk drive. Modern drives range between 100GB up to more than 2TB of storage capacity.

Rotational Speed The rotational speed of the platters is specified in terms of rotations per minute (RPM).

Interface Bandwidth The maximum transfer capacity of the disk specified in megabytes or gigabytes per second. This value is based on the physical protocols used to transfer data from the disk to the electrical data bus.

Average Seek Time The average time necessary to position the head to begin reading or writing a randomly selected disk block. The seek time has two components, the radial seek time and the rotational seek time. The radial seek time is the average time to physically move the head to the correct position from the center of the disk. The rotational seek time is the average time until the platter rotates the selected sector under the magnetic head. The basic formula for the average seek time is $\frac{1}{2}RPM + Radial\ Seek\ Time$.

Table 2.1 provides a number of sample values taken from actual published disk specifications. The disks were selected from a single manufacturer and range from laptop

TABLE 2.1: Sample Disk Properties

Model	Type	Capacity	Rotational Speed	Interface Bandwidth	Average Seek Time
Momentus 5400.6 [15]	Laptop 2.5”	640GB	5400	3Gb/sec	13ms
Barracuda XT [16]	Desktop 2.5”	2TB	7200	6Gb/sec	8.5ms
Cheetah 15K.7 [17]	Server 3.5”	600GB	15000	3Gb/sec	3.4ms

models to high performance models intended for data center operations. The selection of specific disk models is usually based on expected performance requirements and on the purchase cost.

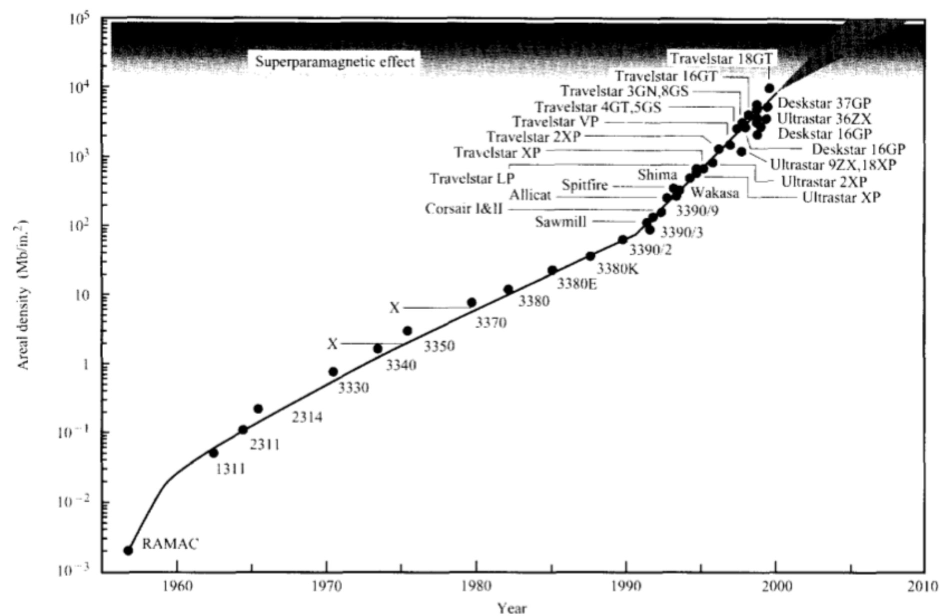

FIGURE 2.2: IBM Hard drive capacity 1960 – 2000 [1]

Thompson [1] reported in Figure 2.2 disk capacities for IBM products released between 1960 to 2000. Thompson attributed the change in slope around 1992 to the introduction of a new recording head by IBM and to increasing competition in the marketplace. Figure 2.3 shows a steady growth of hard drive capacities since 1995, with the earlier data supporting Thompson's finding. Both graphs are plotted on a log scale. Figure 2.2 reports areal density; the number of megabits per square inch. The IBM 350 RAMDAC drive consisted of 50 platters and stored 5 million alphanumeric characters with an areal density of $2000 bits/in^2$. The earliest drives reported in 2.3 stored 100MB with an areal density of $10MB/in^2$. The newest drives hold more than 1TB and have areal densities of more than $150GB/in^2$.

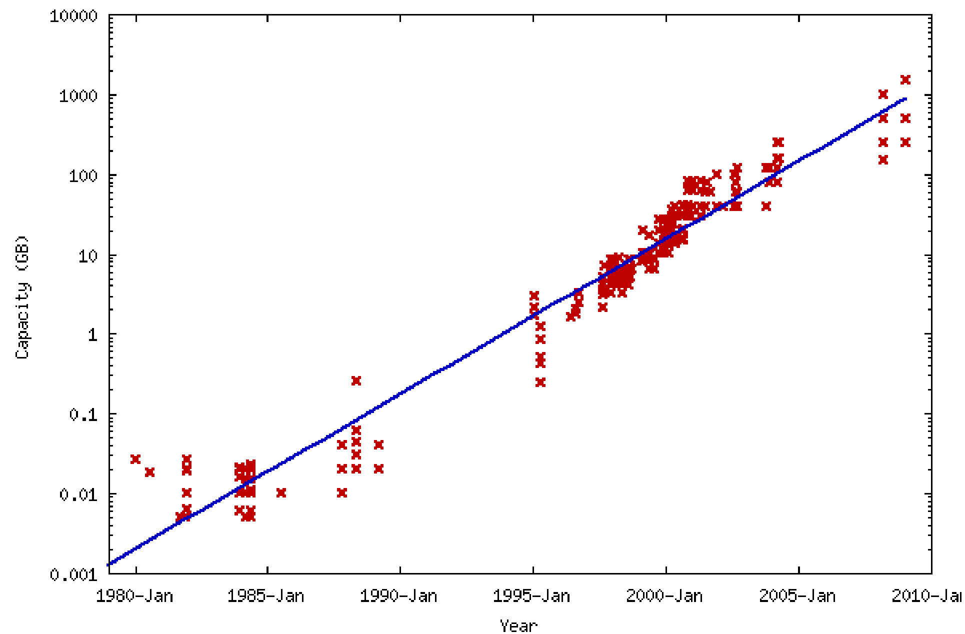

FIGURE 2.3: Hard drive capacity [2]

Over time, smaller disks are no longer manufactured and hence storage administrators will be unable to obtain one-to-one replacements for failed units. New replacements will have larger capacities and different operating parameters. Over time, in-place replacement will become impossible because of ongoing changes to protocols and cabling. Storage devices and enclosures are almost certain to be scheduled for complete replacement every five to ten years. Scalable storage systems that are intended to last more than five years need to support future devices as transparently as possible in order to remain viable past the first or second hardware refresh.

Rotational speed and the related average seek time have remained stable over the past 20 years. The one significant change is that in the 1980's it was possible to purchase disks that rotated at 3600 rpm. These products have been discontinued. More recently, disks intended for mobile applications rotated at 4200 rpm. These disks are being phased out as 5400 rpm mobile disks become the minimum standard.

Serial operations such as reads or writes on spinning disk drives are significantly faster than random operations [18, 19] . The performance difference is due to the physical nature of serial operations, which can continue to operate as the platter turns and the head remains in the same position. This parameter is sometimes know as the maximum sustainable transfer rate (MSTR).

There are a limited number of operations that can be completed on a hard drive during a given period of time. Consider that for each random read request, the controller must move the magnetic head to the appropriate location and wait for the platter to rotate the data under the head. The limit is referred to as the number of Input/Output Operations per second or IOPS. A gross calculation of IOPS may be performed by dividing the average seek time into one second. For example, a drive with an average seek time of 8ms will support approximately 125 IOPS.

2.2.1.2 Power Consumption

Power consumption is hard drives can be specified by four values [20] :

Idle Power Dissipation The power consumed by the drive when it is idle. An idle drive has no requests, but remains spinning at a fixed rotational speed. Idle power consumption is correlated to the drive temperature. The higher the idle power dissipation, the higher the internal drive chassis temperature.

Active Power Dissipation The power consumed when the drive is performing under an 100% random seek load. This value represents the maximum likely power needed by the unit.

12V Maximum Power Dissipation The 12V power rail is used to power the spindle motor. The maximum power consumption occurs when the disk is powered on and spun up to speed.

5V Maximum Power Dissipation The 5V power rail powers the actuators that move the disk arms. The maximum value occurs when moving from an idle to active state. In many disks, the 5V power rail also powers the controller and interface hardware.

Manufacturers are encouraged to publish complete specifications for power consumption. The published values for the previously selected models are found in Table 2.2.

The previous features are almost universal across file systems. In the following sections, we will discuss traditional storage systems and then move on to scalable storage systems. The differences between the two types of storage will help to focus the discussions in the rest of this thesis.

Table 2.2: Sample Disk Power Properties

Model	Type	Idle	Active	12V Max	5V Max
Momentus 5400.6 [15]	Laptop 2.5inch	0.67W	1.57W	NA	5W
Barracuda XT [16]	Desktop 2.5inch	1.35W	1.55W	33.6W	NA
Cheetah 15K.7 [17]	Server 3.5inch	1.85W (5v)	2.5W (5v)	22.8W	2.24W
		9.72 (12v)	12W (12v)		

2.3 Traditional File Systems - TFS

Traditional file systems (TFS) are found in most commercial computer systems, laptops, file servers and high performance computing clusters. They range in size from micro file systems with less than a megabyte of storage up to large institutional filers offering many terabytes of storage.

TFS have a number of common design points:

Random Access Files can be read or written at any point the file, beginning, middle or end.

Modifiable content Files are allowed to be modified, updated, truncated or extended at any time.

External Backup Most TFSs are optimized for normal operation. Disaster recovery is an exceptional situation and is usually performed externally. That is, the backup for a TFS is not part of the system itself.

Most file systems implement these design points using block oriented file systems with relatively small block sizes (4KB to 16KB). Updates require rewriting only the blocks that have changed.

Operation systems further optimize access to file blocks by caching them in main memory. Caching frequently-used blocks in memory significantly reduces the number of accesses to the relatively slow storage units because of the principal of locality and the significant differences in speed between RAM and physical storage medium [21]. The principal of locality [22] states that most programs and operating systems exercise a small set out of the accessible memory locations or file segments. This principal fails for analysis of scientific data, where the entire data set is read in one pass. Traditional file systems are usually optimized for highly localized access.

In addition to reacting to user requests, TFSs implement background maintenance tasks that optimize and manage the existing files. One such task is the de-fragmentation of the storage devices. This process moves blocks around on the storage device so that the blocks of each file are contiguous. This makes reading, updating and deleting each file more efficient since a read of multiple blocks can be performed sequentially without moving the magnetic head.

2.3.1 TFS Indices

FIGURE 2.4: A secure file system [3]

A file system is only useful if the files can be identified and accessed. A filing cabinet filled with concrete is very safe, but its contents are effectively lost for all time. So too, traditional file systems have some form of index that allows files to be found and retrieved. In most file systems (UNIX, Windows, etc.), the file index is in the form of a tree. The root of the tree is usually the base for all files on that physical file system. The root is a form of directory that contains named files or other directories. The sequence of directories from the root to the file's location is called the file path.

Tree structures are very efficient lookup devices. In traditional file systems, the limit on the number of items in a directory is typically more than 10^3 items. Each directory entry can itself be a directory, that is, it is a subdirectory. The resulting structure forms a n-ary tree. Such a file system can easily provide access to billions of files.

File system directory structures are usually stored as part of the file system. The directory structure links into the metadata about each file and hence into the file itself. This organization is particularly helpful for removable media since the index and the media form one integral unit.

Within the past five years, operating system providers have begun to offer additional file system indices based on each file's contents. These systems use a form of inverted index, where the key is one or more words and the values are the list of files that include that word. Inverted indices represent recoverable information. That is, they can be regenerated by reading each of the files. In many systems [23, 24], inverted indices are built as extensions to the operating system and are not stored together with file system.

Searching an index in a traditional file system, whether the file system directory structure or an inverted index, is performed by searching the index directly. That is, the index is a single data structure and the requester scans the data structure.

2.4 Over Capacity

RPM	IOPS
SSD	20000
15K	175
10K	125
7200	75
5400	50

Table 2.3: Average IOPS and disk RPM speeds [4]

As mentioned in section 2.2.1.1, the number of requests per second to a hard drive is limited. Table 2.3 shows the average IOPS that can be expected from modern hard drives based on their rotations speeds. Solid state disks (SSD) are represented on the first line. Their IOPS values are extremely high because there are no moving parts and hence no rotational latency.

Practical read performance rates depend on many factors, including rotation latency, the disk controller, internal bus bandwidth and request patterns. A popular tool for measuring average read performance rates is n2benchw [25]. The tool was written by a German computer magazine to compare disk parameters. It has been widely used, even by competitors because its results are consistent and the tool is easy to use. Using this tool, Tom's Hardware [26], an Internet site that reviews hardware reports that modern disk average between 36 and 102 MB/sec transfer rates for average read performance and 42 to 127 MB/sec for maximum read performance. Using a high end value of

100MB/sec, it would take 60 seconds to read all of the data on a 6GB disk, one hour to read a 360 GB disk and more than three hours read a 1TB disk.

2.5 Big Storage

Scalable storage systems are designed to store and make available huge amounts of data. A system that stores less than 100TB can currently be bought as a single storage unit. Systems over 100TB require distributed solutions that integrate multiple components into a single virtual storage system. The basic discrepancy between storage capacity and network bandwidth implies that once a large storage unit is filled to capacity, it will take days or even months to copy the data to a replacement system. An optimal design would provide the existing system with a natural upgrade path, retaining the basic software infrastructure and replacing physical units as they wear out or become obsolete.

A critical requirement of scalable storage systems is that they scale in bandwidth and in operations per second (IOPS). It is insufficient to create a system that stores a petabyte of data, but allows only a single user or file to be accessed at any one time. Scalable storage systems should leverage their multiple storage components to decentralize operations, allowing a high level of parallelization and increased bandwidth.

Some companies such as IBM and EMC design large storage systems for industrial databases. These applications require extremely fast block access to large data tables. Access is measured in a handful of milliseconds and may reach tens of thousands of operations per second. This thesis focuses on a different type of system, one that provides streaming access to large objects ranging from a few kilobytes up to terabytes in size. In these systems, latency to first access is less critical and may reach up to 1 second per request, but once a transfer is begun, the entire object should be streamed to the client.

Tape systems were the first to provide this type of file storage. The latency was measured as the time it took to locate the tape, mount it on an available drive unit and then seek to the appropriate location on that tape. Access latencies were measuring in the 10s of seconds assuming that the tape was already mounted. If the tape was not mounted, it could take up to 30 minutes for an operator to locate the tape and mount it on an

available drive. Today's systems are significantly simpler; all units are online all of the time and random seeks can be performed on a disk drive within a few milliseconds.

Today's scalable storage architectures utilize hundreds or thousands of storage nodes. Each node is a small computer with a CPU, local memory and a few disks. Each node might provide access to 10 TB of storage using 5 2TB disks. The entire system of storage nodes is connected using a high performance network with switches and routers, so that the bandwidth and latency to and from any node are more or less constant. The system must maintain an index service to identify the location of each object. Instead of funneling all requests through a single front-end server, clients are directed to specific storage nodes for the contents of each file. In a balanced system, the total throughput is the sum of the access capabilities of each storage unit. The central management units provide support and indexing and are not part of the data access path.

2.5.1 Backup

> Question: How do you become a millionaire in Israel?
> Answer: Come to Israel with two million.
> Question: How do you backup a petabyte of storage?
> Answer: Use two petabytes.

When dealing with petabytes of storage, no single device can store the entire contents. Equivalently, no single device can backup the system's contents in case of disaster recovery or system error. For many years, large storage companies sold duplicate systems as remote backups. An organization would stream updates from the main system to a local backup in case of a unit failure and to a remote backup in case of a catastrophic failure. Storage companies could then sell three times the base storage.

Unfortunately for organizations, there is still no better option than to have three copies of each item, two local copies and one remote copy. Instead of treating the copies as warm backups to be used only in case of a failure, modern systems treat the other copies as part of the available system. Clients may be served from any copy. One benefit of this approach is that highly active files can be replicated 3 or more times, providing additional capacity to hungry clients.

Online replicated backups are not free. The first challenge is to maintain indexing methods that can identify a file's location on multiple nodes. The second problem occurs when units fail. The system must update the index and arrange for additional copies to be made so that the replication level is maintained. Finally, the system must deal with simultaneous updates to files. During an update, each copy of the file must be updated and those updates must be permanent before the operation can successfully complete. If one copy is not up to date, a future read might return the wrong data. Commit algorithms like distributed two-phase commit [27] or group commit [28] are used to guarantee consistency, but they introduce significant costs in terms of additional messages and bookkeeping overhead.

For archival systems, this problem is non-existent since updates are not allowed. Some systems support appending data to existing files. New blocks can them be replicated once they are complete. Since blocks cannot change, as long as one block is available, all subsequent reads will return the correct data.

The implication of using the system as its own backup is that available capacity is at best $\frac{1}{2}$ or $\frac{1}{3}$ of the total physical capacity. Thus, a system that stores 500TB of data would be managing 1.5PB of storage, gaining 2 to 3 times the available bandwidth and IOPS, but with an increased probability of unit failure.

2.5.1.1 RAID Storage

RAID [29] stands for Redundant arrays of inexpensive disks. A RAID system consists of a hardware or software controller that virtualizes disk storage across a small number of physical disk units. RAID systems can be categorized according to the mechanisms used to split data across the disks. The following sections describe the different RAID levels [30] and their properties.

RAID 0 - Block-level striping without parity or mirroring

Improved bandwidth because multiple blocks can be read in parallel. No fault tolerance. The entire disk capacity is available to the system.

RAID 1 - Mirroring without parity or striping

Write each data block to two disks simultaneously. Read performance can be

improved by selecting the first responding disk. If one disk fails, the other disk retains a full copy of the data.

RAID 2 - Bit-level striping with dedicated Hamming-code parity

Data is striped so that each sequential bit is on a different disk. A Hamming code is calculated and stored on one or more parity disks. The system can identify and correct the failure of one disk using the parity codes. High transfer rates are possible by streaming data across all disks in parallel. In practice, RAID 2 is never used.

RAID 3 - Byte-level striping with dedicated parity

Data is striped so that each sequential byte is on a different disk. A dedicated parity disk retains error correction data. The system can identify and correct the failure of one disk using the parity code. High transfer rates are possible by streaming data across all disks in parallel.

RAID 4 - Block-level striping with dedicated parity

Data is striped so that each sequential block is on a different disk. A dedicated parity disk retains error correction data. The system can identify and correct the failure of one disk using the parity code. High transfer rates are possible by streaming data across all disks in parallel.

RAID 5 - Block-level striping with distributed parity

Data is striped so that each sequential block is on a different disk. Parity blocks are distributed across the disks. The system can identify and correct the failure of one disk using the parity code. High transfer rates are possible by streaming data across all disks in parallel. The system can continue operation in the event of a disk failure and can rebuild the missing disks from the remaining data. RAID 5 is the predominant method used in production systems.

RAID 6 - Block-level striping with double distributed parity

Data is striped so that each sequential block is on a different disk. Two copies of the parity blocks are distributed across the disks. The system can identify and correct the failure of any two disks using the parity code. High transfer rates are possible by streaming data across all disks in parallel. The system can continue operation in the event of a disk failure and can rebuild the missing disks from the remaining data.

While is might seem that there is little difference between the RAID levels, most production systems utilize RAID 5 as a mechanism for securing data in the face of potential disk failures. RAID 5 operates at the block level, improving read and write performance over RAID 2, 3 and 4, but retains the ability to recover from single disk failures.

Scalable storage systems have moved away from RAID storage over the past ten years. The initial reason as explained by Brewster Khale of the Internet Archive [31] was that RAID controllers would have increased the cost of each storage unit without providing significant benefits in return. An additional reason is that RAID controllers introduce additional failure modes to each system. In a RAID system, if a single failed disk can be rebuilt, but only at the expense of reading from all others disks. During this time, the overall transfer capacity of the RAID controller is significantly reduced, impacting the normal performance. If two disks fail concurrently or more likely, if the controller fails, the entire data set is lost.

Compare this to the simple replicated disk method where replicas reside on separate systems. When a disk fails, it can be replaced and the system remains in operation. The recovery mechanism is at the system level, allowing the system to select the most appropriate location for the new copy. If an entire unit fails, the exact same process is applied.

Engineers prefer to have a single recovery mechanism. It is simpler to apply and is consistent across all units. RAID storage imposes disparate recovery operations and relies heavily on the RAID controller for reporting and recovery. There is an unconfirmed story from Microsoft that a RAID controller silently failed and wrote zeros for all new data. Fortunately, a second copy was maintained elsewhere and no data was lost. If a second copy was available, then why use RAID in the first place.

In practice, RAID systems are attractive for small ($<$ 10TB) datasets, or where high access and transfer rates are required. Database systems may use RAID for fast recovery and high concurrency. For petabyte scale archival systems, RAID introduces additional expenses and failure more without providing any additional benefits.

Chapter 3

What's Driving Storage Research

Since ancient times, man has tried to save information for later use. The earliest cave drawing recorded events in daily life. Clay tablets and papyrus mats were used in the Middle East to record transactions, agreements and historical events more than 3000 years ago. More recently, digital storage offered a new medium for recording information. Magnetic material, beginning with drum storage has provided a medium for storage.

This chapter describes how the development of storage systems is driven by changing capabilities. Changes in storage technology can be attributed directly to the growing capacity of storage media as seen in Figure 3.1. The new capacity offers both a challenge and an opportunity. The challenge is that data structures appropriate for a given limited storage size will either fail or be very slow on larger capacity units. The opportunity is to use the additional space to cache data so that it does not need to be recreated with each operation. The subsequent sections track these changes since the first file system in the early 1960's to the present. This thesis is a direct successor to these efforts and is itself influenced by the capacity of newly available storage units.

3.1 File System Architectures

The earliest known file system [32] was implemented in the early 1960s in Cambridge England as an aid for programming the Electronic Delay Storage Automatic Calculator 2 (EDSAC2) [33]. The EDSAC2 was a vacuum tube based computer with a programmable

micro-controller. The file system was implemented using two magnetic tapes and could hold up to 2047 files. Each file was addressable via its file number from 1 to 2047.

Soon after, the Multics [34] team at MIT developed a general purpose file system [33]that became the predecessor of modern file systems, with a hierarchical directory structure, removable storage media and backups. Files on Multics used a variable length file name and were protected by an access control structure so that data could be protected and restricted to a particular user or to a group of users. The Multics system used a storage drum to store and retrieve data. It also included the capability of moving files from the storage drum to magnetic tape and visa versa.

For some time, file systems remained tightly coupled to the operating system. The first Unix file system, the System V file system (S5FS)[35] was developed in 1974 by Ritchie and Thompson as a component of the new Unix time-sharing system. S5FS was the first file system to have a superblock and inodes (see section 2.2) for organizing the on-disk data structures. S5FS supported 512 and 1042 byte block sizes. As befits the Unix concept, S5FS was extremely simple. It was implemented on two 200MB disks attached to a DEC PDP-11/70. The simplistic implementation was able to utilize only 2% - 3% of the raw disk bandwidth [36]. S5FS supported 8 character file names with a three character extension which later became knows as an 8.3 naming format.

