Using Rabin-Karp Fingerprints and LevelDB for Faster Searches

by

Richard A. Deighton

A thesis submitted in conformity with the requirements

for the Degree of Master of Science

Graduate Department of Computer Science

University of Ontario Institute of Technology

To my sons Brock and Chad,

my parents Hartley and Pat,

and, my wife Elizabeth.

Contents

List of Figures

List of Tables

Observations

List of Algorithms

List of Equations

Abstract

This thesis represents the results of a study into using fingerprints generated according to the Rabin-Karp Algorithm, and a database LevelDB to achieve Text Search times below GREP, which is a standard command-line UNIX text search tool.

Text Search is a set of algorithms that find a string of characters called a Search Pattern in a much larger string of characters in a document we call a text file.

The Rabin-Karp Algorithm iterates through a text file converting character strings into fingerprints at each location. A fingerprint numerically represents a window length string of characters to the left of its location. The algorithm compares the calculated fingerprint to the Search Pattern's fingerprint. When fingerprints are not equal, we can guarantee the corresponding strings will not match. Whereas when fingerprints are, the strings probably match. A verification process confirms matches by checking respective characters.

Our application emerges after making the following major changes to the Rabin-Karp Algorithm. First, we employ a two-step technique rather than one. During step 1, the preprocessing step, we calculate and store fingerprints in a LevelDB database called an Index Database. This is our first major change unique to us. Step 2, the matching step, is our second unique change. We use the Index Database to look-up the Search Pattern's fingerprint and gather its set of locations. Finally, we allow the pattern to be any length relative to the window length. We even created an equation to check if the difference in length is too long for the fingerprint's number system base.

We facilitated our performance experiments by first building our application and testing it against GREP for a wide range of different parameters. Our conclusions and recommendations determine that although we currently only outperform GREP in about half the cases, we identify some promising opportunities to modify some parts of our application so that we can outperform GREP in all instances.

Chapter 1: The Research Problem

1.1.1 Introduction

In general, our task is finding occurrences of a search string in a text file. There are many algorithms and techniques solving this *Text-Search* problem. Our procedure is unique and benefits from synergy. While none of the main parts are particularly new, when we coupled them together in such a new way, we could measure improvements in performance. With that in mind, this thesis represents the results of a study into using Rabin-Karp fingerprints and a database LevelDB to create a text search application that can achieve text-search times below *GREP's* (UNIX's standard text search tool.) We use the remainder of this Chapter to provide an overview of the problem, a description of our Thesis, a discussion of our Motivation behind our Thesis, and a glimpse of our observations showing evidence of our success.

This chapter provides some material to set a context for the experiment. It presents a number of sections covering necessary background for several important objects involved in our work, and touches on their pertinent issues, algorithms, and mathematics of the problem. Subsequent paragraphs in this section will summarize the presentation of this material. Later chapters present the theory and implementation of our work in detail. Hence, to avoid repetition, we try to keep background material in this introductory chapter at an abstract level.

In that light, we outline most of the modifications we intend to make in the traditional Rabin-Karp algorithm. We also present the most basic objects of this work, text strings, by pointing out what they are, and where we encounter them. Next, we introduce the most important object of our work after text strings, the Rabin-Karp algorithm. (Karp & Rabin, 1987) We demonstrate Rabin-Karp's idea of using a numeric fingerprint to represent substrings within a text file is a novel mixture of hashing, search heuristics, and simple arithmetic. After this, we describe the motivation for performing this work. It ties

together the previous discussion with an argument that our new application has promise to solve most, if not all, issues.

The next section gives an overview of the search times we achieved versus GREP. These times confirm our approach has merit in terms of performance over GREP, which are quite encouraging from our point of view.

The final section of this introductory chapter introduces the layout of the remaining document. This discussion will introduce and combine the objects used by the approach, as well as, many of the changes we make made to Rabin-Karp's original algorithm into a concise statement of the solution.

1.2 Thesis

In light of our discussion above, we propose to verify the following Thesis in this work.

> **We can modify the Rabin-Karp Algorithm, and configure the LevelDB database, to create a two-step-text-search-engine algorithm that will outperform the one-step-text-search-engine *GREP* in finding search patterns within local text files.**

In a general sense, the type of solution we investigate employs both sources of improvement mentioned earlier: (1) creating a hash file and (2) reusing its information. In other words, we propose to build a text-search application having two steps, where a preprocessing step uses pieces of the Rabin-Karp algorithm to create fingerprints that we save as a hash file using LevelDB. In addition to "hash file," we sometimes refer to this LevelDB file as, "Index Database," or, "index" for short. In any case, this index is available in any number of subsequent matching steps. We also use techniques from Rabin-Karp's algorithm to carry information forward from one fingerprint calculation to subsequent calculations while building the hash file; thereby significantly reducing the time required to calculate a fingerprint. After building an index, our process uses this

database and several other techniques to find positions for any search text. In the end, our work achieved the following two significant outcomes.

First, our application outperforms GREP in more than half of our test runs. These results alone give our application a great deal of promise. Nevertheless, the best results from our experiments were observations of opportunities to make significant improvements. For instance, we observed enough evidence to make us confident we could modify our application so it could outperform GREP in every case. We list these enhancements in a later chapter covering future work.

The second significant outcome is a result of our design procedure, not our experimentation. Hence, even before programming anything we had to solve our biggest problem, leading our work toward the following equation. It is new to text searching and represents a value we need to know before we can even run one experiment. That is, with everything else held equal, what is the maximum length between an Index Database's window length **WL** and the length a pattern can have **PL**?

$$\Delta_{pat} = \frac{\log_2(modulus)}{\log_2 r \ (radix)}$$

Equation 1: Calculating the maximum number of characters our Index Database's Window Length (WL) can have compared to the length of a pattern (PL.)

This importance of this equation to our analysis can hardly be overstated. It demonstrates several key issues threatening the analysis. First, it draws our attention toward a dependency between the modulus q and the radix r. A requirement of the Rabin-Karp Algorithm is that the two be co-prime; which means one, but not both, must be a prime number. (Gusfield, 1997) A second issue covered by this equation has to do with modulo arithmetic. If we did not prevent a user from exceeding this maximum, they could be in a position where their range's maximum value is actually less than its minimum value. This can happen because the Rabin-Karp algorithm, and therefore our application, both use modulo arithmetic. In any case, if this does occur the application is caught in

somewhat of an endless loop. It would start at what it thinks is the minimum fingerprint value, verify that fingerprint, and continue the same thing for every subsequent fingerprint until it hits the end of the database without ever hitting the maximum value. Having thought that none of the fingerprints exceeded the maximum, it would erroneously include all those fingerprints it passed by on its way to the end. This equation is so valuable that we could not even try to include patterns whose length is less than the index database's window length without it.

1.3 Basic String Terminology

Before discussing text strings in arbitrary language, it is necessary to introduce a few of the issues related to them like their building blocks, characters. Every character (c) belongs to a set of characters called an alphabet (symbolized as Σ.) $|\Sigma|$ denotes the number of characters in the alphabet. We only use alphabets like ASCII where each character has an integer index or *code* unique in the alphabet. We obtain a character's code by interpreting its one-byte bit sequence as a one-byte integer. Hence, having a common byte, the character and its code are two inseparable views of the same object. Character codes are typically used to order characters in an alphabet in lexographic, or alphabetical, order. Our approach respects this use and adds another important role for character codes.

Strings are a set of characters arranged in some order. Each character has a unique position in a string; beginning with zero at the left and increasing by one for each consecutive character moving right along a string. We adopt a custom of naming strings using a single bold upper-case letter to remind us that most strings are arrays of characters. We also use the square bracket operator to access a character at a certain position within a string. For example the character at position **3** in string **S** is **S[3]**.

Since our analysis only deals with a few strings, throughout this document, we introduce them here. The most important string array names we use are as follows. We use **S** to represent an arbitrary string, **T** to represent a text file, **P** to represent a search pattern, and **W** to represent a text window we slide through the text file. Similarly, to avoid confusion

we always refer the length of a respective string by tacking an **L** after its name giving us: **SL** for string length, **TL** for text file length, **PL** for search pattern length and **WL** for window length.

With the above terminology in mind, we present a simple example of the text search problem lying at the center of our work. Let **P**=`abba` (therefore **PL**=4) and **T**=`bbabbaxabbabbay` (therefore **TL**=15). The solution for this example is that **P** *occurs* in **T** at three positions; namely, positions 2, 7, and *10*. positions *7* and *10* in this example demonstrate that occurrences of a pattern may sometimes *overlap*. We also demonstrated that manual processes could perform text search; as readers can attest after verifying by hand the three positions we offered as the solution. Of course, however, when PL > 10 and TL > 5Mb manually process is impossible to defend as a mechanism for finding a solution.

1.4 Strings

Strings are ubiquitous in our society. Everywhere one looks, a string is informing them of something. Further still, our libraries contain huge collections of long text strings we call books. In addition, we have the WWW filled to the brim with strings we generally regard as belonging to a document called a web page. In its simplest sense a string is an ordered set of characters. In most cases, a string occurs within a larger unit usually called a document. As demonstrated above, the work we are presenting deals with finding and reporting the position(s) where a *Search String* occurs in a *Text File* (i.e., a document.)

With so many text strings in our lives, we are unwittingly drowning in a sea of data, and thirsting for information. That means we spend a great deal of time shifting through these documents searching for our information need. In fact, in addition to text search there is a huge spectrum of choices available to help a user hunt for information electronically. The sciences built around such a seemingly straightforward subject are huge. It is important for us to recognize that our work occupies only a small corner of the huge sea of algorithms, techniques, and theories. To help put our corner into context, and

before getting into the text search part of this spectrum, we must introduce a few helpful concepts; most of which come from the field called *Information Retrieval*.

1.4.1 Information Retrieval

Information Retrieval is a science concerned with identifying, extracting, and organizing information for a huge collection of data. (Beaza-Yates & Ribeiro-Neto, 1999) Since our work centers around text search, we thought it appropriate to acknowledge the existence of this vast science that encompasses our work.

People do not usually walk around all day totally focused on finding a specific piece of information. Nevertheless, when a person actually needs something, they really do start looking, usually in earnest. Their drive comes from satisfying a desire to find specific information. Information Retrieval refers to a goal of finding specific information as an *information need*. From this point forward, we regard finding a pattern in a text file is also filling an information need.

Another important concept from Information Retrieval is that users familiar with both their corpus and their toolset, can satisfy their information need with little effort and time. Information Retrieval uses metrics like inverse document frequency to help sort candidate documents from most to least likely to satisfy an information need. Other useful tools include searching for documents using Metadata values and Keywords that can pare down a search's list of candidate documents to a manageable few. (Beaza-Yates & Ribeiro-Neto, 1999)

Therefore, Information Retrieval has a huge spectrum of tools, many of which are proficient at filling an information need. While text search, is a smaller set of tools dealing with finding one string in another. It does fall under the broad heading of Information Retrieval. The following section looks at how Information Retrieval's sub-spectrum containing text searching addresses some information needs.

1.4.2 Text Search covers a Spectrum of Techniques

At one end of the text search spectrum, we have more manual-process-based approaches such as browsing, or surfing, where a user examines page after page of information on the WWW looking for links that will bring the goal (i.e., information need) closer.

At the other end of the text search spectrum-the part we are studying, we have automated approaches where a user simply types their search query and presses [Enter.] Their information need automatically becomes finding occurrences of that string. Producing a list of the position for each occurrence of the search string satisfies the information need. Even though there are many details for such a process, the general idea for text searching is finding and reporting the positions of the search pattern within all documents of a corpus a user is searching. We will reinforce this idea throughout our discussion. In fact, to add more precision to our discussion describing these processes we need to introduce some details of our notational convention, to which we now turn.

1.4.3 One-Step (On-Line) or Two-Step Preprocessing

Text search algorithms typically use one of two to broad categories of analysis techniques, one-step or *on-line* and two-step or *preprocessing/matching*. On-line techniques perform a search at the same time they are processing a text file, which can make them extremely fast. The original Rabin-Karp algorithm is an on-line technique. We have already announced our interest in two-step processes because our approach requires we build an index and perform searches using it. Therefore, we turn our attention to two-step methodologies with preprocessing plus matching. This technique breaks the problem into two types of steps, a preprocessing step, followed by any number of subsequent matching steps. The preprocessing step makes one pass through a text file building and storing some sort of index.

Our solution implements a version of a preprocessing technique using a key-data-pair database called LevelDB. We configured LevelDB to store an index of the Rabin-Karp fingerprints for each position of a text file. The innovation here is that Rabin-Karp originally developed their algorithm as an on-line method, searching for the pattern while

processing the file. (Karp & Rabin, 1987) Our approach builds an index once and allows any number of searches using that index. Later, we give an overview of the Rabin-Karp algorithm, describing its details, and the details of our modifications.

Now we turn our attention to giving an overview of searching in general and text searching in particular.

1.5 Some Search Problems

Having introduced strings and searching, we shift focus briefly to discuss some other text search tasks. We have intentionally avoided including them to keep our project scope manageable. Nevertheless, we recognize that several modifications to our framework would make the following issues candidates for our approach.

Computers have been filling their hard drives and on-line sites at an ever-increasing rate. We mentioned earlier there are many ways to shift through all this content using several popular tools from the broader science of Information Retrieval. Recognizing text search is a popular alternative for many reasons, we contrast it to these other tools in the following discussion.

1.5.1 WWW Search Tools

Having already introduced surfing the internet earlier, we now examine another form of looking for information on it. This time we are discussing Google, or Bing, and a whole host of other internet searching engines available. Internet search tools help sift through large volumes of data (i.e., they all search the entire WWW for occurrences of a search pattern or parts thereof.) They report their list usually in less than one second, but the number of ‘hits’ (i.e., eligible targets) is astronomical. One can hardly call a user’s task of sifting through hundreds of thousands of ‘hits’ as being a very efficient search tool for our problem at hand. However, many users with general queries about arbitrary issues find these internet search tools great advisories. They actually do a very good job of putting the most likely target near the top.

These tools are two-step with huge *preprocessing* steps. For instance, Google uses web 'crawlers' to spend an entire day and night traversing (crawling) the entire WWW. They are building an index for use with the hundreds of thousands of search requests the next day. This is a perfect example of searching with a *preprocessing* scheme, which is the same technique we use in our solution.

1.5.2 Local Search Tools

We have built our solution to perform text searching in a local domain, like a text file on a hard drive. This is similar to the text search functionality of several text search command-line tools provided by operating systems. Earlier we mentioned how UNIX had a tool called *GREP* for just such occasions. Windows has two similar command-line text search tools, *FIND* and *FINDSTR*. All three are functionally equivalent; they find search strings in one or more text files. While "fast," they are all one-step techniques, falling under the *on-line* category of search algorithms. Furthermore, they do not "remember" any information from any previous run(s) they can use in a current run, which is a property of all one-step algorithms. In short, these tools are cumbersome but are robust enough to be in every advanced user's toolbox.

1.6 Rabin-Karp Algorithm

1.6.1 Introduction

Since our work relies so much on Rabin and Karp's ideas and their algorithm, we briefly introduce it now highlighting its main characteristics. Interested readers can refer to the next chapter where we give a very detailed formal description. In 1987, R. Karp and M. Rabin published the *randomized* fingerprint method as a practical and efficient solution to the string-matching problem. (Karp & Rabin, 1987) The *randomized* fingerprint method is a perfect match for our solution because it carries information forward from one comparison to the next, it performs well in practice, and we can generalize it to extend to other related problems. We will refer to their method as simply the *Rabin-Karp*

algorithm, and give a brief overview of its details here using descriptions from (Corman, Leiserson, Rivest, & Stein, 2001) and (Gusfield, 1997).

