A NEW APPROACH TO DATA BACKUP

A Thesis presented to the Faculty of the Graduate School

University of Missouri–Columbia

In Partial Fulfillment
Of the Requirements for the Degree

Master of Science

---

by

JAY HUTCHINSON

MAY 2010

The undersigned, appointed by the Dean of the Graduate School, have examined the thesis entitled

A NEW APPROACH TO DATA BACKUP

Presented by Jay Hutchinson

A candidate for the degree of Master of Science

And hereby certify that in their opinion it is worthy of acceptance.

Professor Gordon Springer

Professor Youssef Saab

Professor William Banks

I dedicated this work to my wife Corrie and my father John. Without their support this thesis would not have been possible.

ACKNOWLEDGEMENTS

Many colleagues, friends, and family have contributed in one way or another to the completion of this thesis. To all of these I am most grateful and give my thanks. Dr. Gordon Springer, Dr. Youssef Saab, and Dr. William Banks took time out of their busy schedules to serve on my thesis committee. Special thanks goes to my thesis adviser Dr. Gordon Springer for reading and commenting on countless drafts of this thesis. Scott Hussey also reviewed an early version of this thesis and offered much insight. Joshua Fraser and I spent innumerable hours over coffee discussing computer science and while perhaps having no direct bearing on this work have been exceedingly valuable to me as a computer scientist and as a friend. This work would not have been possible without the use of free software written by a great many programmers. I want to thank my family who have been incredibly supportive while I finished my thesis. I also thank God whose love for mankind is a thing of wonder.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . . **ii**

LIST OF FIGURES . . . . **v**

LIST OF TABLES . . . . **vi**

1 Introduction **1**

2 Background **3**

2.1 The Backup Problem 3

2.2 Logical vs. Physical Backup 4

2.3 Consistency 5

2.4 Traditional Backup Techniques 7

2.5 Deduplication 8

2.5.1 Whole File Chunking 10

2.5.2 Fixed Length Chunking 10

2.5.3 Content Based Chunking 11

3 Design **15**

3.1 Overview 15

3.2 Anchor Function 19

3.3 Blocks 26

3.4 Archive Interface 30
3.5 Data Integrity 32

4 Implementation 35
4.1 Prototype 35
4.2 Deduplication 37
4.3 Performance 41

5 Conclusion 46

A Block Specification 49
A.1 Root Block 50
A.2 Data Block 50
A.3 Pointer Block 51
A.4 File Block 51
A.5 Directory Block 52

Bibliography 53

LIST OF FIGURES

Figure		Page
2.1	Example of a file broken into three chunks.	12
3.1	Geometric distribution	23
3.2	Jac's anchor function (random data)	23
3.3	Jac's anchor function (Windows data set)	25
3.4	Rabin fingerprinting (Windows data set)	25
3.5	File block and pointer tree.	28
3.6	Directory block and pointer tree.	29

LIST OF TABLES

Table Page

4.1 User directory test results 38

4.2 Windows test results 38

4.3 Linux test results 39

4.4 Bioinformatics test results 40

Chapter 1

Introduction

The threat of data loss has been a concern since data was first recorded and no less so in the digital era. Digital information has advantages over previous analog forms of data storage in that it can be copied much easier. This permits backups of the data to be performed. Redundant information can be recorded separately and later used to restore the original should it be damaged or lost. While the concept of a backup is simple to define, many users still lack a simple and effective method of backing up their data.

This project was motivated by the simple snapshot schemes available on systems such as Plan-9 [29, 31, 33], WAFL [11], and AFS [12]. These systems take a consistent copy of the file system at regular intervals and makes them available to the user as a read only copy. This archival feature allows users to browse earlier copies of their data. These systems all require the use of a special file system.

This paper describes the design and prototype implementation of a command line computer program called Jac. Jac is a backup utility. Development of Jac was motivated by the desire to have a simple command line backup tool that provided the same simple and useful archival features of the snapshot schemes described above. That is, Jac provides a backup tool that lets the user make full copies of the data

to be backed up at regular intervals. In order to do this in a space efficient manner Jac makes use of modern deduplication technology [21, 20]. Deduplication technology takes advantage of the fact that backup data often contains large amounts of redundant data. By removing this redundant data the total amount of storage space can be greatly reduced.

This paper is organized as follows. Chapter 2 gives some background on the backup problem in general along with a discussion of deduplication technology specifically. Chapter 3 discusses the overall design of Jac. Chapter 4 discusses the prototype implementation of Jac along with test results concerning its space efficiency. Chapter 5 summarizes the project.

Chapter 2

Background

2.1 The Backup Problem

Backups are done to prevent the loss of data. Data loss can occur as a result of a disaster, hardware failure, software errors, or user errors. Disasters include a flood or fire at the site destroying the media the data is stored on. A common hardware failure is disk drive failure. Software errors might include a software bug introduced by a software update that erroneously modifies or deletes data before the bug is noticed. User errors include the accidental modification or deletion of important data.

A backup utility takes as input the data to be backed up and produces redundant data as output. This redundant data can be used to reproduce the original input data. This redundant data may simply be a copy of the original data or it may be something more complicated. Since the purpose of creating this redundant data is to restore the original if it should ever be lost, the redundant data is usually stored on a different storage medium and in some cases even at another physical location. In this way, whatever causes the loss of the original data is less likely to also cause the loss of the redundant data. In most cases the data being backed up is open to modification after the backup is performed. The redundant data needs to be updated periodically

so that it is never too far out of date with respect to the original.

Since the redundant data is being updated periodically, it is necessary that the data can be restored from any of several past backup points, not merely the most recent. Imagine that a software or human error deletes a file. This may go unnoticed for some time. If only the most recent backup point is available it might not include the needed file. If however a copy of the data is available from past backup points, a user could simply go back to a previous backup point that occurred before the deletion and restore the file.

2.2 Logical vs. Physical Backup

Backups can be performed in two ways, at the logical level or at the physical level. Logical backups process data at the file system level. An example of this is the Unix Tar utility [10]. Given a directory to backup, the Tar utility descends the file system hierarchy reading directories and files. It serializes them into a formatted data stream that can be stored to disk or tape. The advantage of logical backups is that by using their own storage format they can be made largely independent of the file system and can thus backup and restore to a wide variety of platforms.

The disadvantage is that their independent format may limit their ability to retain certain file system specific meta-data such as extended attributes, access control lists, resource forks, and other data. Another disadvantage is performance. The data on a physical disk drive is laid out in physical blocks of a particular size. File systems store files as a sequence of logical blocks and map these logical blocks to physical blocks on the disk [37]. Since the backup program operates at the file system level it can only access the data as logical blocks. These blocks may not be laid out as sequential physical blocks on the device, resulting in increased read/write head movement to access the data.

Physical backups process data at the device level. As an example, a simple physical backup scheme might copy all of the physical blocks from one physical device to another device with the same physical block size and sufficient capacity. This scheme illustrates two advantages of physical backup. First, it need not understand the contents of the blocks it backs up at all. This means that it will backup the entire file system, all of the data and meta-data will be preserved, regardless of what file system is being used. Second, because the backup operates at the device level it can read the blocks sequentially allowing high performance due to reduced seeking.

The disadvantages of physical backup are also illustrated. Because the backup operates at a lower level and does not understand the file system, the *entire* file system must be backed up and restored as a single unit. It is not possible, without understanding something about the file system, to restore a sub-set of the backup data, such as a single file or directory. Nor can the data be restored to a different file system. Of course the scheme given has a more severe disadvantage. Because it does not know anything about the blocks it backs up it may well be wasting time backing up a large number of empty blocks. Some physical backup systems do have limited understanding of the file system, sufficient to determine which blocks need to be backed up [13, 11].

2.3 Consistency

A concern when performing a backup is that the backup data is a consistent view of the original data. That is, the backup should represent the entire file system hierarchy being backed up as it existed at some instant in time. This is problematic if the file system is in use during the backup procedure. File system modifications during the backup can lead to a backup that does not represent that file system hierarchy consistently. For example, a directory movement during the backup may

cause that entire directory to be absent from the backup depending on the timing [36]. To see this, imagine the following scenario. Let three directories A, B, and C exist in a hypothetical file system. Let C be a sub-directory of directory B. As the backup proceeds it reaches directory A and backs it up. Before reaching directory B sub-directory C is moved to directory A. Later, when the backup reaches directory B, C is no longer there and does not get backed up at all. File systems usually implement move modifications as atomic operations. So directory C always existed in the file system either in directory A or B. Since C appears nowhere in the backup, the backup is not consistent with the file system.

One way to ensure consistency is to limit modifications to the file system during the backup, such as entering single user mode on the computer and running the backup as the lone process or remounting the file system read only for the duration of the backup. In many cases however these options are impractical because the file system in question needs to remain on-line for its users.

The solution to this is to use snapshot technology. A snapshot is a read-only copy of the file system at a single point in time. Many file systems and volume managers now offer the ability to snapshot [16, 11, 17, 25]. Snapshots are performed efficiently using copy-on-write techniques. Rather than copying the entire file system to create the snapshot copy, which would be impractical, a copy-on-write system maintains references from the snapshot copy to the original file system blocks. When a block is written, a copy of that block is made for the snapshot to retain. All unmodified blocks are shared by the snapshot and the live file system. Thus, a snapshot can be created quickly and consumes space roughly equal to the number of blocks modified since its creation. Once a snapshot is made a backup utility can simply perform the backup from the consistent read-only snapshot.

2.4 Traditional Backup Techniques

An example of a traditional backup program is the previously mentioned Tar utility. Tar is a logical backup utility that takes all or part of a file system hierarchy and serializes it into a data stream. Tar can preserve common UNIX file system metadata such as ownership, permissions, file modification time, and other data. In order to reduce the amount of storage space needed to store the backup data the backup stream is often compressed. Data compression tools such as Gzip [5, 9] or Bzip2 [35] are common examples. These compression algorithms can reduce the storage requirements significantly depending on the data set. The resulting data stream can then be written to disk or to tape.

Historically tape drives with removable magnetic tape cartridges have been the media of choice for backups. The cost per megabyte of tape storage has been lower for magnetic tapes than for disk drives. However, this gap is closing. While tapes have relatively high transfer rates for sequential read/write, they perform poorly with random access due to the need to fast forward and rewind the tape in order to seek to a particular location [37]. This necessitates that the data to be backed up be processed into a stream for writing. One downside of the removable tape is that tape management is a hassle and error prone. Manually loading and unloading tapes is time consuming, not to mention that backup tapes can become mislabeled or lost. Robotic tape libraries allow access to a large number of tapes without human intervention. Coupled with backup tape library management, they greatly ease the use of removable magnetic tapes.

The simplest backup scheme would be to simply make a copy of the data to be backed up at each backup point. This scheme is very expensive unless the data to be backed up is very small in size or the number of past backup points to be retained is small. This method uses a lot of storage as the number of copies increases. A normal/incremental backup scheme is one traditional method to alleviate this prob-

lem. First, a normal backup is performed to create a full copy of the data set. Then a series of incremental backups are performed. Each incremental backup consists of just the files that have changed since the previous backup point. One way to identify which files have been modified is to use the modification time available on most file systems. If the modification time for a file is after the last backup point, then the file needs to be updated in the incremental backup.