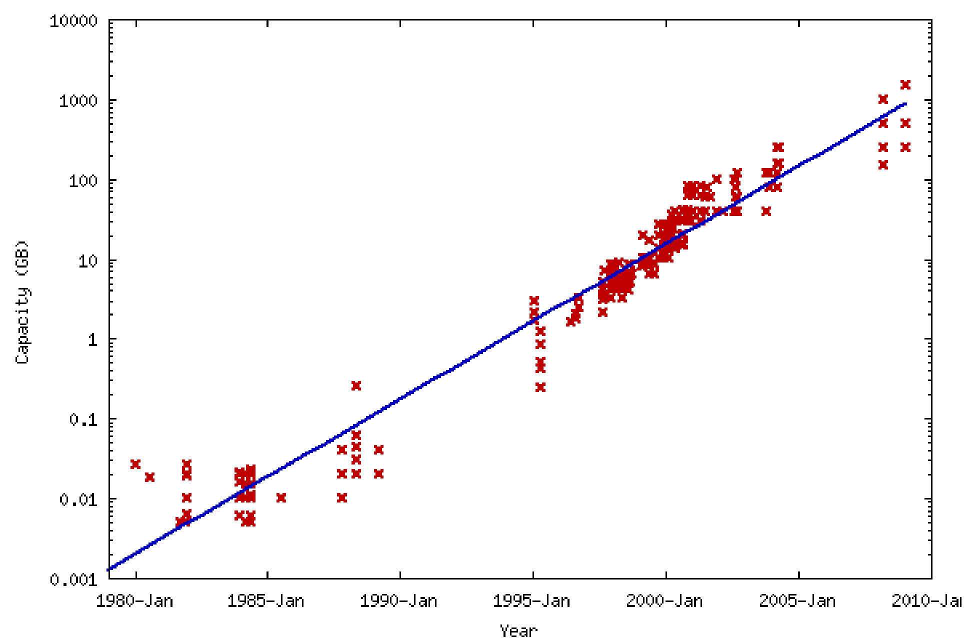

FIGURE 3.1: Hard drive capacity [2]

In 1977, Bill Gates designed the File Allocation Table (FAT) file system [37] in a hotel room in Albuquerque, NM. for Intel. FAT was part of the Basic language being developed for the Intel 8086. The file allocation table maintained a list of clusters. Each cluster was represented as a single 16bit word representing the number of contiguous blocks assigned to that cluster. The original FAT used a 12bit table and was intended for use on floppy disks having up to 32MB of storage and 4096 clusters. FAT was designed for very simple systems that lacked the computational power and infrastructure of Unix and the DEC PDP-11.

Microsoft purchased the rights to this system and then recoded the FAT file system with a 16bit table (FAT16) in 1986. The new file system could manage 65 thousand clusters and was used as part of the first version of the Microsoft® MS-DOS® operating system. A further extension to FAT32 was introduced in 1996 to support up to 1 billion clusters. FAT32 is still in use today as the default file system for USB disk-on-keys, SD memory cards and other removable media that must be broadly readable.

FAT file systems suffered from poor performance because blocks were allocated to clusters based on their availability in the allocation table. As a result, files were spread multiple clusters across the storage media, necessitating many random read or writes for that data. A process of defragmentation was developed that would attempt to move blocks around, aggregating clusters so that files were more or less contiguous. Defragmentation can occur as a background process when the machine is idle, or as an administrative task performed by the user.

Unix broke into two flavors in the late 1970s and early 1980s. One branch was maintained by AT&T as System V. The other branch was developed by the University of California in Berkeley and came to be known as the Berkeley Standard Distribution (BSD). With the split, the BSD team decided to develop a new file system that was more efficient and robust than S5FS. This new file system was called the Berkeley Fast File System (FFS) [38] or alternately, the Unix File System (UFS). Whereas the S5FS and the FAT file system maintained all of the disk information in one table at the head of the disk, UFS spread the metadata across the entire disk, insuring that a single bad block would not destroy the entire file system. UFS used a 4KB block size, but supported the concept of a 1KB fragment so as to avoid wasted blocks. UFS took into account the physical structure of the disk sectors, maintaining replicated structures on separate sectors. UFS was able

to utilize 14% to 47% of the available disk bandwidth, a significant improvement over the S5FS alternative at that time. UFS extended the 8.3 naming format to a uniform 256 byte file name. File extensions were considered part of the file system name.

Disk continued to double in capacity about every two years and there remained two major challenges, robustness and performance. As more data was stored on disks, any system crash could cause the on-media structures to be inconsistent. In the best case, this could be corrected by running a file system check (FSCK) program that read the entire disk and corrected any errors. In the worst case, the data on the disk could be lost forever.

From the perspective of performance, engineers and designers knew that serial reads and writes could significantly improve disk utilization, but they continued to view file systems as databases, with many random read and writes. Users wanted to be sure that a completed write was actually on the physical media so that if the power went out, the data would be safely stored.

Journaled file systems were introduced in 1991 in the IBM AIX operating system. A journal of all modifications to the file system is maintained in parallel to the file system itself. The journal can them be replayed from a known consistent state in the event of a crash. Previous approaches required a complete scan of each file system block in order to identify potentially lost updated and inconsistencies. A journaled file system requires additional storage space and write operations to maintain both the journal and the main file system. The size of the addition storage space can be adjusted based on how often the system needs to checkpoint consistent states. The additional writes can be amortized at the cost of potential lost operations upon recovery.

In 1992, Mendel Rosenblum and John Ousterhout [39] designed a Log based file system that improved disk bandwidth utilization by logging all updates serially. This approach wrote data to the disks in large sequential segments, greatly improving write speeds up to 70% of the available disk bandwidth. The main insight was that multiple write events can be grouped together in a single serial write. If a system is busy, the grouped writes will significantly improve application response times. These systems are slightly slower for infrequent writes.

Sometimes, file systems are designed not because of changing technologies, but because of licensing requirements. In the early 1990s, Linus Torvalds developed the Linux [40] operating system as a rather complete rewrite of the minix operating system. A group of Linux developers took concepts from previous work and developed the Second Extended File System (EXT2) [41] as an open source replacement for the simplistic minix file system. Ext2 removed the concept of fragments introduced by UFS and focused on performance and recoverability. Ext3, added journaling to the Ext2, removing the need to scan the file system after a crash.

One of the problems with file systems at the time was the necessity to update meta-data items on the storage media. Were the system to fail during an update, the affected block might be in any number of states. The record could be untouched. It could contain the entire updated write, or alternatively, the record could contain a mix of original data and new data, or even completely random garbage data written as the disk shut down. This technique was developed by IBM in the 1970s under the term *shadow paging* and was deployed as part the VM/CMS file system. In 1994, Dave Hitz developed a system called the Write Anywhere File Layout (WAFL) [42]. WAFL was developed for NFS servers in order to maintain consistency of large file systems. The basic approach was to use copy-on-write for all media updates. An entire chain of updates could be written to disk without affecting the existing data. At the last moment, a single write would be necessary to move from the old state to the new state. Hitz utilized NVRAM for this pointer swap, avoiding the problem of partial block writes for the master pointer.

SGI developed a new file system called XFS [43] in 1996. Prior to XFS, file systems used a simple recording scheme to local free blocks. A new file would pluck blocks from this pool to fulfill its storage needs. XFS introduced the concept of extents, sequential free blocks that could be allocated contiguously, improving read and write times without requiring a defragmentation process. XFS also addressed the challenge of very large disks. As disk capacity grew, the list of free blocks became a bottleneck. There was insufficient room to store the entire list in memory, necessitating numerous disk requests just to find a free block. XFS borrowed a concept from database systems, using a B+ tree to track free blocks, file blocks, directory structures and inodes. XFS was also the first file system in wide use to support 64bit file systems. XFS was able to utilize 90% to 95% of the available disk bandwidth.

From 1996 to 2008, there were very few breakthroughs in file system design. Most of the development efforts went into improved stability of existing file systems and integration into the different Unix flavors in the marketplace. The popular filesystems now run on variants of BSD, Linux and Solaris.

Microsoft operating systems have far fewer options. They support only the FAT file systems and the NTFS file system. NTFS was developed in the early 1990's for the OS/2 project. Due to disagreements between Microsoft and IBM, it was removed from OS/2 and was first released with Windows NT3.1 in 1993. Minor tweaks and adjustments have been made in the past 17 years with three major releases, but the system remains very similar to its roots.

In 2008, Sun released the Zetta File System (ZFS) [44]. Whereas XFS addressed the problem is very large disks, administrators were now asking to spread a file system across multiple disks. The concept of a volume manager has been around for many years. Volume managers are kernel extensions that abstract the physical disks into virtual disks. A virtual disk volume can span multiple physical volumes. Sun integrated the volume manager into ZFS, allowing ZFS volumes to transparently grow or shrink on demand. ZFS compares favorably to other file systems in terms of performance.

3.2 Distributed file systems

All of the previous file systems were implemented on a single computer with one or more attached disks. These systems are known as DAS (Directly attached storage) systems. As networked computers have become common, attention has turned to Distributed file systems that allow users on one computer to access files that are not physically located on that computer.

The earliest file systems followed the client/server pattern. A server stored files locally and listened on a network interface for file requests. Requests supported directory operations as well as the standard read and write operations. The Datacomputer [45] was one of the first distributed file system, developed for resource sharing on the Arpanet. It was deployed in 1973 on a PDP-10. Other file systems developed in the 1970's include IFS, WFS and XDFS, all from Xerox Parc. These early file systems supported a tens of clients and closely tied a single server to a number of dedicated clients.

The last 1970's saw the formalizations of a number of concepts that would prove essential to distributed file systems. The Remote Procedure Call (RPC) was explored, expanded and solidified into a reliable communication method for distributed operations [46]. At the same time, relational databases were popularized along with the formalized concept of a transaction [47].

Building on RPC and transactions, three file systems tried to develop scalable distributed file systems that could handle multiple servers and thousands of clients. The Athena [48] and Andrew [49] projects attempted to build complete distributed computing environments, including file systems. Their intention was to provide computing support to entire college campuses. The vision behind these research projects was to provide a uniform file experience to all users, regardless of which console they happened to using at that time. Athena and Andrew used RPC as the basic communication infrastructure and transactions to enable concurrent access to files by hundreds and thousands of users. The Andrew file system (AFS) supported the wide area network access, so that users could work from home with the same files that they accessed at their university.

At the same time, Sun, a spin-off from Stanford University, was developing a less ambitious distributed file system called NFS [50] (Network File System) for production use. NFS was designed to allow local users to access file servers. The major insight was that each client could connect to one or more servers, with each file server providing a specific portion of the file system tree. NFS was the standard distributed file system for Sun in the 1980's and with its release in AT&T's SVR4 in 1990, became the standard distributed file system for most Unix based installations up to the present time.

In the 1980's each competing commercial operating system developed their own networking and file system products. The Apple Filing Protocol (AFP) [51], the Netware Core Protocol (NCP) [52] and the Server Message Block (SMB)[53] competed for market share. Of these, the SMB protocol, developed by Microsoft has become the basis for most local distributed files systems including home and office use. The prevalence of SMB is due to the popularity of the Microsoft operation systems. SMB is a closed, proprietary protocol. Open source versions such as SAMBA [54] have been developed, but are constantly playing catch-up with Microsoft's upgrades and modifications.

Distributed file systems are distinguished by their external interfaces. The internal implementation of these interfaces is hidden from its users. The local operating systems

acts as a client of the distributed file system. Users are unaware of the underlying implementation, but may be able to detect a distributed file system due to its longer latencies.

Most distributed file system servers focus on supporting clients and do not try to optimize the placement of files on the servers disks. Systems such as AFS, NFS, SMB act as unprivileged local users using whatever local file system implementation happens to be resident on that machine. The exception to this approach is NetApp, which offers NFS and Samba servers that are implemented on a proprietary internal file system. NetApp can them offer additional features and performance unavailable from less integrated systems.

3.3 Application level file systems

The previous sections describe file systems that were accessed directly from the operating system using system calls into the kernel. As an operating system service, programs can access any supported local and remote file systems through a uniform interface. Application developers have, over the years, requested and implemented many additional file system services that improve performance or increase reliability.

For example, what if two users want to simultaneously access a large file? The results would be unpredictable if both users attempted to write to the same file location. To manage this challenge, modern file systems include range locking operations, enabling each user to request exclusive access. These and other requirements make it very complicated to develop a fully compatible internal file systems.

As a direct result of the complexity of file system interfaces, developer have begun to look at file systems that are separate from the operating system. These new file systems are accessed through custom clients over network connections, instead of through the local operating system. The unifying concept is to create an public interface that can be implemented easily by file system developers.

The first major success on this track was HTTP. In 1989, Tim Berners-Lee wrote a proposal that would become the basis for the World Wide Web [55]. His concept was to have pages of content interconnected using hyperlinks. The hyperlinks were like

references in a thesis, but to other external files and pages. The HTTP [56] protocol that derived from this initial concept is a form of file system protocol. It supports read, write and update of pages.

The HTTP protocol provides a very minimal set of file system interfaces, with only two required interfaces, GET and PUT. The WebDAV [57] protocol, developed in 1999, extends HTTP and provides support for metadata and directory operations such as make directory, move and copy. WebDAV supports concurrent access through LOCK and UNLOCK calls. The WebDAV interface was an Internet Proposed Standard in 1999 and has been implemented as part of Microsoft's Sharepoint [58]. Although a standard, applications that use WebDAV are not very common. WebDAV is a technology that is looking for its killer application.

The Internet Archive [59, 60] developed a private file system in the early 2000s as a way to store petabytes of pages culled from crawls from web sites and media files provided by collectors and individuals. The Archive's file system was a one-off implementation, that could utilize thousands of nodes with tens of thousands of disks. The Archive's file system was intended only as a large storage system. It needed to provide performance sufficient for users browsing the Internet Archive's web site, as opposed to analysis or database services. More details on the Internet Archive's system can be found in Chapter 6.

Some file systems begin with an application large enough and valuable enough to maintain development momentum. The Google File System (GFS) [61] is one of the best known application level file systems. In the early 2000's Google was one of the leading search sites on the Internet. A search site provides access to indices generated over collections of web pages. The system crawls over the Internet, collecting web pages and storing them locally for processing. A successful search site indexes more web pages than its competitors and analyzes those web pages more efficiently. Google engineers needed a scalable file system that could handle multiple copies of web crawls and could also provide efficient analysis. They popularized a framework from the old Lisp language called Map/Reduce [62].

The insight behind Map/Reduce was that for many problems, it is possible to first perform a highly parallel operation on each object in the input set, collect the results and then reduce the results into a smaller result set, or even a scalar value. When applied to

web page analysis, it was possible to apply a parsing operation to each web page and then to generate from those parses an inverted index suitable for web searches. Parallelism in the map operation is the key to Map/Reduce. In principal, a Map/Reduce operation could use thousands of hundreds of thousands of CPUs in parallel. The bottleneck becomes access to the data itself.

Prior to Map/Reduce, the basic paradigm was to move the data to the computation. The previous distributed file systems such as NFS allowed data to be accessed wherever there were free cycles. Map/Reduce supports the inverse operation, moving the computation to the data, wherever that data may reside. These two paradigms are not mutually exclusive. Their use depends on the volume of input and output data and on the quantity of computation per data block. As computations become more intensive, the overhead of moving the data to the computation is amortized. Similarly, if the volume of data is very large compare to the computation, then it is more efficient to perform the computation where the data resides.

The Google File System enabled Map/Reduce parallelism by splitting the file into many large blocks. Web crawls collect multiple pages into large files [63]. It is easier to deal with a few large files than to attempt to manage millions of smaller files. For example, moving a crawl from one location to another with large files requires just a few transactions. Moving the same collection with millions of files requires the successful completion of millions of small transactions. Each large block is replicated one or more times and these replicas are stored on separate machines. The Map/Reduce operations run locally, on one of the machines where a block resides. Map/Reduce and GFS can handle thousands of nodes, each having terabytes of storage and multi-core processors.

GFS and Map/Reduce are proprietary systems. Their value was quickly recognized by competitors and an open-source alternative called Hadoop [64] was developed by Yahoo and others and transferred to the Apache organization. Hadoop and its file system HDFS, are conceptual copies of Map/Reduce and GFS.

All three of these file systems (Internet Achive, GFS, HDFS) are so called WORM file systems: write-once, read many. Studies have shown that even in write-many file systems, most file updates impact only a very small portion of the total number of files [65, 66].

Peer-to-peer file sharing systems are also read-only file systems. Users add files to the system and those file can be read by other participants. No interface is provided for modifying files.

So too, in these very large file systems, files are created in the file system and may be deleted or replaced, but not modified. The WORM paradigm is appropriate for many large file systems, such as Google Mail, an archive of web sites, or scientific data collected by the Large Hadron Collider. In all of these systems, files are created, but never modified.

It is possible to view almost any file system as a WORM system if the system uses a copy-on-write scheme for file updates. Files are not updated, merely copied with changes. The old versions of each file are never deleted.

3.4 Tertiary Storage Systems

In the 1990's scientific data sets were significantly larger than disk subsystems. Researchers turned to hierarchical storage systems where multiple storage technologies were seamlessly integrated into a single very large storage system. The primary storage system was the systems main memory. The secondary storage was usually a relatively small set of high performance disks. The tertiary storage system was magnetic tape robots or optical disk robots. In 1995, hard disks could store up to 1GB, while tapes could store up to 50GB [67]. Tape jukeboxes could store hundreds of tapes, totaling a few terabytes of data. The difference between storage capacities and unit costs made it reasonable to try to store data at the level appropriate for its access frequency.

The challenge represented by tertiary storage is derived from the substantial latency caused first by the jukebox retrieving and mounting the tape and then by the serial scanning of the tape itself until the appropriate segment is found containing the requested data. Not all file accesses could withstand this very slow access method.

The hierarchical storage field produced a number of papers attempting to argue for efficient placement policies. These policies took into account the probability of access for each item, the number of tapes, and the placement of each item on each tape. In 1997, Christodouplakis et. al. [68] explored the mathematical basis for optimal placement on

tape drives and within each tape. This paper has been frequently cited for placement research, but is limited because the domain of research was storage jukeboxes. As such, the primary questions were how to allocate items so that the limited number of mounted disks would, with high probability, contain the requested items. The second issue was how to allocate items to specific tape blocks. Wong, [69] showed that for a serial tape, the optimal placement formed an organ pipe arrangement. The organ pipe arrangement placed the more frequently accessed item in the middle of the tape, with less frequently accessed items to the left and right, alternating until the tape is full. Both Christodouplakis and Wong using as their basic assumption that the probability of access for each file was independent.

Subsequent research has expanded this research to placement strategies for dual-headed disks, and for cases where access patterns are not independent [70, 71]. In this thesis, we consider a specific class of access distributions and assume that all of the disks are equally accessible. Furthermore, location on each disk is no longer an issue because disks are effectively random access devices where the location of each block has little impact on the access time.

Chapter 4

System Model

This chapter develops an analytical model for archival storage systems. It begins with a list of the input parameters that affect the system. These parameters are then used to derive the activity levels at the component storage units. The resulting model exposes a number of unique regions of operation for the system that depend on how frequently objects in those regions are downloaded. We will develop a mathematical definition of these regions and discuss appropriate architectural and design options.

4.1 System Parameters

The parameters in Table 4.1, reflect the status of the overall system and its components. These values set the stage for discussing the behavior of the system and in particular, the activity of the less frequently accessed items in the archive.

The distribution factors, R, H, α and X_{min} describe the activity of the system and the access distribution for the items in the tail. The other factors describe the system infrastructure and its contents.

As noted earlier, spinning disks consume energy even when they are idle. The spindle motor and the controller both consume energy maintaining the constant spin for the disk platters that is required for standard operations. Similarly, starting a disk from a standing stop requires significant energy to spin the disk platters up to operational speed and to initialize the disk controller.

Table 4.1: System Parameters

Parameter name	Symb	Description
Number of unique items	N	The number of unique items stored within the system.
Number of disks	D	The number of available storage disks.
Disk Size	D_{size}	The number of objects that can be stored in each disk.
Disk IOPs	D_{iops}	The number of input/output operations per second sustainable by a given disk.
Minimum Disk Idle Period	D_{idle}	The minimum idle time at which it is more efficient to turn off the disk instead of keeping it spinning.
Requests per second	R	The total number of item requests to the system per second.
Percentage requests in the tail	T	The percentage of all requests that are served by items in the long tail.
Pareto Alpha	α	The slope parameter of the Pareto distribution.
Pareto Minimum	X_{min}	The minimum value over which the Pareto distribution is defined.

The startup cost is a constant, while the operational costs increase monotonically over time. The point at which the operational costs are equal to the startup cost is called the Minimum Disk Idle Period (D_{idle}). After this point, the powered-off disk saves energy.

4.2 Modeling the access distributions

Our model is based on the access distributions for the collection of items in the system. An access distribution reflects the probability that a given item will be accessed. Studies have shown that there is no single probability distribution that fits observed Internet sites or file systems. Most researchers break the access distribution into two components, a distribution that reflects the popular, active items and a distribution that reflects the less popular items.

As will be seen in the following chapter, the number of active items in the first distribution are relatively small compared to the total number of items in the system. For the rest of this discussion, assume that some portion of the items are very active and that they will need special treatment. A common approach is to use some sort of cache, whether as a separate set of disks or an in-memory component to quickly and efficiently

serve the active files. Caching frequently access objects is a well understood research area that has been explored in multiple domains including kernel paging, local file systems and databases.

What is left after the frequently accessed items is a second distribution that has the unique property of being a long-tailed access distribution as described in the following section.

4.3 Long tailed access distributions

There are two conflicting definitions for the long tail of a distribution; the traditional mathematical notion and the popular notion suggested by Chris Anderson in 2004 [72]. The two approaches are confused because they deal with opposite ends of the access distribution. Let us first explore each of these approaches.

4.3.1 Statistical Long-tail

A heavy tailed distribution function F on $(0, \infty)$, with a tail distribution $\bar{F}(x) = \Pr(X > x)$, requires that $\int_0^\infty e^{\epsilon x} \bar{F}(x)\, dx = \infty$ for all $\epsilon > 0$ [73]. A distribution with this property is referred to as long-tailed if

$$\lim_{x \to \infty} \frac{\bar{F}(x-y)}{\bar{F}(x)} = 1, \tag{4.1}$$

for all $y < \infty$. While some [74] require that long-tailed distributions have infinite variance, the results in this chapter are more general and apply to all distributions that support equation 4.1.

The head of the a long-tailed distribution are the items for which $\frac{\bar{F}(x-y)}{\bar{F}(x)} >> 1$ for small values of y. The tail of the distribution are those items for which $\frac{\bar{F}(x-y)}{\bar{F}(x)}$ are close to 1. Items in the head of this distribution has relatively small download frequencies, while items in the tail have extremely high access frequencies. The term $\bar{F}(x) = \Pr(X > x)$ describes the probability that a download will have a higher download count than x.

Figure 4.1, taken from Chapter 5, shows the access patterns as a long-tailed distribution. The x-axis is the number of downloads per file, while the y-axis is the probability that

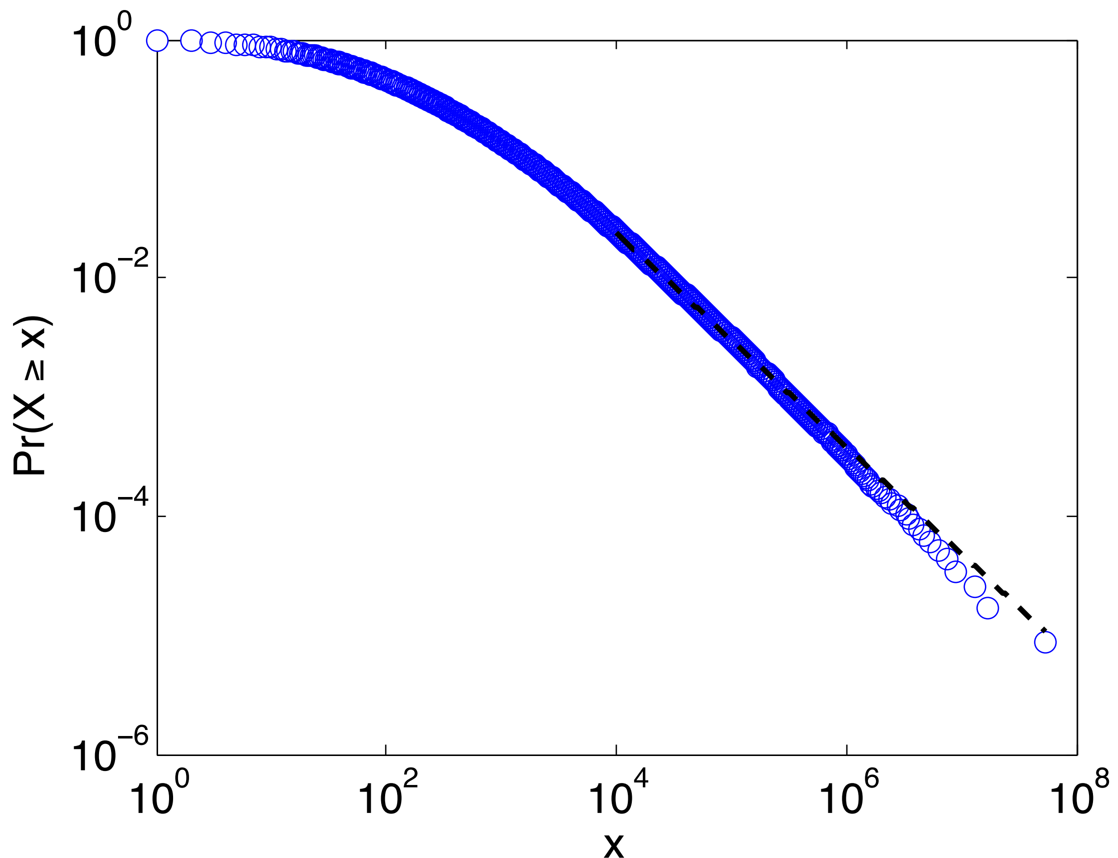

FIGURE 4.1: Statistical Long tail from SourceForge downloads

any given file has a greater download probability than this value. The items at the right of the chart are those files that have very high download counts, while the items at the left have very low download counts. These distributions differ from the idealized formal distributions in that the distribution is only viewed over a finite range because there are only a finite number of objects in any physical storage system.