The Rabin-Karp algorithm uses modulo arithmetic, Horner's Rule, and a number of other innovative techniques to calculate a fingerprint (decimal number) for each substring in a larger text file **T.** The algorithm first calculates pattern **P**'s fingerprint (denoted as **p.**) Then, it iterates through a text file **T** for every location. At each iteration in **T** we are at (offset/position/ or shift) location, denoted as **s**. Now, it calculates a fingerprint for a pattern-length substring beginning at **s**. If a substring's fingerprint is not equal to **p**, the substring will definitely not match the pattern making it a perfect heuristic for string matching. It also has another advantage that helps speedup the comparison process. As we demonstrate in more detail later, comparing integer values for equality is a simple one-step numeric process. Therefore, in one integer comparison we could save having to compare any characters. In addition, small fingerprints (i.e., 32-bit integers) allow an algorithm to take advantage of the speed of small integer arithmetic on modern processors. Incidentally, calculating all fingerprints in **T** and storing them in LevelDB is step one in our proposed approach. Fingerprints are calculated using character codes, radix, and modulus making them a great hash function to the LevelDB database, as we will show later.

The other side of hash files, when the two fingerprint values are equal, we only know one fact; their hash functions put them in the same hash bucket. At this point, we have no idea whether any characters in respective strings match at all. Hence, a pattern's fingerprint value equaling a substring's fingerprint value is at best an extremely good heuristic indicating the *possibility* the two underlying character strings will *match*. We must subsequently confirm or deny an *occurrence* of a match by performing a character-by-character comparison at the particular location in the text. We generally refer to this comparison as *verification*.

In summary, the efficiency of the matching-step in the *Rabin-Karp algorithm* comes about in two ways. Primarily, we eliminate the need to examine each character in the

huge text corpus by the relatively quick method of comparing integer hash values for substrings. Next, we reduce the total amount of character-by-character comparisons required to confirm occurrences. In other words, we only need perform verification at locations in **T** where the fingerprint value of the pattern-length substring in **T** is equal to **P**'s fingerprint value.

1.6.2 Example of Rabin-Karp

The following example uses the ten digits as our alphabet making its radix 10. Cormen uses these values in his rather lengthy example reproduced in Figure 1 and below (Corman, Leiserson, Rivest, & Stein, 2001, p. 913). The top Part (a) of the figure shows a text stream with a five character string 16439 highlighted. The numerical value of that string is 7 modulo 13. Part (b) shows how each location in the text file has a fingerprint calculated one after another. If our search pattern were 16439, we are searching for a fingerprint of 7. Part (b) also shows how a *spurious hit* (false positive) occurs when another substring in the text stream has the same fingerprint of 7 (mod 13). This demonstrates how a completely different set of digits; this time they are, 59534 can give the same fingerprint.. Finally, Part (c) illustrates the constant time process used to shift one place in the text stream and calculate the next fingerprint using information from the current fingerprint. It demonstrates with actual text strings how the Rabin-Karp Algorithm uses Horner's Rule to bring information from the former fingerprint calculation forward to calculate the subsequent fingerprint. The process begins with the current set of digits 16439, the first digit of which we need to remove when sliding our window to the right one digit. Doing this accounts for the -1x10000 part of the expression. After removing the beginning digit of one, we next want to shift the four digit number of 6439 to the left by one position. Hence, the x10 part of the expression, which would have left us with 64390; but, we decided to performed a modulo operator of 13 on the previous two numbers leaving us with 40 (i.e., 70-30.). Finally, we add 8, which leaves us with a value of 9 (mod 13). This is an excellent diagram for highlighting the main processes and components of the Rabin-Karp.

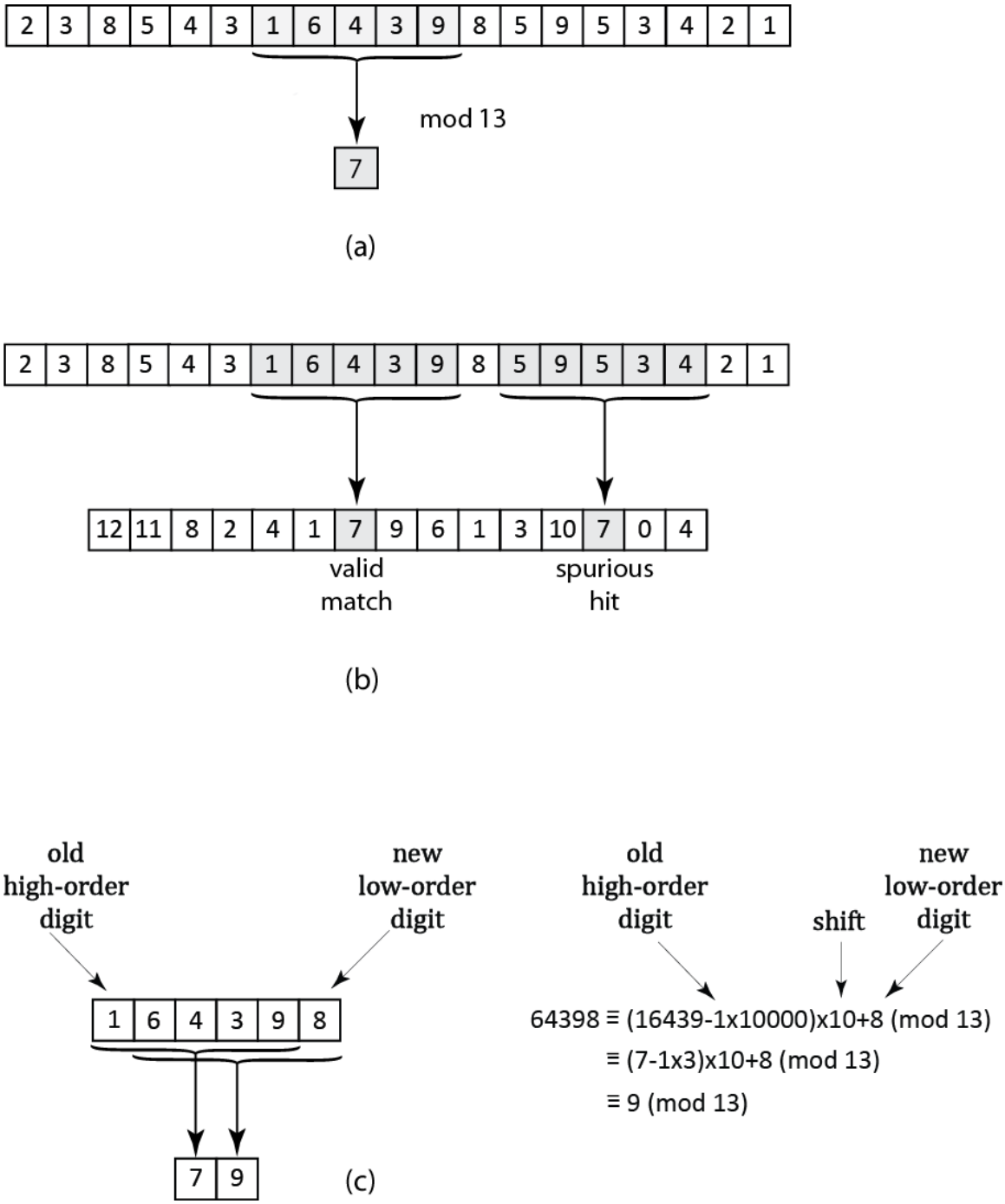

Figure 1: The Rabin-Karp Algorithm. Based on figure 32.5 of (Corman, Leiserson, Rivest, & Stein, 2001, p. 913).

1.6.3 Fingerprints in LevelDB

Using Rabin-Karp principles to make a two-step process requires that we *preprocess* a text file by calculating its fingerprints at every position and saving them for later look-up during step two, the search. We created a LevelDB database for this purpose. LevelDB is an open-source key-value database developed at Google that we describe later.

Originally, we thought a fingerprint would be our primary key. Unfortunately, we soon discovered a duplicate-key problem for LevelDB related to a fingerprint being neither unique nor distinct, making it impossible to serve a role as primary key. Therefore, our primary key consists of a fingerprint's bytes followed by the bytes for file position.

1.7 Motivation

Few, if any, local text search applications in widespread use have two steps. Even though Google, the biggest of them all, is two-steps, it is not very useful for searching local information content. Most current local text search applications, like UNIX's *GREP*, run on-line. This means that work done in a current search is independent of, and sometimes a replication of, calculations and other work done by a previous search. It therefore looks like a very promising direction to build a two-step-text-search-engine to take advantage of the amortization of building a database versus the repeating calculations with a one-step application like *GREP* for local search problems.

Fortunately, researchers have studied the string-matching problem for many years especially in biological areas searching for patterns in DNA. (Gusfield, 1997) They list a huge body of literature describing many different solutions to this problem and demonstrating the naïve method is not optimal. One of the biggest reasons making this method inefficient is that it does not keep any information from the previous comparison when it starts a new comparison. Keeping some information may be valuable. We will show later how it can be a source for more efficient search methods.

Another source of improvement arises from methods that use a two-step process. In these methods, step one is referred to as a *preprocessing* step; and, step two` as the

matching step. (Corman, Leiserson, Rivest, & Stein, 2001) Typically, step one analyzes **T** to produce some sort of index that we save and use later in a matching step. One very useful advantage of having an index is that we can repeat many matching steps to locate occurrences of *any* number of patterns without needing to build a new index each time.

1.8 Description of Work

This section briefly summarizes the work we performed building this application itself. It also gives an overview of how we obtained the performance results comparing our approach and GREP.

Dismantling and reassembling the Rabin-Karp algorithm was an interesting challenge. We used C and C++ in Visual Studio to program and debug our application. Hence, the executable is small (<60k) and fast. We did not use any Windows exclusive functions, APIs, or headers that would preclude it from working in UNIX. Even LevelDB has a C++ API for both operating systems that is easy to switch, making operating on UNIX a very real possibility.

Our work began developing an on-line Rabin-Karp application that worked from the command-line. During this time, we created and implemented a command-line parameter list (See Appendix C) that we kept up-to-date whenever we added a new parameter. Once that was functioning, we divided the application into two parts and began building the first part, including an Index database. After being side tracked trying to use Oracle DB for several months, we switched our database system to LevelDB. Within a week or so, we had LevelDB storing and retrieving our information. This success meant we had completed building our preprocessing step. We then started work on the matching-step. We modified a few parameters that let our application put the positions of all occurrences of the pattern in a results file. With this finished we tested it searching for random phrases from the collection of Mark Twain. (Twain, 2009) With everything tested and verified, we began our performance test using the bible as our text document we downloaded from the internet (The Large Canterbury Corpus, 2001). We constructed our experiment text document by repeating the bible 100 times and saving

the results to a text file. These both gave us a small test text file with one bible for testing, and a much bigger file to calculate performance with 100 bibles. After finding our test search strings we ran GREP for each of them. Later, we compare GREP's time against searching for the same strings using our application. Each time we ran our application, we changed a different parameter while holding others constant.

1.9 Document Layout

This document will introduce and discuss all objects used by our approach. It also highlights many of the significant changes we made to Rabin-Karp's original algorithm morphing it from its original on-line application into a two-step search process. Finally, it discusses some of the major issues we encountered along the way and how we solved them. Readers not familiar with string notation and manipulation can review Appendix A for the needed nomenclature. Otherwise, readers can skip directly to Chapter 2: The Rabin-Karp Algorithm on page 17. Furthermore, readers familiar to Rabin-Karp fingerprints can just scan the headings of Chapter 2.

The discussion develops the concepts in the following order.

Chapter 2: Begins by formalizing the concepts associated with the Rabin-Karp Algorithm. Understanding this algorithm will help with understanding the application we built using it.

Chapter 3: Examines other work in the area of string search in general and Rabin-Karp's fingerprints in particular. This literature review highlights papers done by other authors that have something specific to say about some or all components of our approach.

Chapter 4: Describes the theory and practical implementation of our approach. In particular, we discuss the constituent parameters affecting the results such as modulo and radix. Then we use these parameters to discuss characteristics of a fingerprint such as how we calculate it once for strings like the search string, and recursively for each subsequent substring in a Text File. We also examine some of the properties that not

only influence the fingerprints, but also have themselves influenced by the fingerprint. The chapter concludes that the approach is implementable using a C++ exe program with a size of 58,880 bytes.

Chapter 5: Describes how we designed our experiments, what experiments we conducted, what we needed to support the experiments, and what resulting performance we received for each experiment. In cases where it was obvious and clear why certain results appear, we discuss generally how and why we achieved those results. Finally, we discuss the various perspectives on the results to give an overall picture of how our experiments compare to GREP.

Chapter 6: Discusses our conclusions and recommendations for future work

Appendix A: Formalizing Characters, Strings, and Search

Appendix B: ASCII Table

Appendix C: Command-Line Parameters

Appendix D: Seek Times for each Radix/Modulus Combination

Appendix E: Diagrams of Actual Versus Spurious Hits

Appendix F: Line Graphs of Performance (hits/ms)

Chapter 2: The Rabin-Karp Algorithm

Before we get into the modifications we make to the Rabin-Karp algorithm, we use this chapter to give a formal description of the existing algorithm. We therefore begin with an overview of Rabin-Karp. In it, we expand on the description from the introduction and provide names for techniques it uses like Horner's Rule. We present the Rabin-Karp Algorithm to help highlight some issues that might otherwise have remained hidden under a cloak of complexity. The degree of familiarity offered by these opening concepts helps set a stage for a formal description featuring these techniques and concepts as well as expanding on terms and concepts already mentioned earlier. Finally, once we dispense with a formal definition, we give a detailed example of all processes. Then, we have the tools needed to move to the next chapter that describes in detail changes we made to customize the Rabin-Karp algorithm to meet our Thesis. We based the information presented in this chapter from descriptions given in (Corman, Leiserson, Rivest, & Stein, 2001) and (Gusfield, 1997).

In what follows, it may seem that we are over-emphasizing Rabin-Karp's fingerprint. The reason it would seem this way is the prominence of the fingerprint in the overall algorithm. In fact, we will even expand our earlier mention of how a fingerprint serves as both a hash value for our database and as a heuristic for matching strings. While these particular topics are somewhat removed from the central theme; they are nevertheless, important concepts to understanding the foundation of our work.

2.1 Introduction

The Rabin-Karp algorithm uses modulo arithmetic, Horner's Rule, and a number of other innovative techniques to calculate a fingerprint (decimal number.) We add several features of our own to make it perform even better. Rabin-Karp algorithm follows a specific path in its analysis. First, it calculates a fingerprint (denoted as **p**) for a pattern **P**. Then, it iterates through the text file **T** for every offset or shift (**s**) and calculates a fingerprint representing a pattern-length substrings in **T** beginning at s. We denote this fingerprint as $\mathbf{t_s}$ (for the fingerprint in the text file **t** at shift **s.**) As we demonstrate later,

comparing the fingerprint of a pattern to each of these text file substring fingerprints is extremely efficient. Mostly because comparing integer values for equality is a simple one-step numeric process. In addition, small fingerprints (i.e., 32-bit integers) allow us to take advantage of the speed of small integer arithmetic on modern processors.

When the values for two fingerprints are not equal, we are certain the characters in the pattern do not match the characters in the respective substring of the text file. When two fingerprints values are equal, we know (only), they occur in the same hash bucket. At this point, we do not know the respective strings match because of a possibility that two different strings could produce the same hash. Even if we work diligently to reduce the possibility of hash collisions, we are still in a position of only knowing that the corresponding and respective strings *could* match. Using large modulus values will help reduce the possibility of a collision, but we can never remove it entirely. Hence, a pattern's fingerprint value equaling a substring's fingerprint value is at best an extremely good heuristic indicating the *possibility* the two underlying character strings will *match*. We must subsequently confirm or deny an *occurrence* by performing a character-by-character comparison of **P** with **T[s, … ,s+(PL-1)]**.

In summary, the efficiency of the matching-step in the *Rabin-Karp algorithm* comes about in two ways. Primarily, it eliminates a need to examine each character in the huge text corpus by the relatively quick method of comparing integer hash values for substrings. Next, it reduces the total amount of character-by-character comparisons required to confirm occurrences. In other words, it only needs to compare the characters at locations in **T** where the fingerprint value of the pattern-length substring in **T** is equal to **P**'s fingerprint value. To verify it commences a character-by-character comparison that it abandons whenever we discover mismatching characters.

In the remainder of this Chapter, we present the Rabin-Karp Algorithm from several different perspectives. We begin with a rhetorical description, even the above paragraphs add to this description; but we will expand it below. Next, we formalize the Rabin-Karp Algorithm by giving formal definitions to all its parts that, in turn, contribute to a formal

definition for the whole. Next, we give the Rabin-Karp Algorithm as an algorithmic listing similar to the Brute-Force listing given earlier. We then examine the processes occurring at each line in the procedure. Finally, we give a detailed example.