By storing only the files that have changed, the size of the incremental backups is significantly reduced compared to the normal backup in most cases. This results from the fact that many of the files remain unchanged from one backup point to the next. The disadvantage of such a scheme is that the incremental backups are not self sufficient. To restore the data the last normal backup plus every incremental backup is needed. These may be spread around and require access to many separate portions of the backup file sequence.

2.5 Deduplication

Backup data often contains large amounts of identical data. This occurs in two cases. The first case is temporal redundancy. This results from the fact that given a data set, much of the data from one backup point to the next is the same. Many files are not modified at all and those that are are often modified in small ways. The second case is internal redundancy. This results from redundant data between two different data sets, either on the same machine or on different machines. An example of the first might be an email server. The server keeps messages for its users until delivery. A company wide email could generate an identical message in every user's mailbox on the server. An example of redundancy on different machines can be seen in the following scenario. Assume that the hard drive contents of several different workstation computers are to be backed up to the same central location. If the

computers have the same operating system and many of the same applications, much of the data on these computers can be the same. Bolosky et al. [2] did research indicating that nearly half of the space used by a group of desktop file systems could be reclaimed if duplicate files were eliminated. Deduplication technology attempts to take advantage of the identical data often present in backup data. By identifying these identical portions and storing them just once in the underlying storage space the total amount of storage space needed can often be reduced.

Different data deduplication techniques exist, but most are based on some form of *chunking* and *fingerprinting* [21]. Chunking and fingerprinting work as follows. Each file to be backed up is first divided into a sequence of chunks. Each chunk is a contiguous sequence of bytes from the file. The original file may be reproduced by concatenating the sequence of chunks end to end. Each chunk is then given a unique identifier based upon its contents called a *fingerprint*. The chunk is stored to the backup media and the fingerprint is stored into an index. Every time a chunk is stored the index is first consulted for the new chunk's fingerprint to determine whether or not an identical chunk has already been stored. If it has, storage of the second duplicate chunk is suppressed.

Fingerprints are generated using a cryptographic hash function [22]. SHA1 [28] is a common choice. The entire chunk is used as input and the output of the cryptographic hash function is the fingerprint. An obvious problem with using a hash function in this way is that each fingerprint needs to represent a unique chunk. In theory a cryptographic hash function could violate this condition and produce identical fingerprints given two different chunks. When a cryptographic hash function produces the same output for two different inputs like this it is called a *collision*. In practice the probability of a collision is insignificant, making fingerprints produced by cryptographic hash functions practical unique chunk identifiers [26, 32, 1].

2.5.1 Whole File Chunking

Several different chunking methods exist. The simplest chunking method is to treat each file as a single chunk. This is called *whole file chunking*. The reader should notice the similarities between whole file chunking and the normal/incremental backup scheme already mentioned. Both take advantage of the fact that files often do not change from one backup point to the next. However, whole file chunking has significant advantages. In the normal/incremental scheme, if a file is moved, renamed, or has it's modification time changed without really being changed; the entire file will be stored again. This is not the case for whole file chunking, which will identify the file as a duplicate based upon its fingerprint. Furthermore, if an identical file exists in several places on the same machine or on different machines, because of internal redundancy, only whole file chunking will deduplicate them.

The problem with whole file chunking is that it is unable to take advantage of identical portions of data at the sub-file level. For example, even minor changes to a large file results in the entire file being updated in the backup at the next backup point. The backup then contains two copies of the file which contain a large portion of identical data, thus wasting storage space. An example of this is virtual machine disk images. A single file is used to represent a virtual machine's disk and can be many gigabytes in size. When the machine is run modifications are made to the disk image. These changes may be as simple as updating a small log file within the disk image. Most of the data in the image file is identical to the original. But, the file is now changed, and will be saved to the backup again.

2.5.2 Fixed Length Chunking

One solution to the problem stated above is *fixed length chunking*. With fixed length chunking, each file is broken into a sequence of chunks where each chunk has the same fixed length. In this way a modification to bytes at one location in the file does

not necessarily change the other chunks. Since each chunk is fingerprinted separately only the modified chunks need to be stored at the next backup point. For instance, a log file that is only appended to may have only the last chunk modified.

The problem with fixed length chunking is that modifications can shift the file contents throwing off the alignment of the chunks. To see this consider an example where a large file has a single byte inserted at the beginning of the file. This shifts the original file contents by one byte location. At the next backup point when the file is reprocessed the alignment of the new chunks will be shifted one byte as well with respect to the old chunks. No matches between the original set of chunks and the new set of chunks would occur except by coincidence.

2.5.3 Content Based Chunking

The problem with fixed length chunking results from the fact that chunk boundaries are determined by position. *Content based chunking*, or variable length chunking, divides the files into chunks based upon their content. This general method was used by Manber to identify similar documents in large file systems [19]. It was first applied to chunking by Mazières et al. in the low bandwidth file system [26]. Below is described the basic content based chunking algorithm.

Each file is processed by sliding a 48 byte window across the data one byte at a time from the beginning of the file to the end. Other window sizes can be used. The 48 byte window size is common and seems to have been popularized by Mazières et al.'s use of it [26]. At every window position, a function called the anchor function is computed over the data in the window. If the resulting value matches a predetermined constant, then the data within the window is considered an *anchor*. The point between the last byte of an anchor and the byte immediately following it in the file is considered a *breakpoint*. These breakpoints divide the file into chunks. The beginning and end of a file are always breakpoints. Figure 2.1 shows a file that

has been broken into three chunks. Notice that two anchors have been found in the file creating two breakpoints in addition to the two automatic ones appearing at the beginning and end of the file. These four breakpoints divide the file into three chunks.

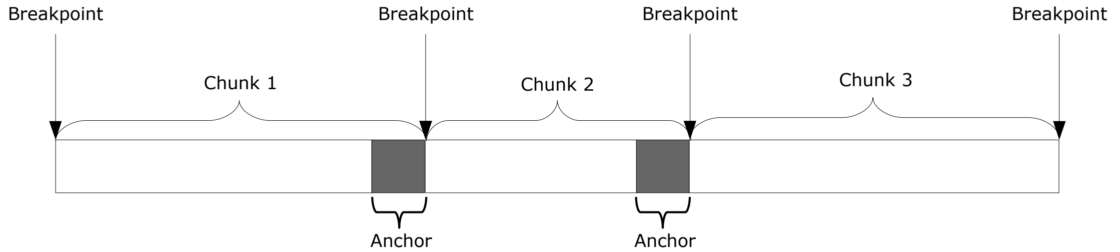


Figure 2.1: Example of a file broken into three chunks.

A technique called Rabin fingerprinting [34] is used as the anchor function. Rabin fingerprints treat the input data as a polynomial in $\mathbb{Z}_2[x]$. That is, as a polynomial with binary coefficients. This polynomial is then divided by a predetermined irreducible polynomial in $\mathbb{Z}_2[x]$. The lower thirteen bits of the resulting remainder are used as the output of the anchor function. Since the 48 byte window is slid the length of the data file and only a constant amount of work is done by the anchor function at each step, the algorithm runs in linear time with respect to the size of the input file.

Using just the lower thirteen bits of the Rabin fingerprint narrows the range of the anchor function to 2^{13} or 8192 possible values. This affects the number of anchors that are set which affects the average size of the chunks. Assuming that the anchor function maps its inputs to its outputs close to uniformly then the function defines a set of anchors equal to roughly 1/8192 of the total possible inputs. Assuming that the input file is random then an anchor will be set roughly once out of every 8192 window positions. This gives an expected distance between anchors of 8192 and therefore an average chunk size of 8192 bytes or 8 KB.

The choice of the average chunk size is a trade off. Smaller average chunk sizes are

more likely to lead to duplicate matches. They also produce more chunks, increasing the size of the index and the time it takes to search the index for each new chunk added. Previous work done by Mazières et al. [26], Zhu et al. [39], and Meister and Brinkmann [21] suggests that 8 KB is a good compromise.

The 8 KB chunk size assumes a random input file. The input data is of course not likely to be random. In Section 3.2 it will be shown that the function performs well in practice even with real data. It is possible however that given the right data the algorithm will produce only a few large chunks or an enormous number of small chunks. For example, if a large file is processed that contains no anchors the function will output one huge chunk the size of the file. A file might also contain an anchor at every byte offset. This would produce as many one byte chunks as the file is long. Neither case is desirable. Large chunks are difficult to work with in memory and limit the possibility of deduplication. Large numbers of small chunks on the other hand fill the index up unnecessarily. To guard against these two possibilities a minimum chunk size of 256 bytes and a maximum chunk size of 64 KB is enforced. If a breakpoint occurs and the chunk is less than 256 bytes the breakpoint is suppressed forcing the chunk to be larger. When a chunk reaches 64 KB a breakpoint is artificially inserted, ensuring that no chunk is any larger.

The use of Rabin fingerprinting as an anchor function is not to be confused with the fingerprints discussed earlier that are produced by cryptographic hash functions and used to identify whole chunks. In fact, calling this usage Rabin fingerprinting at all is somewhat of a misnomer. The irreducible polynomial used can be chosen arbitrarily ahead of time and is fixed for all runs of the algorithm. What Rabin originally proposed was the use of a randomized algorithm, where the irreducible polynomial was chosen at random for each run of the algorithm. The use of the term Rabin fingerprint has stuck however in the context of deduplication.

Content based chunking works such that if two files (or even two non-overlapping

parts of the same file) contain a relatively long identical sequence of bytes, then that sequence will, with high probability, yield many of the same chunks for both files. To illustrate why this is, consider a sequence of bytes that starts with and ends with an anchor, but contains no other anchors. When this sequence is processed it is divided into a chunk consisting of everything after the first anchor. An example of this can be seen in the middle of Figure 2.1. Chunk 2 has an anchor immediately before it and ends with an anchor. Notice that this occurs regardless of where the sequence is found. If the same sequence occurs in several different files the algorithm always divides it into the same chunk. The algorithm does not depend upon the sequences appearing in the same location in both files nor in the files being related in any way. These identical chunks can then be deduplicated by matching fingerprints.

Experimental results confirm that content based chunking performs the best in terms of reducing storage requirements over a wide variety of data sets compared to other chunking methods [21, 20, 30]. In one experiment the effects of different chunking methods was studied on 450 GB of users' directory data. It was found that content based chunking detected close to 35% of the data as duplicate on average [21]. In the same study, content based chunking was run on weekly full backups of the data set to test the amount of temporal redundancy. First, one backup was processed. Then the next week's backup was processed. The amount of non-duplicate data from one backup point to the next with content based chunking was only 1.26% on average. This indicates that content based deduplication is particularly adept at reducing redundancy among a series of periodic backups.

Chapter 3

Design

3.1 Overview

Development of Jac was motivated by the desire for a simple to use backup tool with modern deduplication technology. Originally it was designed with whole PC backup of Linux systems in mind, but its application is more general than that. Jac is a command line program. This gives it a simple interface and one that is similar to existing tools such as Tar. Jac performs logical backups. While the high speed of physical backups is enticing, it limits flexibility. With a physical backup Jac would only be able to backup and restore full file systems and would be unable to cross file systems between backups and restores. Jac relies upon tools provided by the file system or volume manager to perform snapshots and then backs up those snapshots to ensure consistency.