4.3.2 Anderson Long-tail

Chris Anderson wrote a seminal article in Wired, 2004 [72] and a book on the topic in 2008 [75]. He used the term long-tail to describe those items that were not frequently accessed. Figure 4.2 shows a typical long tailed chart. Note that as the rank increases, the number of downloads for that item decreases. The head consists of items that are very frequently downloaded. Anderson used this description to argue that the mass of the long tail frequently outweighed the mass of the active head.

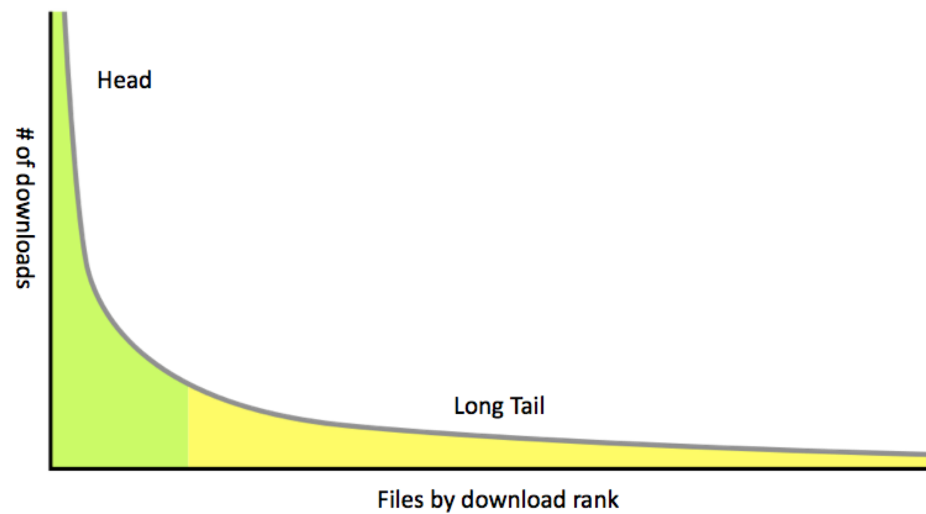

FIGURE 4.2: Anderson Long Tail

This chapter focuses on Anderson's definition of long tail and explores the regions defined by the download count for different files.

4.3.3 Zero class

In the field of Library Science, there has been some discussion on the existence of a zero-class; objects that are never accessed. If the access probability distribution is actually long-tailed, then at some point, every object will be accessed, although that point may take a very long time to arrive. Urquhart's law [76, 77] published in 1976, argued that at some point in the long tailed distribution, the effective probability of accessing a given item may be so small that for all intents and purposes, the book will never be accessed. They used this insight to suggest that all books that fall into this class might be removed from the library stacks.

Items in the zero class are invisible on the traditional mathematical chart where they would be found at the top-left of the graph, right on the y-axis. In Anderson's version, zero class items could be ranked and would lie of the x-axis, extending out to the right of the chart.

Zero class items are interesting because it is possible that given more time, some, if not all, of these items would graduate to active items.

In modern computer storage systems, the price of storage is falling so fast that the cost of storing any given item is negligible and hence all items can be stored indefinitely, even if the probability of accessing any one of them is close to zero.

Note that large providers such as Google and Yahoo have so many hits per day that an object with an access probability of $\frac{1}{10^9}$ might still be accessed every single day.

In this section, we assume that the access distributions follow both the traditional and Anderson long-tailed models and further, we treat all items as potentially accessible. Chapter 5 provides empirical results that support this approach.

4.3.4 Distribution of requests

Using the Anderson Long-tail model, rank the downloaded items from 1 to N, with the item of rank 1 having the highest access frequency and the item with rank N having the lowest access frequency. Define a point x_{min} in the ranking where items with higher access frequencies fit a Pareto distribution with values α and x_{min}. Items with a higher rank are the less active items in the system. Let R represent the number of requests per second to the entire archive. Define T as the fraction of the total system requests R, that are not served by those items that fit the Pareto distribution, and therefore have rank greater than x_{min}.

By definition, the integral of a probability distribution over the range of the distribution is one, i.e. $\int_{min}^{max} \Pr(X = x) = 1$. The fraction T therefore scales the total number of requests, R, to the fraction of the items requested from the long tail.

Figure 4.3 graphically represents the values R, T, x_{min} and the two access distributions.

4.4 Placement policies

Anderson's insight was that Long tailed distributions have a significant portion of their mass in the tail. That is, in our sample system, more than 80% of the files in the system will be served from the tail. Although modern disks store more than a terabyte of data, the 80% tail in our sample occupies close to 400 TB. These items must be spread out across the systems disks and the mechanism to do that is called a placement policy.

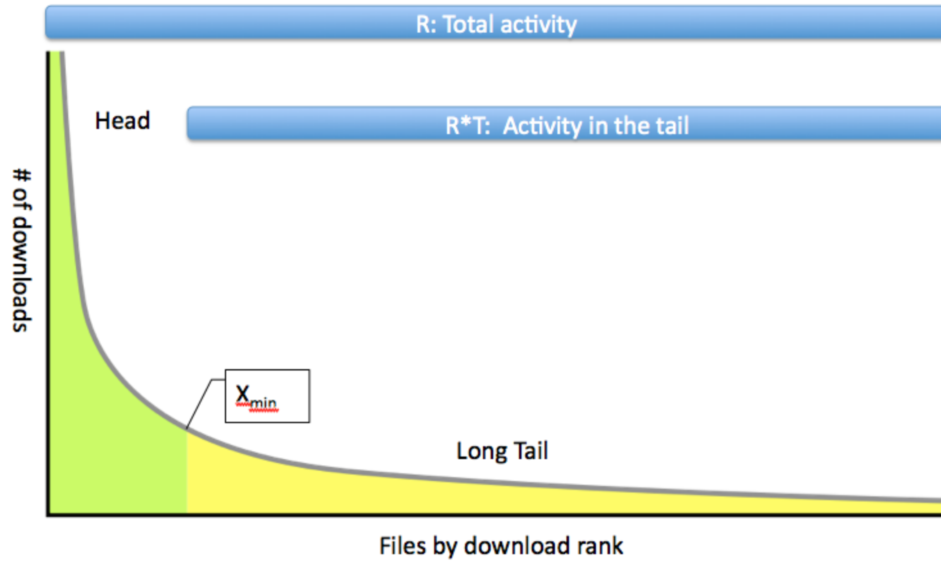

FIGURE 4.3: Access Distribution and Requests to the system

More formally, a placement policy is an algorithm P that operates on a storage system s and an object o. The output of this algorithm is an updated system s' that includes object o. If the object o is already in s, then P may or may not change o's placement within the system.

A placement policy can be evaluated based on the resulting access patterns to the storage units. That is, given a statistical distribution for when files will be accessed, which storage units will service those requests.

The access probability $\Pr_{unit\ access}$ of a given disk i is the probability of accessing at least one item X_a on that disk. This value depends on the specific set of objects on that disk, $Objects_i$ and can be calculated as:

$$\Pr_{unitaccess}(i) = \Pr\left(\bigcup_{a \in Objects_i} X_a\right) \tag{4.2}$$

$$= \sum_{a \in Objects_i} \Pr(X_a) - \sum_{a,b \in Objects_i} \Pr\left(X_a \bigcap X_b\right) + $$
$$+ \sum_{a,b,c \in Objects_i} \Pr\left(X_a \bigcap X_b \bigcap X_c\right) - \cdots + (-1)^{n-1} \Pr\left(\bigcap_{a \in Objects_i} X_a\right) \tag{4.3}$$

$$= \sum_{a \in Objects_i} \Pr(X_a) - \sum_{a,b \in Objects_i} \Pr(X_a)\Pr(X_b) + $$
$$+ \sum_{a,b,c \in Objects_i} \Pr(X_a)\Pr(X_b)\Pr(X_c) - \cdots + (-1)^{n-1} \prod_{a \in Objects_i} \Pr(X_a) \tag{4.4}$$

$$\leq \sum_{a \in Objects_i} \Pr(X_a) \tag{4.5}$$

Equation 4.2 is an equality if the access probabilities for each item are independent. In practice, it should be obvious that there is some dependency between the items if only because the items were collected by a non-random process. The pertinent issue is the magnitude of the dependency. Given that we are dealing with millions or billions of items, even if a small fraction of those items are dependent, the overall disk access probability will still be close to the base value in 4.5. In addition, because by construction, the frequency of access for any given item is extremely low; $\Pr(X = x) < 10^{-7}$. The likelihood of two items being accessed in a single time period is very small and is ignored.

4.5 Oracular Placement

Intuitively, some items are more frequently accessed than other items. If we could group all the frequently accessed items on one set of disks, then the remaining items on other disks would be very infrequently accessed. If the access was sufficiently infrequent, then we could turn off these disks and save the operational costs of energy and coolings when they would have otherwise been idling.

We can formalize this approach by defining P_{oracle}, an oracular placement policy that has a-priori knowledge of the access probability for each object. Objects with a high probability of access are placed on high activity storage units, while low access probability items in the long tail are placed together on less frequently accessed storage units. If the appropriate storage unit is full, the algorithm would displace an existing object or objects with lower access probability to free some space on this unit. It would then insert the current object and restart the algorithm on the displaced objects, placing them where appropriate.

Assuming independence of the access probabilities, $\Pr_{unit\ oracular\ access}$ can be calculated based on the highest and lowest access frequency for the items on disk i, as:

$$\Pr_{unit\ oracular\ access}(i) = \sum_{access_{low}(i)}^{access_{high}(i)} \Pr(X = x) \tag{4.6}$$

The sum of the $\Pr_{unit\ oracular\ access}(i)$ over all disks will not equal 1. This follows because the distribution is not normalized for the specific, finite range of items in our system. Instead, the formal distribution tails off to infinity.

Alternatively, each disk can be viewed as containing items whose access probabilities fall between the most frequently accessed item denoted by $access_{high}$ and the least frequently accessed item denoted by $access_{low}$. The probability of accessing this disk can be expressed as:

$$\Pr_{unit\ oracular\ access}(i) = \Pr(access_{low}(i) < X < access_{high}(i)) \tag{4.7}$$

$$= \Pr(X \geq access_{high}) - \Pr(X \geq access_{low}) \tag{4.8}$$

The two statements are equivalent. I have found that in practice, equation 4.7 is easier to compute once the specific distribution and its parameters are known.

The number of hits to a given disk i over a period of time t, can be specified as:

$$Hits_{unit\ oracular\ access}(i,t) = t * T * \Pr_{unit\ oracular\ access}(i) \tag{4.9}$$

For the number of idle disks over time t we first define an indicator variable I_i, which is 1 if disk i is idle and 0 otherwise.

$$Idle_{unit\ oracular\ access} = \sum_{1}^{D} I_i \tag{4.10}$$

4.5.1 Random Placement

Consider a placement algorithm that was oblivious to the access probabilities for each item. This algorithm might simply place items on available disks as they arrived, or it could try to spread the items over all the non-full disks. Let us call this algorithm a random placement policy P_{random}. For each object to be placed, our algorithm randomly selects a storage unit and assigns the new object to that unit. If the selected storage unit is full, the algorithm would reselect a unit until one is found with sufficient space. As opposed to P_{oracle}, the random placement algorithm never moves items between disks.

The random placement algorithm can be understood in terms of the probability of accessing an item within the range of items bounded by the more frequently accessed item, $access_{max}$, down to the least frequently accessed item, $access_{min}$, in our storage system. That is, the probability of hitting a least one item that is placed by the random placement algorithm $\Pr_{storage\ tail}$ is:

$$\Pr_{storage\ tail}() = \Pr\left(access_{min} < X < access_{max}\right) \tag{4.11}$$

The random allocation policy distributes items over all the disks, leaving each disk with approximately the same number of items. We can then look at the access probability for each disk $\Pr_{disk}$ where each disk has n items randomly selected from the available items. We use the form X_i as the ith random selection from the underlying distribution.

$$\Pr_{unit\ random\ access}(i) = \sum_{x=1}^{n} \Pr(X_x) \tag{4.12}$$

The Central Limit Theorem says that these values will tend to a stable distribution, regardless of the underlying distribution. Thus, each $\Pr_{disk}$ will cluster around some mean value. The specific mean and variance of these derived variables depends only the base distribution.

Note that while we don't know the specific sampled values, it is possible to calculate the total probability of access over the range item items $[access_{min}, access_{max}]$ in the storage system. This value is $\Pr(access_{min} < X < access_{max})$. Since all disks will have more or less the same overall value due to the Central Limit Theorem, we can estimate the probability of access for any disk as close to:

$$\Pr_{unit\ random\ access}() = \frac{\Pr(access_{min} < X < access_{max})}{D} \tag{4.13}$$

$\Pr(access_{min} < X < access_{max})$ is less than 1, as it operates over only a subset of the total range, and hence each disk will get some fraction of those accesses.

The number of hits over a period of time t for any single storage unit can be specified as:

$$Hits_{unit\ access\ random}(t) = t * T * \Pr_{unit\ access\ random} \tag{4.14}$$

Since there is no significant difference between disks, all of the requests T will be spread across the D disks. It follows then that if $t * T > D$, no disk will be idle meaning that all disks must remain active.

4.6 Pareto

The Pareto distribution is named after the Italian economist Vilfredo Pareto who in 1897 used this formula to describe the allocation of wealth among individuals. The distribution is defined by two parameters, x_{min}, the minimum value at which the power-law behavior applies and α, the shape factor. While all agree that there is a value x_{min}, there is some confusion in the literature as to how this factor is represented in the normalizing constant for the distribution. We will use Clauset's formulation as represented in equation 4.16 because the parameters of our empirical distributions were calculated in this form[1].

$$\Pr\left(X = x\right) = f\left(x\right) = \begin{cases} \frac{\alpha-1}{x_{min}}\left(\frac{x}{x_{min}}\right)^{-\alpha} & \text{for } x \geq x_m \\ 0 & \text{for } x < x_m \end{cases} \tag{4.16}$$

The complementary cumulative distribution function (ccdf) of a Pareto random variable with parameters x_{min} and α is:

$$\Pr\left(X \geq x\right) = \left(\frac{x}{x_{min}}\right)^{-\alpha+1} \tag{4.17}$$

$$\Pr\left(X < x\right) = 1 - \left(\frac{x}{x_{min}}\right)^{-\alpha+1} \tag{4.18}$$

Note carefully that these probabilities are normalized for the active head. In order to use these equations in a practical application, the total number of hits to the system must be divided into popular items covered by the Pareto distribution and the less frequently accessed items. For the Pareto distribution, x_{min} is the demarcation point. Items whose rank is less than x_{min} are in the active head. Items greater than x_{min} are in the long-tail.

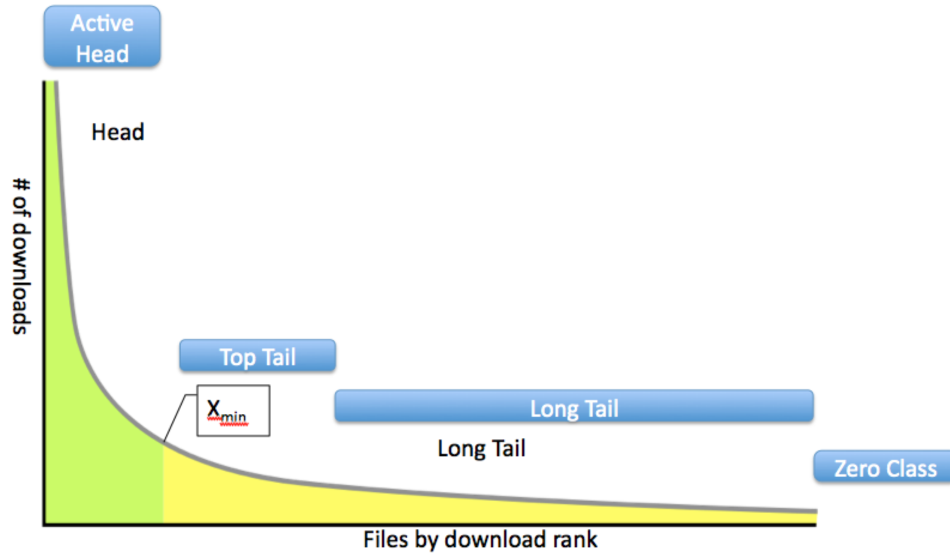

Figure 4.4: Access Regions

4.7 Regions of operation

The previous discussions have identified four distinct groups of files that reside on a large archive system. Each of these groups of files have unique requirements that impact the architecture of the system as well as its operation. The four regions are graphically represented in figure 4.4 and are described as follows:

1. The Head – very frequently access items that account for a significant fraction of the total requests, but represent only a small fraction of the systems files.
2. The top portion of the tail – Items that are governed by a pareto distribution, but whose request frequency is still sufficiently high that these items are not ”cold”.
3. The long tail – items that are governed by a pareto distribution and whose access frequency is very low.
4. The zero class – items that are effectively frozen and never accessed.

The following subsections explore each of these regions and their implications. The final subsection proposes a unified architecture that provides support for each of the regions.

[1]The more common formulation is:

$$\Pr(X = x) = f(x) = \begin{cases} \frac{\alpha x_m^\alpha}{x^{\alpha+1}} & \text{for } x \geq x_m \\ 0 & \text{for } x < x_m \end{cases} \tag{4.15}$$

4.7.1 The Head

Some portion of the files in the system are frequently accessed. These are the files that fit the Pareto distribution and for which we can identify x_{min}. All files which have more downloads than x_{min} are active parts of the Head. For example, the home page of the system and the search engine pages are going to be accessed every time a user enters the system. Items that are linked from popular pages will experience significant access peaks raising the files from inactivity to peaks that may overwhelm the system. The slashdot effect [79], also known as *slashdotting* occurs when a popular website links to a smaller website, causing a massive increase in traffic, frequently overloading the smaller site.

Archives have tremendous internal network capacity, but may have limited external bandwidth. Files in the active head consume the majority of the external bandwidth and should be treated separately from the main archive store.

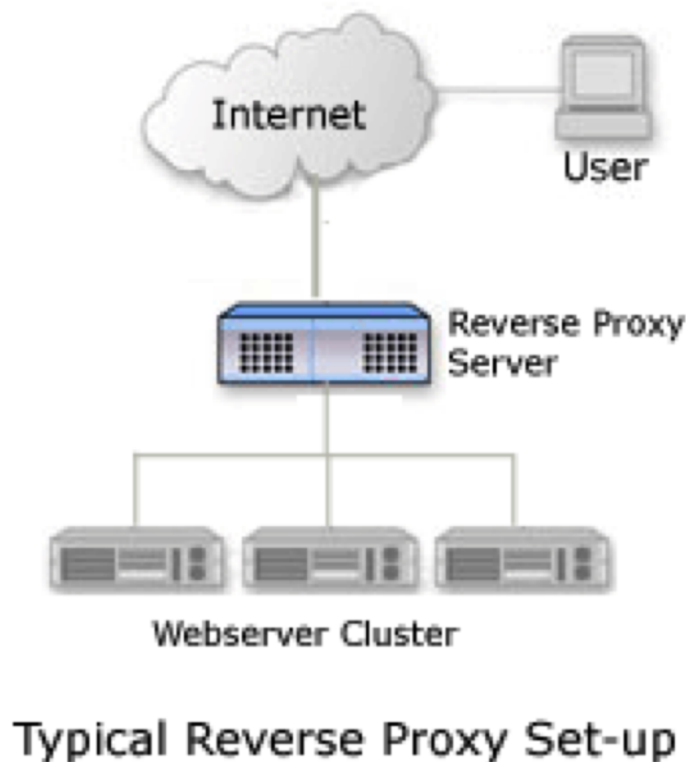

Figure 4.5: Reverse proxy

One option is to use the concept of a static file cache or reverse proxy cache to automatically identify highly active files and to remove their request load from the main archive store. Figure 4.5 shows how the reverse proxy parses all requests to the main system. It identifies frequently accessed items and stores them locally in the reverse proxy server. Requests for these items are then served directly without involving the main system.

Another option is to have the main system segregate the frequently accessed items in dedicated high performance servers. As discussed later in Chapter 7, it is possible to build reasonably high capacity servers using solid state disks. These dedicated servers would have very high throughput capacity. This option differs from the reverse proxy cache in that the dedicated servers are the actual long term storage site for these files. With a reverse proxy cache, the files still need to be stored somewhere in the main system.

4.7.2 The top tail

From the point x_{min} onwards the remaining items fall into Anderson's long tail. Very quickly, as the items rank increases, item number of downloads and therefore its probability of its access becomes exceedingly small. If the system had a perfect oracle, it could assign file locations based on each items future access probability. This would serve to create active disks and inactive disks. Unfortunately, there is no such oracle and hence some approximation will need to be used.

Consider the possibility for moving files around as they are accessed. For each file access, the system would look at the historical access data and move the file either to a more active disk, or to a less active disk. If the destination disk was full, then a potential chain of movements would occur until the system placed all the files appropriately. Eventually, the system would reach a more or less steady state, with only new files causing movement between disks.

The resulting file system would have active disks and inactive disks. Based on empirical studies, a straight priority placement policy might overwhelm the active disks I/O operations per second limit because active disks would have too many downloads over a given time period. A secondary balancing mechanism would them be required to spread active files around in order not to overload any specific disk.

Note that at some point in the rank of items, the download count for items in the tail becomes flat because many files have the same download count. The difference between a random oblivious knowledge placement algorithm and a historically correct ordered placement policy becomes negligible. For these files, there is no difference between the ordered placement and the random placement. The only differences are the few files that have more or less unique download counts.

Take a look at the files that do not fit within this long flat tail. These items are listed as *Top Tail* in figure 4.4. These are files who have larger download counts. If these files are included in the random placement area, they would have significantly higher download probabilities than the rest of the files and hence would make some disks significantly more active.

The specific rank which differentiates between the top tail and the long tail depends directly on the download counts for each item and the selected period of time for which we are optimizing the storage system. Consider a case where a given file is accessed three times a day for the period of one year. Its download count would be 1095, while other files might have only a single download during that time. If the system was optimized for 24 hour periods, this file with 1095 downloads would still be in the long tail. Three times during 24 hours is insignificant in access terms.

On the other hand, consider a system where the x_{min} point is 500,000 downloads per year, or 1369 per day, 57 per hour and almost one per minute. Such a file would automatically make the disk on which it sits active and hence it would be part of the top tail.

The simplest approach is to treat these files along with those in the Head. In any case, the system has to support very frequently accessed files. Treating these top tail files as active files simplifies the system by mandating only two access systems, one for the head and top tail, and one for the long tail and zero class.

4.7.3 The long tail

For those items with sufficiently small download counts over a given period , the oblivious random placement strategy is close to the knowledgeable ordered placement strategy. These items should simply be managed using the random placement strategy. The

benefits of this strategy are that there is no requirement to maintain access histories for each item or to manage chains of file placements as items are accessed. The system is stable from the first moment that items are placed on the file systems.