Even though in previous chapters it seems we have been covering many of the concepts needed for our analysis, in some discussions we were informal, in some we were brief, and others we ignored altogether. Now, we need a set of formal definitions that follow, so we can present a rigorous description of the technical details of Rabin-Karp's approach. We present them here as one collection a reader can refer to rather than jumping around the report. We have adjusted some issues and added new ones in these definitions. We need to postpone elaborating on some of these formalisms until later when a more appropriate context appears. Nonetheless, the details we do present will suffice for now to demonstrate the main concepts behind the Rabin-Karp's technique.

- Appendix A defines an alphabet (Σ.) as a set containing $|\Sigma|$ characters. The Rabin-Karp approach assumes a user has resolved any implementation issues regarding their choice of alphabets. An issue we do have with alphabets Σ and text search is an earlier assumption that it is customary to treat an alphabet's *radix* as $|\Sigma|$. While it is usual to define a radix for an alphabet as the number of characters in it, as the traditional Rabin-Karp Algorithm does, in a few paragraphs below we will challenge this concept by introducing our approach to radix and the number of characters in our alphabet.
- Appendix A also describes characters in general as coming from the ASCII character set that have corresponding codes. Since we made such a huge modification, we must now formally connect a character's code to a concept

called *radix*. Since we treat radix differently, we begin describing Rabin-Karp's original intention, followed by our application's meaning.

- Rabin-Karp Algorithm: Let **r** represent the number *of characters* in the alphabet **Σ** (i.e., **r** = |**Σ**|); and refer to **r** as the *radix* of the alphabet. While our algorithm can accommodate this definition, we extend it as follows.
- Our Modification: Let radix refer to the base of a number system created exclusively for calculating fingerprints. A user establishes a base by looking from the inside out. They look at the size of possible fingerprints with a particular base, only after examining how many characters they want to place in the same hash bucket. In fact, both approaches accommodate the same number of characters. Our approach however has more to do with the hash value calculation of the fingerprint. When getting a code for any character, we use the following formula $b_i = c_i \, mod \, r$ For instance, say we were using ASCII and a radix of eight. The above equation results in having eight different codes each with sixteen characters. (8*16=128) An important observation about his approach comes by realizing character codes are vital for lexographic sorting. The implication for this example is that since each set of sixteen characters have the same code, they all share lexographic positions as well, making sorting a rather messy affair. Essentially, we must use great care when choosing a radix that ends-up having many characters with the same code. At the same time, this modification is important because it gives us more control over later parts of the system; particularly with calculating the minimum length a search can accommodate.

- For any string **S**, let **S(i)** denote a function that returns the integer code (**b_i**) representing the character **S[i]** (note parenthesis versus square brackets.) Both terms use a parameter **i,** which represents the position of the character in **S**. Recall that offsets begin from the left.
- Let **H(S)** be a hash function that calculates a fingerprint for a string **S** of length **k**. **H(S)** is defined with the equation below. A few observations are in order. First, we typically denote a variable for a particular string's fingerprint as the string's letter descriptor in lower-case (using subscripts when required.) Here we see the fingerprint for **H(S)** is equal to **s**. Second, in the previous paragraph we made a point showing all **S(i)** will be less-than-or-equal-to **r**. This means our hash function **H(S)** produces fingerprints that are part of base **r** number system.

$$H(S) = s = \sum_{i=0}^{k-1} r^{k-i} S(i)$$

Equation 2: Hash functions produce Fingerprints (base r)

- Rabin-Karp observed that the above equation for H(S) can be calculated using Horner's rule thereby not only keeping the number of multiplications and additions linear, but also keeping the intermediate values small. (Gusfield, 1997, p. 79) In mathematical terms, Horner's Rule converts the above equation to the following,

$$H(S) = S(k-1) + r(S(k-2) + r(S(k-3) + \cdots + r(S(1) + rS(0))))$$

Equation 3: Calculating hash value using Horner's Rule

For example, in base 10, Horner's Rule multiplies the current value by ten; which is like shifting all digits left one position. Finally, it puts a digit in the one's position by adding the character code for the new character; we will illustrate this concept using a very simplistic example.

Consider an alphabet consisting of the ten decimal digits **Σ**= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. In addition to the digit's character, each character also has a code. To keep it simple, we will deliberately assign the code a value corresponding to the character's integer digit. Consequently, the character '3' has a code that equals an integer 3, and so on. It is important with this example to remember that **S** is just a string of characters even if the string just happens to look like some number. Thus, keep in mind in the following discussion that fingerprints are not strings, but are actual numbers. The difference is significant because codes are numbers that we can compare directly, whereas strings require we perform a character-by-character comparison. We selected this simple example to make it extremely easy to see how we convert a character as an integer in the fingerprint calculation. We even use different fonts to help highlight the difference between the character 3 and the integer 3.

Since the objective of this example is simple, we define radix using a traditional approach of the number of characters in an alphabet. Hence, the radix **r** for this alphabet is |**Σ**|=10. Incidentally, this means our fingerprint number system is base 10 keeping in line with our simple theme. Let **S**=123; then, **SL**=3, and the fingerprint (**s**) for **S** is calculated as follows.

$$s = S(3) + r \times \big(S(2) + r \times S(1)\big)$$

$$= 3+10\times (2 + 10\times 1)$$
$$= 123$$

Recall we have a pattern **P** of length **PL**, and a text **T** of length **TL** where **PL** is very much smaller than **TL**. The Rabin-Karp algorithm is centered on the idea that if there is an occurrence of **P** starting at position **l** of **T** then their fingerprints will be equal (i.e.,

H(P) == H(T_i)). Unfortunately, the converse is not true. To make our earlier proclamation clear, we formalize it as follows:

- When two fingerprints are equal, we do not know if their respective strings will match. We therefore use a character-by-character comparison process to verify whether the corresponding strings match. In the meantime, we know for sure if two fingerprints are *not equal*, the corresponding strings will *not match*. Therefore, there is no need to verify it.

2.2 Rabin-Karp Algorithm

Briefly, the *Rabin-Karp algorithm* converts the string-matching problem into a numerical approach that uses simple integer arithmetic to calculate a numeric fingerprint. We will eventually use this numeric fingerprint as a hash value representing a string of characters that is also an excellent heuristic for finding matches in corresponding character strings. Although the original version uses a *preprocessing time* of $\Theta(PL)$, and has the same worst-case running time the *Brute-Force method*, the Rabin-Karp algorithm reduces the probability of this worst-case running time to be so small that the time for the *matching* step actually becomes linear $\Theta(TL-PL+1)$.

Even though it has a preprocessing step of $\Theta(PL)$, this time is so small compared to the text being searched that we regard the following version of the Rabin-Karp algorithm as *on-line*. We do this because one of our enhancements later will add a more substantial preprocessing step of building a database of all fingerprints in a text file, which we will greatly expand on later. In any case, we present the following procedure to introduce the algorithm and demonstrate its calculations and processing. Once again, we will expand the details of this algorithm later in the report. The inputs are the text file **T**, the search pattern **P**, the radix **r** (typically equal to $|\Sigma|$, but not always,) and the modulus **q** a prime number.

```
Rabin-Karp-Matcher(T, P, r, q)
1 TL ← Length(T)
2 PL ← Length(P)
3 h  ← r^(PL-1) mod q
4 p  ← 0
5 t0 ← 0
6 for  i ← 0 to (PL-1)
7    do p ← (rp + P(i)) mod q
8       t0 ← (rt0 + T(i)) mod q
9 for  s ← 0 to TL – PL
10   do if p == ts
11        then if P[0 … PL-1] == T[s … s+(PL-1)]
12             then print "Pattern occurs at position" s
13        if s < TL – PL
14             then ts+1 ← (r(ts – T(s)h) + T(s+ PL)) mod q
```

Algorithm 1: The original Rabin-Karp algorithm is basis for our work.

The above procedure works as follows. Recall we demonstrated earlier when we introduced characters that they could be interpreted as a symbol or an integer; the procedure interprets all characters as radix-**r** digits. Recalling the use of round brackets for a code and square brackets for a character, the term **P(i)** in line 7 refers to the character code (i.e., an integer) at position **i** in the Pattern rather than its symbol;. Line 3 calculates **h**; a constant used later in Line 14. The variable **h** represents the value of the high-order digit position of a **PL**-digit window. Lines 4 through 8 compute a fingerprint for both the Pattern and the first position of the text file. The method uses Horner's Rule that multiplies every subtotal by **r** before adding the new character. This multiplication by **r** shifts the current value to the left by one digit (i.e., the radix **r**.) Since **r** is the base of our fingerprint numbering system, the effect of this multiplication is to create an empty slot at the right hand end of the number ready to accept another base **r** digit. Adding the value for the new character to this shifted number, places the new character's digit at the right hand most digit of the number. This loop continues until our two fingerprints, **p** and $\mathbf{t_0}$ are **PL**-digit numbers representing their respective strings.

The second part of the algorithm begins with a *for-loop* of lines 9 through 14. This loop iterates through all possible shifts **s** in the text file. In fact, lines 9 -12 are very similar to

the naïve text search algorithm presented earlier. The only difference is the addition of line 10 that checks to see if the fingerprint values equal before checking the strings character by character for a match. Line 12 prints locations for matches. Line 10 is one place where the Rabin-Karp algorithm shows a vast improvement over the naïve method because it reduces the need to compare characters to only those times when their respective fingerprints are equal.

Finally, Line 13 checks to see if the **for** loop on line 9 will be executed again. If so, the algorithm uses another innovation on line 14 that computes $\mathbf{t_{s+1}}$ **mod q** from the value of $\mathbf{t_s}$ **mod q** in constant time using Horner's rule.

2.3 Modulo Arithmetic

In practice, fingerprints could become huge values parameters become too large, certainly too big for simple 32-bit arithmetic. This problem has a potential to render the algorithm useless. That is until, with yet another stroke of ingenuity, Rabin-Karp introduced an idea of using modulo arithmetic in its fingerprint calculations to keep the resulting value within an arbitrary range. They use it in line 3, 7, 8, and 14 in the Algorithm listing above. Modulo arithmetic allows one to reduce a large number of objects into a finite searchable space. Its power comes from allowing one to reduce at any time; which is precisely what we did above in Figure 3 for Horner's Rule. The other nice thing about modulo arithmetic is that the heuristic and equality features of fingerprints still hold in this environment.

As mentioned earlier, but until now not demonstrating; we can finally see how the addition of modular arithmetic has led to the fingerprint being a *heuristic* as to whether or not the pattern appears at a particular location. We can also observe how efficient the heuristic can or cannot be since if $H_q(T_o) \equiv H_q(P)$ we must do a character-by-character evaluation on the substring at that location *o* to verify whether or not there is actually an occurrence of *P*. These observations lead to the following definition.

- If $H(P) \equiv H(T_l)$ but P does not occur at position l of T, then we call this a *false match* or a *spurious hit*.

The key to implementing the Rabin-Karp algorithm then becomes choosing a modulus q small enough that the arithmetic is kept efficient, yet large enough that the probability of a *false positive* between P and T is kept small. While easy to use and having the ability to keep fingerprint*s* values smaller than 32bit integers, modulo arithmetic is a very expensive function that we try to minimize the use of in our implementation. We elaborate on these shortcut procedures in the following Chapter.

Chapter 3: String Search Literature Review

3.1 Introduction

Since Rabin and Karp introduced their fingerprint calculation algorithm, it has enjoyed much success. We list a variety of vastly different applications below. Our search, however, did not yield any cases where researchers used the same approach as ours. In fact, even with string searching itself, the algorithm works so efficiently on-line there seems no purpose in trying to improve its impressive time and space numbers. Actually, it remains a leading on-line string search algorithm mentioned in many algorithm and string processing textbooks; we illustrate four textbooks below.

3.2 Textbook References

All of the authors of following four textbooks say they aimed them at graduate and upper undergraduate level courses. Consequently, the last three textbooks combined, contain an exhaustive descriptions of all areas of stringology. Whereas the first textbook on algorithms had the best description of the Rabin-Karp Algorithm, hence, we relied heavily on it for our work.

3.2.1 Introduction to Algorithms

The textbook Introduction to Algorithms (Corman, Leiserson, Rivest, & Stein, 2001) is a "must-have" for all computer science students, as well as some professionals, researchers, and teachers. It gives an wide-ranging primer for studying modern computer algorithms with. It covers many algorithms in considerable depth and mathematical rigour, and includes a C-like pseudo-code listing for each. Additionally, it has coherence with respect to cross-referencing algorithms with one another. While some readers may use the book as a "cookbook" for the most popular algorithms, it offers much more. With the objective of targeting an audience from students to researchers, and everything in between, it presents their material so the algorithm's design and analysis is accessible to all levels of readers. Somehow, the authors did not sacrifice depth of coverage or

mathematical rigor while keeping explanations elementary. (Corman, Leiserson, Rivest, & Stein, 2001)

This textbook provided us with a description of how to implement the Rabin-Karp algorithm. In fact, we use one of its diagrams later to demonstrate the concepts behind a fingerprint.

3.2.2 Algorithms on Strings, Trees, and Sequences

The textbook Algorithms on Strings, Trees, and Sequences demonstrates (Gusfield, 1997) how to combine computer science with molecular biology using string computation as a common thread. They provide a very rigorous treatment of algorithms for character strings and molecular sequences. They give formal definitions from both sciences for fundamental techniques and objects. They present an exhaustive general-purpose description and analysis of all subject matter so others may apply bits and pieces from their tapestry of algorithms and proofs.

This rigorous treatment of the field of deterministic algorithms operating on strings and sequences was impressive because it gives complete proofs of behaviours like worst-case time, correctness, and space. Indeed, later we use some of their formal definitions inspired they formality we used to develop our work.

We used this book as a guide to our formal definition of characters and strings.

3.2.3 Algorithms on Strings

The textbook Algorithms on Strings (Crochemore, C., & Lecroq, 2009) illustrate the correctness proofs for fundamental text processing and matching algorithms and methods for evaluating their performance. They develop their topic by focusing on a generic sense of organizing text in a computer environment with limited memory and slow hard drives. Their topics create an algorithmic and technical framework required in fields like information retrieval, automatic indexing for search engines, the compression of text, and more generally the practical software system, including its edition and its treatment. This framework applies to a plethora of fields such as pattern matching, automatic processing

of natural languages, treatment and analysis on genome sequences, analysis of musical sequences, safety and security related data flows, and management of a textual databases. Their basic approach is to sew their collection of basic algorithms together to create a combinatorial underpinning for all string searching activities like pattern matching, indexing textual data, comparing texts by alignment, and searching for local regularities. (Crochemore, C., & Lecroq, 2009)

We suspect this to be on most Stringologist's bookshelves just as it sits well-worn on ours. It mentions the Rabin-Karp algorithm in the context of string searching using a hash function.

3.2.4 Flexible Pattern Matching in Strings

The textbook Flexible Pattern Matching in Strings (Navarro & Raffinot, 2007) presents the string-matching problem from a practical point of view. While many algorithms have extremely good theoretical complexity and space scores, some perform badly in practice; often slower than the naive approach of checking every character at every position. According to the authors, their impetus for the book was to focus on on-line algorithms and implementations performing best in practice. As such, they cover topics for matching simple, multiple, and extended strings; moreover, they also cover regular expressions. In most topics, they present techniques for both exact and approximate matching. They provide an in depth description of the most practical algorithms, and promise a normal programmer can implement their approaches in a few hours. Like the previous textbook, this one mentions the Rabin-Karp algorithm in the context of using a hash function for string searching. (Navarro & Raffinot, 2007)

3.2.5 Modern Information Retrieval

The textbook Modern Information Retrieval (Baeza-Yates & Ribeiro-Neto, 1999) has an interesting twist. Six of the leading researchers in their respective fields write six of its fifteen chapters. Yet, the content is both modern and cohesive throughout with a carefully designed content and organization. While its coverage is broad, its detail contains the richness many textbooks lack. The book is both rigorous and complete, and

approaches modern Information Retrieval a computer scientist's point of view. (Baeza-Yates & Ribeiro-Neto, 1999)

We used this textbook to provide background information for some of information we used in Chapter 1 of our work. Recall, in Chapter 1 we explained our motive and justified our thesis.