In the past, the cost of storage space made it unrealistic to retain all past backup points. Jac was designed with the philosophy that today's low cost storage coupled with deduplication technology changes this assumption. The policy of retaining all past backup points has been used in the Plan 9 file system as part of its snapshot backups [31, 33]. The advantage of such a policy, beyond users having access to all

past backup points, is simplicity. No method is needed to decide when and which backup points to remove, preventing removal of backup data a user may at some time need. For most user data this is a practical policy. The cost of storage space has continually fallen allowing each user to have an excess of capacity. Because user data does not change that much and deduplication technology allows removal of temporal redundancy, each backup past the first consumes little additional storage space. This is not to suggest that all user data is to be backed up and retained indefinitely. Some files may simply be too large to backup at all. Some data is derivative or temporary by nature. The user may of course decide to exclude such data from being backed up.

Jac stores all of the data to be backed up by dividing it into blocks that are inserted into an *archive*. All archives share a common interface, allowing Jac to support many different archive implementations. The simplest type of archive uses a single formatted file to store the blocks and is described in Chapter 4. An archive may contain more than one entry. Each entry represents a directory structure that a user wants to backup. One example might be a user backing up his user directory every week. Each backup point is a separate entry in the archive. Multiple entries are also the result of different data sets being backed up to the same archive. For example, the hard drive contents of several different machines may be backed up to the same archive. All of the data in an archive is deduplicated against all of the other data in the archive.

Archives use fingerprinting with a cryptographic hash function to detect duplicate blocks. For each block in the archive, its fingerprint is stored in an index maintained by the archive. When a block being added has the same fingerprint as a block that already exists within the archive, no new space in the archive is allocated for the block. Rather, the archive returns the fingerprint for the already existing block. In this way duplicate blocks are detected and never stored twice.

An archive's index is a mapping of fingerprints to blocks. Not only can it be determined if a fingerprint is already in an index, but given a fingerprint the associated block can be retrieved from the archive. In this way fingerprints act as unique *keys*. A block's fingerprint and its key are the same thing and the two terms are used interchangeably throughout the rest of the paper.

For each file to be backed up, Jac breaks the file into a series of chunks using content based chunking. Each chunk is a contiguous sequence of bytes from the original file. The original file may be reproduced by concatenating the sequence of chunks end to end. A header is placed on each chunk creating a *data block*. Each data block is then stored into the archive. Files that have much of the same data in them will have many identical byte sequences. These will yield many of the same chunks and thus many of the same data blocks. These duplicates are detected when they are inserted into the archive and do not consume any extra space.

Of course, more than data blocks need to be stored within an archive. Some method is needed to organize the data blocks within an archive into the correct order so that an entire file may later be restored. There needs to be a way to record the contents of directories so that directory structures can be restored. There also needs to be a way to store the data associated with files and directories such as file and directory names, permissions, ownership information, et cetera. All of this information is encoded into blocks too and are inserted into the archive. The different types of blocks, including data blocks, are described in Section 3.3. The archive does not distinguish between block types and thus duplication among any type of block is eliminated.

Storing the associated data, or meta-data, of a file system poses a challenge for any backup utility because different file systems support different meta-data. For example, some file systems have access control lists for fine grained configuration of authorization settings. Other file systems do not support this at all. These differences

create a problem of translation. If a backup is performed on data from a file system of type A and the data is restored to a file system of type B then some translation of the meta-data will have to take place because file system A and B may support different types of meta-data. The challenge of translation is two fold. First, some translation may not be possible. For example, file system A may support access control lists while file system B may not. Second, even if some type of translation is possible, the number of translations the software must support is $O(n^2)$ where n is the number of file systems supported. Therefore, supporting more than just a few file systems becomes a significant software challenge.

On the other hand, the backup utility can not simply throw away all of the meta-data. Much of it is quite valuable and needs to be restored. The solution used by Jac is to support a file system independent format for recording meta-data. Jac uses the same format as the Tar utility. This format is able to record traditional Unix attributes such as file and directory names, permissions, ownership, modification times, et cetera. This does not of course handle all of the possible meta-data that every file system may support, but it records much of the important meta-data. Future versions of Jac may support more extensive meta-data.

Section 3.2 describes the content based chunking algorithm used in Jac to divide files into chunks. Section 3.3 describes the different types of blocks that are stored in an archive and how they are used to represent the data being backed up. Section 3.4 describes the common interface used by archives and how they work. Section 3.5 describes how the design of Jac provides a built in way to check the integrity and authenticity of the stored data.

3.2 Anchor Function

Jac uses content based chunking, like that described in Section 2.5.3, to break an input file into a set of byte sequences of varying sizes called chunks. Most content based chunking algorithms use Rabin fingerprints as the anchor function. It has proved effective in practice and is used by a wide variety of systems, but any hash function that can be updated quickly can be used [14, 4, 15]. Jac uses a different anchor function, one that is similar to the Fletcher [7] and Adler-32 [6] checksum functions. It was chosen because it is simple and fast to compute. The function is described below and then compared to Rabin fingerprinting with various tests.

The chunking algorithm must compute the anchor function at every byte offset. To be practical, this function must be fast to compute. Since the window slides over the data, a function that can be updated quickly is desired. Specifically, given the contents of the window, the associated function's value, and the next byte in the file; it should be possible to compute the next value without reprocessing the entire contents of the window at the next position.

Given a sequence of bytes X, of length n, the function is defined as follows:

$$f(X) = \sum_{i=1}^{n} i \; X_{n-i+1}$$

This is equivalent to b_n below with the recursive form:

$$a_n = \begin{cases} 0 & \text{for } n < 1 \\ a_{n-1} + X_n & \text{otherwise} \end{cases}$$

$$b_n = \begin{cases} 0 & \text{for } n < 1 \\ b_{n-1} + a_n & \text{otherwise} \end{cases}$$

This can be seen by expanding the terms:

$$a_n = X_1 + X_2 + \cdots + X_n$$

$$\begin{aligned} b_n &= a_1 + a_2 + \cdots + a_n \\ &= (X_1) + (X_1 + X_2) + \cdots + (X_1 + X_2 + \cdots + X_n) \\ &= nX_1 + (n-1)X_2 + \cdots + 2X_{n-1} + X_n \end{aligned}$$

This expansion can be used to derive formulas for updating a and b as the window is moved across the data. Let the window size be n. Let X_k be the left most byte at the old window position and X_{k+n} be the right most byte at the new window position. Then, new values for a and b can be computed by:

$$a' = a - X_k + X_{k+n} \quad (3.1)$$

$$b' = b - nX_k + a' \quad (3.2)$$

Instead of the 48 byte window size used in many Rabin fingerprint based system, Jac uses a 64 byte window. With a window size of $n = 64$ bytes, a power of two, the multiplication in Equation 3.2 can be performed with a left shift. The anchor function can therefore be updated with two integer additions, two integer subtractions, and one left shift; which is quite fast. Like other implementations, only the lower thirteen bits of the function's output are used, resulting in an 8 KB average chunk size.

Two sets of tests were performed to compare the merits of Jac's anchor function with that of Rabin fingerprinting as an anchor function. The first test was designed to test the speed of the two functions. The speed test consisted of running each function over a 1 GB file of random data mapped into memory. Both functions were coded in C and stripped down to their essentials to run as fast as possible. No work was performed by the code except that a counter was incremented upon every anchor match. This was done to prevent the compiler from optimizing away the entire loop. The Rabin fingerprint code was based upon code from the low bandwidth file system project [26].

As a baseline comparison, code was written that simply looped through the 1 GB of data adding up 64 bit blocks at a time in a counter. This test was performed simply to get an idea of how long it took to touch 1 GB of data on the test machine. All of the code was compiled with GCC with the `-O3` option. For each function the test was run three separate times to ensure that system load was not effecting the run times. The three separate runs always ran within 1 second of each other. The tests showed that the baseline test took 1 second to run, Jac's anchor function took 5 seconds to run, and the Rabin fingerprint function took 13 seconds to run.

Jac's anchor function runs more than twice as fast as the Rabin fingerprint function. Being faster is worthless though if it is a poor anchor function. For example, if the anchor function places too few anchors the chunk sizes will be huge. In this case the maximum chunk size will reduce the chunks to 65 KB by inserting artificial breakpoints so that chunks do not become arbitrarily large, but if this happens all the time then the algorithm is essentially reduced to fixed length chunking. Of course both Rabin fingerprinting and Jac's anchor function are just hash functions. For each of them there is some input that will cause such undesirable behavior, but a good hash function will work well in most cases.

The second set of tests was done to determine how well each function performed at breaking the data into chunks. The functions are compared to each other and to an ideal model. As stated earlier in Section 2.5.3, the expected chunk size of 8 KB is based on the assumption that the anchor function maps its large domain to its relatively small range in a uniform manner and that the input file is random. The assumptions about the functions with random input data gives an ideal model for comparison. At each window position the probability that it will be an anchor is 1/8192. Each window position can therefore be thought of as a Bernoulli trial where the probability of success at each trial is 1/8192. Each chunk can then be considered a sequence of trials leading up to a success. Therefore the size of the chunks follow

the geometric distribution with an expected value of 8192.

First, both functions were run on 1 GB of random data. With random data, the functions should give results very close to the geometric distribution unless they are internally biased in some way. Both anchor functions gave results very close to the ideal. Jac produced an average chunk size of 8193 (or 0.01% larger than the expected 8192). Rabin fingerprinting produced an average chunk size of 8226 (0.42%).

A probability distribution was created for the Jac test run and for the ideal geometric distribution. The results are plotted in Figures 3.1 and 3.2. Each vertical bar represents the number of chunks that occurred within that range of sizes. The width of each bar on the x-axis is 128. So, the first bar represents all of the chunk with sizes in the interval [0,127]. The height of each bar on the y-axis represents the number of chunks as a fraction of the total number of chunks.

Figure 3.1 shows the geometric distribution. The data was simply computed based on the formula for the geometric distribution. Figure 3.2 shows the distribution of chunk sizes produced by Jac given the 1 GB of random data as input. Comparing the probability distributions shown in the figures allow us to compare the entire distribution to the ideal geometric distribution, not merely the average chunk size. The two figures confirm the assumption that Jac's anchor function closely follows the geometric distribution given random input data.