This approach can be extended to include those items in the Head and the Top Tail. Assume that all items are placed using the random placement strategy. The cache or reverse proxy server will pick up the active files once they start to be accessed. At that point, their placement on the main storage system uses only disk space and does not increase system load or I/O operations per second.

In addition, such a simplified strategy enables equivalently simple backup strategies. Each disk could have one or more exact duplicates that could support disaster replication and/or load balancing. Because items are never moved from one disk to another the duplicates remain static and stable. The drawback is that the loss of a disk requires a complete rebuild from the one or more remaining replicas.

Alternatively, two or more complete storage systems could each implement the random placement strategy. The random nature of each system would mean that in the event of a disk failure, the rebuild would pull from most if not all of the remaining disks, amortizing the load and reducing specific bottlenecks. The drawback is that such a rebuild requires a bill of lading for each and every disk so that the lost files can be identified and the disk rebuilt.

4.7.4 The zero class

If the system is sufficiently large, then some files will be accessed only once in a lifetime. These files make up the zero class, with zero downloads during most long periods. If these files could be identified at insertion time, they could be written to very low cost, high latency devices such as laser disks, tapes or other tertiary storage devices. Unfortunately, there is no a-priori way to identify these files and hence they should be placed along with the long tail items.

In the worst case, these files take up disk space and are backed-up along with all the other files. As a benefit, the unified architecture means that only one main storage architecture is needed for the entire system, regardless of where the file's access probability falls.

Chapter 5

Empirical Findings

This chapter explores the existence of long-tailed distributions in actual large scale file systems. Whereas the previous chapter began with the premise that such access distributions existed, this chapter will provide empirical proof that such distributions exist. There are three case studies in this chapter, a multi-terabyte open-source software repository, a half-petabyte collection of media files and a half-petabyte collection of archival web pages. Each system is described briefly along with the details of the data collection and analysis methods. The observed distributions are presented along with approximations to model statistical distributions.

5.1 SourceForge

SourceForge www.sourceforge.net, is a repository for open-source software development. The site provides for each project, source control facilities, an issue tracking system, project specific wikis and a download repository. The most significant requirement [80] when hosting a project on SourceForge is that the project must have an open-source license [81]. SourceForge has been operational since 1999 and provides researchers with access to some of its internal database records. As of February 2009, SourceForge hosted more than 230,000 projects with 1,676,535 unique downloadable files occupying a total of 5,134,401,994,129 bytes or a bit over 5 terabytes.

Each project on SourceForge can publish files for download. These files may be project documentation, full releases or partial components of a project's releases. For example,

the Pidgin project, a chat client that integrates most popular instant messaging services, offers hundreds of downloadable files for versions 2.0.0 through 2.7.1. Each of these files is tracked in SourceForge database, with fields for file size, and total downloads.

The dataset [82] used this research was collected by Greg Madey and his group at Notre Dame. The dataset consists of selected tables copied on a monthly basis from the SourceForge site. The ER diagrams, schemas and tables are published for all available data sets. More than a hundred papers [83] have been written based on this dataset, beginning in 1999 and continuing until the present day.

The datasets available from SourceForge have changed over time. Up until March 2005, the tables included daily download statistics for each file. After that date, only monthly aggregate values have been published. The research in this section aggregates the older daily data with the newer monthly data into a single monthly dataset that covers downloads from November 1999 until December 2008. There were 3,588,755,259 downloads of 1,676,535 files over a period of 111 months.

Of the 1.6 million files in SourceForge, almost one third, 505,951, have no recorded downloads. These files would be candidates for eventual deletion from the system since the chance for access falls over time, as the project becomes older and more obsolete. For the purposes of this study, we will remove these files and focus only on those files that have at least one download during the 10 year study. The remaining files will be referred to as *active files*.

Figure 5.1 presents the download activity for SourceForge from 1999 to 2009. The graph presents the number of unique files downloaded per month, the total number of downloads and the number of active files that existed at that time. As can be seen, there is a gap between the total number of files and the files that were downloaded.

Figure 5.2 provides more insight into the percentage of files accessed over time. The bottom-most line represents the gap seen in Figure 5.1 and ranges from 15% up to 35% of the active files. Over time, more and more of the total active files are accessed. The maximal access seems to be about 65% when viewed over three years, and that value is very stable. This suggests that for the files in SourceForge that have at least one download, 65% of the files will be downloaded. This value is indicative of a long tailed download distribution since longer samples find additional files. Yet, because the

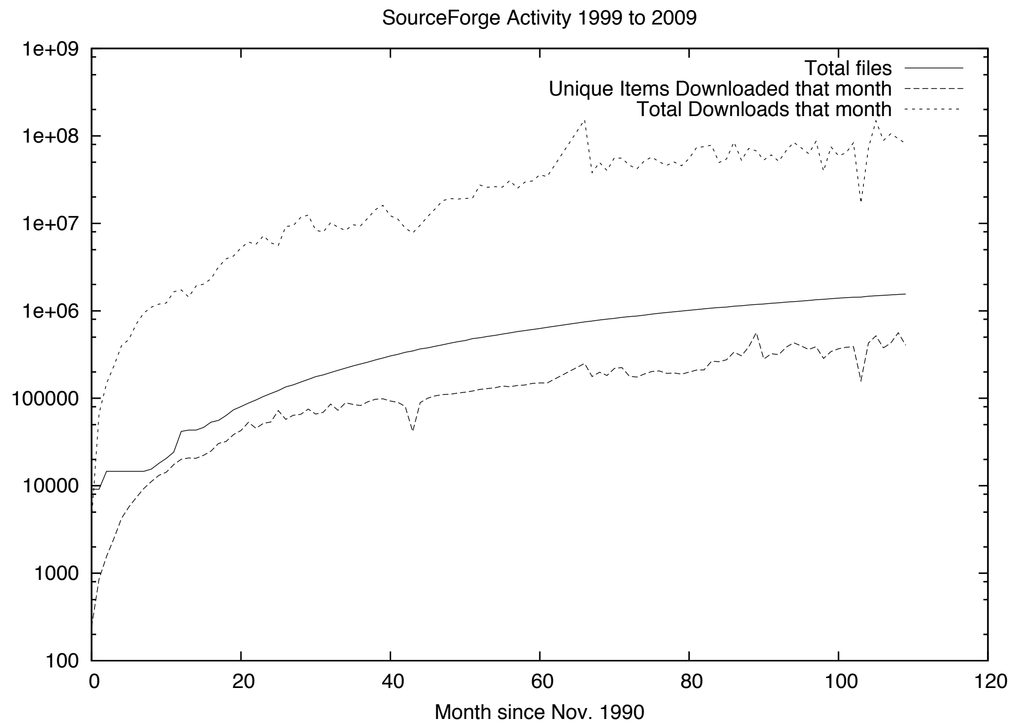

Figure 5.1: SourceForge Activity 1999 to 2009

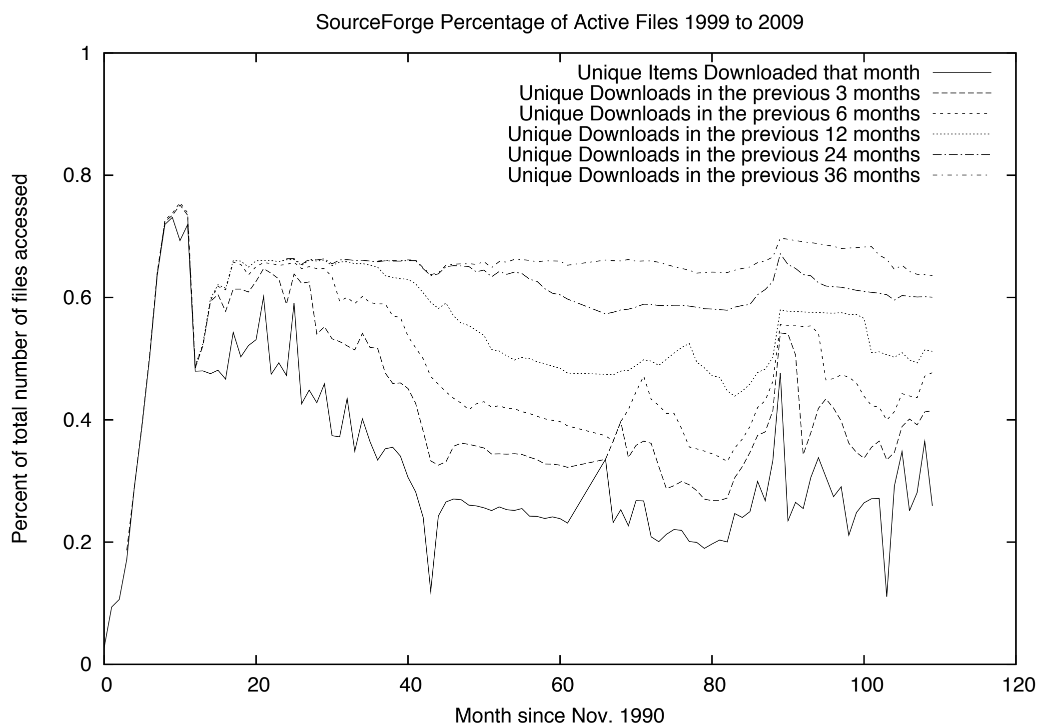

Figure 5.2: SourceForge downloads as a percentage of the total number of active files

maximum value seems to be 65%, there may indeed be files that are cold enough that they could be removed from the system, or moved to slower, more efficient media.

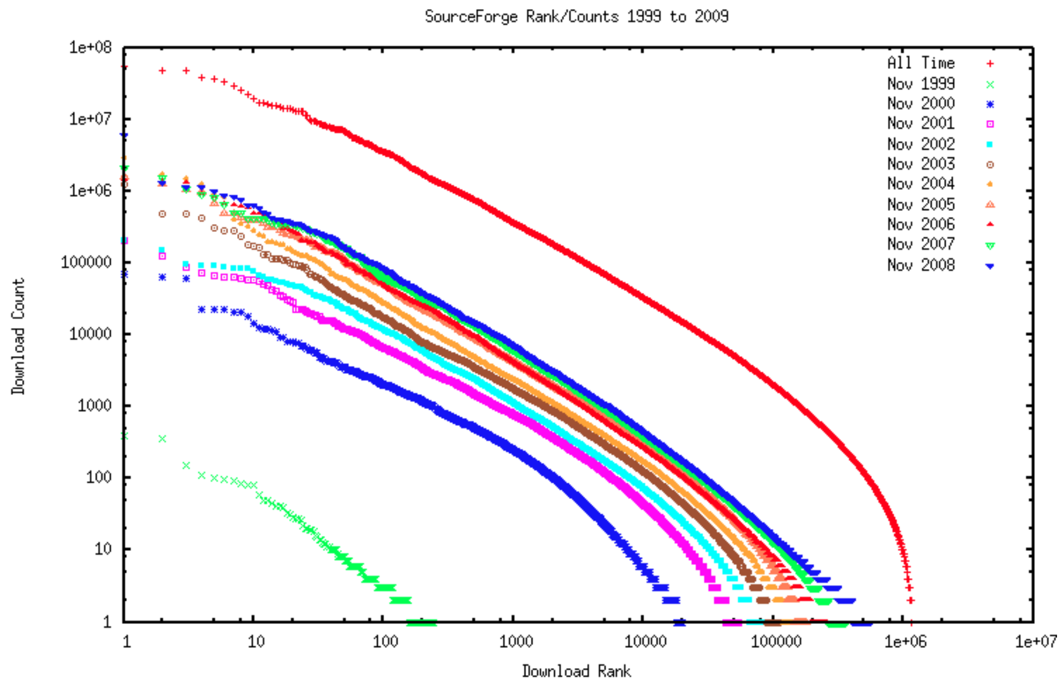

FIGURE 5.3: SourceForge downloads by count and rank

The hypothesis that the download distribution is long-tailed can be eye-balled in Figure 5.3. This chart shows the rank of each file by download count. The rightmost line is the overall plot for all time, while the internal lines are the plots for the first month of each year of operation. In any given month, the plot approaches a straight line from beginning to end. The difference between the monthly lines and the "All Time" line are those files that were not accessed in that month.

5.1.1 Distribution Fitting

A common method for the initial identification of a distribution as long-tailed is to plot its probability density function on a log-log scale. A long-tailed distribution will be visible as a more or less straight line with a strong negative slope. Figures 5.4 and 5.5 show the difference between a non-long-tailed distribution (normal) and long-tailed distributions (pareto). The blue circles are samples, while the black line is the nearest fit to a pareto distribution.

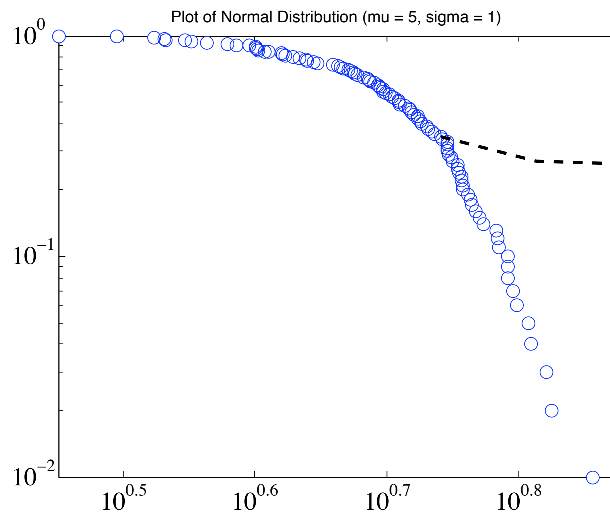

FIGURE 5.4: Log-log plot of a normal distribution

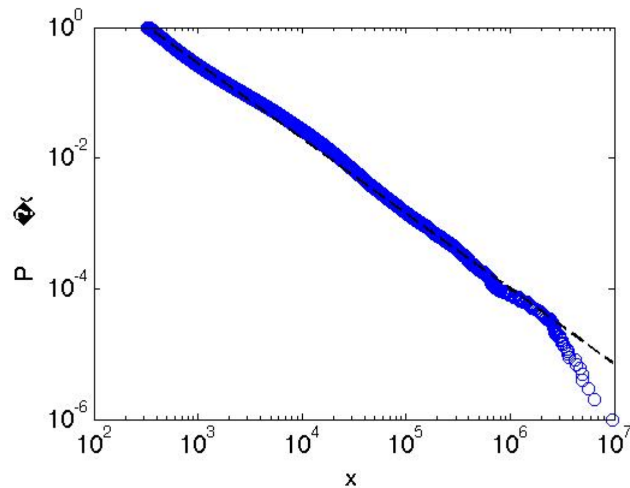

FIGURE 5.5: Log-log plot of a pareto distribution

A more formal approach is to estimate the parameter α of a pareto distribution and to determine the goodness-of-fit values between the derived distribution and the data set. The first challenge when fitting a power law is that the head of the distribution may not behave appropriately. It is for this reason that the pareto distribution includes a parameter x_{min}, to specific the first point in the distribution. We use a software package [84] that applies a power-law fitting algorithm to the entire data set and then to progressively shorter segments of the data set until it finds the best fit.

Clauset et.al. [84], provide a recipe for analyzing power-law distributed data. The formula for a power-law distribution is defined in equation 5.1 for $\alpha > 1$ and $x_{min} > 0$.

$$\Pr(X = x) = p(x) = \begin{cases} \frac{\alpha-1}{x_{min}} \left(\frac{x_{min}}{x}\right)^{\alpha} & \text{for } x \geq x_{min} \\ 1 & \text{for } x < x_m \end{cases} \tag{5.1}$$

The first step is to estimate the parameters x_{min} and α. We then compare the power law with alternate hypothesis via a log likelihood ratio test. For each alternative, if the calculated log likelihood ratio is significantly different than zero then its sign indicates whether the alternative is favored over the power-law model. That is, negative log likelihood values indicate a preference for a power-law distribution over the alternative models.

Clauset derives a maximum likelihood estimator (MLE) for estimating the scaling parameter α. For the case where the number of samples n, is very large, $\hat{\alpha}$, the estimate of the true value α, can be computed using the formula:

$$\hat{\alpha} = 1 + n \left[\sum_{i=1}^{n} ln \frac{x_i}{x_{min}} \right]^{-1} \tag{5.2}$$

The estimator assumes a known value of x_{min}. Clauset uses a Kolmogotov-Smirnov (KS) statistic which represents the maximum distance between the cumulative distribution functions (CDFs) of the data and the fitted model.

$$D = \max_{x \geq x_{min}} |S(x) - P(x)| \tag{5.3}$$

Where $S(x)$ is the CDF of the observed data with values at least x_{min} and $P(x)$ is the CDF of the power-law model that best fits the data in the region $x \geq x_{min}$. The estimated value $\hat{x_{min}}$ is the value of x_{min} that minimizes D.

The log-likelihood estimator value is derived by Clauset as:

$$L = n \log \frac{\alpha - 1}{x_{min}} - \alpha \sum_{i \geq x_{min}}^{n} \log \frac{i}{x_{min}} \tag{5.4}$$

A negative value indicates a preference to a power-law fit, while a zero or positive value indicates a non-power-law distribution.

Clauset has published a MATLAB program called plfit.m that calculates the values α, x_{min} and L. A companion program, plplot.m, graphs the actual distribution against the computed values. In the rest of this chapter, we will report values as derived by these two programs. The size of our datasets were large enough that the programs as provided were unable to report values after 8 hours of clock time. A sample of each data set was taken, using every 10th or every 1000th value. The resulting smaller dataset was amenable to analysis by these programs. Empirically, the resulting value for α provided a good fit against the full distribution. Because the full sampled data was not used, the values of x_{min} were only loosely related to the actual dataset. We report the values as an indication of the size of the head of the distribution, but recognize that the actual values are likely to be significantly larger.

The derived values for the SourceForge dataset are $\alpha = 1.898$ and a log-likelihood value of -3.24. The value of x_{min} for a sampled dataset of one tenth of the data was 10002. With approximately one million values in the full data set, this suggests that the head may include as much as 100,020 values, or almost one tenth of the total sample. Figure 5.6 shows the graphs of the empirical and derived distributions.

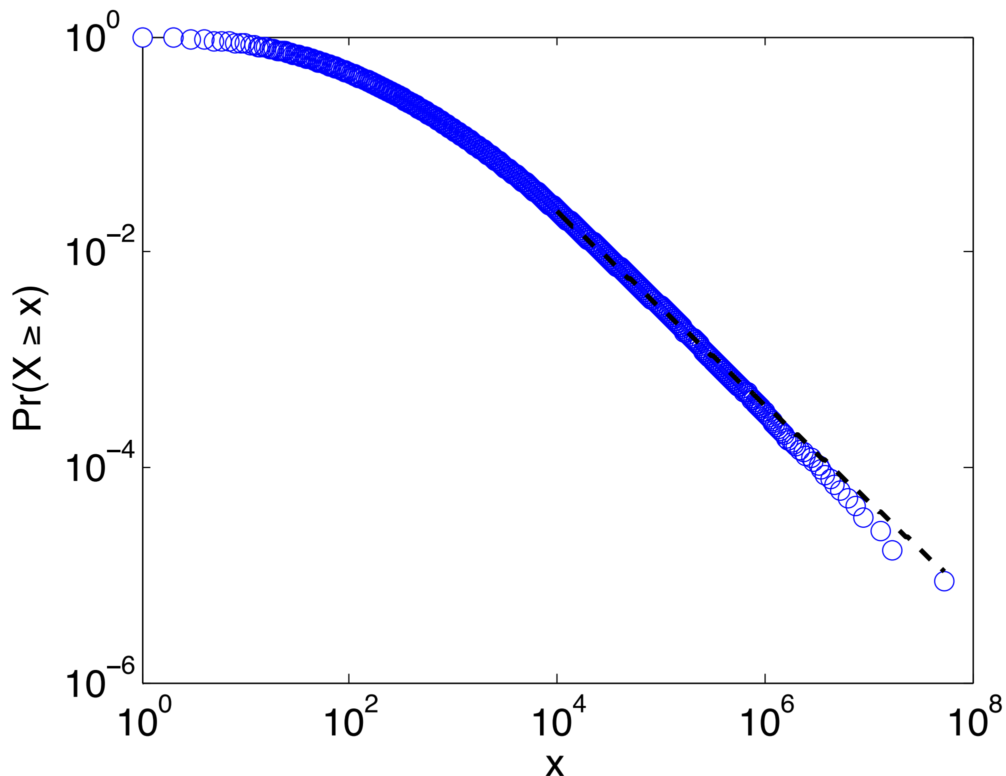

FIGURE 5.6: SourceForge Downloads vs. Pareto Distribution

5.1.2 Mass Count Disparity

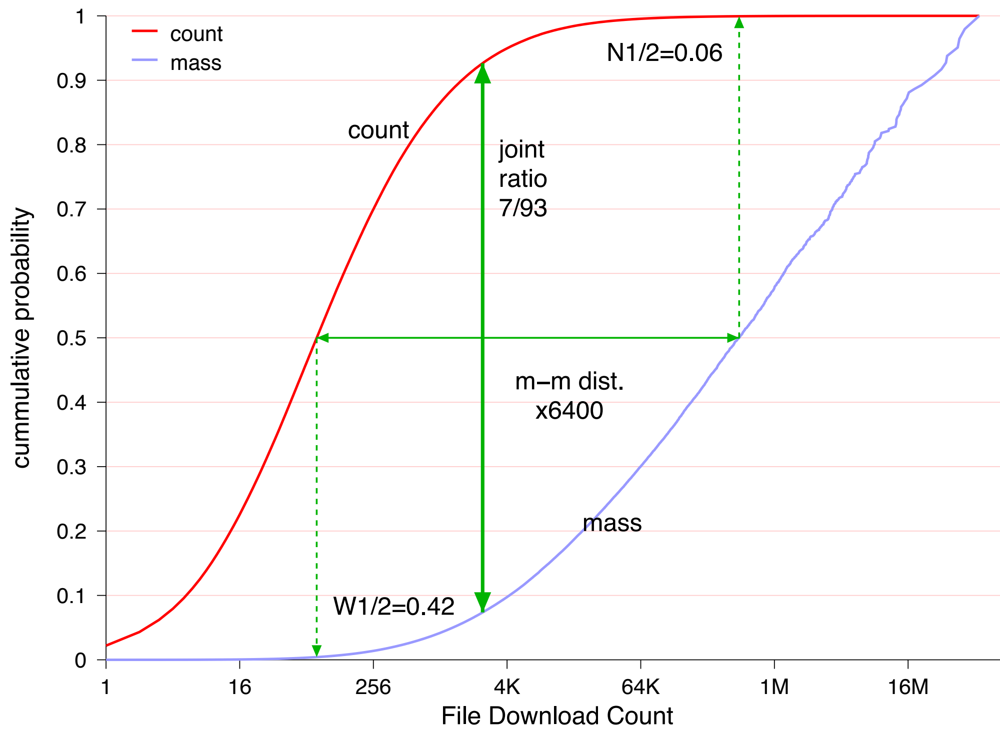

FIGURE 5.7: Mass Count disparity for SourceForge downloads

One way to visualize an access distribution is to the mass-count disparity plot[85]. The plot is based on two distributions, the first being the common CDF, $F_c(x) = Pr(x < X)$. This is called the count distribution as denoted by the subscript c. The mass distribution is to weight each file by its contribution. Feitelson describes the mass distribution in terms of the base distributions pdf, $f(x)$.

$$F_m(x) = \frac{\int_{-\infty}^{x} x' f(x') dx'}{\int_{-\infty}^{\infty} x' f(x') dx'} \tag{5.5}$$

In our case, the count distribution reflects the probability that a given file is downloaded. The mass distribution reflects the contribution of each item to the total number of downloads. The formula for the number of count distribution based on the empirical data. Define the function $mass(x)$ as the number of downloads for an item of size x. The two distributions are then:

$$F_c(size) = \frac{\sum_{x=1}^{size} mass(x)}{\sum_{x=1}^{maxsize} mass(x)} \tag{5.6}$$

$$F_m(size) = \frac{\sum_{x=1}^{size} x \; mass(x)}{\sum_{x=1}^{maxsize} x \; mass(x)} \tag{5.7}$$

The mass-count disparity can be helpful in eyeballing two common rules. The first rule is the 80/20 rule, also know as the Pareto principal. Pareto noted in 1906 that 80% of the land in Italy was owned by 20% of the population. In Computer Science, the Pareto principal has been used to suggest that 90% of the work will take 10% of the time and the remaining 10% of the work will take 90% of the time.