3.3 Selected Paper References

Rabin-Karp fingerprints have found their way into a broad range of areas, not just pattern matching. The following list of papers show a selection of applications that demonstrate their use in practical areas such as data compression and network flows, as well as improved hashing.

3.3.1 Data Compression

Rabin-Karp fingerprints have found success in many fields of stringology. For example, we begin with the field of compressing data. Papers for other fields follow.

3.3.1.1 U.S. Patent Compression Method

Two employees at Lucent Technologies Inc. invented and patented an on-line method and apparatus for achieving relatively low compression ratios in streaming data. Their approach is very interesting because it uses both fingerprints and a hash table. Their input is any data stream, which they divide into equal-length "blocks." The blocks usually range between 40 and 1000 bytes. They process their streaming data by calculating a fingerprint for every block (using Rabin-Karp's equation). Next, they place every fingerprint, location pair into a memory-resident hash table. It does not save these hash tables. In fact, the data stream's "process history," defines how many blocks they keep in this hash table. The history length is a number of blocks one can go backward in the data stream looking for the longest possible repeating string. Their purpose is to compress this data by replacing repeating strings with a pointer to the nearest location of that string. Readers will recognize that the sliding window process is different than Rabin-Karp's window that slides one character at a time using the last fingerprint to

calculate the current one. The block approach used in this paper must calculate fingerprints from scratch at every block This document was very interesting despite its legalese. (Bentley & McLiroy, 2003)

3.3.1.2 Longest Common Extensions

At a level of calculating fingerprints, the process in this paper is similar to a previous paper's compression process. (Bentley & McLiroy, 2003) The previous paper calculated fingerprints for a set of fixed-length blocks. This paper calculates fingerprints for a set of variable-length blocks that just happened to be prefixes. A similarity emerges when each application jumps ahead to its next block getting a fresh set of characters to calculate the next fingerprint. The fingerprint independence from one block to the next deprives these processes of the performance enhancing benefit from the faster constant time sliding window calculation introduced in (Karp & Rabin, 1987). Nevertheless, this paper is interesting and offers some useful insights. The following gives a brief overview of the paper in terms of the mechanics of fingerprint operations.

This paper is more theoretical than mechanical in that the authors study the time-space trade-offs for the longest common extension (LCE) problem more than how to implement them. In particular, they focus on the space used for the data structure versus the worst-case time for answering an LCE query. While they prove a large number of bounds and times throughout the report, we feel it better for interested readers to consult the original for nomenclature and other issues. At the same time, one very important observation is that they claim to have not only provided the first smooth trade-offs for the LCE problem, but also they matched previously known bounds at the extremes when $\tau = 1, or\ \tau = n$. (Bille, Gørtz, Sach, Vildhøj, Kärkkäinen, & Stoye, 2012)

From a broad perspective, the LCE process works in the following manner. For any string of text, T, the longest common extension of suffix i and suffix j, denoted LCE(i, j), is the length of the longest common prefix of the two suffixes of T starting at position i and position j and going as far as possible. The LCE problem is to preprocess T into a compact data structure supporting fast longest common extension queries.

Interestingly, the LCE problem is a basic primitive appearing as a sub-problem in a wide range of string searching, indexing and matching issues such as compression, cyphering, exact and approximate string matching, exact or approximate tandem repeats and computing palindromes. In many of these applications, the LCE problem is the computational bottleneck. Hence, there is a clear need for enhancements such as those presented and proven this paper. In short, using fingerprints as a heuristic rather than the traditional character-by-character comparison is enough to produced huge rewards for this on-line application.

Two parts of this paper are very interesting topics: (1) approximate string matching and (2) the Aho–Corasick automaton. While neither is immediately relevant to our application, in the long term, both may contribute to its computational and performance improvement. Aho–Corasick is a string-matching algorithm that contains a finite set of strings. It locates any of those lines within an input text. The second interesting issue deals with approximate string matching. This algorithm allows up to k mismatches to occur at the end of the block. Again, not very useful in our immediate case, but some of the ideas may contribute to making our algorithm deal with approximate strings.

3.3.2 Improved Hash for String Matching

This paper is an example of the kinds of modifications the Rabin-Karp algorithm undergoes in the name of efficiency. The authors prove they "can accelerate the computation of fingerprints by bitwise operations." (Fuyao, 2009, p. 1) They first demonstrate that the "Rabin-Karp algorithm is still inferior to other string matching algorithms in practice." Then, they demonstrate that "the reason is the complex arithmetic operations rather than checking for false matches that circumscribe the algorithm's performance." (Fuyao, 2009, p. 2) To improve the situation they make two insightful modifications. First, they replaced Rabin-Karp's set of arithmetic operations for calculating a fingerprint with an equivalent set of bitwise operations. This replacement works magnificently as long as **m <= w** *(*where **m** is length of pattern and text window, and **w** is the length of a machine word.) Their second modification extends

the first for situations where **m > w**. Specifically, they take B *least significant bits* (LSB) from each character, which bitwise operations does efficiently.

Whereas the Rabin-Karp algorithm uses a character's entire code in its arithmetic operations that calculate a fingerprint; their version uses a constant number (B) of low order bits from a character's code and bitwise operations to speed up the string matching process considerably. The authors prove that the probability of a hash collision is very low, and the complexity of running time on average is linear. (Fuyao, 2009)

At first, the performance improvements demonstrated by these simple alterations were attractive for our work. Unfortunately, we show later that the percent of time calculating fingerprints is insignificant compared to building the index database. In the end, since the modifications will not be significant compared to the effort implementing it, we decided not to add these to our application.

3.3.3 Threat Signatures from Network Flows

ARAKIS is a CERT Polska project sponsored by NASK (Research and Academic Computer Network) a research and development company in Poland. According to their website, ARAKIS is a project "that aims to create an early warning and information system concerning novel network threats. The system developed as part of the project focuses on detection and characterization of new automated threats with a focus primarily, though not only, on exploits used in the wild, not malware. Currently the system detects threats that propagate actively through scanning. The public dashboard of the project shows a snapshot of network activity observed by the system. (NASK (Research and Academic Computer Network), 2012).

The system generates network threat signatures used in intrusion detection and prevention systems. Currently, this is a mostly manual process, thus prone to errors and slow. The system described in this paper uses Rabin-Karp fingerprints to not only detecting network packets that are an attack threat but also to extract a network threat signature in one step, thus is fast and less error prone. (Kijewski, 2006)

This paper demonstrates the breadth of topics using Rabin-Karp fingerprints.

Chapter 4: Our Application Using Rabin-Karp

4.1 Introduction

The concepts for this project involve a greater depth of understanding than first meets the eye; especially considering how simple the 14-step Rabin-Karp Algorithm shown earlier appears. This Chapter is about describing how we modified, inspired by this depth of understanding, Rabin-Karp to create our own text search application. As the number of sections in this chapter attests, we made a great number of modifications, many of which helped speed-up individual steps. We discuss these later. However, the biggest modifications we made were NOT for speed. Instead, they were, (a) adding both a preprocessing step and a matching step, and (b) creating a command-line application. Following this section, we first present topic (a) then, we present the other topics before finishing the chapter with a section on topic (b.) The main reason for placing topic (b) at the very end is, it provides a setting that allows us to not only review each parameter required to make our application run, but also to connect these parameters to their underlying techniques that we just presented.

At the same time, other modifications we make have big impacts on the technique's performance. For instance, we talk about issues regarding the selection of LevelDB database, fingerprints, Radix, and Modulus. We try to keep the discussion focused exclusively on covering new capabilities. Unfortunately, sometimes we must not only explore issues for situations demanding a background context, but also digress to explain how the application works and why it requires certain parameters. We address two of these types of reasons in the next several Sections. Their content does not fit with other topic lines, so we placed them here to describe how, the first reason (sorting fingerprints) helped our modification, and the second one (Endianness) stood in our way from making any progress unless and until we fixed the problem.

4.2 Important Side Effects

An important issue needs a brief description before we get started on the details. We are talking about several very important side effects of our Algorithm's two-step approach: (a) collections, and (b) sorting. They are important to note because of their far-reaching effects and incredible benefits.

The first issue concerns a benefit we obtained by going from a one-step to a two-step process. Now, our algorithm always returns a file with a collection of locations (file positions) after every run. Dealing with a collection allows us to process and analyze an entire group of similar entities mostly key byte arrays; and not just during the actual run itself, but also during post-processing analysis including calculations that produce some of the useful statistics we show later.

Having a collection requires a container. We have two containers, a *database* for persistence, and a *vector* for our C++ programming. Elsewhere we described how LevelDB automatically uses its key byte array to sort database keys lexicographically. This means LevelDB automatically sorts our keys first by fingerprint then, if two consecutive fingerprints are equal, by file position. Since we use iterators to traverse our data, we perform our analysis by always skipping one position at a time in a forward direction. An immediate benefit of this appears when we are building vectors that hold our collections, LevelDB has already sorted our source data in an order we need for analysis and reporting. Having data in our vectors already sorted the way we need them allows us to do some processes rather quickly like deleting false-positives before the verification step in several of our matching algorithms below.

4.3 Endianness

During development, we discovered an unplanned modification by uncovering a bug in LevelDB. We made this modification because of a concept called *Endianness*. In a simple sense, this refers to how a hardware configuration forces an operating system to order numerical bytes in memory. The changes to our application described below work

automatically behind the scenes, and do not rely on any input parameters. Briefly, the application performs a quick test on a simple integer in memory, and determines whether it is running on a Little or Big Endian host platform. With this knowledge, the application automatically adjusts the byte order for its integers. We explain the details of Endianness and our modification in more detail below.

Before we explain the problem, however, we are compelled to mention we only have a problem because we manipulate integer values like fingerprints and file position, on a byte-by-byte level for several reasons. Consider a primary key in an Index Database as a case in point. Elsewhere we explain that a primary key in LevelDB is a *byte array* that looks and acts the same as a string. LevelDB will not accept any other data type for a primary key value than a byte array. In any case, our application uses a *primary key* that is also a composite key consisting of bytes from a fingerprint value followed by bytes from a file position value. We built functions to transform the bytes of numeric values to a composite key, and back from the composite key to numeric values. All of this low level byte manipulation brought the Endianness to the fore.

After considerable investigation, we discovered that our Index Database was not sorting its records in the order suggested by their fingerprint integer values. After significant investigation and debugging, we found that since we were using Windows, at the hardware level (i.e., Intel-based system), it was actually using a native format for storing integer values called "*Little Endian*." Whereas, LevelDB internally assumed data was all stored in "*Big Endian*" format; this is probably because LevelDB developers originally built it for UNIX hardware (that uses Big Endian) and only recently 'ported' it to work on Windows hardware. In any case, we had discovered the bug and now set out to fix it.

We derived the descriptions and examples explaining Endianess from the web page. (RapidTables.com, 2011) We also used an Intel White Paper for further material. (Intel Corporation, 2004) Before beginning our description, we need to define several terms that play a significant role in Endianess. The first is *least significant byte* (LSB), and the second is *most significant byte* (MSB).

- **LSB** is a byte representing the smallest quantity or weight of all the bytes making up a number. (Intel Corporation, 2004).
- **MSB** is a byte representing the largest quantity or weight of all the bytes making up a number. (Intel Corporation, 2004)
- The name ***Little Endian*** literally means little end first, meaning the LSB is stored on the left hand end of a number's multi-byte value. (RapidTables.com, 2011)

In other words, Windows rearranges the bytes in a word so the LSB is at the left-hand end and the MSB on the right hand end. It turns out that Little Endian is not an ideal format when it comes to sorting numbers inside an application. Unless, a programmer writes a sorting routine to accommodate the bytes being in backward order (which LevelDB does not), sorting is backwards.

The following tables describe how a simple integer such as, "**`0x0D0C0B0A`**" is stored in Little Endian format.

Address	Data	Example
0	byte0	0A (LSB)
1	byte1	0B
2	byte2	0C
3	byte3	0D

Table 1: Little Endian numbers are stored from least significant byte (LSB) in low memory address to most significant byte (MSB) in high memory address.

- *Big Endian* sorts in lexicographic order. The reason for this is predictably, that the name ***Big Endian*** literally means big end first, meaning the MSB is stored on the left hand end of a number's multi-byte value. (RapidTables.com, 2011)

Most UNIX hardware stores integers in Big Endian format. The following table shows how the same integer as above "**`0x0D0C0B0A`**", is stored in Big Endian format.

Address	Data	Example
0	byte0	0D
1	byte1	0C
2	byte2	0B
3	byte3	0A (LSB)

Table 2: Big Endian numbers are stored from most significant byte (MSB) in low memory address to least significant byte (LSB) in high memory address.

LevelDB expects all values to be in Big Endian format without explicitly saying so in any written material. We actually consider this a bug in LevelDB because Windows users like us may actually be using LevelDB on Windows and experiencing subtle errors and not even realize it. We were fortunate enough to uncover this bug during our testing. In fact, we only caught this bug because we built our test cases by hand and knew how many duplicate hash values to expect for given locations.

We based our fix on a commitment that we will store all LevelDB keys (and values) in Big Endian format.

- No byte order switching takes place if our application is running on Big Endian hardware (likely using UNIX). If, on the other hand, our application is running on Little Endian hardware (likely using Windows,) we set a global flag we use to ensure every numeric value's bytes are 'reordered' first switching from right-to-left going into LevelDB, then switching from left-to-right coming out of LevelDB.

4.4 Beginning a two-step Process

4.4.1 Introduction

Creating a two-step algorithm from a one-step algorithm presents some interesting challenges. It follows that we should perhaps discuss both steps in this one section. We

do not. Instead, we give each step its own section following this one. This way, we can use this section to consider issues we could not specifically relate to one particular step or the other.

Before we entertain these issues, we introduce a diagram showing how a two-step algorithm works, in general; and, what our two-step algorithm looks like in particular. Figure 2 shows the two steps, one on the right, and one on the left. The right hand side shows how we read a text file, calculate fingerprints, and write the Index Database. The bottom of the left had side shows how a Search String is provided. Then, its fingerprint is calculated, which, in turn, is searched for in the database. Matching fingerprints yields a list of corresponding locations, which we check in the text file to verify the string in it matches the search string. Either this verification step returns a match if the two strings are the same. Otherwise, it returns a mismatch (also called false positive or spurious hit.)

4.4.2 Impact of Two-Step Process

There are a number of issues arising from the modification of dividing the one-step Rabin-Karp Algorithm into two steps. The basic idea of a two–step approach is to build an *Index Database* during step one. Then in step two, use that Index in all subsequent matching requests, of which there can be many. We introduce in this section some of the changes we made to accommodate this separation of functionality. Below, we examine *window length*, as well as the two parameters *radix* and *modulo* before introducing several enhancements that speed-up our application's performance.

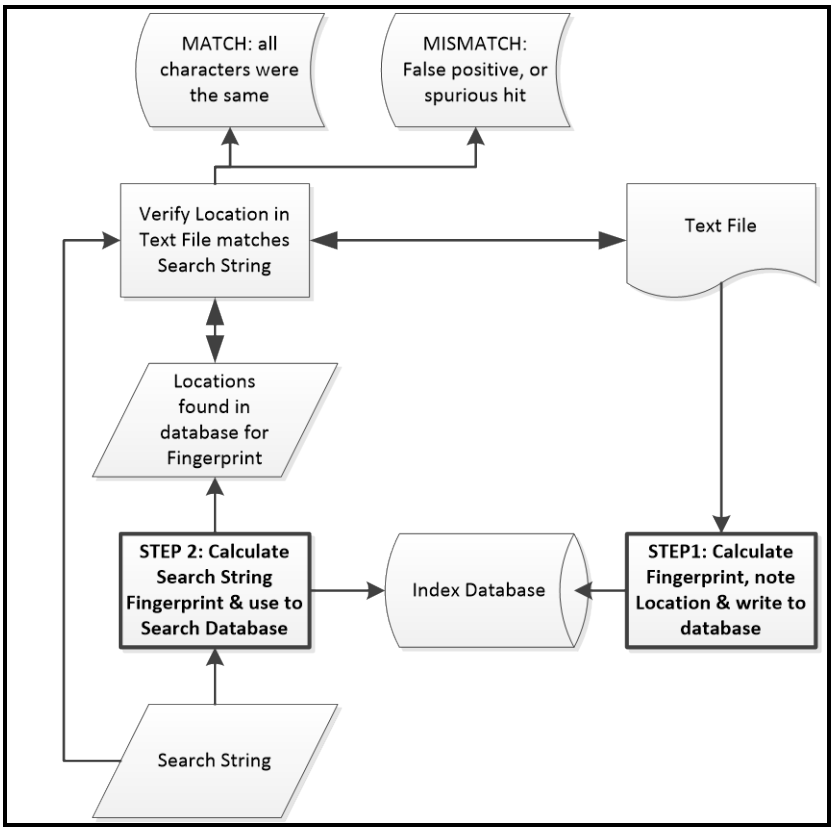

Figure 2: A chart showing the components of a two-step search technique and the relationship amongst them.