Next the functions were compared to the geometric distribution and to each other using data gathered from machines. Two data sets were used. The first is 3 GB of data composed of files from the author's user directory. These files include text files, PDF's, source code, images, etc. Jac produced an average chunk size of 8314 (1.49% larger than the expected 8192) and Rabin fingerprinting produced an average chunk size of 8288 (1.17% larger). The 3 GB of data was created by concatenating the collection of data files into a single large file for each function to process. This was done because in practice files are sometimes smaller than the expected 8 KB chunk

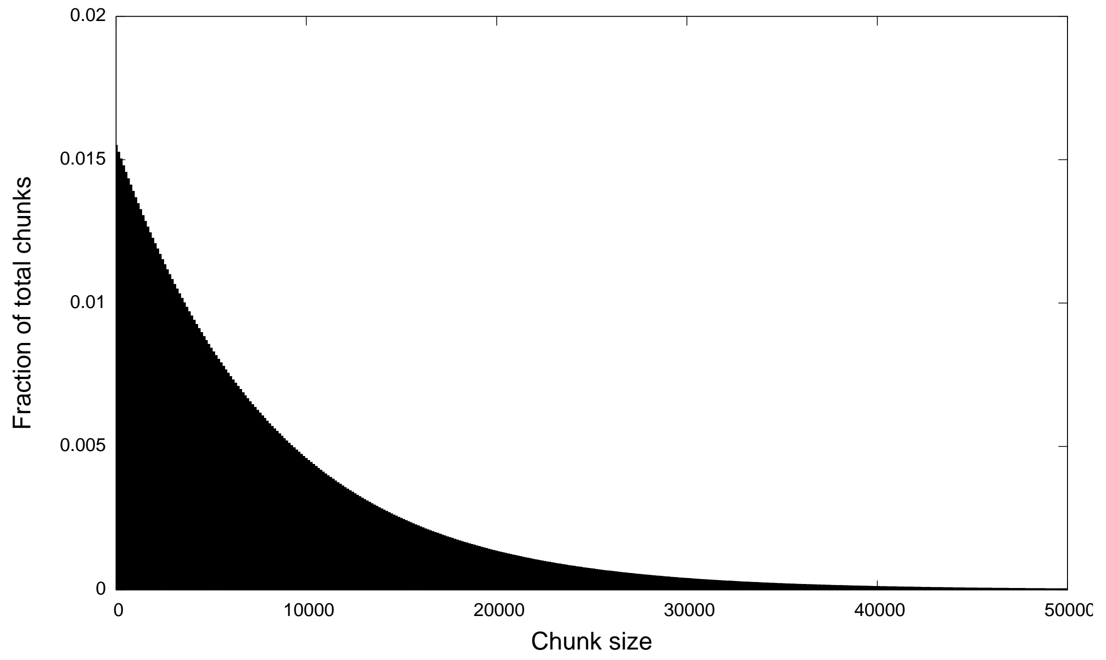


Figure 3.1: Geometric distribution

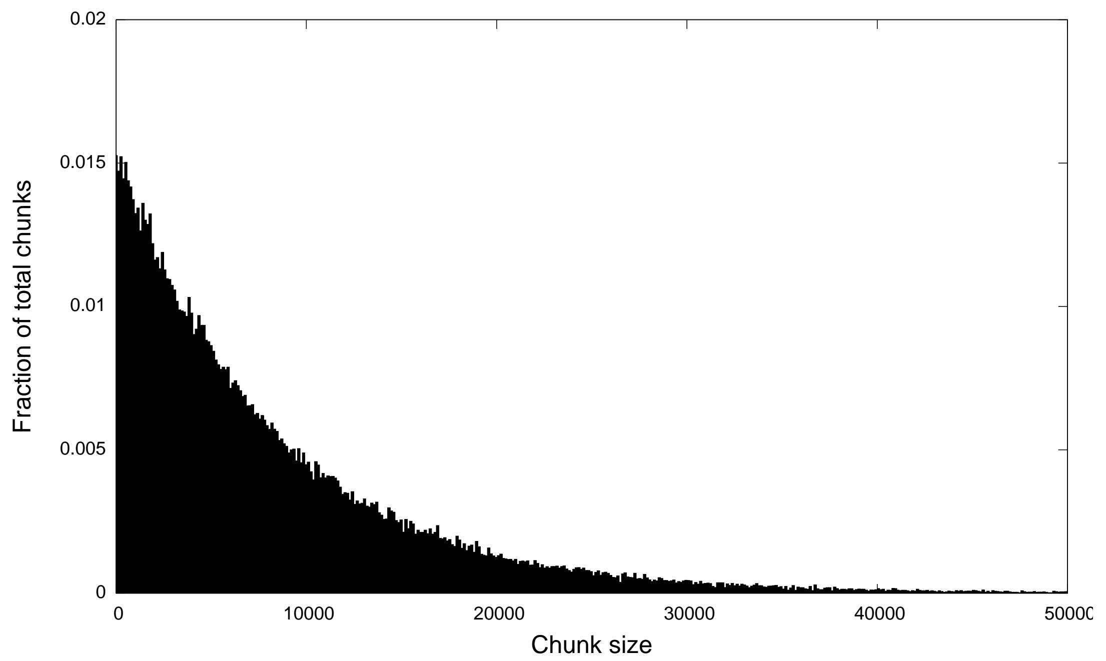


Figure 3.2: Jac's anchor function (random data)

size. Since the beginning and end of every file is a breakpoint by definition, small files have the effect of adding breakpoints causing chunks to be smaller than they would be on a single large file. Normally this is fine but for this test we want to be able to compare the anchor functions to the geometric distribution. By concatenating all of the files together the skewing effect of small files is eliminated.

The second data set is a 6 GB Windows OS backup file. The file is the result of a complete backup of a Windows XP desktop machine. This includes the operating system, system configuration files, application binaries, user data, and all other ancillary data found on a machine. The user data in this second test is different than in the first. Jac produced an average chunk size of 8784 (7.23% larger than the expected 8192) and Rabin fingerprinting produced an average chunk size of 7239 (11.63% smaller). Figure 3.3 shows the distribution of chunks resulting from Jac on the Windows data set. Figure 3.4 shows the chunk distribution for Rabin fingerprinting. Note that in Figure 3.4 the first two bars extend much higher, but have been cut off at 0.02 to make comparison with the other graphs easier. Comparing Figure 3.3 and 3.4 shows that the two different anchor functions behave similarly. Also, comparing the two figures to Figure 3.1, shows that both function behave surprisingly close to the geometric distribution.

To summarize, testing shows that the simple anchor function used in Jac is more than twice as fast as the Rabin fingerprinting function. The chunk size output of the two function both follow the geometric distribution very closely given random input data. With data sets collected from machines both functions followed the geometric distribution equally well. Jac's anchor function is therefore comparable with the Rabin fingerprint method already widely used in practice with the advantage of being faster.

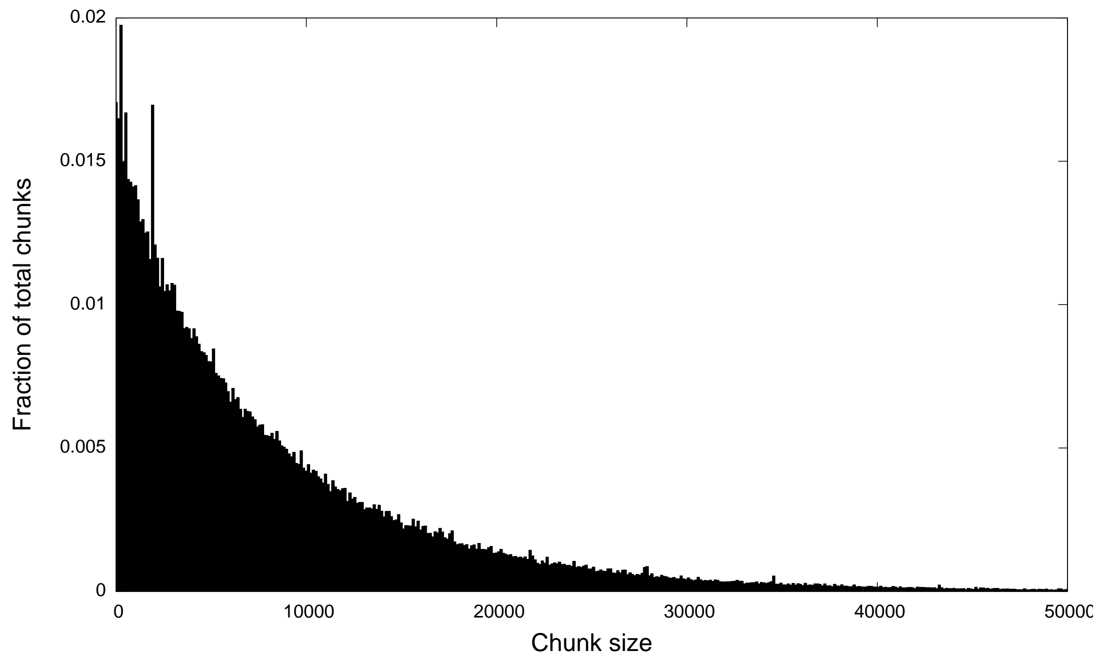


Figure 3.3: Jac's anchor function (Windows data set)

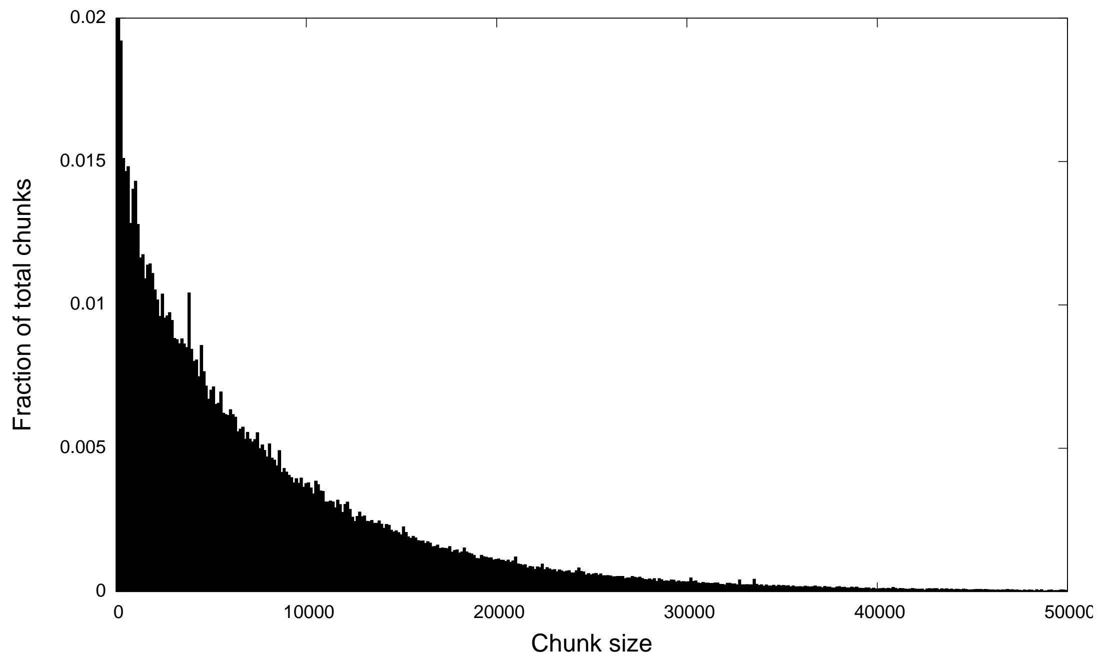


Figure 3.4: Rabin fingerprinting (Windows data set)

3.3 Blocks

Jac stores all of the backup data by dividing it into a set of blocks that are then inserted into an archive. There are five different block types: root blocks, directory blocks, file blocks, data blocks, and pointer blocks. A root block represents a single entry stored in the archive. A directory block and a file block represent a directory and a file stored in the archive respectively. A data block represents a single chunk of data from a file. Pointer blocks are used by file and directory blocks to organize the data they represent. A low level description of each block type's layout suitable for writing software to encode/decode them can be found in Appendix A.