The second rule is 0/50% rule, which suggests that fully half of the objects are so small that their contribution is negligible. This rule helps to focus the efforts of the developer or designer to the 50% of the objects that do have an impact.

Mass-count disparity charts expose these two rules visually. The joint-ratio presented at the center of the chart is the Pareto principal values. This value is the unique set of points at which the two distributions sum to 100%, that is p% of the items account for 100-p% of the mass and 100-p% of the items account for p% of the mass.

The 0/50% rule is represented by two metrics, $N_{1/2}$ and $W_{1/2}$. The metric $N_{1/2}$ quantifies the percentage of items from the tail that are needed to account for half of the mass:

$$N_{1/2} = 100\left(1 - F_c\left(F_m^{-1}(0.5)\right)\right) \tag{5.8}$$

$W_{1/2}$ quantifies the percentage of the mass represented by half of the items:

$$W_{1/2} = 100\left(1 - F_m\left(F_c^{-1}(0.5)\right)\right) \tag{5.9}$$

A final contribution is the median-median distance which identifies the distance between the medians of the two distributions. The further apart, the heavier the tail of the distribution. The absolute values depend on the units used in the distributions. We therefore express the distance as a ratio.

With this background, we can discuss Figure 5.7, representing the mass-count disparity for the SourceForge downloads. The joint ration of 7/93 notes that a very large percentage of the mass is represented by just 7% of the files. That is, there are a fraction

of files that are very heavily downloaded, while 93% of the files are only infrequently downloaded.

The $W_{1/2}$ and $N_{1/2}$ values are both extremely small. 50% of the files account for less than 0.42% of the downloads and conversely, 50% of the download mass is captured by just 0.06% of the files.

Finally, the median-median distance ratio is 6400, indicating a heavy tailed distribution.

5.1.3 Heat Maps

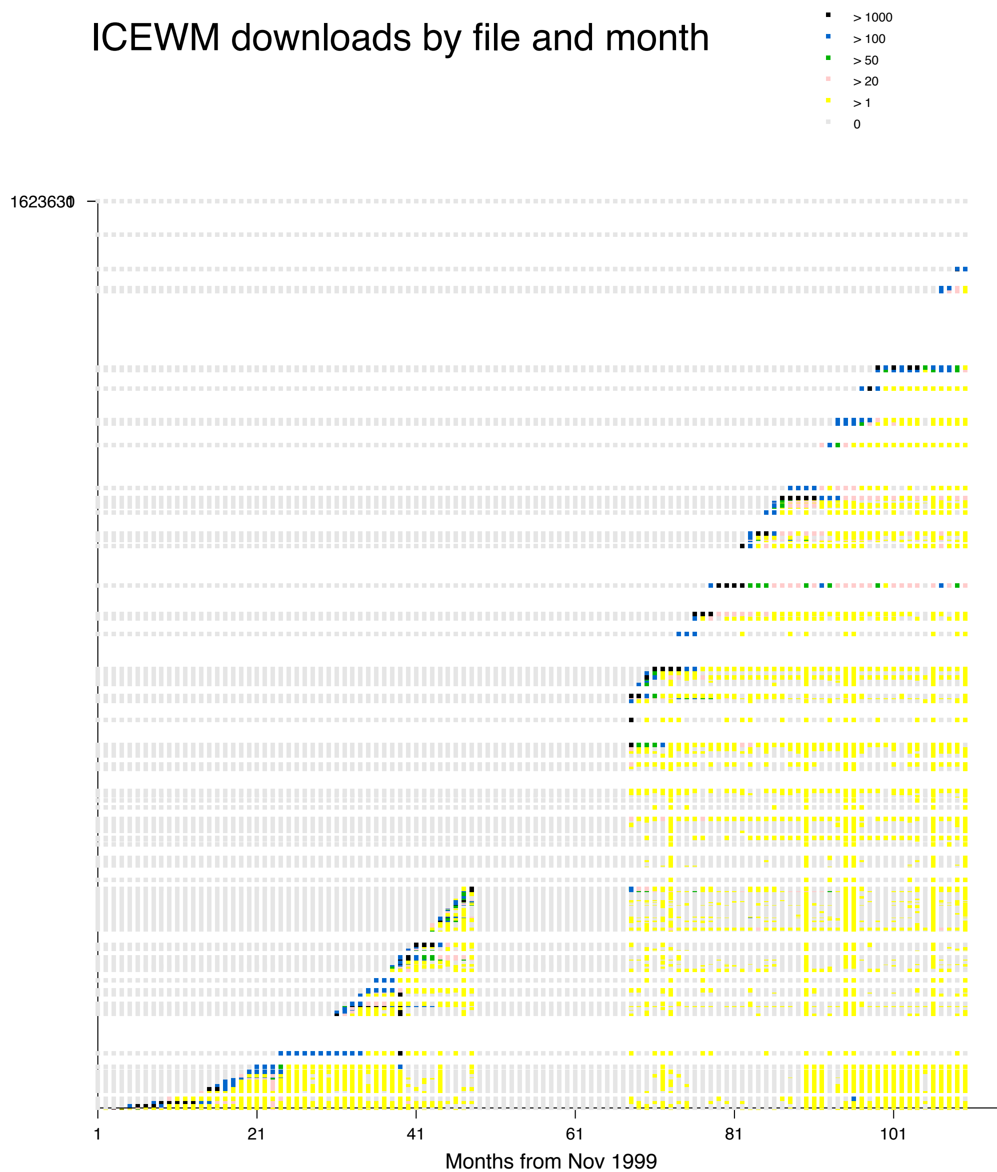

FIGURE 5.8: Access to ICEWM project files

A complementary method for viewing long tailed distributions is to build a heat map of the items over item. Figure 5.8 is a heat map of file accesses within the ICEWM project on SourceForge. The vertical lines represent the files in the project, each line is a different file. The x-axis is the months since November 1999, when the dataset began. Each colored block represents a number of hits to this file during the corresponding month. The darker the block, the more hits occurred. As each file was released, there were many hits and that continued for a number of months. More interesting are the yellow blocks, representing 1 to 19 hits. As can be seen in the later months, there are a significant number of yellow boxes even for files which were very old. Furthermore, the hits are across most of the files.

This heat map shows that file are accessed even though they are old, but that those accesses are minimal. It might be argued that file hits are because of internal maintenance efforts, but the dataset derives from the access database, which only records accesses via the standard web site.

5.1.4 Discussion

SourceForge is an archive, and it behaves as expected, with long-tailed access distributions. The heat maps show that for at least some projects, there is regular activity for all files, even if those files are ten years old.

From a practical perspective, the total disk space on SourceForge is only 5TB. Using modern technology, a single server can handle all 5TB with 5 single 1TB disks. Backup and Load balancing can be provided by duplicating this node one or more times.

Because the dataset is relatively small, the placement algorithms in this thesis are a non-issue. There is simply not enough data to require a full blown disk level placement and replication scheme. As will be seen in the following examples, one there are a few hundred terabytes of data, no single node can maintain all of the data and hence placement schemes are relevant.

5.2 Internet Archive Wayback Machine

The Internet Archive Wayback Machine is an Internet web site that provides access to archived copies of web pages from the Internet since 1996. There is a minimal web interface that allows users to display available the versions of a given URL. The user can then view one or more of those versions.

The specific architecture and design of the Internet Archive is provided in Chapter 6. For the purpose of this discussion, it is necessary to know that web pages are stored in ARC files, aggregated files that contain millions of web pages and are usually around 100 MB in total size. There is an index on the main servers that maps between URL and ARC file, but the location of the ARC file within the file system is not recorded. Instead, the system broadcasts a request for a given ARC file using its canonical name. The storage units that contain the broadcasted ARC file name will respond to the server. The server can then make direct requests for a URL within that ARC file.

For administrative reasons, I was unable to get access to the web logs for requests to the Wayback Machine. Instead, I implemented a sniffer that recorded the broadcast requests on the internal Internet Archive network. The requests were aggregated into 10 minute (600 second) time periods, listing the requested ARC files and the number of requests during that time period. This approach was chosen because it provides limits on the memory footprint of the sniffing process. Every 10 minutes, the current data was written to a log file and a data set was begun.

The script that implemented the sniffer ran continuously for 9 months, from September 24, 2006 at 19:59:18 GMT until July 1, 2007 at 05:49:20 GMT. The resulting log file contained 763,276,975 entries and required 40GB of disk space. Each entry lists the time at the beginning of the sample, the length of the sample (always 600), the name of the ARC file and the number of times that it was accessed. For example, the first record in the data set is:

```
1159127958      600     wb_urls.cgi2.20031104070938.arc.gz      6
```

The first number is the time in milliseconds since the Unix Epoch: Jan 1, 1970. The final number is the number of hits to this file during the 600 seconds of this sample. The

file accessed was *wb_urls.cgi2.20031104070938.arc.gz*, which was created in November 2003. Most files include a year and month in the file name. This is the only available mechanism for dating ARC files without direct access to the Internet Archives databases, which are not public and were unavailable to our research group.

There were a total of 11,305,266 unique ARC files referenced in the log. If each ARC file were indeed 100MB, the total size would be 1.1PB. In actuality, the total Wayback storage is less than .5PB.

5.2.1 Distribution Fitting

Using the Clauset software, the derived power-law fitting values for the Internet Archive Wayback Machine dataset are $\alpha = 2.41$ with a log-likelihood value of -24.8643. The value of x_{min} for a sampled dataset of one hundredth of the data was 3060. The log-likelihood value is negative and large, strongly suggesting a power law distribution, but the relatively large size of x_{min} and a full dataset size of approximately 11 million values suggests that the head of the dataset is approximately 2.8% of the total dataset. Figure 5.9 shows the graphs of the empirical and derived distributions. As can be seen in the distribution graph, while the head does seem large, it none-the-less falls quickly, even if it is not a pure power-law.

5.2.2 Mass Count Disparity

Figure 5.10 represents the mass-count disparity for the Wayback Machine downloads. The joint ration of 12/88 notes that a significant percentage of the mass is represented by just 12% of the files. More than one tenth of the files are frequently downloaded, while 88% of the files are only infrequently downloaded.

The $W_{1/2}$ and $N_{1/2}$ values are both extremely small. 50% of the files account for less than 1.45% of the downloads and conversely, 50% of the download mass is captured by less than 1% (0.81%) of the files.

Finally, the median-median distance ratio of 260 indicates a heavy tailed distribution, but not one that is as long as the two other datasets in this chapter. This distinction

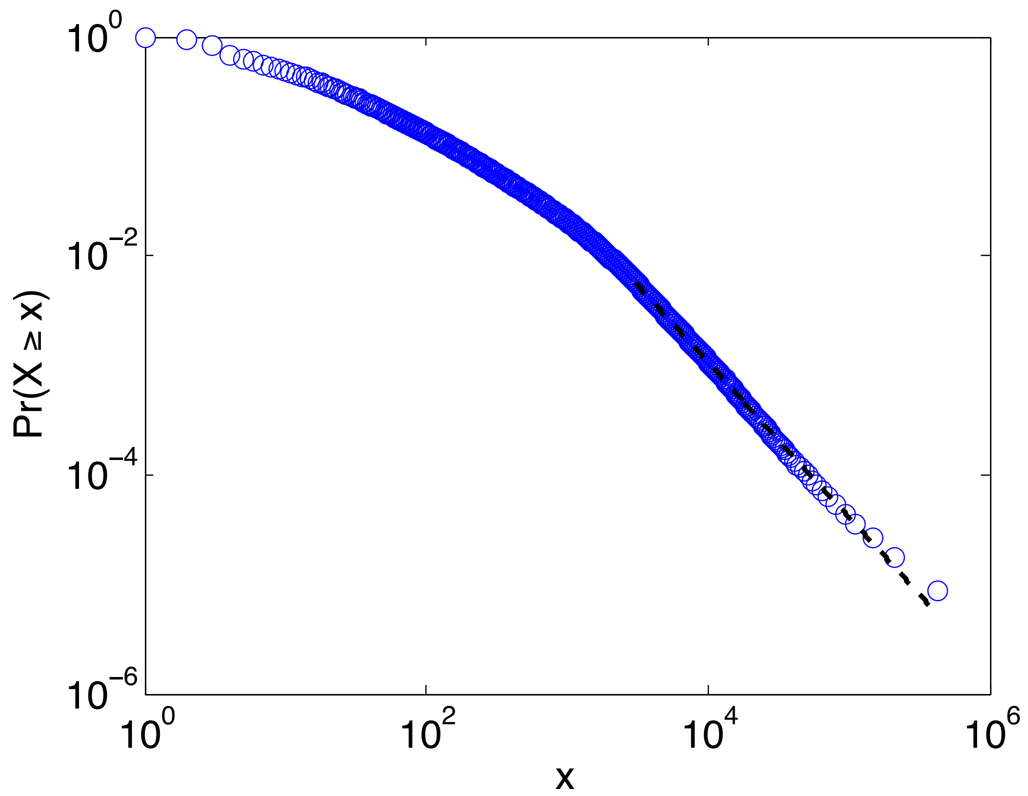

FIGURE 5.9: Internet Archive Wayback ARC file access vs. Pareto Distribution

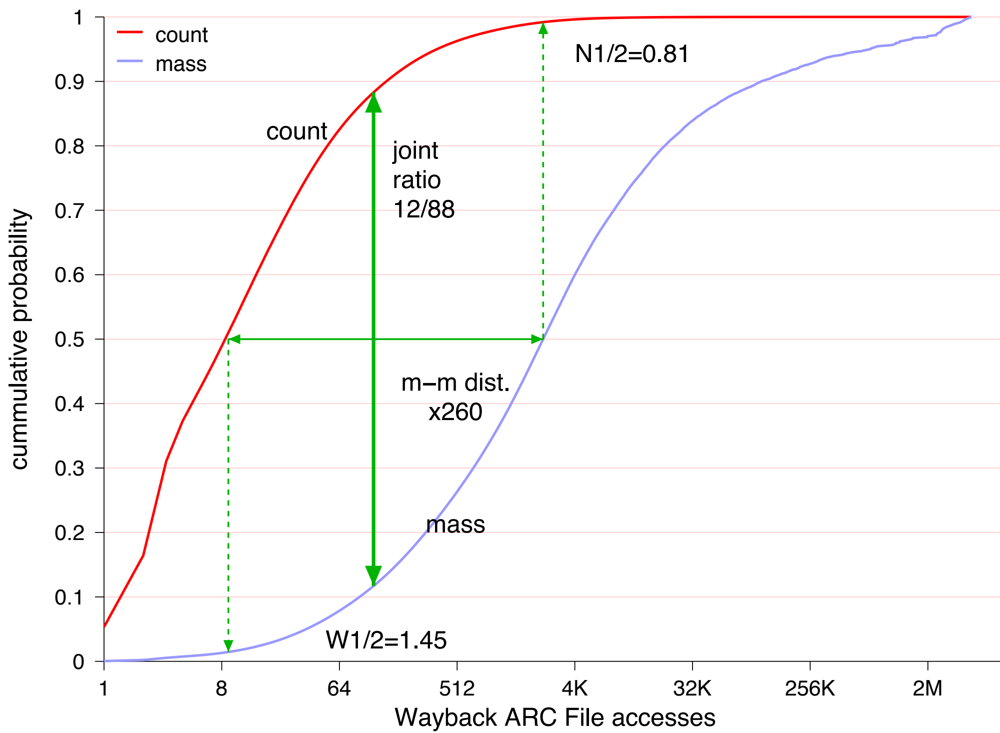

FIGURE 5.10: Internet Archive ARC File Mass/Count distribution

implies that there is some differentiation between hot and cold files, but that many files are warm instead of the very cold files in the IA Media collection.

5.2.3 Access over time

One perspective on a long-tailed access distribution is that over long periods of time, more and more of the data set will become accessed. This follows because the probability of access for any given file is very small. Over time, more and more of the low probability files will be accessed. Figure 5.11 shows the cumulative fraction of files accessed over the 9 months collected in this dataset.

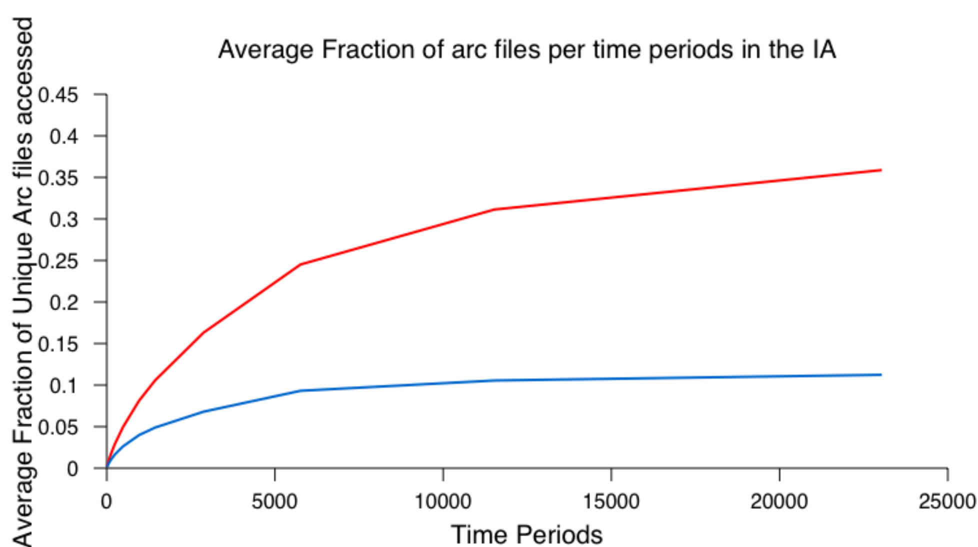

FIGURE 5.11: Internet Archive Wayback Cumulative ARC file access over time

5.3 Internet Archive Media Collection

The Internet Archive maintains two public archives. The Wayback machine collection was explored in section 5.2. The main drawback to the Wayback data is that there is no record of access to specific objects within the ARC files. It is possible to see what files are accessed, but not to develop a model that includes the size of each request.

The second public archive maintained by the Internet Archive is a 1/2 petabyte collection of media files; images, books, audio and video recordings. The requirement for the Media

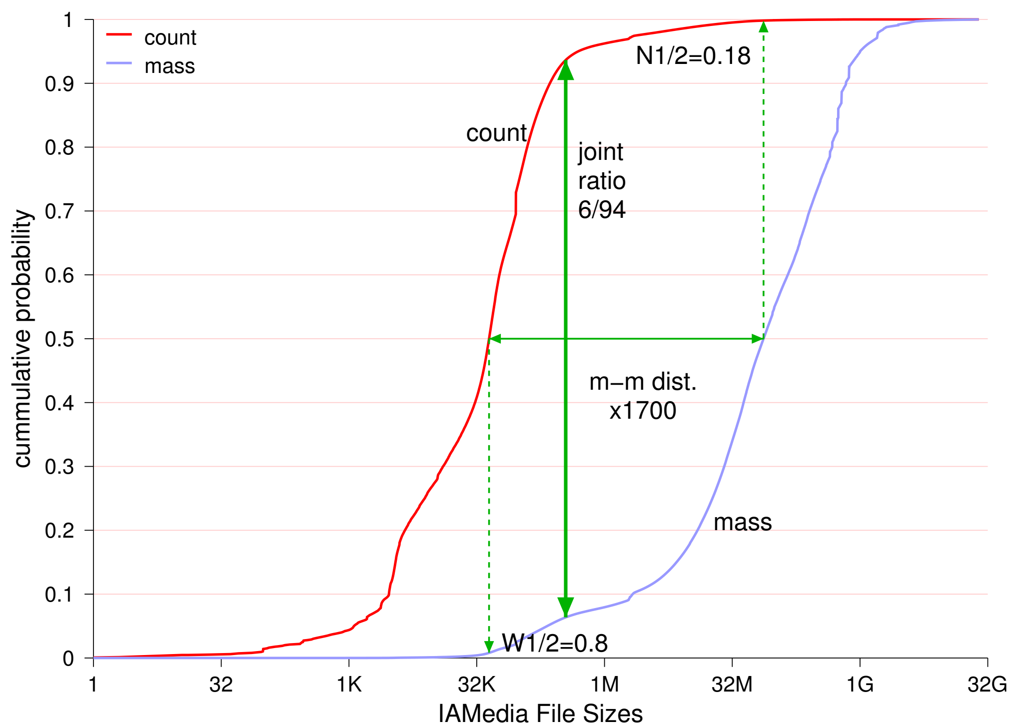

Figure 5.12: Internet Archive Media files by size

Archive is that all files be in the public domain or covered by a Commons license[86]. The range of file sizes can be seen in Figure 5.12 displayed as a mass/count plot.. The largest downloaded file is 26GB and there are 246 files larger than 4GB. More than 90% of all files on the system are larger than 2MB, but 50% of the downloads are for files less than 64KB.

The Media Archive consists of a set of front-end web servers that direct traffic to the server containing each requested item. The front-end servers poll the set of storage servers to see which server has the requested file, and then choose the server which first responded. The heuristic assumes that machines which response faster than other machines have more free cycles and bandwidth.

Each storage server runs a lightweight HTTP server and logs all requests. These requests are then aggregated into a single merged file for each day of operations. The logs remain on the Media Archive's server for 30 days. I wrote an automated script that downloaded these logs to our local servers for long term storage and analysis. Over a period of two years (730 days), I downloaded 662 logs.

The size of the log files and the large number of files posed significant analysis challenges. I used the Hadoop system to store each log and then developed a phased algorithm to parse the W3C logs, extract the file name and the sizes, and then create a merged map file. Even the size of the file was not obvious because the W3C log reported only the amount downloaded, even if the download was interrupted or if the download was only for a range of the full file. The analysis program computed the file size based on the largest observed download size for that file across all data sets.

Due to the variable length of the URLs, I first normalized the paths by removing extraneous HTTP parameters and leading paths specific to the node on which the data was hosted. For example, beginning with the following row in the log file:

```
41aeofjF6kgd.41Pkuv/uyatm2412P0TgB3uVkU41POTVm.IzUgI
ia311206.us.archive.org
 - [01/Jan/2009:22:36:34 +0000]
"GET /1/items/Hkyat23/Hkyat23.gif?cnt=0 HTTP/1.1"
404 4038 "http://www.archive.org/search.php?query=subject:"""
"Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1;
SIMBAR={D9CDDDFC-C1B0-4B1D-836C-8706DB8357EE};
FunWebProducts; Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1) ;
.NET CLR 2.0.50727; 3P_UVRM 1.0.14.4; IEMB3; IEMB3)"
```

The software extracts the URL "/1/items/Hkyat23/Hkyat23.gif?cnt=0" and converts it to "Hkyat23/Hkyat23.gif". The download size is 4083. bit the return value is 404, which indicates a permission denied error. This line is they dropped from the dataset, as only successful downloads with a return value of 200 or 206 are accepted.

After reduction and analysis, there are 390,418,951 unique files in the dataset, accounting for 569 TB of file data.

The histogram of downloads by file size can be seen in Figure 5.13. The graph was produced by putting each file into a bin of 2^n. As can be seen in the graph, the most popular download size is between 8K and 16K.