4.4.2.1 Window Length

Thus far, throughout this report we made it seem natural that calculating fingerprints from a text file is done for the same length substring as the search pattern length (**PL**.) This was no fluke, as we could have used any length for calculating text file fingerprints. We used the search pattern length to avoid discussing the very issue we must now deal with; using a length for calculating fingerprints from a text file that is different from the search pattern length. We are now at a point where this issue becomes important because it has a tremendous effect on building an Index Database and using it to search for strings. This is the point in our analysis where we will start making more use of the two concepts defined below. The reason is that they both refer to the length of the string used to calculate fingerprints in a text file.

- During all analysis involving a text file we use a text file Window (**W**) that has a Window Length (**WL**) associated with it. **W** always contains a substring from a text file containing **WL** characters.

A typical user of step two only wants locations for occurrences of their search string. They never expect an application to issue an error message saying, "Pattern length too long (or too short) cannot use this Index with that search string." Since this scenario could happen, we created a few extra algorithms and made a few new enhancements. We discuss these below.

One of the first modifications we performed was to make Window Length (**WL**) a user input for step one. This way, we can vary the length to examine what differences arise for various lengths holding everything else constant. We discuss this change further in the section below about step one. Another modification we needed to make was to create three algorithms that can use any Index Database to find search patterns of (almost) any length. The three algorithms deal respectively with the following three situations: **PL<WL**, **PL=WL** and, **PL>WL**. Each of these circumstances has a different mechanism to find and verify matches. Once again, we will describe these in detail in the section below about step two. It is worth noting that the Rabin-Karp Algorithm works only when **PL=WL.**

We designated fingerprints earlier as heuristics to whether or not a search string matches a substring from a text file. When a fingerprint from a text file equals a search string's fingerprint, we have the following two possibilities. When the respective characters do match, we refer to the incident as a *hit*. Whereas, when the characters do not match we call it a *spurious hit*. Thus, when the values of two fingerprints equal, we only know there is a *possible* match with the respective strings. The extra step we must execute to ensure the substrings do match we call *verification*. When we have a spurious hit, we do not know it until after we run *verification*, which is expensive, so we do not want to run it

too often. The best way to ensure this is to minimize spurious hits, which we discuss below.

4.4.2.2 Examining Radix and Modulo

Searching for a fingerprint in the Index Database is fast $O(1)$. Verification, on the other hand, is considerably slower $O(n^2)$ and is responsible for much of the *performance cost* of having many spurious-hits for a given fingerprint value. Our experiments confirm that the size of the hash, or, said differently, the maximum value for a fingerprint, plays a key role to the number of spurious hits to expect. Three parameters contribute to the size of the hash, *window length*, *radix,* and *modulus*. Explaining the details of these effects plus others below, is precisely why we dedicate this sub-section to deal with *modulus* and *radix* concepts. For example, radix and modulus are at the heart of another change we made. We changed our application to create several look-up tables for calculation of fingerprints using these two parameters. We will describe these tables later in our discussion. In any case, following this section, we return the discussion back to the two-step technique and describe each step in a section of its own.

4.4.2.3 Command-Line Inputs

In Appendix C, we describe how a user enters all command-line inputs. This section, however, we only highlight the following three; among other values, a user inputs a *modulus*, a *window length,* and a *radix* on the command-line. All three inputs are mandatory. However, if a user is unsure what prime number to use for modulus, they can simply enter a zero on the command-line. A value of zero causes the application to use the radix as is and to assign a prime value of 1,073,499,991 automatically to the modulus; which is the largest prime value our application can use without running a risk of overflow.

While we are discussing modulus, this is a good place to digress for a paragraph to make the following observation about modulus. One of the major changes we made to Rabin-Karp was allowing a user to supply any modulus they deemed as suitable, provided it was a prime number. The original Rabin-Karp Algorithm paper used probabilistic analysis

when selecting a prime modulus for a particular run. Recall the original algorithm is on-line and each run could do one search in one text file at one time. It would be difficult to do real-life experiments on modulus in this scenario. In their paper, Rabin-Karp also relied on probability to calculate the theoretical time and space values for their Algorithm. (Karp & Rabin, 1987) We do not currently have probabilistic random number generation in our application; but it would be an excellent enhancement to make it an option. Currently we assume the higher the modulus the better to avoid collisions, hence verifications. The number given above is the high value prime number we use regularly, although our experiments use several of them.

4.4.2.4 Program Inputs

A user can input the size of both fingerprints and file position; do not forget these are both part of the database key. Unfortunately, since these sizes are data types supplied as *typedef*s, we cannot have a user enter them as command-line parameters. Instead, a user changes them by assigning a *data type* for two program *typedef*s called, *HashValue_t* and *FilePosition_t.* These *typedef*s are located at the top of an application file called *RKUtils.h*. A user can choose between C's "long" or "long long" data types for HashValue_t and "[unsigned] long" or "[unsigned] long long" for FilePosition_t. A warning is appropriate because assigning these as 64bit integers instead of 32bit will double the size of an already large Index Database.

The original reason for Rabin-Karp to use modulo arithmetic was to keep calculated values small enough to capitalize on the speed with which modern processors perform one-word integer arithmetic. As mentioned briefly above, we sidestepped the speed gain from modulus calculations by approaching the problem more broadly than RK with look-up tables.

We can still use 32-bit fingerprints, but we calculate them differently; different enough to allow us a maximum modulo of a prime number around 2^{31} without risking overflow. To maximize the size of available modulus and radix, we used some precisely placed 64-bit integer parameters within several utility functions overloading them to the environment.

Additionally, the environment comes into play again when each function must decide to return 32-bit or 64-bit integer values based on how a user has setup their system (e.g., if HashValue_t = long long the function would return a 64-bit integer.)

Our two-pronged rational was as follows. First, higher modulus allows bigger hash values, and higher radix allows wider hash separation. To increase one or another, or even both, we need to increase the size of the numbers used to make certain calculations. We did just that. Moreover, we are pre-calculating all values and using look-up tables for all calculations making the 64-bit calculation penalty negligible. Since the look-up tables involve radix as well, we introduce and describe them in the next section.

4.4.2.5 Tying Radix and Modulus

4.4.2.5.1 Introduction

Radix is a value in Rabin-Karp that is at the center of most calculations. The biggest reason is that it is the base for the fingerprint number system. The higher the radix, the more characters can exist in an alphabet without having overlapping character codes. We provide a description of radix and all its roles in the following sections.

4.4.2.5.2 Limiting Alphabet Size with Radix

In our application, we define radix as the *base* of a number system created exclusively for calculating fingerprints. This means that a 'digit' in a fingerprint's number system ranges from zero to radix. Additionally, instead of using the actual character code when calculating a fingerprint, we use a value that is the character code modulus the radix. In other words, when retrieving a code for any character in an alphabet, we use the following equation:

$$b_i = c_i \bmod radix$$

Equation 4: Equation for calculating character codes in our application.

To understand the significance of this equation, let us assume we were using ASCII and a radix of eight. This combination of circumstances causes the following interrelated facts:

(a) the number of different *character codes* dropped from 128 to 8; (b) the number of different *digits* in the fingerprint's number system dropped by exactly the same amount from 128 to 8; and, (c) since 128/8=16, each hash bucket will have sixteen different characters in it. Said differently, this means that one character code now represents sixteen different characters, which is likely to result in a significant number of spurious hits.

We defined the formal definitions for strings in Appendix A. In that writing we provide the following definition for **S(i)**:

> For any string **S**, let **S(i)** denote a function that returns an integer code representing of the character **S[i]**.

This definition is still valid and leads to the following modifications to our application. When a character is involved in a process concerning character comparison like *verification*, our application uses $\mathbf{c_i}$ for both characters. This assumption helps keep the number of spurious hits to a minimum. Whereas, characters involved with any calculation related to fingerprints use the function, **S(i)**. Dropping any reference to $\mathbf{b_i}$, we define the **S(i)** function as follows:

$$S(i) = c_i \; mod \; radix$$

Equation 5: Equation for restricting character codes to be in a range from a minimum of 0 to a maximum of radix using modulo arithmetic.

Even though our application makes the implementation of the above concepts useful and workable, there is still a need to emphasize the care a user must take selecting a radix. For instance, to satisfy our experimental analysis we originally thought we would alter the value for radix to range from 2 to 128. We ran a modest analysis to build an index using a radix of two. After twelve hours of running with no end in sight, we aborted the attempt. In the end, the smallest radix we could use ended up being four, as will be shown later in our results section.

At the same time, this modification is important because it gives us more control over later parts of our system; particularly with calculating the minimum difference in length a search can accommodate. Although counterintuitive, having a high radix means, we cannot accommodate large differences between **WL** and **PL.** We elaborate later during our description of the matching step.

4.4.2.5.3 Demonstrating 32-Integer Results

Adding to the above decision is a complication of selecting *radix* and *modulus* together to avoid overflow in calculations using 32-bit integers. Even though we discuss 32-bit integers elsewhere, this particular topic is worth mentioning because we include a modulus in our experiment that shows the performance of imposing the 32-bit restriction juxtapose to the performance of not imposing that restriction. Therefore, our task is to discuss how to select a modulus for a given radix so as not to cause overflow.

Calculating the lowest modulus for a radix requires first knowing a maximum value for a modulus given a radix. That is, we need to know which calculation's result may trigger an overflow. Our analysis showed us the biggest risk occurs when the application multiplies a modulus by a character code. Even though this result is passed to the mod() function, the value passed in must be a 32-bit integer. Consequently, to calculate the biggest modulus that avoids overflow, we first recall that the largest character code is the radix and the maximum result is 2^{31} (the maximum value for a signed 32-bit integer.) Hence, the maximum value that a modulus and accompanying radix can take to avoid overflowing 32-bit arithmetic is (2^{31} / radix). In our experiments, we selected a modulus that was the largest prime number we could find that was less than this value for the radix we selected.

4.4.2.6 Creating Look-Up Tables

As a final part of this subsection, we describe how we use look-up tables to rid our application of a great deal of unnecessary calculations. We must be forthright and confess the idea was not ours. Indeed, as the next section illustrates, the idea originates in the original Rabin-Karp Algorithm itself. When we considered the savings in number

of calculations over all characters in a text file, the value was significant. For one thing, every calculation we make has a modulus function at the end of it. This modulus calculation is expensive because of everything happening in the background. Consequently, we borrowed the concept of look-up tables that we calculate before anything else in our application, to get rid of many mod calculations.

4.4.2.6.1 Small Integers from Large Integers

Before we get to the actual look-up tables, we revisit one of the biggest advantages for our application. Look-up tables not only save calculations during a match step, but also prevent several side effects of these calculations. For example, we will look at 32-bit versus 64-bit integer calculations that we use in our application. Above, we demonstrated that the value of a modulus must be such that when multiplied by any other value must produce a value that is less than a signed 32-bit integer; otherwise, an overflow would occur in an intermediate calculation. Similar arguments exist for other values such as radix and even character codes.

Currently, for our application we only ever use 32-bit modulo; the last experiment used a 32-bit signed integer to demonstrate that our application will not fail due to an overflow. When calculating the values in a look-up table we call a function whose parameters cast each 32-bit input value to 64-bit numbers. We use these to do all our calculations within that function. This allows values inside the function to be large without causing an overflow. Finally, after taking the modulus of the resulting value, we can cast the result back to 32-bit integer and pass it back as such. This means that within our utility function we have performed 64-bit arithmetic to calculate and return a 32-bit value. Of course, using a 64-bit modulus is computationally very expensive. However, if we only do it a few times, the calculation is worth the investment. It also allows us to use very large modulo to reduce the problem of spurious hits, explained elsewhere.

4.4.2.6.2 High Order byte

We include this section as an illustration of the not only Rabin-Karp's look-up table, but also to illustrate their creation and use in general. Earlier, we provided a listing for the

Rabin-Karp Algorithm on page 24. Line 3 of the listing demonstrated look-up table functionality (albeit with only one value) that seemed interesting and useful for our application. The Rabin-Karp Algorithm calculated a value for **h** on line 3 and saved it for repeated use in later lines. We replicate the equation for **h** below as a reminder of how many calculations are involved in just this one variable; and to demonstrate the power of using look-up tables that can sometimes save millions of calculations.

$$\boldsymbol{h = radix^{(WL-1)} \% modulus}$$

Equation 6: Calculating Rabin-Karp's h program constant

Even though this particular look-up table only has one cell, the savings in calculations are enormous. Without the variable **h**, Rabin-Karp's application would have had to use the above equation once for each character in a text file. When one considers the calculation produces the same result every time, it only makes sense to calculate it once and look it up every time we need it.

The value **h** represents is a high-order- byte multiplier used to remove a fingerprint's high order digit when the fingerprint window is slid right one character. Interested readers should refer back to the actual algorithm on page 24 above for more information. Another point we are stressing is that calculations we repeat many times producing the same result, can be pre-calculated, and stored in an array. This way, we can perform a look-up to get a particular value rather than a huge expensive calculation. Cumulatively, this trade-off saves incredible number of calculations.

Our application uses the following two look-up tables.

4.4.2.6.3 Radix Powers

Many places throughout the program we perform calculations that involve fingerprints. These calculations happen during indexing, during searching, and during calculations determining fingerprint ranges, to name a few. In every one of these cases, the calculation required raising a radix to a certain power and taking the modulus of the

result. We saw an opportunity for a look-up table with so many repetitions of the same calculations. We base this one-dimensional table on *window length* with each *position* in the array also being the respective array index. The value in each cell is the modulus of the radix raised to the position (or address), as shown in an equation below. To accommodate very large windows and search patterns, we gave the array an arbitrarily large number of entries; where the index **i** ranges from 0 to 100. This range would presumably be bigger than any reasonable **WL** or **PL**, we are likely to see in our application. This means that we can be reasonably certain the value for the variable **h** we talked about above is contained in a cell of the powRADIX[] array. The following expression defines the values in our powRADIX[] look-up table.

$$\boldsymbol{powRADIX}[\boldsymbol{i}] = \boldsymbol{radix}^{\boldsymbol{i}} \,\%\, \boldsymbol{modulus}$$

Equation 7: Calculating value for cell in the radix power look-up table

We use this look-up table in our application. We also use it in our algorithm listing below. In the listing, we treat the powRADIX[] array somewhat like a function. Thus, if a token like 'powRADIX[12]' occurs in a listing, we interpret it as asking for the result of performing the calculation listed above for i=12.

4.4.2.6.4 Changing Characters in Window

We now turn to the window sliding through the text file one character at a time. After we slide the window one character to the right, the leftmost character is left dangling. Recall from an earlier discussion that Rabin-Karp's **h** variable is the high-level byte multiplier for removing this character. Rabin-Karp pre-calculates **h,** our application goes one step further by multiplying **h** by a character code value and storing the result in an array. This array contains our look-up table called charOUT[] with one address for each available character code in the alphabet (**i** = 0 to 127 in our case.)