One block can point to another block by containing the unique key of the block it points to. For instance, let a block A contain the key to block B. Given block A the key for block B can be extracted. The key can then be used to retrieve block B from the archive because archives map keys to blocks. In this case block A points to block B. In this context block A is called the parent and block B is called the child. Archives store the blocks and maintain a mapping of keys to blocks. Other than that the archive does not organize the blocks in any way. The ability of blocks to point to each other allows them to be organized into more complex tree structures as will be seen below. For instance, these structures help organize separate data blocks into a single file.

The simplest type of block is the data block. Jac's chunk algorithm divides files into variable length chunks of data. Each data block contains one chunk of data.

Pointer blocks point to other blocks by containing a list of their keys. The number of keys a pointer block contains is variable. Pointer blocks can point to any type of block. By pointing to other pointer blocks it is possible to create pointer trees where the internal blocks of the tree are pointer blocks and the leaves are some other type of block. These pointer trees can allow one block to organize and point to an arbitrarily large number of other blocks by simply pointing to the root of the pointer

tree. The number of keys that the pointer blocks contain affects the size of the tree. If a pointer block is allowed to contain an arbitrarily large number of keys then the tree will contain a single pointer block at the root that points to all of the leaf blocks directly. On the other hand if each pointer block only points to a few other blocks, the tree will contain many pointer blocks as internal nodes and have longer paths from the root to the leaves. Both extremes are undesirable. Large pointer blocks are difficult to work with in memory. Lots of small pointer blocks are undesirable as well since many intermediary pointer blocks have to be accessed from the archive in order to reach the desired leaf block. The current implementation restricts the number of keys a pointer block contains to a maximum of 200. This keeps the number of internal pointer blocks small compared to the number of leaf blocks in the tree. It also ensures that the largest possible pointer block is just under 4 KB in size.

An example of a block pointing to a pointer tree is the file block. Each file block represents a file stored in the archive. The file block contains the meta-data associated with the file. Currently this is limited to Unix style attributes such as the file's name, permissions, ownership, et cetera. The file's data is stored separately in data blocks containing the relevant chunks of data. The file block points to the root of a pointer tree which in turn points to all of the file's associated data blocks. An example of this can be seen in Figure 3.5. Note that in the figure, for the purpose of illustration, pointer blocks only point to at most three other blocks. Size restrictions do not make it possible to illustrate each block pointing to 200 other blocks. The concept is never the less the same. A left to right walk of the tree gives the data blocks at the leaves of the tree in the order that they should be concatenated in to rebuild the file.

Files can optionally be compressed before insertion into the archive. When this is done the chunk data in each data block associated with the file is compressed with a compression scheme before being inserted in the archive. The file block contains a

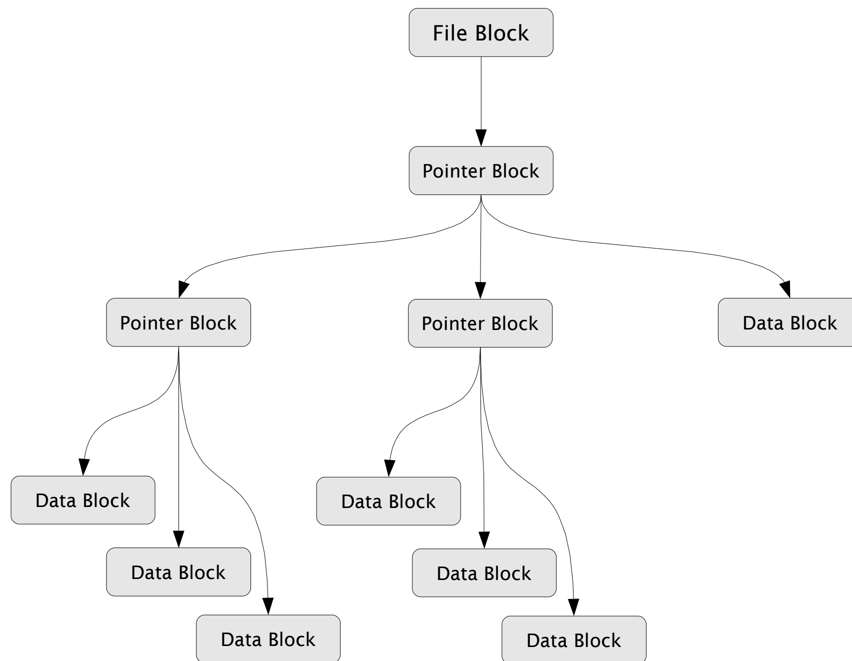


Figure 3.5: File block and pointer tree.

field indicating what, if any, compression scheme was used. Two compression schemes are supported at present, Gzip and Bzip2. Because compression is done on individual data blocks and not on the entire archive, data can be stored and restored from the archive more easily. For instance, it is not necessary to decompress the entire archive in order to retrieve a single file. Instead only the data block associated with that file needs to be decompressed. For large archives this is an advantage.

Each directory in the archive is represented by a directory block. Like the file block, a directory block contains the meta-data associated with the directory such as the directory's name, permissions, et cetera. The contents of the directory are represented by separate file and directory blocks. In order to point to its contents, a directory block points to the root of a pointer tree where each leaf is a directory or file block. Figure 3.6 shows an example of a directory and its pointer tree. Each leaf block represents a file or directory within the parent directory. In the figure the leaves of the tree are directory and file blocks. These block would normally point to

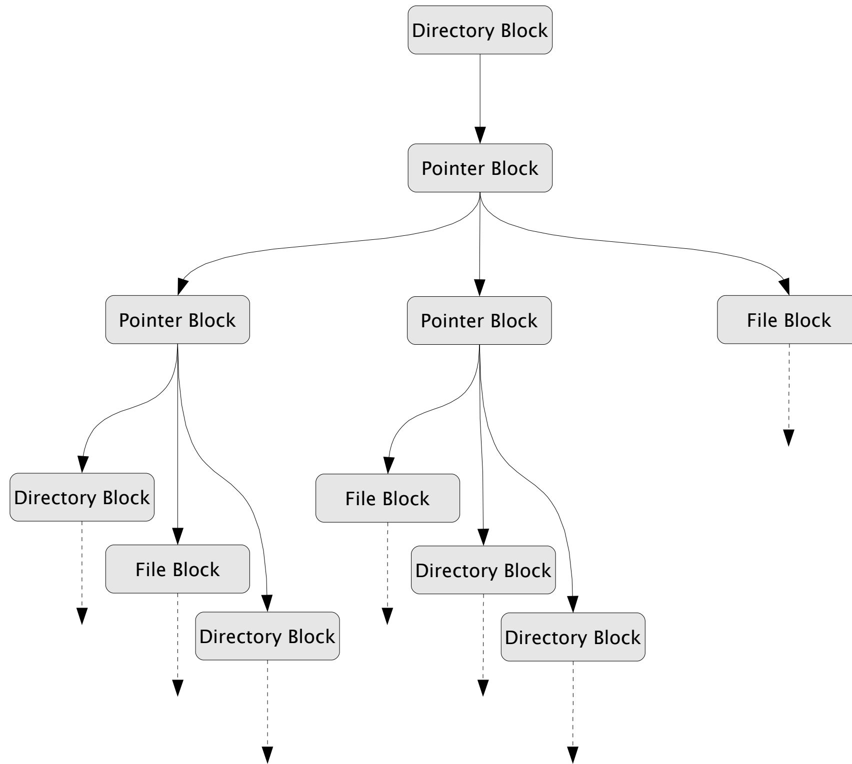


Figure 3.6: Directory block and pointer tree.

yet more blocks, represented in the figure as dashed lines. There is simply not room to display each of the sub-trees rooted at each of the file and directory blocks.

A single archive may contain many entries. Each entry in the archive is represented by a root block. The root block points to the directory block representing the root of the entry. Root blocks also contain a version number. This tells the software reading the archive which version of the block specification was used to write the entry. Currently version 0 is the only specification defined and is described in greater detail in Appendix A. Increasing the version number allows the definition of new enhanced specifications in the future without redefining the old specification and breaking backwards compatibility with data written to the archives under the old specification.

3.4 Archive Interface

Jac stores all of its data as blocks of varying length. These blocks are stored in archives. Archives may be implemented in many different ways, but they all share the same interface. This interface will be described. A description of one specific archive implementation can be found in Chapter 4.

Archives keep track of the blocks that they store by maintaining a mapping of keys to values. These mappings can be implemented efficiently by any appropriate data structure such as a hash table or balanced tree. The data of each block is stored as a value associated with a unique key. The archive interface has only two commands for dealing with these key-value mappings. The `put(block)` command stores a single block and returns a unique key for that block. The `get(key)` command takes a key and returns the associated block.

Each key is derived from the block itself. Keys are produced by computing a SHA1 [28] cryptographic hash function over the block. The SHA1 function takes a variable amount of input data, the block, and produces a 160-bit output, the key. If two blocks contain identical data they will produce an identical key. This is how duplicate blocks are detected. When a block is added to the archive, with the `put` command, its key is computed. The archive is searched for that key among all of the key-value mappings. If it does not exist, the block is stored in the archive and a new key-value mapping is made. If it does exist, it is assumed that the block already exists in the archive and no new space is allocated. Rather, the archive simply returns the unique key for the already existing block.

The archive interface enforces a write once read many policy. Once a block is stored in an archive it is immutable. The interface does not allow for a block to be removed or changed. This policy fits well with a backup utility designed to preserve all past backup points and helps prevent accidental removal of data. It also makes the design simpler. Since blocks can not be deleted, archives need no method for

reclaiming free space from deleted blocks.

An archive may contain more than one entry. Each entry in an archive represents the root of a directory structure that a user backed up. When a user wishes to access an entry, some method is needed for him to indicate which entry he wants to access. To illustrate the problem consider the following example. A user Fred backs up his user directory named `fred` once a week. Each backup point is a separate entry in the archive allowing Fred to retrieve data from any past backup point. Each entry in the archive needs a unique name of some sort so that Fred can indicate which backup point he is interested in. But, each entry in the archive can not be named `fred`, there would be no way to distinguish the different backup points. Jac allows the user to name each entry when it is placed into the archive and prevents the user from choosing a name that already exists, ensuring that each entry has a unique name. These names allow the user to access archive entries using easy to remember names of their choosing. For example, Fred might name an entry `2009-09-21` to indicate when the backup point was created.

To implement the naming of entries, the archive interface supports a second mapping separate from the one already discussed. Names are mapped to the key of a root block. As the reader will remember, each root block represents an archive entry. This mapping therefore allows a name to be used to access an entry in the archive. Names may be any byte string. The `putName(name, key)` command creates a mapping from the given name to the given key. The name must not already exist and the key must be the key of a root block stored within the archive. The `getName(name)` command returns the associated key.

Being able to name entries with user chosen names goes a long way toward helping the user organize the archive entries. Jac goes one step further and allows the names to be organized into a hierarchical directory structure. Like a file system, Jac treats the ASCII '/' in entry names as a separator character. The namespace of

entry names form a top level directory structure within the archive where each endpoint is an entry within the archive. Each archive entry is itself a directory structure. Therefore, all of the data in an archive can be thought of as part of a single file system like hierarchy. This has two advantages.