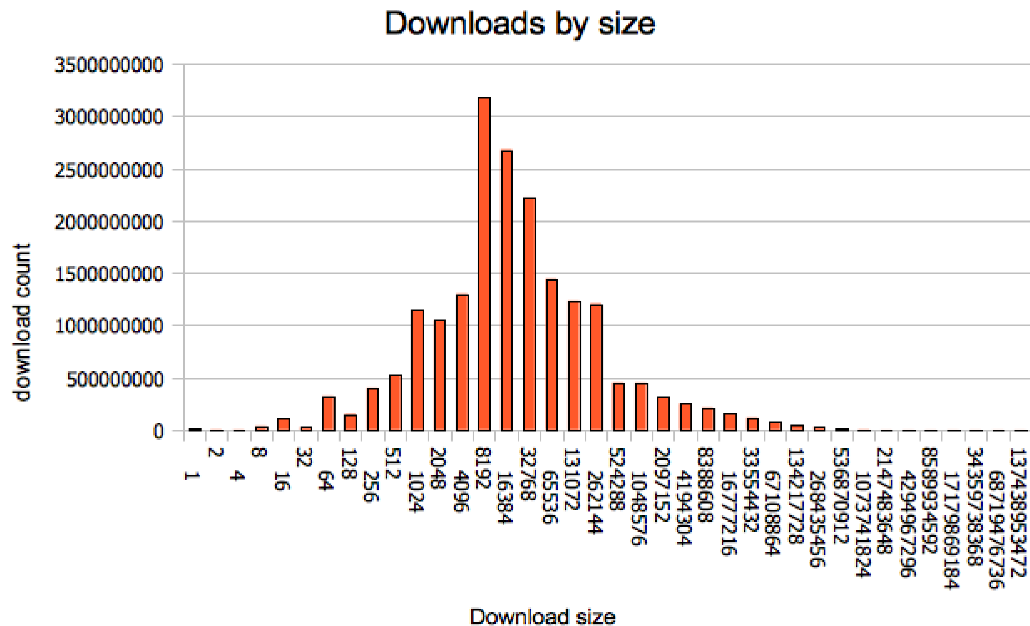

FIGURE 5.13: Internet Archive Media downloads by size 2008-2009

5.3.1 Distribution Fitting

Using the Clauset software, the derived power-law fitting values for the Internet Archive Media collection dataset are $\alpha = 2.07$ with a log-likelihood value of -27.6015. The value of x_{min} for a sampled dataset of one thousandth of the data was 551. The log-likelihood value is negative and large, strongly suggesting a power law distribution, but the relatively large size of x_{min} and a full dataset size of 390 million values suggests that the head of the dataset is very small, accounting for less than 1/700th of the total dataset The extent of the head and tail can be seen in Figure 5.14, the graphs of the empirical and derived distributions.

5.3.2 Mass Count Disparity

Figure 5.15 represents the mass-count disparity for the Internet Archive Media collection downloads. The joint ration of 6/94 notes that a significant percentage of the mass is represented by just 6% of the files. 94% of the files are infrequently downloaded.

The $W_{1/2}$ and $N_{1/2}$ values are both extremely small. 50% of the files account for less than 0.8% of the downloads and conversely, 50% of the download mass is captured by less than 0.18% of the files.

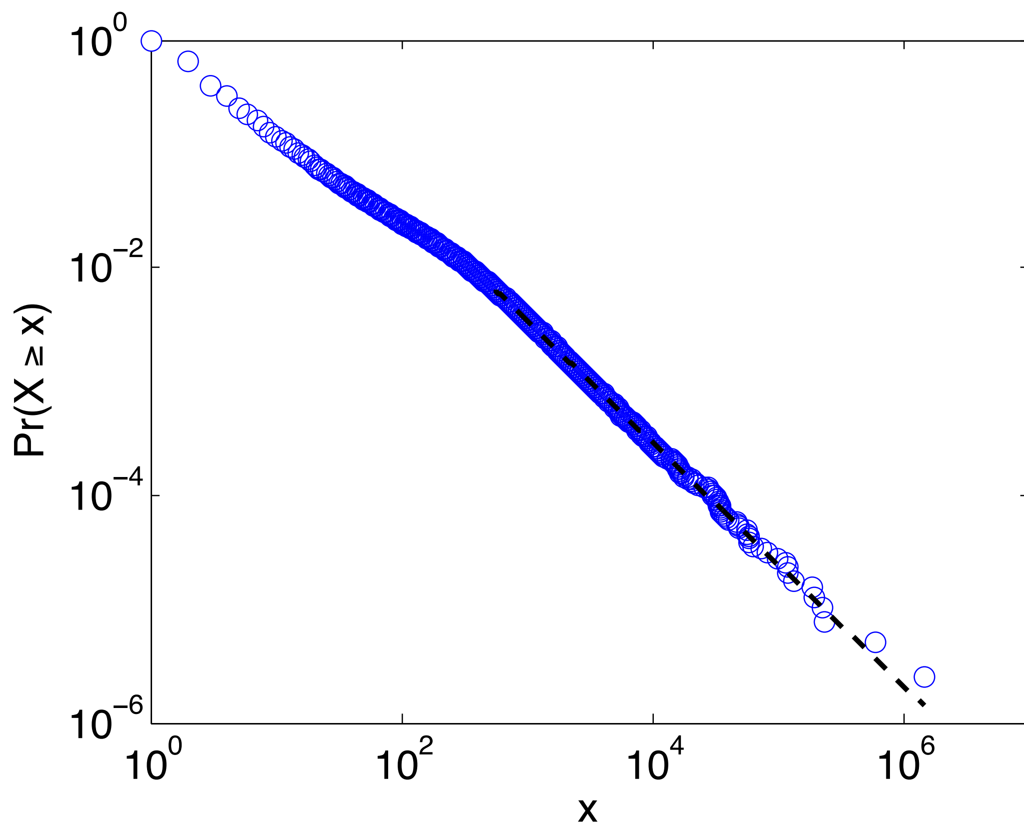

FIGURE 5.14: Internet Archive Media files access vs. Pareto Distribution

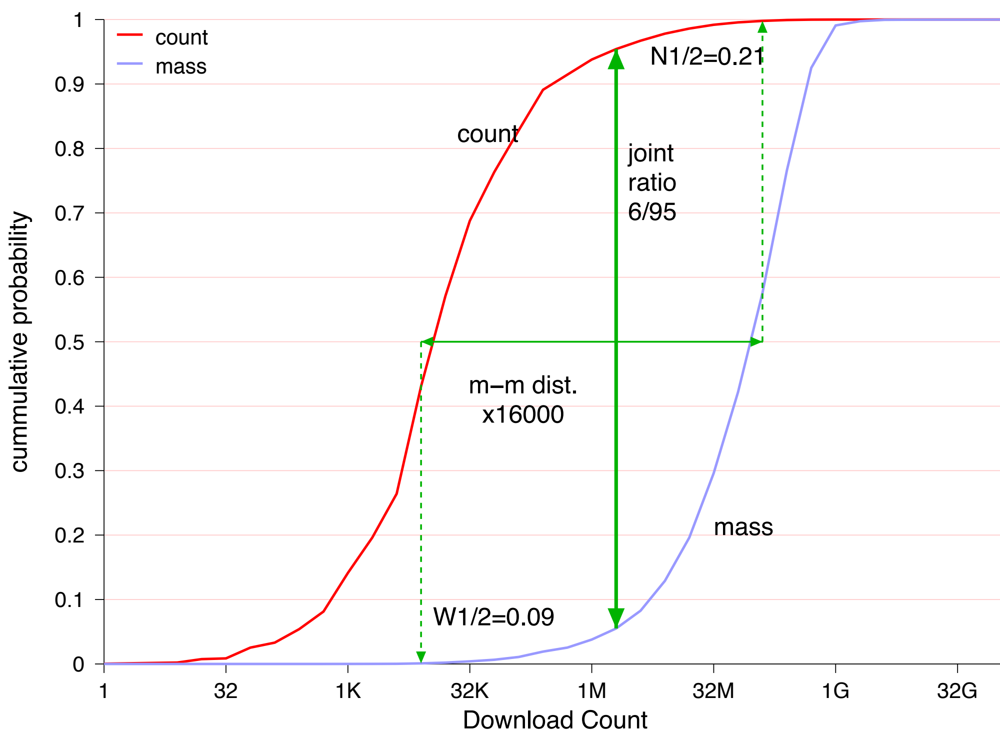

FIGURE 5.15: Internet Archive Media Mass/Count distribution

Finally, the median-median distance ratio of 1700 indicates a heavy tailed distribution. This distinction implies that there is significant differentiation between hot and cold files.

5.4 Conclusions

Table 5.1: Empirical analysis summary

	SourceForge	Wayback Machine	Media Collection
Number of items	1.67M	11.3M	390M
α	1.898	2.41	2.07
x_{min}	100,020 (10%)	306,000 (2.7%)	551,000 (0.14%)
Log Likelihood	-3.24	-24.8643	-27.6015
Joint Ratio	7/93	12/88	6/94
$N_{1/2}$	0.06	0.81	0.18
$W_{1/2}$	0.42	1.45	0.8
Median-median ratio	6400	260	1700

This chapter has presented empirical analysis of three datasets. Table 5.1 summarizes the pareto and mass-count values for the three distributions. All of the datasets exhibit long-tailed properties. The SourceForge dataset is the smallest and has the largest number of items not included in the tail of the distribution. Given that this dataset covers the longest time period (10 years), it may be that small datasets do not have as large a distinction between items. On the other hand, the SourceForge data has the largest median-median ratio, suggesting that in SourceForge, hot items are much more likely to be accessed than hot items in the other datasets. SourceForge's $N_{1/2}$ and $W_{1/2}$ values are the smallest of the three datasets, indicating that the files are more heavily skewed. Hot files are hotter and cold files are colder. This reflects the nature of software products. Older versions are less in demand and new versions can become very heavily accessed due being slashdot'ed or featured on a news show.

The Internet Archive datasets are clearly long-tailed, with very large negative log-likelihood values. The Wayback Machine files are less skewed than the other datasets. That is, there is a distinction, but the head itself is not very active. This makes sense because the Wayback files are less active than the other datasets. Whereas in SourceForge or the Media collection, a single file can have hundreds of thousands of downloads, an active file in the Wayback machine might have a few thousand downloads.

The Media collection is the largest dataset in terms of number of items. It also has the smallest tail as a percentage of the total number of files. The actual size of the head is within one order of magnitude of the other datasets, even though the total number of items is one or two magnitudes larger.

5.4.1 Impact of placement strategies

In the previous chapter, we derived the formula for x_{tail} as:

$$x_{tail} = \left(\left(\alpha - 1 \right) \left(|Objects_i| - 1 \right) x_{min}^{\alpha} \right)^{\frac{1}{\alpha+2}} \tag{5.10}$$

Table 5.2 shows the values for the three datasets that we have explored.

TABLE 5.2: x_{tail} values

Dataset	x_{tail}	% of dataset
SourceForge	276,576	16% (25% of active files)
Wayback Machine	8,586,483	75%
Media Collection	20,151,282	5.2%

In order for a placement policy to be effective, it must at the same time apply to enough files to make the storage interesting and it much leave no more files active than can be handled by a reasonable cache. If there are too few files cover by the algorithm, then the architect would be better off building a single algorithm that could handle both hot and cold files. The next chapter describes such an architecture in use by the Internet Archive.

The files in the head of the distribution must be managed differently than those in the tail. They must somehow be available on-demand for highly frequent access. The easiest method is to develop a large cache that can handle the items in the head of the access distribution while leaving the tail to the slower, idle disks.

The calculated values for x_{tail} expose two ends of this spectrum. The Media Collection data clearly fits the long-tailed random placement algorithm. While 5.2% still represents 29.5TB of storage, this is still within the range of a caching system.

On the other hand, the 75% of the files that remain active in the Wayback Machine dataset preclude using any placement algorithm. It is better to use a single algorithm to

cover all the files. In practice, the Internet Archive uses a random placement algorithm for all of this data as described in the next chapter. The reason that such an algorithm is still effective is that there are no heavy hot spots in the Wayback Machine corpus. By randomly spreading the data around, the system spreads the load, even though all disks are still accessed and items will still have a reasonable probability of access.

The SourceForge data represents a borderline case. In the specific implementation of SourceForge, there are only 5TB of data. The total dataset is too small to warrant any placement policy. The complete dataset can be mounted on 5 disks, and a simple cache can handle any of the active files. The two values represent the percentage of active files and the percentage of all files. Since almost 1/3 of all files are non-active, the 16% value is a better estimate for our purposes. If the system has significantly more files, the architects would need to make a judgement call. Can the 16% of active files be handled in a cache with 84% in a random placement store, or should a different architecture be developed.

In summary, the specific distributions have a significant impact on the placement strategies. Designers and Architects need to understand these distributions early enough in the development phase to prepare for scaling and capacity.

Chapter 6

The Architecture of the Internet Archive

In order to understand the impact of placement strategies on large scale archives, it is helpful to take a look at an existing archive that uses a similar architecture. This chapter provides a detailed description of the Internet Archive Wayback machine and follows with a discussion of the benefits and drawbacks of its implementation. This chapter concludes with a proposal for modifications based on the placement strategies outlined in the previous chapters.

6.1 What is it?

The Internet Archive (www.archive.org) is a petabyte scale public Internet library. It contains two major collections, The Wayback Machine providing access to approximately 500 TB of historical web pages collected from the Internet beginning in 1996, and a Media Collection containing more than 500 TB of public domain books, audio, video, and images. The Internet Archive has been in continuous operation since 2000. In its current state, the system handles tens of millions of requests totaling more than 40 TB of data each day, year round.

The Internet Archive is not only a good example of a large scale Internet site, but also an architectural and operational success story. One of the most interesting elements of the Internet Archive is that less than five employees are involved in operations

and maintenance. Surprisingly, the Internet Archive has accomplished this feat while avoiding almost all of the popular approaches for performance enhancement and storage minimization in favor of a simplest-solution-first strategy.

Many systems [87] implement a reverse proxy cache to improve performance and to reduce the load on its storage units. A typical proxy cache is located at a customer site or ISP and it proxies all web requests outside of the local domain. As requests are served, they are cached so that subsequent requests to that page will not necessitate a wide-area network request. A reverse-proxy cache resides at the hosting site. It's purpose is to remove the load from the site servers by proxy-ing and caching requests for frequently accessed items.

The Internet Archive has no such reverse-proxy cache. We will show that implementing and operating such a cache is not cost effective using currently available technology. However, newly available Solid State Disks (SSD) are shown to be a promising option for future implementations.

The case study presented here draws on our experience through access to the Internet Archive and discussions with its architects and operations staff. Please note that I have never worked in a technical capacity at the Internet Archive and am not privy to any internal operational decisions or policies. My position is one of remote researchers with very limited observational capabilities.

6.2 System Architecture

Using an approach similar to Kruhchten's [88], we describe the System Architecture through a number of different views. The requirements section details the goals and constraints which drove and continue to drive architectural decisions. The Logical view exposes the basic system objects. The Process view presents the ongoing activities involved in delivering the basic service and maintaining its integrity. The Development view describes the implementation of these processes. Finally, the Physical view exposes the hardware and software components that implement the system.

6.2.1 Requirements

The Internet Archive's mission is to be an Internet Library; reliably storing large amounts of data and delivering that data to users on the Internet. The system must be scalable, storing many billions of objects and petabytes of content. The only bottleneck to content delivery should be the Internet Archives connections to the external network. That is, the internal system should be able to scale based on customer demand and available outgoing bandwidth.

The system requires a search and index mechanism to enable users to locate specific items. The designers choose to take this requirement in its minimal interpretation. Items should be searchable by title and by pre-specified keywords. Detailed search at the content level is not necessary.

A library or archival system should make an attempt to maintain its data for many years and to retain that data in the face of component failures. The designers understood that there is a direct relationship between system cost and its level of reliability [89]. Basic reliability can be achievable with linear cost, but as the requirements grow, the cost to achieve those enhanced goals increases dramatically. The designers therefore intentionally set the minimum requirements rather low: Try not to lose data, but don't try too hard.

Its creator, Brewster Kahle established a few basic design requirements in support of the system's core mission. These requirements are still in force today and have significantly colored the architecture and operations of the archive.

- The system should use only commodity equipment.
- The system should not rely on commercial software.
- The system should not require a PhD degree to implement or to maintain.
- The system should be as simple as possible.

6.2.2 Logical View

The basic building block in the archive is a content element which represents a particular item such as a book, web page or video. Elements are grouped into aggregates such as

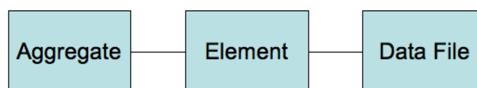

Figure 6.1: Logical View

collections or crawls. A collection might be a set of books as found in the Million Book Project [90], or movies such as The Prelinger Archives [91]. Other types of collections include web crawls performed by third parties or by the Internet Archive itself.

Each element may be composed of multiple data files. For example, a book may have hundreds of pages represented as images and as plain text. A video element may be retained in multiple formats for ease of access. Web pages typically reference other web objects as links or embedded objects. For web objects, each and every item is referenced as a separate element.

6.2.3 Process View

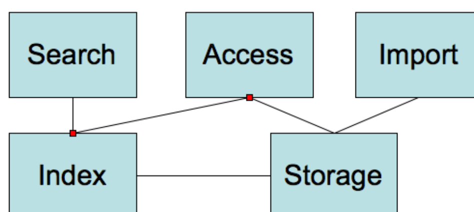

Figure 6.2: Process View

There are relatively few processes involved in the Internet Archive. The Storage process maintains the integrity of the elements as storage components fail or are retired from service. The Import process accepts items for storage and integrates these new items into the system. The Indexing and Search processes create and maintain the indices over items while enabling users to find specific items. Finally, the Access process delivers those items on demand. Figure 6.2 shows the relationships between these processes.

6.2.4 Development View

In this section, we describe the implementation for each of the processes listed above. The presentation is bottom up, first describing the basic components and then utilizing them in subsequent descriptions.

6.2.4.1 Storage

Logical elements are stored in one or more predefined root directories in the local file system on each storage node. Each root directory represents a entire hard disk, and each element has its own directory. Data files are then stored as files in their element's directory.

Web crawl elements are not stored as a directory because the number of items in that directory would stress most Linux file systems. Instead, these elements are stored as a relatively small number of ARC [63] files. Each file is a set of uncorrelated web pages usually totaling close to 100MB. For each web object, the crawler that gathers these objects appends to the ARC file a header followed by the content of that object. Note that the header appears in the ARC file directly before each item and not in some form of index. This process continues until the file reaches its maximal size at which point the crawler closes that file and opens a new file. One of the challenges in working with ARC files is that they are completely unindexed. The only way to search or access one of its web files is to sequentially scan the entire ARC file. ARC files are stored in their original unmodified form on the node.

In principal, each element is stored on at least two storage nodes. A monitoring service [92] identifies nodes that have failed and disks that are failing [93]. When an error is detected, the operators begin an automated process that copies the contents of the old node to a new node, either from a replica or from the failing node itself. In the past, some data has been lost because two nodes crashed at the same time in the lone data center. The current Internet Archive is replicated across three geographically remote sites, enabling, if necessary, retrieval from one of the remote sites.

An automated recovery system would be an obvious extension to the existing architecture. Such a system would have to manage many issues, including automated error detection and the provisioning of new hardware. In keeping with the aggressively simple

implementation approach, such a system was never implemented. Another perspective would be to note that such a system was never needed. The number of failures has never been high enough to cause undue load on the administrators. More details about the Internet Archives hard disk failure rates were published by Shwarz et.al. [31].

Finally, the storage systems provides a form of load balancing. The Internet Archive contains some very popular movies. When one of these movies becomes popular, the storage nodes may be unable to keep up with the number of concurrent requests. The operations staff manually watches for these spikes and copies the files to additional nodes. When filling the nodes, some extra space is set aside for just such situations. There is no need to ever remove these duplicates.

6.2.4.2 Import

New items arrive at the archive by many paths. They can be uploaded by Internet users, delivered by truck to the Internet Archive's facilities, provided as bulk transfers by partners, or created internally through web crawls or book scans. Regardless of how the data arrives, the process begins by locating the current import nodes; two twin nodes are dedicated to newly imported items. Imported items are stored in parallel to these nodes until they reach a ”fill level”, some percentage points shy of 100%. When that level is reached, two new empty nodes are provisioned and the process continues.

Where possible, the element, its data files and any metadata are collected during the import process and integrated into the content indices. There is no need to update any indices or metadata related to the new item's location because this data is dynamically determined by the Access process.

6.2.4.3 Index and Search

The Index process maintains a list of each item in the Internet Archive and any available associated metadata. There are at least two different index implementations.

Each web crawl includes millions of URLs. The total number of URLs in the library is between 2 and 10 billion items. The Internet Archive may store multiple copies of a web page if it was retrieved by distinct crawls. References to each URL are tagged by date.

The Wayback Machine first displays the available versions of each URL. Users can then choose to view a specific version from the Internet Archive's collections.

The original design requirements severely limiting the use of specialized software or hardware were taken to mean that the system should not use a database system. In response, The designers chose to store the URL index data in flat sorted files. The system is limited to searching for complete URLs and so the URL name space can be divided into similar size buckets. The system first removes common prefixes such as www and is then sorted by the remaining URL string. Hence the first bucket might contain all URLs that start with A, B or C. The next bucket would contain URLs that start with D, E, F and G, and so forth. Multiple index files are maintained, physically distributed across the main web servers. Requests are statically routed to the appropriate web server and index section based on the requested URL.

The web index was originally designed to be built by a batch process that was to be run each month. The process required reading every unique ARC file, extracting the URLs, sorting the results and finally splitting the index into sections. Until recently, index scans were performed very infrequently because each index scan caused the permanent loss of up to 10 hard disks. The specific cause of the disk failures seems to have been related to insufficient data center cooling capacity. Actively accessing the disks raised the machine room temperature by at least 5 degrees fahrenheit. This problem was addressed by moving the majority of nodes to a more capable data center. More recently, an improved process was developed that can incrementally update the index.

A major challenge for the index and search components are the sheer number of items in the system. There are more than a million ARC files. Assuming that the average page is only 20 kilobytes, each ARC file would then contain on average 5000 page objects. The total system would need to support more than 5 billion pages. The current index has close to that number of entries and requires more than 2 TB of storage.

The separate index is maintained for all other content, including books, movies, and audio recordings. It contains only searchable meta-data such as titles, authors and publication dates. This index is very small when compared to the Wayback Machine. It contains only a few million records and is currently implemented as a MySQL database.

6.2.4.4 Access

There is no central index detailing the location of elements and data files within the system. To find an item, a request for that object name is sent as a UDP broadcast to all data nodes. A small listener program on each storage node maintains the list of all local files in main memory and looks up each request. Those nodes that have the requested item reply to the broadcast with the local file path and their own node name. The requesting node then redirects the client browser to the appropriate storage node. Each storage node runs a lightweight web server that can efficiently deliver local files.

In effect, the system implements a distributed index that is very robust in the face of failures or updates. Moving a file from one node to another requires only updating the in-memory index on those two nodes. Failure of a node is invisible to the searcher. That node will simply not respond to any requests. This approach also creates a minimal form of load balancing. Heavily loaded nodes will naturally be slightly slower to respond to broadcast requests. Faster, unloaded responses will then be used instead of the subsequent responses from slower nodes.

The process is slightly different for web pages. When a particular URL is requested, the system uses a URL/ARC file index to identify the specific ARC file that contains this URL. The ARC file is then located using the broadcast mechanism and the client browser is redirected to that storage node. The light weight web server then spawns off a small process to open the ARC file, retrieve the page and return it to the browser. While this process is relatively expensive, the number of concurrent requests to ARC files per machine is small. There is sufficient local memory on each storage node to maintain an entire ARC file in RAM. This enables the kernel to prefetch the ARC file and to maintain it locally in case there are temporally close requests for items in that same ARC file.

6.2.5 Physical View

The Internet Archive architecture is composed of a small number of front-end web nodes and a large number of back-end storage nodes as shown in Figure 6.3.

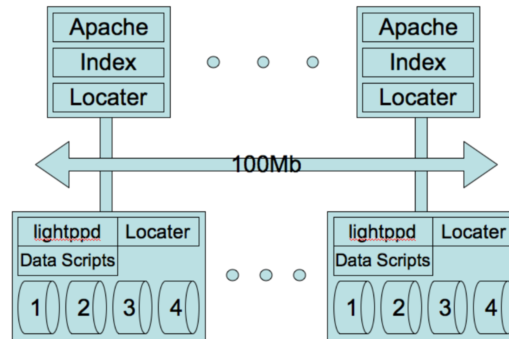

Figure 6.3: Physical View

6.2.5.1 Web Nodes

The Web Nodes are implemented on Apache web servers running in Linux on commodity hardware. A standard load balancer is used to parcel requests between front-end nodes. Since the Web nodes never deliver bulk data, they are tuned for high volumes of short requests. With the exception of search results, all other pages are static, further reducing the computation overhead.