Recall earlier we defined **S(i)** as a function accepting character code as input (**i**) and returning character code mod radix. Notice that function appears in the subsequent equation as well as the radix raised to the power of for the leftmost character. We use the

following equation to calculate each cell's value before any other calculations in our application:

$$charOUT[i] = (S(i) \times radix^{WL-1}) \,\%\, modulus$$

Equation 8: Calculating value for cell in the leftmost character/radix power look-up table

After we fill this array with values, we simply provide a character code, and get back a value for removing that particular character from the current fingerprint. Not having to perform above calculations for each character in a text file is now possible. While sliding a window through a text file calculating fingerprints, a simple look-up is all that is required to get the value necessary to remove the leftmost character. Performing this calculation for each character in a text file would be a huge endeavor compared to retrieving a variable's value from an array. The point is using these two pre-calculated look-up tables save a great deal of CPU calculations later during the actual runs. They also give us the advantage of overloading the functions based on 32-bit versus 64-bit calculations. Getting a value from a look-up table takes the same time for 32-bit as for 64-bit.

We use both these look-up tables in our applications. We also include it in our algorithm listing below to demonstrate how and where we implemented it during Step one. In the listing, we treat the charOUT[] array somewhat like a function. Thus, if a token like 'charOUT[i]' occurs in a listing, we interpret it as asking to look up and return the value needed to remove the character with code = i from the left hand end of the current fingerprint. Line 14 of the listing below illustrates its use.

4.5 Step-One Building an Index

4.5.1 Introduction

Now that we have introduced a few issues of our two-step process, we can examine the changes we made for step one of the process; called Building an Index. We have already

briefly introduced LevelDB, the database engine we use; and will talk about it further below. We have also introduced the fingerprint and file position that we combine for our index database's primary key. All these issues including the original Rabin-Karp Algorithm contribute to step one of our process.

4.5.2 Index Building Algorithm

4.5.2.1 Introduction

The best way to demonstrate the changes we made to Rabin-Karp Algorithm is using algorithm listings. The following listing shows our algorithm for step one that can be compared to the original algorithm listing called, "Rabin-Karp-Matcher" on page 24. We will do that below. In the meantime, looking at the overall listing one can observe that our algorithm steps through a text file one character at a time, the same as the original. A change we made was to add processes and functions necessary for building an Index Database during that same stepping process. More precisely, as we add one character to a fingerprint we also add one record to the database in the same step. We will have more to say on this later. Finally, notice how no more lines of code remain dedicated to checking a pattern's fingerprint, or verifying characters match. That is because step one only builds an index. Step two is where all those other activities takes place like searching, and matching.

```
Index-Building-Algorithm(T, W, r, q)
1 TL ← Length(T)
2 WL ← Length(W)
3 for i = 0 to 127
4    do charOUT[i] ← (i mod r) * r^(WL-1) mod q
5 t0 ← 0
6 for  i ← 0 to (WL-1)
7    do t0 ← (rt0 + T(i)) mod q
8 for  s ← 0 to TL – WL
9    do   s = correctEndian (s);
10   do   ts = correctEndian (ts);
11   do   key = copy bytes (ts + s); (to DBkey structure)
12   do   insert key and s into LevelDB (Populate database)
13   If (s+1 > TL-WL) then EXIT ;
14   do   ts+1 ← (r(ts – charOUT[T[s]]) + T(s + WL)) mod q
```

Algorithm 2: Our Index-Building Algorithm, Step-One

4.5.2.2 Creating the Index Database

The above procedure works as follows. Recall we demonstrated earlier when we introduced characters they could be interpreted as either a symbol or an integer; the procedure interprets all characters as integers. Recalling the use of round brackets for a code and square brackets for a character, the term **T(i)** in line 7 refers to the character code (i.e., an integer) at position **i** in the text file rather than its symbol. The *for-loop* starting on line 3 populates the charOUT[] array; a constant used later in Line 13. Lines 6 and 7 compute a fingerprint for the first position of the text file. The method uses Horner's Rule that multiplies every subtotal by **r** before adding the new character. This loop continues until our fingerprint $\mathbf{t_0}$ is **WL**-digit numbers representing its respective string.

The second part of the algorithm begins with a *for-loop* of lines 8 through 15. This loop iterates through all possible shifts **s** in the text file performing the following calculations. First, Lines 9 & 10 check the Endianness and rearrange the byte orders if required. Then, Line 11 concatenates the two integers, fingerprint and file position, by treating them as byte arrays and copying the bytes to another temporary byte array. This temporary array

is now in a format to copy to a **key** byte array suitable for insertion in the database. Line 12, inserts this value as the key and the position (s) as the value into our Index Database.

Finally, Line 13 checks to see if the **for** loop on line 8 will be executed again. If not, it exits to prevent overflow. If so, the algorithm uses another innovation on line 14 that computes $\mathbf{t_{s+1}}$ **mod q** from the value of $\mathbf{t_s}$ **mod q** in constant time using Horner's rule.

Now that we have looked at how the process calculates and formats its data, we are in a much better place to have a little closer look at the database engine we used called LevelDB. The next section covers LevelDB.

4.5.3 Building our Index Database using LevelDB

4.5.3.1 Introduction

Throughout the document, we have been mentioning LevelDB as a tool we used to store our preprocessing information. It probably stands alone as the biggest modification we made to create our application. We also mentioned earlier that LevelDB was not our first choice. Originally, we worked with Oracle DB for several months before concluding we could not configure it properly to get the results we expected. Having found no reason why this phenomenon was occurring, we aborted Oracle DB and decided to try LevelDB.

LevelDB is an Open-Source Key-Value pair database created at Google. As such, it is NOT a relational database. While the records are automatically sorted by their key byte array, and keys can be located quickly with a function like **Get**(), there is no functionality typical relational databases provide. For example, a user cannot build an index on the value byte array, they cannot perform joins to secondary tables, and they cannot use SQL to perform any of its usual functions. LevelDB is just a fast, reliable, and easy way to store values in a table that automatically sorts its records by its key and allows many useful lookup and iterating functions based on key values. Below, we give a brief description of LevelDB including where to get it, how we set it up, and how a user works with this database, both voluntarily and otherwise.

LevelDB's web page is (Google Inc., 2012) (see http://code.google.com/p/leveldb/) where program download and documentation are both available. Unfortunately, there is not too much documentation for users in general. As for programmers, there is a three-page set of examples. Although, well written and filled with appropriate concision this "Detailed documentation" (Dean & Ghemawat) as they refer to it falls way short of serious programming documentation. Perhaps the documentation was trying to follow the same idea behind the database itself of small footprint and fast results. In any case, the documentation does not expose many of the nuances and fine details that would assist or even allow for any kind of strategic optimization of the setup parameters like cache and block sizes. Because of this lack of information, we did not attempt to check LevelDB's sensitivity to these parameters in particular during our experimentation. Aside from the fact that our experiment needed to vary our own parameters like radix, pattern length and modulus for instance, we had absolutely no direction as to how the block size or cache size ultimately affect the speed and size of LevelDB. Understandably, we put this kind of effort in our later chapter on future work and enhancements.

4.5.3.2 Google's LevelDB Description

The web page we just introduced above contains lists of LevelDB features and limitations. Interested readers should refer to the LevelDB project page at the following URL: http://code.google.com/p/leveldb/

4.5.3.3 LevelDB Functions and Properties

Before we get into our use of LeveDB, we must review a few characteristics we need to accomplish our task. We will begin with several properties we need in our analysis, and end with a list of its most important functions.

LevelDB is a very useful database as the above list declares. Throughout our brief discussion, we will review several of the major features from the list as we introduce new ones. We begin with the fact that LevelDB is an open-source C++ library. Its open-source license is a BSD-styled license that allows us to use the library and documentation

freely for our research, that is, as long as we acknowledge the owners. To install it, we simply supplied the path to its libraries and header files, put in its main header, and compiled within several minutes. Then, the work began by us building a C++ application that performed all our calculations.

LevelDB allows its *key* to be an arbitrary byte array they call a **slice**. We take advantage of this when we construct our key by concatenating the file position bytes to the fingerprint bytes; in the next section, we describe how we build a key byte array in detail. LevelDB also keeps the records sorted by the key as it **puts()** them. This means LevelDB will keep our records primarily sorted by fingerprint, with file position breaking any ties. This is exactly the order we need our output, so iterating through keys having fingerprints of equal value, we get a sorted list of file positions automatically. Incidentally, this sorting process is precisely one of the places we needed Big Endian byte arrangement. It is also how we discovered our integers were in Little Endian.

Having sorted records helps during a step two analysis when we are looking for particular fingerprints. LevelDB has a built-in *iterator* similar to STL's iterator that allows us to skip through collections of keys according to our own criteria. It also allows us to find keys with its **SeekToFirst(key)** method. The iterator has every other method needed to skip thorough the database starting wherever we wish, and ending wherever we wish. During runs in step-two, the iterator provides us with all the functionality we require. One of the characteristics of matching short values is a significant asset for our application. In our case, we mostly search LevelDB for a particular fingerprint, which is only half as long as the entire key (the other half is file position.) Nonetheless, the iterator will simply stop at the *first* record with a fingerprint matching ours. We can then continue iterating to subsequent records having the same fingerprint to get a list of all locations that single fingerprint has. Since LevelDB has sorted the records, the iteration results in a sorted list of file positions for that particular fingerprint. We will expand on this characteristic throughout this section.

LevelDB has the following components: Options; Status; ReadOptions; WriteOptions; and, WriteBatch. These components have functions that give a user quite liberal and thorough access to the database's capabilities. We do not need to expand on what each one is and how it works because of the breadth of features. The fact is these functions/components contain all the functionality one would expect a key-value database to have.

Finally, as described above, three of LevelDB's main functions give users access to enter, retrieve, or delete a record from a database. They are: `Put(key, value)`, `Get(key)`, `Delete(key)`. These functions have the same functionality as comparable functions in typical databases; not the least of which is the Put() function sorting records by its key byte array.

4.5.3.4 Our LevelDB Database

With our recent work using LevelDB, as far as we can tell, all functional and characteristics claims are true. The Detailed documentation opens with the following description: "The `LevelDB` library provides a persistent key value store. Keys and values are arbitrary byte arrays. The keys are ordered within the key value store according to a user-specified comparator function." (Dean & Ghemawat)

In this section, we will define our key and value byte arrays. Then, we will discuss why we do not use a comparator function. The key and value byte arrays are how an application trades data with LevelDB. It is up to the application to get everything in the correct format within the byte array.

4.5.3.4.1 Key and Value Byte Arrays

We discussed these earlier so we will not get into too much detail. Nevertheless, formal definitions for both byte arrays are as follows:

- The **key** for our LevelDB database is the bytes of a fingerprint followed by the bytes of a text file *position*, with the bytes arranged in Big Endian format.

Currently, the program can work with fingerprint prime values between 1 and 2^{31} (2,147,483,648) without overflow problems.

- A ***key byte array*** is a data structure called a slice used by LevelDB to usher the primary key into and out of a database. In our application, it consists of the key defined above.
- Since we have control over the type (hence, size) of the fingerprint, through *HashValue_t*, and the text file position, through *FilePosition_t*, (see section called Program Inputs on page 44), we have ultimate control over the size of the key byte array.
- A ***value byte array*** is a data structure called a *slice* used by LevelDB to usher value data into and out of a database. In our application, it consists of the same text file *position* as is included in the key above. This integer is also in Big Endian format.

The *key* and *value byte arrays* are very important data structures for LevelDB. They play an important role. Whenever data goes in to a database or out of one, these two data structures are the only conduits for that data. The key byte array is also the variable LevelDB uses in comparisons for database searching and browsing.

- The default sort order is lexicographically on the key byte array; that is, this is the sort order used by LevelDB when a user does not provide a *user-specified comparator function*. Integers in Big Endian format sort from smallest to largest lexicographically, as we would expect.

4.5.3.4.2 Functional Dependency

A digression is in order explaining the variable's value we placed into the *value byte array*. Although, we already declared LevelDB was not a relational database, some of the relational model's rules are helpful for other types of databases such as LevelDB. One relational rule in particular applies to many types of databases. Repeating the same value on the same record in a database breaks a relational rule preventing records from having *functional dependencies* among their fields. A functional dependency means that we can calculate a value on a record from one or more values on the same record. The concept behind this rule is for users to calculate the needed values when required rather than store them. Functional dependencies open a door to allowing a user to cause a database to lose its integrity. For now, we will just have to admit that our application has functional dependencies. Several of the recommended improvements will find a need for the value byte array, thereby removing these dependencies.

4.5.3.5 Administrator Functions

A user can create an *Index Database* anywhere in a directory tree they have a security access of 'modify.' Hence, when a user creates an Index Database, the command-line input requires a parameter giving a name for this directory (where the corresponding LevelDB database will exist.)

As LevelDB is building the Index Database, it uses many temporary files for moving data around. For instance, one of our Index Databases has 1,934 files in its database directory (described above.) Unfortunately, the names LevelDB uses for these files are the same for every database. These two issues make it a minimum REQUIRMENT for users to create new directories for each database. Alternatively, a user could at least empty a directory from a previous database before building a new one on that particular subdirectory. Thus, the security needs are to at least give a user permission to both create and delete directories as well as empty existing ones.

4.5.3.6 No Duplicated Keys

Some might call LevelDB, “light-weight” because it does not drag along very much overhead. For instance, it does not have support for duplicate keys; something we very much need to deal when we encounter hash collisions. We therefore needed a mechanism to overcome the fact that our Index Databases represent many records having the same fingerprint that we originally planned on as being our database key. The duplication of fingerprints comes from the simple fact that fingerprints are not unique for a given text file. Elsewhere in the report, we discussed two primary reasons for duplicate fingerprints that we reproduce here for convenience. (1) Two substrings with matching characters and length will always have the same fingerprint, and (2) depending on radix, and modulus, two completely different character substrings can end up having exactly the same fingerprint by chance.

Whatever the cause, the solutions to us was obvious (and already discussed above.) We created our own database key by concatenating the bytes for file position to the bytes for fingerprint. This action eliminates the chance of overlap completely because the file position is unique for each record in a text file.

4.6 Step-Two: Match Patterns to Text

4.6.1 Introduction

This section is about the tools and techniques we use to match a pattern (**P**) of length (**PL**) with text in a window (**W**) of length (**WL**) from a text file (**T**) of length (**TL**.) The chapter is about step-two of our approach to text search. The main obstacle in this part of the application is not building an Index Database beforehand; the obstacle is designing an Index Database meeting user needs in the first place. Designing an Index Database requires decisions for all parameters like radix, alphabet, window length, etc. that represent a great deal of work. Nevertheless, once a user builds an Index Database, they can use it repeatedly to search for all kinds of text.

4.6.2 Implementing Matching

Below we will see we have exactly three categories of matching available to us. Each one depends directly or indirectly with **WL**, **PL**, **r** and **q**. When we discuss each category later, we assume a reader is familiar with the following facts.

All of the functions in our application that find locations return a *vector* containing sorted file positions. We refer to this *vector*, both here and in the pseudo-code, by the arbitrary name **Locs**. By now we know step one of our application scans an entire text file once, building an Index Database containing fingerprint and file position information. We know as well, that the Index Database never has any duplicate keys because the second part of a key byte array value is a unique file position. Therefore, LevelDB will sort the records first by fingerprint then by file position. Incidentally, this sort order is correct because of our Big Endian format change described earlier.

Finally, when comparing a short string to a longer string for a match. If every character in the short string matches the respective character in the longer string, then the shorter string *matches* the longer string, but not the other way around. These types of partial matches are how we can locate fingerprints in an Index Database without needing a file position.

4.6.3 The Matching Issues

In this section, we investigate and describe a generalization we created allowing us to use one Index Database for locating patterns of different lengths. In fact, with a few exceptions we can use almost any window length (**WL**) and still find patterns of almost any length from the same Index Database. To accomplish this, our application operates with three categories of matches: (1) Pattern length is shorter than Window length (**PL<WL**); (2) Pattern is same length as Window (**PL==WL**), and (3) Pattern length is greater than Window length (**PL>WL**.) We use the remainder of this section to explore each of the three categories and describe solutions we developed for each.

4.6.3.1 Pattern Shorter than Window Length (PL<WL)

We begin by examining the toughest of the three categories, finding patterns whose lengths are smaller than the Index Database's window length. Consider a user that first selected a window length of **WL=5**, now wanting to find locations for patterns shorter than **WL** say **PL=3** (**PL<WL**.) This is the most difficult to solve of the three matching problems because we need to work with partial fingerprints. We will quickly review an example before moving on to bigger issues in this topic. We begin our discussion below with an example using a three character pattern "123", with **WL=5** and **PL = 3**.