First, it allows the entry names to be organized with common file system semantics. For example, our imaginary user Fred might store consecutive backups of his personal computer under the names `pc/2008-01-01` and `pc/2008-02-01`. Fred's laptop backups might be under names like `laptop/2008-01-01`.

Second, a user can use file system semantics when specifying a particular file or directory from an entry. For example, let us assume that Fred from above accidentally deleted an important file from his PC and wants to restore it from one of his backups. He would need to specify both the name of the entry, let's say `pc/2008-01-01` and the full name of the file within that entry that he wants to restore. Let us say that that file is `/Development/projectX/main.c` relative to the root of his home directory and thus the root of the backup point. Since everything in the archive is part of a single file system like hierarchy Fred can simply combine the two and enter `pc/2008-01-01/Development/projectX/main.c` to identify the desired file.

3.5 Data Integrity

Backup data is depended upon to restore the original if anything should happen to it. Thus, it is desirable to be able to verify the integrity of the backup data. That is, there should be some way to determine if the data in the archive has been accidentally altered in any way from what was originally stored there. Alterations could occur due to hardware errors associated with the underlying storage media or a software error that inappropriately writes to the archive.

Due to the way data is stored in the archives, determining the integrity of the

data in an archive is straight forward. Each block is stored in the archive under a key that is a SHA1 cryptographic hash of its data. Blocks are organized into tree structures by pointing to each other using the keys as pointers. Therefore, each tree of blocks composing an entry forms a hash tree [23, 24].

Given any block's key, that block can be retrieved from the archive. The SHA1 hash value of the block can be computed. The key used to retrieve the block is itself the SHA1 hash value of the block when it was originally inserted into the archive. These two values can then be compared. If they differ the block has been altered in some way. If they are the same the block in the archive can be assumed to be unaltered.

A block's key is contained in any block that points to it. Therefore, any block can be verified given a verified block that points to it. Each entry forms a tree of blocks pointing to each other with a root block at the root of the tree. The key to the root block is contained in the mapping for the entry's name and can be used to integrity check the root block. The tree can be recursively descended from the root block all the way down to the leaves. At each step a block's recomputed key is checked against the key found in its already verified parent block.

Since each entry forms a hash tree, it is possible, with the above method, to verify not only the integrity of the data but also the authenticity of the data. Verifying the authenticity of the data ensures that not only have no accidental errors occurred, but that no deliberate modifications have been made either. The only difference is that some method is needed to authenticate the root block's key. This is because the root block's key, from the name mappings, is the first key in the chain of verification. Someone could conceivably circumvent the archive interface and alter the underlying blocks directly along with all of the corresponding key values located in the parent blocks. They could alter the blocks they wanted and all corresponding blocks on a path up to and including the root block itself and its key in the name mapping.

To protect against this, the current implementation allows the user to print the root block's key out upon storing an entry. It is then left to the user to maintain a separate and authentic copy of this key. When verifying an entry, the user can check the root block's key in the archive against their separate copy.

Research into SHA1 has uncovered weaknesses in its design as a cryptographic hash function [38, 3]. While these discoveries do not affect the use of SHA1 as a hash function to fingerprint blocks or to verify their integrity, it does cast some concern on its use to authenticate data as described. This may require a move to stronger cryptographic hash functions in the future.

Chapter 4

Implementation

4.1 Prototype

Using the design described, a prototype of Jac was implemented. This prototype was then used to test the design of Jac. Two types of tests were performed. The first test was designed to see how well Jac deduplicates data in practice. The second test was designed to see how fast the Jac prototype processed data in practice and whether or not Jac's design can be implemented in a way that is sufficiently fast to be practical.

The prototype provides a command line interface that allows a user to store a directory, including all sub-directories and files, into an archive. Given a target directory Jac walks the directory structure converting it into a tree of blocks representing the target directory. These blocks are then inserted into the archive where they are deduplicated against all of the other block in the archive. Directory structure, file and directory names, and Unix attributes such as user/group ID and permissions are preserved. Special files such as FIFO, device, and symbolic links are preserved as well. Users can store as many separate entries to an archive as they wish and name these archives as they choose with unique names. A user can optionally compress the data within an archive entry with Gzip or Bzip2 to further reduce the amount of

storage required. The prototype also implements the reverse process. Given an entry name it can convert the associated tree of blocks to an actual directory structure and write it to disk. A user can also restore just part of an entry such as a single file or sub-directory.

The prototype is written almost entirely in the Python [8, 18] programming language. Python was chosen because it is a high level language with a large number of existing libraries, allowing fast development for a prototype project. During development, initial testing of the prototype indicated that the chunk algorithm, that loops over every byte of data processed, was too slow when run in the Python interpreter. To optimize this, the chunk algorithm was rewritten in C and is called from the Python code.

Since Jac's design defines a common archive interface, many different archive implementations can be written that will work with Jac. The prototype implements one such archive. This archive stores the data to a single formatted file. The implementation relies upon the GNU dbm library [27] (Gdbm). Gdbm is a persistent hash table implementation that provides an application programming interface (API) for storing and retrieving key-value pairs to and from a single file. Since an archive is barely more than that itself, the prototype's archive is implemented as a loose wrapper around the Gdbm API. When a `put(block)` command is made to the archive implementation it computes the SHA1 hash of the block producing the unique key. The Gdbm API is then called to check whether or not the key already exists in the Gdbm file and, if not, the key and block pair are stored in the file. The `get(key)` command simply calls the Gdbm API directly with the key and Gdbm retrieves the block from the file. The only twist is that the archive implements two mappings. One set of key-value pairs is for key-block pairs. The other is for name-key pairs associated with naming archive entries. These two mappings are different and should be kept separate. To do this the `put` and `get` commands add the ASCII '0' character

to the beginning of a key before calling Gdbm to add or retrieve a mapping. Likewise the `putName` and `getName` commands add the '1' character to the beginning of each name. Using a unique prefix for the different mappings like this ensures that the two separate key-value mappings stay separate within the single set of mappings stored by Gdbm.

4.2 Deduplication

The prototype was used to test Jac's ability to deduplicate data. Of course Jac's ability to do so depends on the degree of redundancy in the input data. If there is no redundancy then there is nothing to deduplicate. The results give an indication of how much Jac can reduce the storage space in a real backup scenario. Four different data sets were tested. Each data set consists of a sequence of snapshots of some data source. In the tests each of these snapshots, or backup points, was added to an initially empty archive one at a time. By comparing the size of the input data to that of the resulting archive file, the amount of deduplication can be determined. No compression was used in the tests. If it had there would be no way to tell if a smaller archive file was the result of deduplication or ordinary compression.

For each data set the results are given in a separate table below. For each backup point the table shows the date of the backup, the size of the input directory on disk, the size of the archive after it was added, and the patch size for that addition. The patch size is the amount of change in the archive's size divided by the size of the input. For example, on the second line of Table 4.1 10.33 GB of new data is added to the archive and the resulting archive size is 11.23 GB. From the previous line we see that the archive's size before the addition was 11.12 GB. It is easy to see that much of the 10.33 GB input was deduplicated as the archive's size increased very little. The patch size expressed this change. The patch size for this addition is (11.23

GB - 11.12 GB) / 10.33 GB = 1.06%. This represents how much deduplication has taken place. Smaller patch sizes indicate more deduplication and visa versa. Since it is expressed as a percentage of the input size the patch size can be used to compare deduplication between data sets of different sizes. No patch size is given on the first line of each table because the archives are initially empty.

Date	Input Size	Archive Size	Patch Size
2008-05-03	10.21 GB	11.12 GB	N/A
2008-06-06	10.33 GB	11.23 GB	1.06%
2008-07-04	10.46 GB	11.38 GB	1.43%
2008-08-01	10.46 GB	11.39 GB	0.09%
2008-09-05	10.48 GB	11.42 GB	0.28%
2008-10-03	10.53 GB	11.47 GB	0.47%

Table 4.1: User directory test results

The first data set consists of a single user's working directory at six different points in time spaced roughly a month apart. The directory contains approximately 10.5 GB of data. This data consists mostly of data files such as source code, text files, application documents, PDF's, audio files, pictures, and user application settings. The results can be seen in Table 4.1. Deduplication was quite effective in this test. Given 62.47 GB of total input data the final archive size is only 11.47 GB. Each additional backup point of roughly 10.5 GB results in a meager increase in the archive's size. This can be seen by the small patch sizes. The average of the patch sizes was 0.67%. For this data set, on average, given 1 GB of input the archive needs to only store about (1 GB) * 0.67% = 6.82 MB of data to represent it.

Data	Input Size	Archive Size	Patch Size
2008-04-07	6.74 GB	5.78 GB	N/A
2008-05-03	6.89 GB	6.55 GB	11.18%
2008-06-06	7.02 GB	7.30 GB	10.68%

Table 4.2: Windows test results

The second data set consists of the backup files from three full backups, each

taken roughly a month apart, of a desktop machine running Windows XP using the standard Windows backup utility. Each of the backup files is uncompressed and approximately 7 GB in size. This test differs from the first in two important ways. First, this data set provides a wider range of data. The first was composed of a large amount of user data. This data set contains all of the files needed to restore a PC. This includes the operating system, applications binaries, and system settings. Second, in this test the input data has already been serialized by the Windows backup utility into a single file. It is this file, not the original directory contents that Jac processes as input. Jac does not understand the Windows backup file format and will process it the same as any other single file. Table 4.2 shows the results. Deduplication is quite effective. The archive's final size of 7.30 GB is significantly smaller than the total size of the input files at 20.65 GB. The patch sizes of 10.93% on average indicate that the amount of deduplication is not as good as it was in the first test.

Revision	Input Size	Archive Size	Patch Size
2.6.0	200.8 MB	186.2 MB	N/A
2.6.1	200.1 MB	207.1 MB	10.44%
2.6.2	204.2 MB	248.5 MB	20.27%
2.6.3	206.2 MB	285.4 MB	17.90%
2.6.4	207.7 MB	325.0 MB	19.07%
2.6.5	209.1 MB	366.8 MB	19.99%

Table 4.3: Linux test results

The third data set consists of the Linux kernel source tree at six consecutive revision points (v2.6.0 - v2.6.5). Each revision is a directory structure using about 200 MB of disk space and containing more that 15,000 files and directories. This data set was chosen because it is very different from the others. From one revision point to the next much of the source code is bound to be the same. However, because it is a source tree of a very active project, there are lots of small changes to many of the files from one revision to the next. Table 4.3 shows the results. Given 1.2 GB of input data the final archive file size is only 366.8 MB. The average patch size was

17.53%.

Directory	Input Size	Archive Size	Percent of input
1	1.3 TB	0.975 TB	74.96%
2	5.7 TB	4.832 TB	84.77%
1 & 2	7.0 TB	5.807 TB	82.96%

Table 4.4: Bioinformatics test results

The final data set is scientific data from biological experiments. The data is the output generated by a high throughput DNA sequencing machine. This consists mostly of four color dye images of the DNA bases making up the genes or portions of a genome along with other ancillary data such as scoring values to indicate how reliable the results are. Each run generates roughly 1-5 TB of data per experiment depending on the experimental parameters. The deduplication test input consists of two separate directories. Each directory is the output of one biological experiment run.