The URL index is stored on these same nodes. It is split into segments and grouped by the first letter of the URL. Each segment is on at least two nodes in case one node should fail. The total index is more than 2 TB of data because of the large number of archived URLs.

The number of Web nodes is dependent on the maximum number of concurrent page requests, but not on the number of concurrent downloads. There are always a minimum of three operational nodes in case of a localized node failure. In practice, the Internet Archive has approximately six web nodes running at all times. These nodes have provided sufficient capacity for all existing needs.

6.2.5.2 Storage Nodes

Each Storage Node consists of a low power CPU and up to four commodity disks. Each node runs Linux and a lighttpd web server. Files are stored in the local file system. Data nodes are self-contained and do not depend on the activity of any other component in the system.

A simple program implements the location responder. To make things as simple as possible, the location responder performs a name lookup on each file request. The Linux kernel caches file names and so very few searches ever require a physical disk access.

There are more than 2500 storage nodes in the current primary data center, sufficient to keep at least two copies of each data file. The Bibliotheca Alexandrina [94] in Alexandria, Egypt and the European Archive citeeuroparchive in Amsterdam and Paris act as partial replicas.

6.2.6 Upgrade Path

There are no architectural decisions that strictly depend on a specific software or piece of hardware. As new hardware becomes available, it can easily be integrated into the Internet Archive's network. At the same time, old hardware can be thrown away. The built-in failure management processes will replicate the data on new hardware at the beginning of its life cycle.

When the Internet Archive began major operations, the size of the largest disk was around 30 Gb. Today, it is possible to purchase 1.5 Tb disks. The architecture is independent of the size or number of disks. The operations staff has the flexibility to purchase the most cost effective disks at any given time without concern for software or hardware interactions.

There are no dependencies on the specific hardware platform. Any system that is supported by one of the popular Linux distributions will necessarily include an Apache web server and the Perl interpreter. The Internet Archive requires that all programs be written in a portable scripting language for just this reason. The Internet Archive purchases only commodity hardware and avoids any specialized cards or add-ons. Because the hardware is standard and simple, there is a very high probability that it will be supported by a recent Linux distribution.

6.3 Actual Performance

The Internet Archive currently includes more than 2500 nodes and more than 6000 disks. Outgoing bandwidth is more than 6 Gb/sec. The internal network consists mostly of 100Mb/sec with a 1Gb/sec network connecting the front-end web servers.

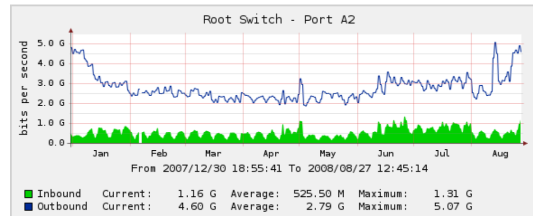

FIGURE 6.4: Network load from Jan 2008 through August 2008

Figure 6.4 presents the daily average network load as incoming and outgoing traffic for the entire archive. This switch is the main concentrator for the US operations. Peak loads are slightly over five gigabits per second with average loads around 2.8 gigabits per second. Data passing through this switch includes all access to the Internet Archive's main data center.

For the remainder of this section, we will focus specifically on non-webcrawl files. The Internet Archive logs accesses to URLs in the Wayback machine separately and we currently do not have access to those records.

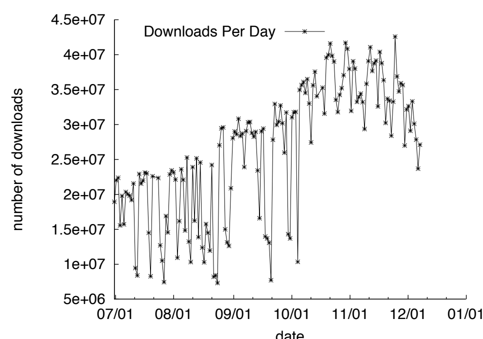

FIGURE 6.5: Downloads from July 2008 through early Dec 2008

As seen in Figures 6.5 and 6.6, during the instrumented period, the Internet Archive served between 2.3 and 48 terabytes per day. The daily download count ranged between 7.3 million and 42.5 million downloads per day.

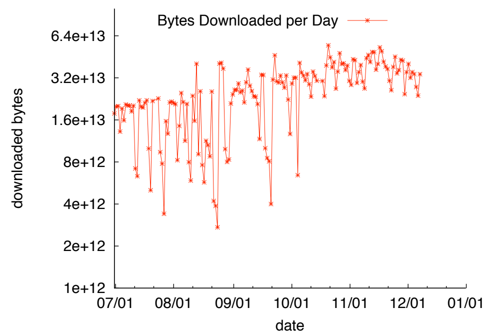

FIGURE 6.6: Bytes Downloaded from July 2008 through early Dec 2008

The local network supports 100Mb/sec and is used on a regular basis for the location broadcasts. Each request includes the name of the requested object, the IP address of the sending node, and a unique 32 bit ID for this request. The response includes the name of the responding node and the local path to the requested file. Empirically, the paths range from 10 to 300 bytes long, with 90% of requests having paths less than 160 bytes. There are on average approximately 800 requests per second with peaks around 1400 requests per second.

The Internet Archive uses a switched fabric for its networks. Thus UDP broadcasts will pass through all nodes, but responses will only travel between the requesting and responding nodes. The current load imposed by the locator mechanism is the number of requests per second times the UDP packet size. At current loads, this corresponds to less than 3% of the total capacity. The system starts to become loaded around 30,000 requests per second, or 48% of the available capacity. Upgrading to a 1Gb/sec network enables this algorithm to scale to more than 300,000 requests per second at 50% capacity.

6.4 Existing Placement Strategies

As explained in 6.2.4.2, the placement strategy for items in the Internet Archive consists of filling two identical machines with the incoming data stream. The incoming files are ARC files may be ARC files or Media files. ARC files each contain on the order of 100,000 web pages. These web pages were collected by a crawling agent over a relatively short period of time.

For Media files, the placement algorithm is equivalent to the Random placement algorithm proposed in this thesis. As files are depositing in the Media Archive, the majority of files will fall into obscurity, becoming part of the long tail. One of the problems with the existing architecture is that there is no cache, and so popular files will make the two nodes on which they reside very active.

The story for ARC files is different. First, while each node contains many ARC files, even recent ARC files are very infrequently accessed. Our review of the logs indicates that there is no correlation between the time of the ARC file and its access frequency. Thus, all data is cold from the moment it enters the system.

Secondly, it might be believed that there would be high correlation between files in a given ARC file. If this were the case, we would expect to have seen clumps of accesses to each ARC file. In practice, each ARC file was accessed once or twice within each 10 minute recording period. As each file is an web page and as each web page might include many images, the observed access pattern suggests that each file is being accessed on its own merits.

The ARC file placement is therefore also close to the proposed random placement algorithm.

6.5 Should ARC files be unpacked?

Internal discussions with Internet Archive personnel raised the question of whether it would be better to unpack the ARC files and store each page as a separate item. The reason given for unpacking each ARC file is that each page could then be managed according to its own access patterns.

The current approach requires that an ARC file be accessed for any of its component pages. This is particularly challenging because ARC files are not indexed. A request for a given page requires that the ARC file be read from the start until the item is found, potentially reading 100MB for a 1K web page. Unpacking each ARC file would then require only a direct read of the requested item.

The counter-argument is that managing billions of files is much harder than managing a few million ARC files. ARC files can be moved as a single unit without having to

worry about each component. If each page had to be managed directly, it would require massive overhead to make sure that copies and moves were completed for each and every object.

A more significant issues is that by design, ARC files should be cast in concrete and cannot be modified in order to provide an argument for their pedigree. Each ARC file represents a crawl on a given date. A checksum on that ARC file strongly suggests that no item in that ARC file was modified.

We can consider an alternative design, where each page is checksum'ed and that value is stored in one or more secure files that are never changed. ARC files can them be unpacked, but should they?

It is certainly the case that randomly placing archived web pages would result in a more uniform placement of files on the available disks. Given that experience has shown that there are few if any popular archived web pages, the difference between managing each file separately and managing a single ARC file seems to be minimal. That is, the probability of accessing an ARC file is equivalent to accessing each any of the files contained in that ARC file. Since a random placement algorithm could have placed all those items on the same disk, there is no practical difference in the access probabilities for that disk.

We can extend this argument to active files if there is a high performance cache within the system. Popular items will be accessed a small number of times from the disk where it was stored and all further requests will be served by the cache. Thus even popular items can be stored in the equivalent of ARC files.

This is the approach taken by Google. Google's page cache stores many files within a single GFS block. While GFS blocks are not ARC files, they are append-only and thus very similar to ARC files. The pages that Google has cached in GFS can then be served directly from the GFS block if they are infrequently accessed, while popular items can be served from a much smaller system level cache.

As a final note, the Internet Archive has moved to a new format called WARC[95], that includes an index reducing the need to scan the whole file for one of its components.

Chapter 7

Implications and Opportunities

7.1 Energy

The initial motivation for this thesis was that by identifying idle nodes, it would be possible to turn those nodes off and hence save on energy. Chapter 4 suggests that the entire contents of the archive system should be spread randomly across all available nodes. This has the benefit of spreading out the access, but directly contradicts the possibility of turning off idle nodes. The random placement policy guarantees that all nodes will be accessed at a more or less uniform level.

Yet, there remains a possibility of energy savings when one considers the methods for backing up these large data stores. As mentioned earlier, the only currently viable method for backing up a petabyte store is to copy the item to a second petabyte store. For more reliability, a third store is recommended. Thus, there are two or three copies of each item spread throughout the system. With planning, it is possible and even beneficial to turn off the backup volumes until and unless they are needed.

One approach to backup of large volumes when using a random allocation policy is to randomly allocate each item in each copy of the system. Thus, an item might be on disk 53 in the main system, disk 105 in the first backup copy and disk 6 in the second backup copy. In such a system, turning off disks is complicated because it is unclear which disks would need to be re-activated in case of a failure in the primary system. The storage system would have to maintain an accurate representation of the location of each and every item in all disks. Systems like the Google File System and HDFS indeed

maintain these mappings. When a disk fails, the controlling system must identify all the lost items and arrange for additional replicas to be created from a stable copy.

The Internet Archive uses a simpler approach, creating duplicates or triplicates of each disk. Since the entire disk is replicated, it is simple to turn off each backup copy and to re-active that copy in case of primary disk failure or increased load on the primary disk. The replica can double the available bandwidth and I/O operations for all items on the original disk. Similarly, if the primary disk fails, only one disk in the backup system must be reactivated and a single disk copy operation is required to recreate the primary system.

In either case, if the access volume is manageable with less than the full compliment of replicas, then one or more of those replicas can be de-activated. At that point, the system will save energy because the disk is not running and it will save on cooling because that disk is no longer generating heat.

One optimization to this approach is to regularly switch between active and backup disks. For example, a system might reanimate one copy of the backup disks and declare them to be the primary copy. The original primary could then be deactivated. In this way, each disk is used only one half or one third of the time, saving on MTBF and energy. By keeping all copies active at some level, the operators increase their confidence in the stability of each disk and its data.

7.2 Managing the head of the access distribution

What happens to the items that are frequently accessed? Where are they stored? If the set of frequently accessed items was static, then it would be reasonable to create a separate data store specifically for the active items. In reality, items rise and fall in their access patterns. For example, an old paper on simulated annealing might stay inactive for years. If the author were to win a prestigious prize, then that item might suddenly become very popular, at least for a few days or weeks.

A better approach is to store frequently accessed items along with the infrequently accessed items. Create backup copies of each item according to a uniform backup plan, with random copies or duplicate disks based on preference. In order for the system to

support high access loads, the frequently accessed items should be placed in a cache, where the huge majority of accesses will occur. The natural cache algorithms for both an LRU or LFU will remove items that become infrequently accessed and pull in new items that become active.

With this in mind, we turn to the challenges of creating a high performance cache for high access volume archival stores.

7.3 Caching

Many systems [87] implement a reverse proxy cache to improve performance and to reduce the load on its storage units. The Internet Archive has no such reverse-proxy cache. We will show that implementing and operating such a cache is not cost effective using currently available technology. However, newly available Solid State Disks (SSD) are shown to be a promising option for future implementations.

Many Internet systems include a reverse-proxy cache for static content. Such caches are used to reduce the load on dynamic web servers and to speed up access to content. These issues are not relevant to the Internet Archive because the system already offloads static content to the storage nodes. There is no indication that the existing disk speeds, or the internal or external network bandwidths are bottlenecks to content delivery. A reverse-proxy cache might significantly reduce the load on the storage nodes, potentially enabling these disks to be idled or even shut down for extended periods of time. If the cache is operationally cost effective, it could result in significant cost savings due to wear and tear on storage nodes and due to energy savings on these same nodes.

7.3.1 Empirical Requirements

To understand the empirical requirements for caching static files at the Internet Archive, we analyzed Media Collection W3C logs collected between July 1, 2008 and December 1, 2008. Each compressed log is between 1Gb and 3Gb with between 7.4 million and 42 million records per file. There are more than 50 million distinct objects referenced in these files. Each object reference is a URL path between 10 and 300 characters long.

For our detailed analysis, we used a specific seven day period from November 1, 2008 through November 7, 2008. During that time, the Internet Archive served 270 million requests and delivered 240 terabytes worth of data. Using consecutive dates balances any unusual activity and improves the simulated cache performance.

For each log file, we parse the log and extract the date, URL, http status and download size. We discard all records that have an invalid size, that represent failed downloads (not a 200 or 206 return value), that were not for the HTTP protocol, or that were not a standard HTTP request (GET, HEAD, or POST). These items should never get to the storage nodes and hence would not effect a cache. In any case, these requests represent less than 4.5% of the requests and no more than 0.00015% of the total download bandwidth.

We produce three data files for each log: a mapping of object IDs to file sizes, a list of object ID's in access order, and a file containing three fields: access time, object ID, and request size. The first two files are used by the stkdst [96] program to compute the priority depth analysis. The third file is used by the webtraff [97] package to compute standard web statistics.

Converting the W3C log files into our three data files was non-trivial. The problem was to maintain a hash table with each object's URL and its mapped ID. As each URL was referenced, we could look them up in the hash table. We found that this table would not fit into the 2Gb memory footprint of most Linux programs.

Our first solution was to implement a distributed hash using memcached [98]. We deployed memcached on four machines, allocating 2Gb of memory to each server. We turned off the LRU replacement feature of memcached, thus turning it into a static hash table. The resulting 8 Gb cache was large enough to process at least 10 days worth of logs. The process took a number of hours to complete and was very fragile.

We finally implemented the conversion algorithm in Hadoop[99]. The process required two map-reduce passes. The first pass parsed the log files and generated key-value pairs where the key was the URL and the value was a vector of the record time and download size. The reduce phase ran as a single reducer and assigned each unique URL key to a new ID. The mapping was written to a secondary file in the Hadoop file system. The second map-reduce phase inverted this process, mapping each record into a time key and

a vector value with the id and download size. Hadoop automatically sorts the output by key and so the reduce phase was simply the identity mapping.

The stkdst program implements an LRU stack algorithm [100], generalized to include the size of the requested objects. For each item in an access trace, the algorithm computes the size of the cache in terms of objects and bytes that would be necessary to have stored that item. By sorting the output on priority depth and then computing the cumulative distribution of number of items or item sizes as a function of priority depth, we respectively obtain the hit rate and the byte hit rate as a function of cache size.

The webtraff package is a collection of scripts to compute standard web trace analyses such as popularity, size distribution, bytes and requests per interval, and inter-arrival times.

Both stkdst and webtraff required some modifications. Neither system was designed for the large number of records and items referenced in our logs. For example, both systems limited file sizes to 2 gigabytes in size. We were able to address these problems by porting the code to use 64 bit long values for all relevant operations.

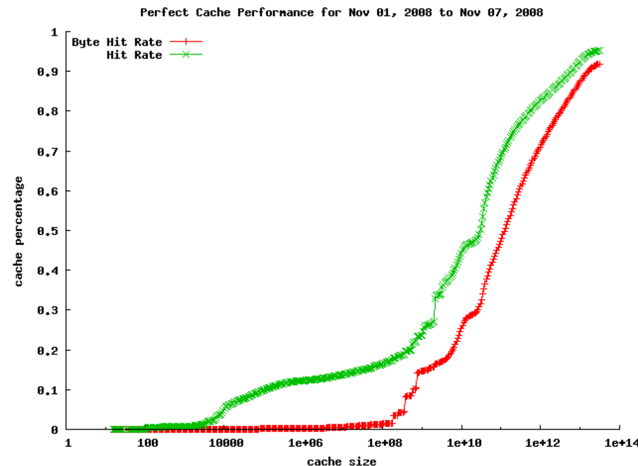

FIGURE 7.1: Cache Rates vs. Cache Size from Nov 1, 2008 to Nov 7, 2008

Using priority depth analysis as described in [101], we derived the cache hit rate as a function of the cache size in terms of the percentage of the files served and the percentage of bytes served. The results are shown in Figure 7.1. Table 7.1 provides a numeric summary of these charts.

Table 7.1: Coverage achieved as function of cache size.

cache size	% bytes	% hits
100 GB	48.09%	69.09%
200 GB	56.37%	74.79%
300 GB	61.09%	77.30%
400 GB	63.85%	78.63%
500 GB	65.78%	79.65%
600 GB	67.60%	80.73%
700 GB	68.77%	81.38%
800 GB	69.87%	81.96%
900 GB	70.97%	82.52%
1 TB	71.53%	82.79%
2 TB	76.79%	85.48%
3 TB	79.72%	87.17%
4 TB	81.61%	88.35%
5 TB	83.43%	89.58%
6 TB	84.32%	90.21%
7 TB	85.69%	91.21%
8 TB	86.14%	91.58%
9 TB	87.04%	92.32%
10 TB	87.93%	92.93%
15 TB	90.13%	94.35%
20 TB	91.20%	94.91%
25 TB	91.68%	95.13%
30 TB	91.81%	95.19%

7.3.2 Sizing the Cache

Using the byte hit rate and taking into account that each download may only be for some fraction of the total file size, we see that the maximum effective cache size is 30 TB. Such a cache would be able to serve 228 terabytes of data, accounting for 91.81% of the downloaded bytes. The other 4.91% of the hits and 8.19% of the downloaded bytes were accessed only once during this period and hence there is nothing to be gained by caching them.

The incremental improvements as the cache sizes grows begin to level off over for caches over 5 TB. A six-fold increase is cache size from 5 TB to 30 TB nets only a 8.38% increase in the number of cached bytes, which is 21 TB of delivered data. We will come back to the cache size once we understand the I/O operations per second and the bandwidth per second that our cache will need to support. Those values are related to the number of cache hits and the size of the requests, but not to the disk footprint of the cache.

7.3.3 I/Os per Second

As a simplification, let us assume that the cache hits are uniformly distributed during any given period. For the period in question, there were between 107 and 1259 requests per second with a average rate of 447 requests per second. Our cache will thus serve some percentage of that value as determined by the size of the cache.

There are two scaling issues with our proposed cache: bandwidth and requests per second. Bandwidth is an issue at many architectural levels. From the network perspective, a single 10Gb ethernet would be more than the existing external network and hence would be sufficient for existing network traffic. Adding multiple such interfaces or using more than one caching node would provide for expansion room. The other two areas of concern for bandwidth are the disk access bandwidth and the bus speed.

One of the major bottlenecks in current non-memory caches is the disk subsystems. We must consider the disk transfer rates as well as the number of I/O operations per second (IOPS) to the disk subsystem. The traditional approach to limitations in disk bandwidth or IOPS is found in database systems that utilize a large number of small disks. By spreading the data, the database can use all of the disks in parallel, thus multiplying the bandwidth and IOPS by the number of disks.

In order to see if IOPS are an issue for our cache, we will need to translate the number of hits to the cache into IOPS at the disk level. For the purposes of this exposition, let us assume a worst case scenario where the local memory on the caching server cannot cache all active files. That is, each file request will result in at least one disk request.

In most hardware caching scenarios, the block size is fixed [102]. This means that all cache hits return exactly the same amount of data. Our cache must return a variable amount of data because our file sizes are highly variable. Furthermore, our cache must support the current HTTP 1.1 protocol, which allows the requester to download any range of bytes from within the file. Almost half of all requests to the Internet Archive are for ranges of data. Thus, while we cache whole files, we must assume that each request will perform a random seek to the beginning of the requested file segment.

Modern operating systems attempt to pre-fetch parts of the file. The Linux operating system begins with its default block size of 4KB [103]. For each subsequent sequential read, it doubles the size of the prefetch buffer up to a maximum of 128KB. The prefetch

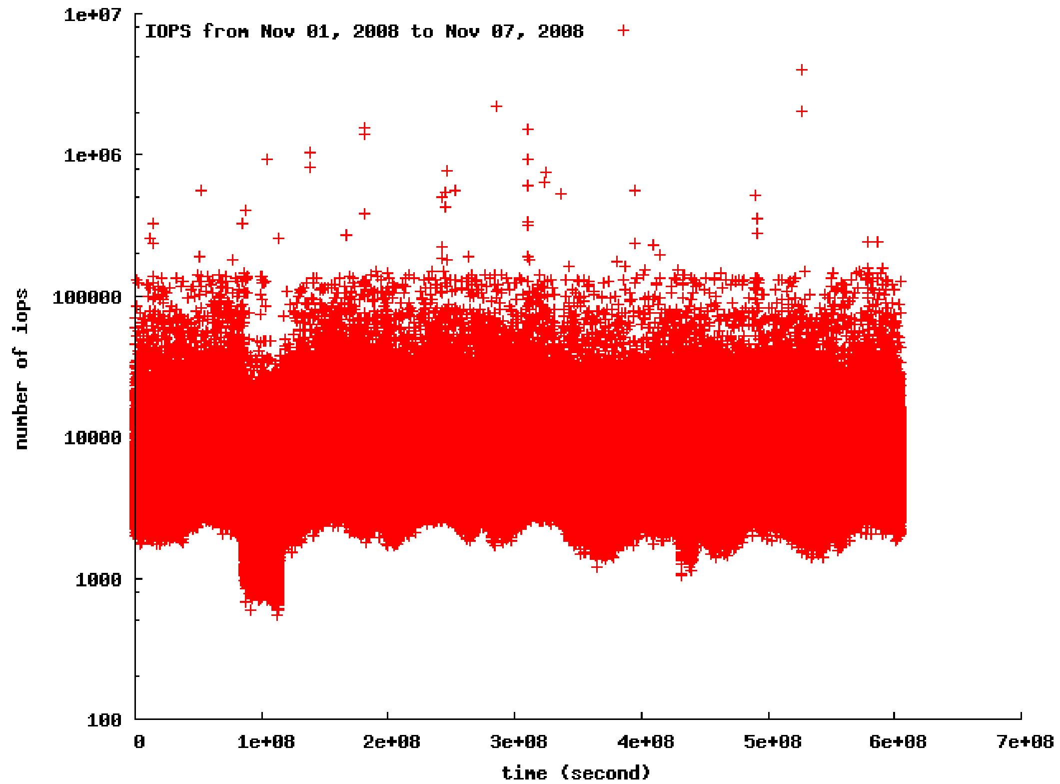

FIGURE 7.2: Estimated IOPS over time from Nov 1, 2008 to Nov 7, 2008

algorithm is intended to increase the throughput of the disk by performing sequential reads which do not require additional disk seeks.

Let us assume that the kernel maintains a prefetch buffer for each and every open file descriptor. That is, each open file is treated separately by the kernel and is prefetched on demand. This assumption may be true in practice given the large number of open files served by our cache.

We simulated the prefetch algorithm, counting the number of IOPS necessary per second based on the available trace data. Figure 7.2 shows the results on a log scale. The minimum value was 552 IOPS. The maximum value was 4021411 IOPS. The average value was 7734 per second. As can be seen, there were a significant number of seconds with more than 50000 IOPS.

It is critical to note that these values are only an approximation. The major challenge to these results is that we assume that all IO operations for each request occur immediately upon arrival of the request. In actuality, the file access is spread over the time necessary to deliver the data to the requester over the Internet. For large files, this can take hours and even days.