4.6.3.1.1 Example of Matching a Short Fingerprint

We have already stressed how important this function is because the search pattern is shorter than the window length. It is somewhat easy to grasp through example than it is through formal definitions. Hence, we present the following example. To demonstrate how we solve this problem, assume we have a search pattern, **P** equal to **`123`** (an integer.) Since the pattern has three characters, its pattern length **PL**=`3`. Our alphabet consists of all digits, **Σ** =`{0,1,2,3,4,5,6,7,8,9}`; which means radix **r**=`10` (because that's how many characters are in the alphabet.) Remember, in our application a user can set **r** to any value. In this example, we have set the radix to the number of characters in the alphabet, not only because it is a standard approach, but also, to make our example's characters act like digits and digits to act like characters. Furthermore, let **q**=`13`, *and* **WL** = `5`. Continuing with our example, the hash function is for this set-up is: $\mathbf{H}(P) = (10^{3-1} \times 1) + (10^{3-2} \times 2) + (10^{3-3} \times 3) = \mathbf{123}$. Finally, the text file from which we will perform our search is, **T** =**`0123456789123`**.

Solving this seems trivial at first because we can manually scan the text file looking for `123`. Doing that is as doing the Brute Force Algorithm described earlier. We will nevertheless continue our example and solve it through our algorithm. Our **WL** is `5` so we need to search for all fingerprints looking like `123xx`, where `xx` are any digits. The manner in which we do this is by recognizing there is a range of values for this format. In other words, we want to find locations in **T** where fingerprint values exist in the range

from a minimum of `12300` to a maximum of `12399`. Hence, anytime we want to find a shorter pattern, we have a convenient mechanism as a guide. More formally, the matching mechanism where **PL<WL** works as follows.

Notice that to calculate the range's minimum value of `12300`, we simply shifted the known value **H***(P)* of `123` (with PL=3) to the left two positions. We accomplished this in this example by simply multiplying **H***(P)* by $\mathbf{r^2}$. In general, to calculate the range's minimum value we multiply the pattern's fingerprint **H***(P)* by $\mathbf{r^{WL-PL}}$. To calculate the maximum value we do the same shift (i.e., `12300`) but this time we add $\mathbf{r^{WL-PL}-1}$ to that result (which in our example is $\mathbf{r^2-1}$, *or* `100-1=99`).

4.6.3.1.2 Setting a Range of Fingerprints

With the specific example now calculated out, it is easier to see the interplay of the variables and parameters in the following two equations that define the minimum and maximum values of a fingerprint range. Any fingerprint falling between these two values may indeed contain the short pattern we are looking for. Unfortunately, being within the range does not say anything about being an actual match, all it says is that it *might* match. Therefore, after collecting a vector full of positions having a fingerprint within the range, our application compares the characters in each substring against the characters in our pattern using our *verification* process. The equations for both ranges follow.

$$H_{min}(P) = H(P) \times r^{WL-PL} \bmod q$$

Equation 9: Calculating the minimum fingerprint value for a short pattern range

$$H_{max}(P) = H(P) \times r^{WL-PL} \bmod q + (r^{WL-PL} - 1) \bmod q$$

Equation 10: Calculating the maximum fingerprint value for a short pattern range

Even after discarding the fingerprints outside this range, there still a possibility that the range is so huge there may be thousands of possible fingerprints that are nowhere near our target value. This scenario means we would be verifying many file locations that do not have strings matching our pattern. Even with our simple example, dropping the pattern length by one leads to a range from `12000` to `12999`. The factor of ten difference is because our radix is also ten. Even still, however, adding one digit we jumped from a range of one hundred possible fingerprints to a range of one thousand. Now consider a user with a radix of 127 (with its corresponding new base of `127`.) Since 127^3 is roughly equal to two million we have that many possible fingerprints. Consider the size of the range in our previous example when we shortened the pattern's length by one, and then apply it to this last example. Two million times 127 is a big number. It is easy to see how big a range can get, and to speculate how many of the fingerprints in that range are false positives (spurious hits.)

4.6.3.1.3 Pattern and Window Length Difference

Another way to screen whether a shorter pattern can be found for a given window length in a particular database is to calculate an allowable difference between their lengths. This is a higher-level approach than above because it tells us if calculating a range is even possible to begin with. This section demonstrates how our application calculates allowable length differences.

We begin by subtracting the two range equations from above ($H_{max}(P)–H_{min}(P)$) gives the maximum difference between two fingerprints (denoted as $\Delta_{max}(P)$). We have already simplified the right hand side of the resulting equation as follows:

$$\Delta_{\max}(P) = (r^{WL-PL} - 1) mod\ q$$

Since (`mod q`) implies a range from `0` to `q-1`, to have an ability to distinguish fingerprint values, we want to ensure that the right hand side be as large as possible. Since `q-1` is its maximum value anyway, we use it. Substituting, `q-1` for $\Delta_{max}(P)$ and

dropping the `mod q` (because when the RHS is equal to `q-1` that is precisely the value `mod q` would return) leaves us with the following equality,

$$q - 1 = (r^{WL-PL} - 1)$$

Next, we simplify by adding one to both sides, yielding,

$$q = r^{WL-PL}$$

Simplifying even further, we take the $\log_2$ of both sides; rearranging the RHS gives us,

$$\log_2(q) = (WL - PL)\ \log_2(r)$$

Next, let Δ_{pat}=(**WL-PL**) denote the *maximum difference between window length and pattern length.* After making that substitution, we simply divide both sides by the log of **r**. This leaves us with the following equation to calculate the maximum length difference between an Index Database's **WL** and a shorter Pattern **PL**,

$$\Delta_{pat} = \frac{\log_2 q}{\log_2 r}$$

Equation 11: Pattern and window length difference

This very handy equation is precisely the tool we need to issue a message like, "Pattern length too short cannot use this Index." to users. Our application uses the value from this equation before accepting a short pattern for analysis. If the difference is too large, there is little sense even beginning any calculations to begin with so, the application issues a message and terminates

4.6.3.1.4 Parameter Influences

While this equation is informative, it does not help us with ready-made conclusions about how large **r** or **q** should be without tackling a confusing interplay of conflicting

assumptions. A basic trade-off exists between maximizing both **r** and **q** together. Yet, any user of our application will ultimately have to decide on values for several parameters including **r** and **q**. Nevertheless, this section looks at the maximum length difference in terms of various values of **r** and **q**. More importantly, we made several assumptions when contemplating how to illustrate any useful lessons. In the end, we decided a table would give readers an idea of some reasonable values we can expect to see for Δ_{pat} when we assign typical values to both **r** and **q** in practice. This section will examine this question in some detail.

Table 3, shows the maximum value we can expect for Δ_{pat}=*(***WL-PL***)*, using various practical values for a radix **r** (rows) and a modulo **q** (columns.)

r ↓ / q →	257	32,173	15,485,863	1,073,676,287	2,147,483,647
2	8.0	14.9	23.9	30.0	31.0
16	2.0	3.7	5.9	7.5	7.7
32	1.6	3.0	4.8	6.0	6.2
127	1.1	2.1	3.4	4.3	4.4

Table 3: Δ_{pat} maximum for practical values of both r and q. These maximum differences are likely to occur in practice.

We considered the following ideas when selecting the values for **q** and **r** in this table. Earlier, we established that **q** must be a prime number. We also mentioned performance demands require that modulo arithmetic (and its intermediate values) be performed on integers that can fit into a single computer word (i.e., $\mathbf{q} \leq 2^{32}$); although later, we tested and expanded it to accommodate 64-bit math using loop-up tables. Further, since some intermediate modulo values can be negative, **q** must be of a signed integer type. Since **q** cannot be any of the unsigned integer types, our fit-in-a-computer-word-limit shrinks to $\mathbf{q} \leq 2^{31}$.) Therefore, to get an appreciation for the effects various values of **q** have on the

maximum length difference, in our experiment we selected a range of prime numbers that are near values that are related to computer words, such as: 2^8(=256); 2^{16}(=32,168); 2^{24}(=16,777,216); 2^{30}(=1,073,741,824); and, 2^{31}(=2,147,483,648).

Similarly, to get an appreciation of the sensitivity for the maximum difference with respect to values for our radix **r**, we selected several values that are candidates for our problem. For instance, by reducing the characters in **T** to two values, then we have a binary system where **r**=2. But, for an alphabet that includes all ASCII characters, our radix should be **r**=128. When using this value for radix there are no requirements to have all 128 characters actually appear in a text file. Nevertheless using ASCII character codes keeps the alphabet so every character value sorts as we would expect (i.e., alphabetically.) Finally, to indicate the effect of values for **r** between these two extremes we included **r**=16 and **r** = 32. Unfortunately, as we mention elsewhere, using an **r** of 2 was not practical at all. In fact, after some ten hours of executing we aborted the run and moved our lowest value for **r** up to 4.

There are several main points made in this and earlier sections worth review. First, LevelDB keeps its records sorted by values in its key byte array, which in our case consists of the fingerprint bytes followed by the file position bytes in Big Endian format. Second, when we ask LevelDB to find a fingerprint without giving a file position, it will place a cursor at the first record whose fingerprint portion of its key byte array matches the given fingerprint. Third, there is a calculation that supplies values telling us the largest allowable length difference between **WL** and **PL** that can exist when searching a pattern whose length is shorter than the database's window length. Finally, earlier we discussed LevelDB's iterator that has many of the functions as STL iterators.

4.6.3.1.5 Getting File Positions

Understanding iterators allows us to see how our application finds locations for shorter patterns that have passed the above length test but are still shorter than **WL**. Briefly, we use an iterator to find the smallest and largest fingerprint in the range given by Equation 11 on page 65 above. The range of $\mathbf{H_{min}(P)}$ and $\mathbf{H_{max}(P)}$ could not only be a very large

spread depending on the modulus **q**, the radix **r**, and the length difference (**WL-PL**), but also produce a large number of spurious hits. Whatever the case, this fingerprint range allows us to (a) use an iterator's function **SeekToFirst(H_{min}(P))** to search for the first occurrence of the value of **H_{min}(P)** in the Index Database, and(b) loop through the sorted records until the cursor passes the value of **H_{max}(P)**. During this looping process, our application stops at each record and appends the file position value into a vector we already introduced called **Locs**. Every file position in **Locs** is a candidate for matching the pattern. Therefore, our application iterates through **Locs** and performs *verification* for every file position. It will delete any position from **Locs** that is a false positive, leaving us with a vector containing exactly what we need; a list of file positions where the pattern occurs.

4.6.3.2 Pattern Equal to Window Length (PL==WL)

If the above was the toughest of the three situations, this one is the easiest; finding patterns whose lengths are the same as the Index Database's window length. In this case, we still use a cursor, but we do not need all the confirmation steps we saw in the previous situation where (**PL<WL**.) Our first step in this process is to calculate the pattern's fingerprint **H(P)**. Next, we simply use an iterator's function **SeekToFirst(H(P))** to search for the first occurrence of the fingerprint **H(P)** in the Index Database. Then, we iterate through the sorted records until the cursor comes to an entry whose value is not equal to the pattern's fingerprint **H(P)**. When that happens, we are finished gathering locations. In exactly the same fashion as above during this looping process, our application stops at each record and appends the file position value into a **Locs** vector. Also as above, every file position in **Locs** is a candidate for matching the pattern. Consequently, our application performs *verification* for each entry in the vector. During the verification process, the application deletes any position from **Locs** that is a false positive. After this verification step, our vector **Locs** has a list of file positions where the pattern occurs.

4.6.3.3 Pattern Longer than Window Length (PL>WL)

If the above scenarios were the toughest and easiest of the three situations, it makes sense to assume this one falls between the two. It does not. It is closer to the toughest than the easiest. In this scenario we have a task of finding a pattern whose length (**PL**) is bigger than the Index Database's window length (**WL**). While we give plenty of details later, our basic angle of attack for solving this problem is first to chop the pattern into an ordered set of substrings whose lengths are all equal to **WL** even the last one. Then, for each substring, we calculate a fingerprint. In addition, once we know a beginning file location, we can quickly and easily calculate the locations for each substring in the pattern. With these two broad concepts as a backdrop, we now turn to the details.

To begin with, we still use a cursor, but will have a few confirmation steps to perform before we get that far in the analysis. Our first step is to calculate a set of fingerprints for the entire length of the search pattern. Each fingerprint represents a substring of **P** that is exactly **WL** characters long (even the last one.) The first fingerprint, therefore, is for a substring of the first **WL** characters in the pattern. We denote it as $\mathbf{H(P_0)}$. Next, we calculate the second fingerprint from our long pattern, $\mathbf{H(P_{WL})}$. We continue calculating each successive fingerprint in the long pattern until we reach its end. The last fingerprint is tricky because we want it to be the same length **WL** as the other substrings. Thus, no matter where the fingerprint for the second last position ends, our last fingerprint will always be calculated as: $\boldsymbol{H(P_{PL-WL}^{WL})}$. It implies there most certainly will be some overlap between the last position's substring and the second last position's substring.

Now we have a set of fingerprints for our pattern. The first process finds all file positions for the first fingerprint in the pattern by iterating through the Index Database. As usual, we put the resulting file positions into a vector called **Locs**. Think of this vector as a sort of master set of file positions because at every step in our process, it contains a file position for every occurrence along with false positives. The entire idea of our process from here on is determining the file positions in this set that are candidates for an occurrence of a match and removing the file positions that are spurious hits. As we will

elaborate later, a key to identifying match potentials versus spurious hits is calculating how far away a fingerprint is from the beginning of the pattern.

Now that we have a starting point, we can begin a process of whittling down the file positions in **Locs** using the remaining fingerprints and a special comparison. The second step is virtually identical to the first step. The two differences are: (a) finding file positions for the last substring in the pattern instead of the first, and (b) placing these positions in a vector called **NextLocs** instead of in **Locs**. The notion behind this and the remaining parts of the process is as follows: if a file position in **Locs** is part of an occurrence, then it must have a 'corresponding file position' in **NextLocs** and vice versa. We calculate the 'corresponding file position' in **NextLocs** by adding the file position in **Locs** to the number of characters between the two substrings. If there is such a file position in **NextLocs,** we save that file position in **Locs** for the next round because it has a higher probability of being part of a match than it did before the comparison. Otherwise, **NextLocs** does not have a corresponding file position. That means we can remove the respective file position from **Locs** because it can never be the beginning substring of an occurrence when we know there are no ending substrings that matchup**.**

We continue processing each file position in **Locs** by adding the respective distance to its file position and searching for an entry with that particular file position in **NextLocs**. To do this, we iterate through **Locs** keeping all entries whose corresponding file position occurs in **NextLocs** and removing the remaining entries. We based the process on a view that every occurrence of the pattern must include a match for the first fingerprint, the last fingerprint, and every fingerprint in between.

After finishing with the file positions for the last fingerprint, we empty **NextLocs** and repeat the entire process for the second last fingerprint in the pattern instead of the last. Keep in mind that the number of entries in the **Locs** vector keeps growing steadily smaller with each step. In any case, we repeat this process for each fingerprint in the original long pattern. Finally, after performing all these steps for each of the long pattern's fingerprints, our **Locs** vector will only contain file positions where each

substring's fingerprint occurs in the proper location with respect to one another. Since this is the definition of an occurrence, our **Locs** vector contains a list of file positions showing where every occurrence of the pattern exists in the text file.