In the first three deduplication tests each backup point was a snapshot of a single data source in time. Because of this there was an expectation of duplicate data from one backup point to the next. This is not the case for this test. The two input directories may not share any duplicate data. Because of this each directory was deduplicated first separately and then together. Table 4.4 shows the results. The resulting archives are smaller than the input data by 15% or more in all cases. Adding the separate directory archive sizes and comparing it to the archive size of the directories together shows that most of the deduplication occurred within the directories not between the directories. Given 7.0 TB of total input data Jac reduced this to a resulting archive of 5.807 TB.

Processing all of this data took 44 hours of CPU time. In order to conserve space on the test machine for this test no data was written to the archive, rather the block sizes and the duplicate blocks were simply tabulated. So, the 44 hours even underestimates the real time. Given the large time requirements and the small

percentage of space saved, compared to the other data sets, it does not look as though Jac is a very good match for this type of data.

4.3 Performance

Jac was designed to deduplicate data and thus be able to store backups in a more space efficient manner not to necessarily be faster than current backup techniques. It is however important to ensure that the design of Jac is not such that all implementations will be so slow as to prohibit their practical usage. Using the prototype, tests were performed to see how fast it processes data. Being a prototype written mostly in an interpreted language there is no expectation that it will run particularly fast. But, if the prototype is fast enough for practical use then more optimized implementations will be as fast or faster.

The tests were carried out on off the shelf hardware. The test machine had a 2.2 GHz Intel core 2 Duo processor, 2 GB of RAM, and a single Serial-ATA hard disk. For each of three data sets both Jac and Tar were run and the run times analyzed and compared. Tar was chosen because it is very similar to Jac in its operations. Tar walks the target directory structure converting the entire structure into a serialized form that is written to a file. Jac does the same writing to a single file archive. Since Tar is a widely used and mature application comparing it to Jac should give a meaningful idea of how close Jac's performance is to being usable in practice. Each test was run three times to ensure that system load was not effecting the results. The times were then averaged and rounded to the nearest second. The only exception to this is the third test. Due to the long run time it was run only once.

In the first test a single 1 GB file of random data was used as input to both Jac and Tar. This gives somewhat of a baseline as it shows how fast Jac can process data when there is nothing to deduplicate. Jac took 3 minutes 35 seconds giving it a

processing speed of 4.76 MB/s. Tar took 1 minute 24 seconds giving it a processing speed of 12.19 MB/s. Thus Jac's performance is only 39% of Tar's. The results are to be expected. Since the input is a single file, Tar is doing little more than copying the data from one file to another. Jac on the other hand is doing significant work in a fruitless effort to deduplicate a random file.

In the second test both Jac and Tar were run on a 1.2 GB directory of Linux source code. The directory contained six successive revisions of the Linux source tree (2.6.0 - 2.6.5). This is the same data set used above in the deduplication tests. This data set gives Jac some opportunity for deduplication. Jac ran in 5 minutes 49 seconds resulting a processing speed of 2.93 MB/s. Tar, coincidentally, ran in 5 minutes 49 seconds as well. Both programs ran slower in this test than in the previous test. For one thing, they both had to walk the directory structure, opening and closing many separate files. There is an overhead to that compared to the first test where it simply opened and processed one large file.

The results of the second test are a little surprising. In the test Tar has far less work to do than Jac. While both programs must read in the same data from the input directory, from that point on they do very different things. Tar must serialize the input data before finally writing it to disk. Jac on the other hand must break the file into chunks, process them into blocks, fingerprint the blocks with a cryptographic hash function, and test each key against the index before finally writing the block (assuming it is not a duplicate) to disk. Furthermore, Tar is a mature program written in C while Jac is a prototype with much of its code running in the Python interpreter. The trick, so to speak, is that while Jac spends more time processing it spends less time writing data to disk due to deduplication. Looking at the output files confirms this. The Tar file was 1.1 GB in size while Jac's archive file was a mere 366.8 MB. Given duplicate data in the input, Jac is able to trade I/O time for CPU time. Given the speed of CPUs compared to hard disks this trade off is advantageous.

In a third test Jac was used to process three Windows backup files totaling 20.65 GB. Again, this is the same data set that was used above in the deduplication tests. Jac took 235 minutes and 13 seconds giving it a processing speed of 1.5 MB/s. Tar took 30 minutes and 43 seconds or 11.47 MB/s. Tar ran at about the same speed as it did in the first test. Which makes sense. In both tests Tar is processing large files. Jac on the other hand ran extremely slow.

Why is Jac so much slower in the third test compared to the first test when there was more opportunity for deduplication? The author's hypothesis was that Gdbm was performing poorly due to the large archive sizes produced. To test this theory a "dummy" archive was implemented. This archive when called with a `put` command computes the unique key for the block using SHA1 and returns it. It does not however check to see whether a duplicate block has previously been written nor does it write anything to disk, it simply discards the block. This dummy archive essentially tests how fast Jac processed data minus the time that Gdbm takes. Using this dummy archive Jac was again run with the Windows data set. This time Jac took 11 minutes 33 seconds giving it a processing speed of 30.51 MB/s. Comparing the 11 minute time in this test to the 235 minute time of Jac using the Gdbm archive indicates that an inordinate amount of time is being spent by Gdbm to index and write the data to disk.

In order to narrow down where the time is going a little more another test was performed. In this test Gdbm was used to maintain the index of keys and check for duplicates, but the blocks themselves were not written to disk. This was then ran on the Windows data set. It ran in 173 minutes 5 seconds. Comparing this to the 11 minute time from the previous test indicates that, at least with larger inputs, Gdbm is very slow at indexing. Unfortunately there is no documentation on Gdbm's file format or its internal workings short of reverse engineering from its source code. It is therefore difficult to tell exactly why Gdbm performs so poorly when dealing with

larger indexes. However an analysis of the problem of indexing in general is helpful.

Let us assume that the entire index can fit into memory. To test this another archive was tested. This archive does not use Gdbm at all. It simply appends all of the non-duplicate blocks to a single file. The index is kept entirely in memory using Python's built in dictionary type which is implemented as a hash table. For each key the index records a pair, the offset within the file where the block is located and the length in bytes of that block. When the program finishes it uses Python's built in routines to serialize the index and write it to a separate index file. With this test archive Jac ran the Windows data set in 18 minutes and 15 seconds giving it a processing speed of 19.31 MB/s. This is 68% faster than Tar's 11.47 MB/s on the same data set.

As long as the index resides in memory Jac is quite fast. But as the index grows it will at some point no longer fit in main memory. At this point secondary storage will have to be accessed which usually means magnetic disks. How quickly can data be indexed on disk? Let us assume that each index search requires just a single disk access. Each disk access will require a seek and a read. On the test machine the average random read seek time is 8 ms. As an example, given 5 GB of input data broken into 8 KB data blocks results in 655360 blocks. At 8 ms per block this will require 5242 seconds or about 1.5 hours and that is just the seek time. Higher performance disks may of course have faster seek times, but the difference between main memory and the fastest magnetic disk is still great.

The problem is that while Jac can reduce disk I/O by eliminating the need to write the duplicate blocks to disk, for large indexes, it introduces disk I/O when it has to read parts of the index from disk. Because of the slow seek times of disks, checking the index for duplicate blocks becomes a performance bottleneck. A common technique for improving performance when dealing with the differential in speed of main memory and disk is caching. Assuming a high rate of cache hits most index

searches will be handled out of memory and never go to disk thus greatly improving performance. Good cache performance however depends upon locality of reference, either temporal locality or spacial locality. Unfortunately, Jac exhibits neither. In this context temporal locality is the expectation that a key searched for once is likely to be searched for again in the near future. If there are many duplicate blocks and they are positioned close to each other in the input stream this would be true. But that would be mere coincidence, there is no reason to expect such a scenario. Spacial locality is the expectation that given one key search other keys "near" it, by some measure of the key space, will be searched for as well. Again, this is not the case. Because the keys are SHA1 hashes they are essentially randomized.

Determining Jac's performance is not simple. Two conclusions can be drawn from the testing and analysis. First, performance depends on the amount of duplicate data in the input. If there is no duplicate data in the input using Jac is pointless and slow compared to a simple utility like Tar. If there is duplicate data Jac can be competitive because duplicate data means less data to write to disk and therefore higher performance. Second, performance depends on the size of the index and whether in can fit in main memory. If the index will not fit into memory performance quickly degrades as even a single disk seek per index search is a bottleneck.

It is important to remember that the point of Jac is to reduce the overall space usage of backups not necessarily to be faster than traditional schemes. Given the large and increasing sizes of main memory the indexes of many data sets will be able to fit in memory. For some applications even the slow performance of a disk index may be practical if the data set is highly duplicated and reducing storage space is paramount. New technologies may also change the picture in the near future. For example solid state drives have much faster random access times than conventional disks drives as there is no read/write head to move.

Chapter 5

Conclusion

Jac provides modern deduplication technology in a simple command line backup utility. Jac was originally motivated by the desire to have a simple snapshot style backup system like those found in some operating systems and file systems, but in the form of a command line tool. These types of systems allow the user to retain all past backup points. To do this in a space efficient manner Jac relies upon deduplication technology. Deduplication techniques provide an interesting way to solve the backup problem compared to traditional techniques. By deduplicating common pieces of the underlying data, significant savings in the amount of storage needed to preserve the backups can be achieved.

Deduplication in Jac is performed with content based chunking and fingerprinting. Jac uses a custom anchor function for performing content based chunking rather than the common scheme based on Rabin fingerprints. The two functions were compared and it was found that the function used in Jac is more than twice as fast as the Rabin function. Tests were also done to compare the behavior of the two functions when chunking data. Tests show that both functions follow the idealized model of a geometric distribution fairly closely with a wide range of data.

Jac stores all of the data as variable length blocks which are then stored into

archives. This includes the data from the files along with UNIX attributes such as file names, ownership, and permissions. Blocks are organized into tree structures through pointers between the blocks. This organization of blocks along with the use of cryptographic hash functions for fingerprinting provides Jac with strong built in integrity checking of the archive data. Jac defines a simple archive interface for all archive implementations. This allows many different archive implementations to be written for storing the block data. The archive interface allows archive entries to be named with user chosen names and for these names to be organized into a directory like hierarchy.

A fully usable prototype of Jac was implemented. This prototype was then used to test Jac's ability to deduplicate data. Four different data sets were used. The results for the first three confirm Jac's ability to deduplicate data leading to significant space savings when given consecutive backup points. The final test was composed of data from biological experiments. While Jac was able to reduce the storage space, it was unable to do so to the extent seen in the first three experiments. The prototype's performance was also tested. Testing the performance proved tricky. The amount of duplicate data in the input effects the processing speed. With no duplicate data, Jac's performance suffers compared to more traditional tools as it must do more work. With duplicate data Jac can perform as well or better. This results from not having to write the duplicate data to disk, thus saving significant time doing I/O to disk. Performance is also affected by the total size of the archive. As the archive grows so to does the index. When the index is too large to fit in memory extra disk access must be done to perform index look ups for deduplication processing. These extra disk accesses can quickly become a performance bottleneck.