In support of our argument, we observe that there are no quiet times in this trace. Spreading the load over time will very likely reduce the very high IOPS rate, but it will also serve to increase the lower values. Furthermore, by spreading the load over time, we expect to see a reduction in kernel prefetch activity, once again increasing the number of IOPS.

Our calculation of IOPS also enable us to determine the required I/O bandwidth. For the period in question, the values ranged from a minimum of 0.007 Gb/sec to a maximum of 263 Gb/sec, with an average value of 0.41 Gb/sec per second. The bandwidth exceeded 1 Gb/sec approximately 5% of the time, and exceeded 2Gb/sec 0.65% of the time. Note that the total available outgoing bandwidth is less than 10 Gb/sec. Our reported value of 263 Gb/sec is a result of assuming that the entire file is downloaded at the moment of the request.

At this point, we note the now well publicized difference between traditional hard disks and solid state disks. In a traditional hard disk, data is stored on rotating platters and read by floating magnetic sensor heads. In order to perform a seek, the sensor must be moved to the correct track on this platter and the platter must complete its rotation to bring the data under the sensor head. For 5400 RPM disks, the average seek time is around 8 milliseconds. If each file access required one seek, the disk would be able to service only 125 operations per second, assuming that transfer time was negligible. In practice most commodity disks service about 100 operations per second with some enterprise disks offering up to 250 IOPS. These same enterprise disks offer sustainable transfer rates of 125MB/sec [104] (1 Gb/sec) over 3Gb/sec SATA interfaces.

Solid state disks have no moving parts and therefore are not subject to rotational seek latencies. Current solid state disks boast between 7000 and 35000 IOPS. Perhaps due to being early on the product curve, Solid states disks use the same 3Gb/sec SATA interfaces and support transfer rates very similar to traditional hard disks.

7.3.4 Implementation options

Let us now return to the determination of the cache size. As can be seen in Table 7.1, a one terabyte disk cache would satisfy 82.79% of the hits and 71.53% of the content. There are several ways that such a cache can be implemented.

Web caches are usually implemented using RAM. In-core memory would easily deliver the required IOPS, and also reduce latency relative to disks. Current operating systems limit the size of main memory to between 64 and 128 Gb. Building a 1 TB RAM cache would necessitate implementation as a distributed system, increasing the complexity of the implementation and introducing more points of failure.

An alternative would be to use the existing RAM of each node as a cache. In the Internet Archive, there are 2500 nodes and each node has 512MB of storage for a total of 1.28TB of RAM. Ignoring the fact that some portion of this RAM will be needed for the kernel and for running applications, we might consider using this memory as a in-place cache. Perhaps the biggest challenge would be that many files are larger than 512MB. To cache these files, we would need to split them up across nodes. While possible in principal, we believe that using existing RAM as a cache would violate the keep-it-simple approach in the Internet Archive and would likely reduce the performance of the existing system due to swapping and memory contention.

We might consider using enterprise-class hard disks with 250 IOPS. A single one terabyte hard disk would store all the data, but would never be able to supply the thousands of IOPS that we need. Even if we use ten 100 gigabyte disks, we would have only 2500 IOPS, which is also below our requirements. We could build a ten terabyte cache with one hundred 100 gigabyte disks. This would provide 25000 IOPS, which would support all but the the peak requirements. Unfortunately, using such a large number of disks becomes self defeating because we might as well just continue to use the original storage nodes.

It is in these circumstances that a solid state disk with 20000 IOPS would be a very good fit in terms of IOPS. Having two such disks would likely provide sufficient over capacity to handle even the peak times. There remains the question of disk bandwidth. Our simplistic calculations suggest that two solid state disks would have sufficient bandwidth for more than 99% of the sampled time periods. The proposed calculations that include the downstream bandwidth would likely lower the peak times into the supported range.

Chapter 8

Epilogue

8.1 Summary

This thesis covered the basics of computerized storage systems, beginning with the concepts developed over the past 50 years and reviewing the major advancements in the field during that time. The focus has been on how storage systems have scaled, and on how scaling has changed the focus from fast but simple systems that significantly under-utilized the existing hardware to modern systems that utilize not only the physical storage media, but also the computational power attached to that media and the network that connects the devices to their data consumers.

The primary focus has been not on high performance storage designed for databases, but on large archival systems that maintain data for extended periods of time. We have explored three such systems, each having a different corpus of content and each displaying a similar long-tailed distribution.

With the understanding that long tailed distributions can characterize the access patterns of archival systems, we then explored how these distributions impact the architecture of very large storage systems. It immediately became clear that large archival systems are two or three times the size of their basic content, because the only way to backup these systems is by duplicated that data. Once there is a duplicate, there is no reason not to make it available online. After all, the same hardware and equipment is used in both the primary and backup system.

With these very large storage systems in mind, we discussed the appropriate methods for placing files on the storage system's component disks. We discussed two placement policies, one which put all the high access items on a limited number of disks, and left the other disks idle. The second placement policy randomly spread all files across the disks with no concern as to the actual access probability of each item.

Chapter 4 developed the regions of access probabilities for the items in the system into segments that might benefit from optimized architectures. The chapter concluded with a unified architecture that could provide all things to each region of access frequency, from the very frequently accessed items to the cold items.

The Internet Archive was presented as an example of a uniform architecture. That system utilized a uniform placement algorithm similar to the random placement strategy. The Internet Archive serves as an empirical example of how the random placement policy works in practice and is a testimony to the policy's efficacy. Extensions to the architecture were suggested that could improve performance and extend the life of the storage units.

Chapter 7 addressed the issue of energy savings and the challenge of handling highly popular files with a uniform architecture. It showed that current spinning hard disks are unsuitable for high performance, large scale caches. Solid state disks were offered as an viable alternative due to their supporting orders of magnitude more operations per second than traditional hard disks.

8.2 Future Research

The model derived in this thesis serves to focus architectural efforts for large scale archival storage systems. The segmented approach to file access frequencies enables architects to search for designs that offer better efficiency and/or performance. More importantly, the field of large scale archival storage is just beginning to become a practical problem. Google adds one petabyte each week to their storage systems and that data is being used by YouTube and Gmail, two systems that have definite archive aspects. The same is true of Microsoft Live and Yahoo Mail.

Looking further afield, many national security agencies capture and store digital recording of email and telephone communications for analysis and investigation. These systems will naturally expand and the opportunity to do longitudinal analysis over months and years will become more practical.

Storage technologies continue to advance, but the speed of light is a constant. Moving data from one location to another is and will remain expensive. At least one copy of every piece of archival data needs to be moved to some remote location for disaster recovery. But our work on placement strategies implies that instead of storing a complete copy at some remote location, it will be better to spread the data across different sites. Not only does this improve disaster recovery, but we have also shown that the access patterns to these remote items will remain balanced. Google, Yahoo and Microsoft have or will encounter these issues in the coming years.

8.2.1 Databases

Jim Gray was one of the main motivators for database systems and database architectures. He lived in a world where there was never enough memory or disk bandwidth. There was insufficient memory to store the database in RAM, and hence many random accesses were necessary to read data from disks. Since the accesses were random, the disks operated in their most inefficient mode and hence could serve only a few hundred requests per second at best.

Just as this thesis has discussed archival file systems, so to there is an analogous challenge for archival database systems. What happens when a single database spans hundreds of disks and still needs a backup system? Why can't the backup system be put online and become part of the database's resource pool. The benefits of this system would be to increase the number of available disk operations and to spread the load across more disks, since each data item would reside in multiple locations.

The drawbacks of such a system would be significantly greater write costs, because databases are very careful about data integrity. Such a system would also require sophisticated dynamic query optimizers that take into account not only the location of each item, but also the current activity level within the system. Having multiple copies

enables the supervisor to select the least loaded disk, the least loaded network and the least loaded processor for each request.

Our research group has begun investigating these issues with a slightly easier problem, distributing Map/Reduce operations across data centers. We have begun to develop an infrastructure to capture near-realtime empirical performance data from running tasks. It is important to know the effective bandwidth between two nodes, as opposed to the potential bandwidth at the network interface card. This work will be supported by LAWA, a European Community FP7 STREP in cooperation with the European Internet Archive, The Max Planck Institute and others in Europe.

Bibliography

[1] D. A. Thompson and J. S. Best. The future of magnetic data storage technology. *IBM J. Res. Dev.*, 44(3):311–322, 2000. ISSN 0018-8646. doi: http://dx.doi.org/10.1147/rd.443.0311.

[2] Wikipedia. Hard drive capacity, March 2008. URL http://commons.wikimedia.org/wiki/File:Hard_drive_capacity_over_time.png.

[3] Phil Dowd. Pc security, May 2010. URL http://phildowd.com/?p=85.

[4] Yellow Bricks. Iops, December 2009. URL http://www.yellow-bricks.com/2009/12/23/iops/.

[5] Elliot Jaffe. MULTIPLE IDENTITY ATTACKS ON DISTRIBUTED SYSTEMS. Master's thesis, The Hebrew University of Jerusalem, School of Computer Science and Engineering, 2005.

[6] Library of Congress. About the library, May 2010. URL http://www.loc.gov/about/generalinfo.html.

[7] Oliver Johnson. Text, bytes and videotape, May 2010. URL http://plus.maths.org/issue23/features/data/index2.html.

[8] Thom Hickey. Entire library of congress, May 2010. URL http://outgoing.typepad.com/outgoing/2005/06/entire_library_.html.

[9] Jeff Dean. Designs, lessions and advice from building large distributed systems. In *Proceedings of the ACM SIGOPS LADIS*, 2009. URL http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf.

[10] Devin Coldewey. Nsa to store yottabytes of surveillance data in utah megarepository, April 2010. URL http://www.crunchgear.com/2009/11/01/nsa-to-store-yottabytes-of-surveillance-data-in-utah-megarepository/.

[11] James Hamilton. Designing and deploying internet-scale services, January 2008. URL http://mvdirona.com/jrh/talksAndPapers/JamesRH_AmazonDev.pdf.

[12] Silicon Valley Leadership Group. Data center energy forecast. Web, July 2008. URL https://microsite.accenture.com/svlgreport/Documents/pdf/SVLG_Report.pdf.

[13] Michael Garey and David Johnson. *Computers and Intractability*, page 226. W.H. Freeman, 1979.

[14] Vikay Vazirani. *Approximation Algorithms*. Springer Verlag New York, LLC, 2007. URL http://search.barnesandnoble.com/books/product.aspx?r=1&ISBN=9783540653677&r=1.

[15] Seagate Technology. Momentus 5400.6 sata product manual, 2010. URL http://www.seagate.com/staticfiles/support/disc/manuals/notebook/momentus/5400.6%20(Wyatt)/100528359e.pdf.

[16] Seagate Technology. Barracuda xt series sata product manual, 2010. URL http://www.seagate.com/staticfiles/support/disc/manuals/desktop/Barracuda%20XT/100586689c.pdf.

[17] Seagate Technology. Cheetah 15k.7 data sheet, 2010. URL http://www.seagate.com/docs/pdf/datasheet/disc/ds_cheetah_15k_7.pdf.

[18] RH Katz, GA Gibson, and DA Patterson. Disk system architectures for high performance computing. *Proceedings of the IEEE*, 77(12):1842–1858, 1989.

[19] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Wang. Modeling hard-disk power consumption. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 217–230, Berkeley, CA, USA, 2003. USENIX Association.

[20] StorageReview.com. Drive performance resource center: Benchmark database, May 2010. URL http://www.storagereview.com/Testbed4Compare.sr.

[21] K. Li, R. Kumpf, P. Horton, and T. Anderson. A quantitative analysis of disk drive power management in portable computers. In *Proceedings of the 1994 Winter USENIX Conference*, pages 279–291, 1994.

[22] P.J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, 2005.

[23] D. Pogue. Google takes on your desktop. *New York Times*, 2004.

[24] E. Cutrell and S.T. Dumais. Exploring personal information. *Communications of the ACM*, 49(4):51, 2006.

[25] c't magazine. h2benchw benchmarking software, May 2010. URL ftp://ftp.heise.de/pub/ct/ctsi/h2benchw.zip.

[26] Tom's Hardware. 3.5 hard drive charts - average read transfer performance, May 2010. URL http://www.tomshardware.com/charts/3.5-hard-drive-charts/Average-Read-Transfer-Performance,658.html.

[27] M. W. Young, Dean S. Thompson, and E. Jaffe. A modular architecture for distributed transaction processing. In *USENIX Winter*, pages 357–363, 1991.

[28] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):53, 1993.

[29] D.A. Patterson, G. Gibson, and R.H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116. ACM, 1988.

[30] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: high-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, 1994. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/176979.176981.

[31] Thomas Schwarz, Mary Baker, Steven Bassi, Bruce Baumgart, Wayne Flagg, Catherine van Ingen, Kobus Joste, Mark Manasse, and Mehul Shah. Disk failure investigations at the internet archive. In *MSST2006: 23rd IEEE, 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, May 2006.

[32] MV Wilkes. A programmer's utility filing system. *The Computer Journal*, 7(3): 180, 1964.

[33] Maurice V. Wilkes. Edsac 2. *IEEE Ann. Hist. Comput.*, 14(4): 49–56, 1992. ISSN 1058-6180. doi: http://dx.doi.org/10.1109/85.194055. URL http://portal.acm.org/ft_gateway.cfm?id=612476&type=external&coll=GUIDE&dl=GUIDE&CFID=91438394&CFTOKEN=88599456.

[34] E.I. Organick. *The Multics system: an examination of its structure*. MIT Press, Cambridge, MA, USA, 1972. ISBN 0-262-15012-3.

[35] D. M. Ritchie, D. M. Ritchie, and K." Thompson. The unix time-sharing system. *COMMUNICATIONS OF THE ACM*, 17:365–375, 1974. doi: 10.1.1.100.7314.

[36] Val Henson. A brief history of unix file systems, March 2005. URL http://www.lugod.org/presentations/filesystems.pdf.

[37] R. Duncan. Design goals and implementation of the new high performance file system. *MICROSOFT SYST. J.*, 4(5):1–14, 1989.

[38] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[39] M. Rosenblum and J.K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1): 26–52, 1992.

[40] Linus Torvalds. What would you like to see most in minix?, 1994. URL http://groups.google.com/group/comp.os.minix/msg/b813d52cbc5a044b?dmode=source&pli=1.

[41] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux*, pages 90–367, 1994.

[42] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, 1994.

[43] M. Holton and R. Das. Xfs: A next generation journalled 64-bit filesystem with guaranteed rate i. Technical report, SGI Corp., 1994. URL http://www.sgi.com/Technology/xfs-whitepaper.html.

[44] R. Strobl and O.S. Evangelist. Zfs: Revolution in file systems. *Sun Tech Days*, 2009:2008, 2008.

[45] T. Marill and D. Stern. The datacomputer: A network data utility. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 389–395. ACM, 1975.

[46] Alfred Zalmon Spector. *Multiprocessing architectures for local computer networks*. PhD thesis, Stanford University, Stanford, CA, USA, 1981.

[47] Alfred Z. Spector and Peter M. Schwarz. Transactions: a construct for reliable distributed computing. *SIGOPS Oper. Syst. Rev.*, 17(2):18–35, 1983. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1041478.1041481. URL http://portal.acm.org/ft_gateway.cfm?id=1041481&type=pdf&coll=GUIDE&dl=GUIDE&CFID=90317559&CFTOKEN=87126334.

[48] Edward Balkovich, Steven Lerman, and Richard P. Parmelee. Computing in higher education: the athena experience. *Commun. ACM*, 28 (11):1214–1224, 1985. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/4547.4553. URL http://portal.acm.org/ft_gateway.cfm?id=4553&type=pdf&coll=GUIDE&dl=GUIDE&CFID=90317559&CFTOKEN=87126334.

[49] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. Rosenthal, and F. Donelson Smith. Andrew: a distributed personal computing environment. *Commun. ACM*, 29(3): 184–201, 1986. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/5666.5671. URL http://portal.acm.org/ft_gateway.cfm?id=5671&type=pdf&coll=GUIDE&dl=GUIDE&CFID=90317559&CFTOKEN=87126334.

[50] Alex Osadzinski. The network file system (nfs). *Comput. Stand. Interfaces*, 8 (1):45–48, 1988. ISSN 0920-5489. doi: http://dx.doi.org/10.1016/0920-5489(88)90076-1.

[51] G. Baker. Talking appletalk: Unstacking the apple lan protocol stack. *LOCAL AREA NETWORK MAG.*, pages 82–87, 1988.

[52] I. Novell. *NetWare system interface technical overview*. Addison Wesley Publishing Company, 1990.

[53] Microsoft Corporation. Microsoft networks smb file sharing protocol (document version 6.0p). Technical report, Microsoft Corporation, January 1996.

[54] J.D. Blair. *Samba: Integrating UNIX and Windows*. Specialized Systems Consultants, 1998.

[55] TJ Berners-Lee. The world-wide web. *Computer Networks and ISDN Systems*, 25 (4-5):454–459, 1992.

[56] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol–http/1.0, 1996.

[57] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. Http extensions for distributed authoring–webdav. *Microsoft, UC Irvine, Netscape, Novell. Internet Proposed Standard Request for Comments (RFC)*, 2518, 1999.

[58] M. Users. SharePoint Fundamentals. *Microsoft SharePoint*, pages 65–101, 2007.

[59] The Internet Archive Foundation. The internet archive, 2010. http://www.archive.org.

[60] E. Jaffe and S. Kirkpatrick. Architecture of the internet archive. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–10. ACM, 2009.

[61] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, December 2003. ISSN 0163-5980. URL http://portal.acm.org/citation.cfm?id=1165389.945450.

[62] S. Ghemawat and J. Dean. Mapreduce: Simplified data processing on large clusters. *Usenix SDI*, 2004.

[63] Mike Burner and Brewster Khale. WWW Archive File Format Specification, 2002. http://web.archive.org/web/20021002080721/pages.alexa.com/company/arcformat.html.

[64] A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki*

at http://lucene. apache. org/hadoop, 2005. URL http://lucene.apache.org/hadoop.

[65] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 93–109. ACM, 1999.

[66] Drew Roselli and Thomas E. Anderson. A comparison of file system workloads. In *In Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54. USENIX Association, 2000.

[67] Wikipedia. Storagetek tape formats. Wikipedia, 2010. URL http://en.wikipedia.org/wiki/StorageTek_tape_formats.

[68] Stavros Christodoulakis, Peter Triantafillou, and Fenia A. Zioga. Principles of optimally placing data in tertiary storage libraries. In *In Proceedings of the 23rd VLDB Conference*, pages 236–245. Morgan Kaufmann, 1997.

[69] C. K. Wong. *Algorithmic Studies in Mass Storage Systems*. W. H. Freeman & Co., New York, NY, USA, 1983. ISBN 0716781417.

[70] J. Li and S. Prabhakar. Data placement for tertiary storage. In *NASA CONFERENCE PUBLICATION*, pages 193–208. Citeseer, 2002.

[71] Athena Vakali and Evimaria Terzi. Video data storage policies: an access frequency based approach. *Computers & Electrical Engineering*, 28(6): 447 – 464, 2002. ISSN 0045-7906. doi: DOI:10.1016/S0045-7906(00)00068-9. URL http://www.sciencedirect.com/science/article/B6V25-4691K6D-2/2/2222294713c36e4a0589c668d47fd5e2.

[72] C. Anderson. The Long Tail, Wired. *October*, 12:2004, 2004.

[73] S. Asmussen. *Applied probability and queues*. Springer Verlag, 2003.

[74] W.H. DuMouchel. Stable distributions in statistical inference: 1. Symmetric stable distributions compared to other symmetric long-tailed distributions. *Journal of the American Statistical Association*, 68(342):469–477, 1973.

[75] C. Anderson. *The long tail: Why the future of business is selling less of more*. Hyperion Books, 2008. ISBN 1401309666.

[76] JA Urquhart and NC Urquhart. *Relegation and stock control in libraries*. Taylor & Francis, 1976.

[77] S.J. Bensman. Urquhart's Law. *Science & technology libraries*, 26(2):33–69, 2005.

[78] J. Nolan. Stable distributions: Models for Heavy Tailed Data. 2009. URL http://academic2.american.edu/~jpnolan/stable/chap1.pdf.

[79] Slashdot effect. Wikipedia, August 2010. URL http://en.wikipedia.org/wiki/Slashdot_effect.

[80] SourceForge. Terms of use, May 2010. URL http://sourceforge.net/apps/trac/sitelegal/wiki/Terms_of_Use.

[81] Open Source Initiative. The open source definition, May 2010. URL http://opensource.org/docs/osd.

[82] M. Van Antwerp and G. Madey. Advances in the sourceforge research data archive (srda). In *Fourth International Conference on Open Source Systems, IFIP 2.13 (WoPDaSD 2008)*, Milan, Italy, September 2008.

[83] SourceForge Research Data Archive (SRDA): A Repository of FLOSS Research Data. Papers - open source research, May 2010. URL http://zerlot.cse.nd.edu/mediawiki/index.php?title=Papers.

[84] A. Clauset, C.R. Shalizi, and M.E.J. Newman. Power-law distributions in empirical data. *SIAM review*, 51(4):661–703, 2009.

[85] Dror G. Feitelson. Metrics for mass-count disparity. In *MASCOTS '06: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 61–68, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2573-3. doi: http://dx.doi.org/10.1109/MASCOTS.2006.30.

[86] Creative Commons. Creative commons cco - no rights reserved. web, 2010. URL http://creativecommons.org/about/cc0.

[87] Brian D. Davison. A survey of proxy cache evaluation techniques. In *WCW99: Proceedings of the Fourth International Web Caching Workshop*, pages 67–77, 1999.

[88] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Softw.*, 12(6): 42–50, 1995. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/52.469759.

[89] Wei Hou and O.G. Okogbaa. Reliability and availability cost design tradeoffs for HA systems. *Reliability and Maintainability Symposium, 2005. Proceedings. Annual*, pages 433–438, 24-27, 2005. ISSN 0149-144X. doi: 10.1109/RAMS.2005.1408401.

[90] Carnegie Mellon University Libraries. Frequently Asked Questions About the Million Book Project, 2008. http://www.library.cmu.edu/Libraries/MBP_FAQ.html.

[91] Rick Prelinger. www.prelinger.com, 2008. http://www.panix.com/~footage/.

[92] Carson Gaspar. Deploying Nagios in a Large Enterprise Environment. In *LISA*. USENIX, 2007.

[93] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[94] Bibliotheca Alexandria, 2009. http://www.bibalex.org.

[95] ISO 28500:2009. Information and documentation - warc file format. Technical report, ISO, Geneva, Switzerland, 2009. URL http://www.digitalpreservation.gov/formats/fdd/fdd000236.shtml.

[96] T. Kelly. Priority depth (generalized stack distance) implementation in ANSI C, 2000. http://ai.eecs.umich.edu/œtpkelly/papers/.

[97] N. Markatchev and C. Williamson. WebTraff: A GUI for web proxy cache workload modeling and analysis. In *MASCOTS '02: Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, page 356, Washington, DC, USA, 2002. IEEE Computer Society.

[98] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, 2004. ISSN 1075-3583.

[99] Apache Software Foundation. Hadoop Core, 2008. http://hadoop.apache.org/core/.

[100] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78, 1970.

[101] T. Kelly and D. Reeves. Optimal web cache sizing: scalable methods for exact solutions. *Computer Communications*, 24(2):163 – 173, 2001. ISSN 0140-3664. doi: DOI:10.1016/S0140-3664(00)00311-X. URL http://www.sciencedirect.com/science/article/B6TYP-423RH1W-6/2/7e5b18c36b771889708741e5337cb614.

[102] Tien-Fu Chen and J.-L. Baer. A performance study of software and hardware data prefetching schemes. *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*, pages 223–232, Apr 1994. doi: 10.1109/ISCA.1994.288147.

[103] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. *SIGMETRICS Perform. Eval. Rev.*, 33(1):157–168, 2005. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/1071690.1064231.

[104] Seagate Technology. Seagate technology - cheetah hard drive family, 2009. http://www.seagate.com/www/en-us/products/servers/cheetah/.

Acknowledgements

The acknowledgements and the people to thank go here, don't forget to include your project advisor...

Remember to thank Scott, Dror, Bruce, Jim Gray, Brewster, the IA, etc.