Jac is a prototype and there is still much work to be done. Jac would benefit from multithreading. A lot of CPU processing and a lot of I/O are done during deduplication. As a single threaded application, nothing else can be done while waiting for an

I/O operation to complete. Using multiple threads would allow I/O and CPU operations to overlap, taking better advantage of the computer's resources. Further more, current CPUs in even modest hardware have multiple cores. Being multithreaded would allow Jac to use all of these cores, not just one. Jac's design lends itself to multithreading. Since deduplication is detected at the archive, concurrency issues can be handled there in a single place. For example, adding a simple reader/writer lock around the archive would allow multiple worker threads to work in parallel on different files. A thread can read a file in, divided it into chunks, process it into blocks, and apply the necessary compression without any concurrency issues.

Beyond improvements to robustness and speed, many features and enhancements are possible. For instance, an exclude option that would allow the user to specify file patterns to be excluded from the archive. This would be useful to avoid backing up temporary directories or derivative files. For instance, many web browsers store a cache of recently visited web pages in a subdirectory somewhere within the user's home directory. An exclude option would allow that subdirectory to be specified and prevent the temporary cache files from being backed up each time. Similarly, a compression exclude option would allow the user to express patterns to be excluded from compression. This would be useful for image, audio, and video files that have already been compressed. For example, it is unlikely that the general compression methods of Gzip or Bzip2 will compress an mp3 audio file by a significant amount. It wastes CPU time trying to further compress these types of files. A '*.mp3' exclude pattern would cause the files to be backed up, but no compression would be applied.

Jac puts modern deduplication technology into a simple command line program. While designed to solve the problem of space efficient backup, such a solution might prove useful beyond that problem domain. The archive entries in Jac can be any directory structure not just backup points. This makes Jac a more general deduplication tool. The author hopes that others will find Jac useful in many ways.

Appendix A

Block Specification

Jac stores all of the data as blocks that are inserted into archives. These blocks can point to one another by containing each other's unique keys. A high level description of these blocks and how they point to each other can be found in Section 3.3. This appendix gives a low level description of each block's layout suitable for writing software to encode/decode them. The specification described here is version 0. Newer specifications, with higher versions numbers, may be defined in the future.

Five different blocks are defined: root blocks, data blocks, pointer blocks, file blocks, and directory blocks. Each of the block types is described in turn. A diagram of each block is given showing the structure of each block's layout. The diagram shows the relative position of the fields to each other with lower memory addresses on the left. In the description below each diagram the fields are explained. There is no padding between fields to bring short fields up to some minimum length such as 4 bytes or 8 bytes. For example, if a 1 byte field is followed by a 4 byte field then the 4 byte field immediately follows the 1 byte field giving a total of 5 bytes. Fields that are multi-byte integers are encoded most significant byte first or big-endian.

Two of the fields are of note as several block types use them. First, each block, except for the root block, begins with a Type field. When a block is read from the

archive the software needs some way to determine what type of block it just read so that it will know how to decode it. The Type field is the first field in each block, and indicates what type of block it is. Then the software can correctly decode the rest of the fields. The Type field is a 1 byte unsigned integer. The Type field is unnecessary for root blocks because they always exist at the root of an entry and their type can be deduced from their location.

Second, both the file block and directory block contain a Meta field. This is a variable length field of bytes representing the attributes, or meta-data, associated with the file or directory. Attributes include things such as the file or directory name, permissions, ownership, modification times, and the like. The layout of the Meta field is the same format as the Tar header block used by the Tar utility. See Appendix D of the Tar manual [10] for complete details and descriptions.

A.1 Root Block

Version	Key
1	20

A root block represents a single entry within the archive. The Version field is a 1 byte unsigned integer indicating what version of the block specification was used to write the entry. Currently only version 0 is defined. The Key field is a 20 byte key that points to the root of the entry's directory structure. This is almost always a directory block. If the entry contains only a single file, it will be a file block.

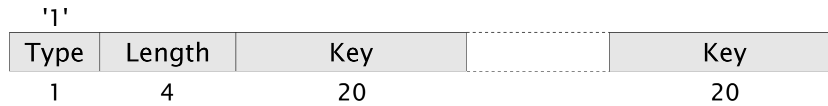
A.2 Data Block

'0'	N	
Type	Length	Data
1	4	N

Each data block represents a single chunk of data produced by running the

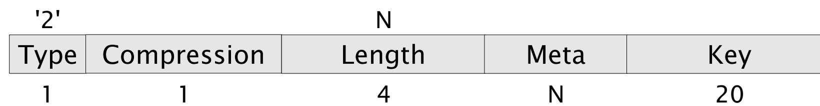
chunking algorithm over a file. The Type field is always 0 for a data block. The Length field is a 4 byte unsigned integer giving the length in bytes of the Data field. The Data field is variable length and contains the raw bytes of the chunk represented by the data block.

A.3 Pointer Block



Pointer blocks are used to point to multiple blocks and create the tree structures that link blocks together into archive entries. The Type field is always 1 for a pointer block. The Length field is a 4 byte unsigned integer indicating how many key fields follow it. The Key fields are 20 byte keys pointing to other blocks.

A.4 File Block



A file block represents a single file stored in an archive. The file block contains the attributes associated with the file. A file block does not contain the data of the associated file directly, but rather points to the data blocks that contain the data chunks for that file. The file block also indicates whether or not the chunks within the data blocks have been compressed and if so with what compression scheme.

The Type field is always 2 for a file block. The Compression field is a 1 byte unsigned integer indicating what type of compression, if any, was used to compress the Data field of the data blocks associated with the file. The integer 0 indicates that no compression was used, 1 that Gzip was used, and 2 that Bzip2 was used. The Length field is a 4 byte unsigned integer indicating the length in bytes of the Meta

field. The Key field is a 20 byte key pointing to a pointer block. This pointer block is the root of a pointer tree. The leaves of this tree are data blocks. A left to right walk of the tree gives the data blocks (their chunks) in the order that the should be concatenated to reproduce the original file. As a special case, if the file contains only a single data block, the key will point to it directly. This saves producing an intermediary pointer block that points to a single data block.

A.5 Directory Block

'3'	N		
Type	Length	Meta	Key
1	4	N	20

A directory block represents a directory stored in the archive. The directory block contains the attributes associated with the directory. Directories contain various sub-directories and files. A directory block represents this by pointing to the directory and file blocks that represent those sub-directories and files.

The type field is always 3 for a directory block. The Length field is a 4 byte unsigned integer indicating the length of the Meta field in bytes. The Key field is a 20 byte key pointing to a pointer block. The pointer block is the root of a pointer tree. The leaves of this tree are directory and file blocks. Each of these blocks represents a sub-directory or file in the directory. If the directory has only a single entry, either a single directory or a single file, the key field of the directory block will point to the respective directory or file block directly. This avoids an extra pointer block pointing to a single directory or file block.

Bibliography

[1] J. Black. Compare-by-hash: A reasoned analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 85–90, 2006. http://www.usenix.org/event/usenix06/tech/black.html.

[2] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. *SIGMETRICS Perform. Eval. Rev.*, 28(1):34–43, 2000.

[3] Martin Cochran. Notes on the Wang et al. 2^{63} SHA-1 differential path. Cryptology ePrint Archive, Report 2007/474, 2007. http://eprint.iacr.org/.

[4] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.

[5] P. Deutsch. Gzip file format specification version 4.3. RFC 1952, May 1996. http://www.ietf.org/rfc/rfc1952.txt.

[6] P. Deutsch. Zlib compressed data format specification version 3.3. RFC 1950, May 1996. http://www.ietf.org/rfc/rfc1950.txt.

[7] J. G. Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, COM-30(1):247–252, January 1982.

[8] Python Software Foundation. The Python home page, 2009. http://www.python.org.

[9] Jean-loup Gailly and Mark Adler. The gzip home page. http://www.gzip.org.

[10] John Gilmore and Jay Fenlason. *GNU tar: an archiver tool*. Free Software Foundation, March 2009. http://www.gnu.org/software/tar/manual/.

[11] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the 1994 Winter USENIX Technical Conference*, pages 235–245, January 1994.

[12] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[13] Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley. Logical vs. physical file system backup. In *Proceedings of the USENIX Third Symposium on Operating Systems Design and Implementation (OSDI)*, pages 239–250, February 1999.

[14] Navendu Jain, Mike Dahlin, and Renu Tewari. Taper: Tiered approach for eliminating redundancy in replica sychronization. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, December 2005.

[15] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)*, pages 5–5, Berkeley, CA, USA, 2004. USENIX Association.

[16] Steve Lammert. The 3.0 AFS backup system. In *Proceedings of the 4th USENIX Conference on Large Installation Systems Administration (LISA)*, pages 143–148, October 1990.

[17] A. J. Lewis, Joe Thornber, Patrick Caulfield, Alasdair Kergon, and Jochen Radmacher. Logical volume manager HOWTO, 2006. `http://www.tldp.org/HOWTO/LVM-HOWTO/`.

[18] Mark Lutz. *Learning Python*. O'Reilly, 4 edition, September 2009.

[19] Udi Manber. Finding similar files in a large file system. In *Proceedings of the Winter 1994 USENIX Technical Conference*, San Francisco, CA, January 1994.

[20] Nagapramod Mandagere, Pin Zhou, Mark A Smith, and Sandeep Uttamchandani. Demystifying data deduplication. In *Companion '08: Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, pages 12–17, New York, NY, USA, 2008. ACM.

[21] Dirk Meister and André Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–12, New York, NY, USA, 2009. ACM.

[22] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*, chapter 9. CRC Press, 1996. `http://www.cacr.math.uwaterloo.ca/hac`.

[23] Ralph C. Merkle. *Secrecy, authentication, and public key systems Ph.D. dissertation Department of Electrical Engineering*. PhD thesis, Stanford University, 1979.

[24] Ralph C. Merkle. Protocols for public-key cryptosystems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–133, April 1980.

[25] Sun Microsystems. OpenSolaris documentation, 2009. `http://opensolaris.org/os/community/zfs/docs/`.

[26] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 174–187, 2001.

[27] Philip A. Nelson and Jason Downs. The GDBM man page, v1.8.3, 1999. `http://directory.fsf.org/project/gdbm/`.

[28] National Institute of Standards and Technology. FIPS 180-1: Secure hash standard, April 1995. `http://csrc.nist.gov`.

[29] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, 1990.

[30] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, July 2004.

[31] Sean Quinlan. A cached worm file system. *Software Practice & Experience*, 21(12):1289–1299, 1991.

[32] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2002. `http://www.usenix.org/publications/library/proceedings/fast02/quinlan.html`.

[33] Sean Quinlan, Jim McKie, and Russ Cox. Fossil, an archival file server. `http://www.cs.bell-labs.com/sys/doc/fossil.pdf`.

[34] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard University, 1981.

[35] Julian Seward. bzip2 and libbzip2: A program and library for data compression, December 2007. `http://www.bzip.org/docs.html`.

[36] Steve Shumway. Issues in on-line backup. In *Proceedings of the 5th USENIX Conference on Large Installation Systems Administration (LISA)*, pages 81–88, September 1991.

[37] Avi Silberschatz and Peter Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., 5 edition, 1997.

[38] X. Wang, A. C. Yao, and F. Yao. Cryptanalysis on SHA-1. Presented by Adi Shamir at the rump session of CRYPTO 2005. Slides may be found currently at `http://csrc.nist.gov/groups/ST/hash/documents/WangSHA1-New-Result.pdf`.

[39] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies(FAST'08)*, pages 269–282, 2008.