4.6.4 Pattern Matching Algorithm

4.6.4.1 Introduction

Now that we have provided the necessary background information, we present the following listing to show the algorithm we implemented and demonstrate its calculations and processing. This algorithm listing is somewhat larger and more complex than previous listings because of the three different types of matching. Once again, we will expand the details of this algorithm in subsequent sections of the report. The inputs are a text file **T**, a search pattern **P**, a window length **WL**, a radix **r,** a modulus **q**, and an Index database **DB**.

```
Rabin-Karp-Matcher(T, P, r, q, WL, DB)
##-1 prepares program constants like look-up tables
1 TL ← Length(T)
2 PL ← Length(P)
3 p ← 0
4 powRADIX[0] = 1
5 charOUT[0]  = 1
6 for (i = 1 to 100)
7     do powRADIX[i] ← (powRADIX[i-1] * r) mod q
8 for (i = 0 to 127)
9     do charOUT[i]  ← ((i mod r) * powRADIX[WL-1]) mod q
##-2 works on problems where search pattern length is less than Index Database window length PL<WL
10 if ((WL-PL) > 0 and ((WL-PL) ≤ (log2(q)/log2(r)))
11    p  ← 0
12    for  (i ← 0 to (PL-1))
13          do  p ← (rp + P(i)) mod q
14    LL ← (p * powRADIX[WL-PL]) mod q
15    UL ← (p * powRADIX[WL-PL]) mod q + (powRADIX[WL-PL]–1) mod q
16    get s from DB for (LL ≤ key and UL ≥ key) and save as vector
Locs[]
```

```
##-3 works with patterns whose lengths are the same as the window length  PL == WL
17 if ((WL-PL) == 0)
18    p  ← 0
19    for  (i ← 0 to (PL-1))
20          do p ← (rp + P(i)) mod q
21    get s from DB for p == key and save as vector Locs[]
##-4 works with patterns whose lengths are larger than the window length PL > WL
22 if ((WL-PL) < 0)
23    p ← 0
24    for  (i ← 0 to (WL-1))
25          do p ← (rp + P(i)) mod q
26    get s from DB for p == key and save as vector Locs[]
27    p2  ← 0
28    for  (i ← (PL-WL) to (PL-1))
29          do p2 ← (rp2 + P(i)) mod q
30    get s2 from DB for p2 == key and save as vector NextLocs[]
31    keep s in Locs[] for all s=s2-WL in NextLocs[]
32    v ← 0
33    for (k ← WL to PL–2WL)
34          do p3  ← 0
35          do v  ← v+1
36          do for  i ← (s) to (s + WL-1)
37                do p3 ← (rp3 + P(i)) mod q
38          do  get  s3  from  DB  for  p3  ==  key  and  save  as  vector
NextLocs[]
39          do keep s in Locs[] for all s=s3-WL in NextLocs[]
40##-5 Remove 'spurious hit'
41 SpuriousHits ← 0
42 for each s in Locs[]
43    if P[0 … PL-1] == T[s … s+(PL-1)]
44          then print "Pattern match at position: " s
45    else SpuriousHits ← SpuriousHits + 1
46          delete s from Locs[]
47 print "Total Spurious Hits:" SpuriousHits
```

Algorithm 3: Step-Two of Our Approach to Pattern Matching

4.6.4.2 Algorithm Description

4.6.4.2.1 Character Symbols and Codes

Before beginning a description of step two of our algorithm, we need to point out a few general topics about confusing concepts we want to make completely clear. First, throughout this document we have stressed that a character has two components, a symbol, and an integer. This algorithm in particular uses both side-by-side. When we need to refer specifically to a code or a symbol, we use parentheses or square brackets respectively. Remember as well that all character codes range from 0 to r – 1, which are

also digits in the r-based numbering system used in calculating fingerprints. In the listing, we clearly see an example of not only parentheses on Line 13 during fingerprint calculations, but also square brackets on line 43 during verification.

4.6.4.2.2 Vectors

Next, earlier we referred to a *vector* of values without really defining what a vector was or does. Since the algorithm listing uses several vectors, in particular to hold locations for later processing, we should provide at least a general definition for them. In a general sense, a vector is a C++ data structure similar to an array. In particular, we use the term vector to refer to a data structure having the same characteristics and properties as an STL (Standard Template Library) vector. (Prata, 2005) The algorithm listing identifies vectors by putting the square brackets immediately after their names, typically without any parameters. We use the same names in the listing that we did earlier while discussing the matching theory. In any case, throughout this algorithm we use a phrase "**`and save as`** `vector X[],`" to indicate that we want to store all of the values we just obtained into a vector called 'X.'

The algorithm uses two vectors, namely Locs[] and NextLocs[]. The first vector contains a sorted list of positions that satisfy whatever information need a user was trying to achieve. The second vector plays a support role when a search pattern is longer than an Index Database's window length. The basic procedure our algorithm follows is first to fill a 'main' vector called Locs[] with file positions having fingerprints that match the first one. Next, we fill another vector NextLocs[] with file positions whose fingerprints match the fingerprint from a corresponding place in the pattern. Finally, the procedure checks each position from Locs[] to see if the second vector has a corresponding position a certain number of bytes away. If so, we keep the file position value in Locs[], if not, we remove it from Locs[].

4.6.4.2.3 Index Database

Additionally, there is little mention of the Index Database in the listing other than in the **get** statements. We have already defined our algorithm for building this database earlier.

Now that we are examining our matching algorithm listing, we can see how an Index Database is actually used. To begin with, the DB parameter passes the Index Database into the algorithm. We can therefore we assume it already exists from an earlier run of our algorithm's step one process, and is filled with data from the text file **T** (also passed in.) We use a **get** command (see Line16 for example) to access its data. For instance, we use a **get** command to return values for '***s***', the location portion of the key. The '***s***' is significant because it stands for 'shift,' which represents the same value as file position; but, it is used mostly when the context is how far a sliding window has moved.

4.6.4.2.4 Algorithm Parts

In addition to the above general statements about the algorithm, we can get on with describing some of its specific features and characteristics. First, notice how we placed five bold comment lines (that begin with "##") throughout the listing. These comments delineate the major parts of the algorithm; which are: (1) program constants and initialization, (2) search when search string smaller than Index Database's window, (3) search when search string is same length as window, (4) search when search string is longer than window, and (5) verification and output.

The general flow of control is as follows. Execute Lines 1 through 9 to load program constants and look-up tables. Next, choose a situation between comments two, three, or four. Execute the lines for selected comment. Finally, place results into vector called Locs[]. The difference between comments two, three, and four are related to the concept we introduced earlier about the three scenarios for matching (PL < WL; PL == WL; and PL > WL.) The matching process depends which scenario is present, and the comments delineate the three processes. Comment two, lines 10 through 16 deal with the scenario PL < WL. This is a complicated process and can end with no positions in the Locs [] vector. This can occur when the length difference is too small. The second part of the **if** statement on line 10 demonstrates, no values are placed in the vector. Comment three, Lines 17 through 21, deals with the scenario where PL == WL. Finally, comment four, lines 22 through 39, deal with the scenario where the search pattern length is larger than the window length.

4.6.4.2.5 Verification and Output

After the above processes are completed, we have enough information to send our Locs[] vector through verification. That means we execute the lines for the last comment. Lines 41 through47 we call *verification* because that is precisely what happens. For each file position value in the Locs[] vector, we go to that position in the text file and check to see if that character matches the first character in the pattern. If they match, we move to the next characters and check to see if they match. We continue this checking until either a mismatch occurs or we reach the end of the pattern. If all corresponding characters match, a message prints the file position; otherwise, a spurious hit occurs and its counter is incremented.

4.6.5 Application Output

The application uses appropriate command-line output throughout the analysis to keep a user informed of progress. None of this output is of any consequent to an analysis, so we will ignore its details. Instead, the application produces three result files containing all information needed to perform an analysis. Step one creates one file containing its output, while Step two creates two files. We give a brief description of all three files in this section.

First, in all three cases, during an execution, the application checks for the output file(s.) If a file exists already, it is open for modification. If a file by that name does not exist in the database directory, the application creates a new file and opens it for modification.

4.6.5.1 Output File Description

4.6.5.1.1 File *.src

The application places the results of searches in a CSV (Comma Separated Values) file located in the database directory (same as command line parameter.) This file also has the same name as the database, but it has a ".src" extension tacked on its end. There is one record in this file for each search. Each record contains values for the following columns separated by commas:

a) **Matches**:

   A count of the number of occurrences of the pattern were found.

b) **Spurious Hits:**

   A count of the number of times an occurrence was flagged but was not found.

   This is also known as False Positive.

c) **Window Length:**

   Length used to calculate the fingerprint for building the LevelDB Index Database.

d) **Pattern Length:**

   Length of the pattern being sought for this run

e) **Radix:**

   Base of fingerprint numbering system, also maximum number for character code.

f) **Modulo:**

   The maximum number a fingerprint can be. All calculations use (% modulus.)

g) **Time (milliseconds)**

   The actual time it took to perform the search. A value of zero is used if radix and modulus are too small to allow search to proceed because length difference is too large.

h) **Number of repeated searches to get time over 10 ms**

   We discovered that sometimes searches happen so fast we could not get a time measurement other than zero. To remedy this problem we keep performing the same search in a loop until the total time exceeds 10 ms.

i) **Search string:**

   A reprint of the string used in original search pattern

4.6.5.1.2 File *.wrt

The application places the results of building an Index Database in a CSV (Comma Separated Values) file located in the database directory. This file also has the same name as the database, but it has a ".wrt" extension tacked on its end. This file contains a log of how many bytes the application has read from the Text File and how long it took to read them. The command-line parameter "*--report-every=*" "*(or -e=)*" controls the number of bytes to be read before writing the next record in this file. There is one record in this file for each byte count span. Each record contains values for the following columns separated by commas:

a) **Radix:**

   Base of fingerprint numbering system, also maximum number for character code.

b) **Modulo:**

   The maximum number a fingerprint can be. All calculations use (% modulus.)

c) **Window Length:**

   Length used to calculate the fingerprint for building the LevelDB Index Database.

d) **Bytes read:**

   Cumulative bytes read from Text File, so we have a running total of the bytes processed thus far.

e) **Time (seconds):**

   Time taken from the last write operation in this file until this write operation. Note, unlike the bytes read in the previous field, this value is not cumulative and is just the time taken to process the last increment of bytes.

4.6.5.1.3 File *.wtot

The application places the results of building an Index Database in a CSV (Comma Separated Values) file located in the database. This file also has the same name as the database, but it has a ".wtot" extension tacked on its end. This file contains a log of how many bytes the application read from the Text File in total, how many bytes the LevelDB Index Database is, and how long it took to create the LevelDB Index Database. There is one record in this file for each byte count span. Each record contains values for the following columns separated by commas:

a) **Radix:**

   Base of fingerprint numbering system, also maximum number for character code.

b) **Modulus:**

   The maximum number a fingerprint can be. All calculations use (% modulus.)

c) **Window Length:**

   Length used to calculate the fingerprint for building the LevelDB Index Database.

d) **textFileSize (bytes):**

   Number of bytes processed in the Text file during the Index Building step.

e) **Time to create Index Database:**

   Total time taken to build the Index Database in LevelDB.

f) **Size of Index Database (bytes):**

   Total number of bytes in the resulting Index Database.

Chapter 5: Performance of Our New Application

This chapter describes the results of our experiment. It highlights our algorithm's performance and compares some aspects of that performance with the performance of GREP (the de facto standard text search application for UNIX). In this chapter, we describe GREP. We also present the parameters, their ranges, and all the combinations and permutations we included in our experiment. Furthermore, we describe the hardware and the text file(s) we ran the experiments on. Finally, we present the experiment outcomes. Throughout the presentation, we highlight and keep track of some of the most interesting observations we made while conducting our experiments.

5.1 Experiment Design & Implementation

The previous chapters have demonstrated that we changed several major aspects of Rabin-Karp's original algorithm into a new application using LevelDB. Having written and tested our application, our primary goal now is to determine whether our newly created application can outperform GREP. To determine this, we designed an experimental approach based on varying one parameter at a time holding all other parameters constant. To begin with, we built 36 different Index Databases to accommodate different radix, modulus, and window length parameters as shown in Table 4 below.

> *The layout and contents of Index Databases is made up of each radix and modulus combination, of which there are nine, each of which has four Index Databases, one for each of the four window lengths; making a total of 36 different Index Databases.*

Observation 1: We based the layout and configuration of our experiment's Index Databases on radix, modulus, and window length.

Radix = 4, m =				Radix = 8, m =				Radix = 16, m = 134,207,779			
WL	WL	WL	WL	WL	WL	WL	WL	WL	WL	WL=	WL=
Radix = 16, m =				Radix = 32, m =				Radix = 32, m =			
WL	WL	WL	WL	WL	WL	WL	WL	WL	WL	WL=	WL=
Radix = 128, m =				Radix = 128, m =				Radix = 128, m =			
WL	WL	WL	WL	WL	WL	WL	WL	WL	WL	WL=	WL=

Table 4: This table summarizes the parameters for all 36 Index Databases we created for our experiment. The table has nine boxes, each of which has a radix, and a modulus on the top row, and four window lengths on the next row. Information from the top row plus one of the window lengths under it uniquely describes one of the 36 Index Databases. We tested 11 pattern lengths (listed below) on each database giving a total of 396 tests.

Another important observation we can see in this table is as follows:

> *For each of the five radix values (4, 8, 16, 32, 128), the lowest modulus we used in our experiments is equal to the largest prime number that would not allow any calculation to exceed the value of a signed 32-bit integer. The equation to calculate this maximum value is (2^{31}/radix). Therefore, the smallest modulus is the biggest prime we could find that does not exceed this value.*

Observation 2: The lowest modulus we used for each radix was the largest prime number that is less than (2^{31}/radix).

Once we create an Index Database, we can search it repeatedly keeping in mind the Window Length (WL) used to create it. With this in mind, we introduce the next experimental parameter, Pattern Length (PL), to account for each of the searches in our experiment. One issue we kept in mind when selecting pattern lengths was to ensure we obtained a mixture of searches that included each of our three scenarios described in the previous chapter as PL<WL, PL==WL, and PL<WL. Since we created a different search strategy for these same three criteria, it only makes sense to keep this in the foreground while considering other issues.

In our experiments we decided to use eight different pattern lengths (PL) {2, 4, 8, 10, 12, 16, 32, 64} for patterns that we know exist in the text file. We also used three different pattern lengths {4, 8, 12} for patterns we know for sure do not exist in the text file. These last three patterns are substrings of, "ZZZZZZZZZZZZ" having a length of 4, 8, and 12.

Observation 3: Our experiment was to search for a variety of search pattern lengths. We used eight search pattern lengths for patterns we know occur in each Index Database, and three lengths for patterns we are sure do not occur.

With 36 Index Databases and 11 different pattern lengths, we needed to perform 396 different searches to accomplish our experimental objectives. Perhaps not surprisingly, both organizing the experimental runs using a batch file; and, analyzing the mountain of resulting data from each run individually and all runs together consumed a great deal of resources. Not to mention the difficulty we encountered deciding how to formulate, and format, all of these results. On the one hand, we needed to present convincing evidence for our conclusions; on the other hand, we could not clutter the report with 396 different graphs showing each experiment against GREP.

Nevertheless, the remainder of this chapter explains our approach in more detail, and describes the highlights from the observed results.

5.2 Source Text Files

During our deliberations, we needed several text files, one text file for our testing and verification, and another text file for the actual experiments. We will discuss the test text file later; for now, we introduce our experimental text file. After considering many sources of English text available, we decided on using the King James Bible. (The Large Canterbury Corpus, 2001) First, the document has old English style prose making it easy to find substrings with different numbers of occurrences ranging from one, to hundreds, to even thousands. Second, by making our experiment's text file from one hundred copies of the bible one after another, we end up with file positions for our search strings

at predictable distances, and we can predict the number of occurrences for each of our search strings by multiplying the numbers from the single copy by 100. These two features gave us reliable devices for quick and easy verification of our results. Finally, a single copy of the Bible only occupies some 4,050,944 bytes (3.86 MB) of disk space, which helps our text editor load it and count occurrences for any search string very quickly. The text file with 100 copies occupies 404,742,144 bytes (385 MB) of disk space.

> *We used a text file containing 100 copies of the King James Bible for all experiments. This file occupies 404,742,144 bytes (385 MB) on a disk. It was large enough to produce meaningful time differences for most patterns.*

Observation 4: Our text file contained 100 copies of the King James Bible in 3.86 MB.

With regard to application speed, we will compare the actual numbers with GREP below. In the meantime, since some searches were so fast, we had to add a function in our application that kept repeating a search until the cumulative elapsed time exceeded 10 milliseconds. This made sure we had big enough time numbers to guarantee a search did not return a value of 0.00 ms. The following observation illustrates where we desperately needed this function.

> *For a radix of 4, a PL of 4, a modulus of 1,073,499,991 and a search Pattern of "ZZZZ" our application had to repeat the search 1329 times just to get a cumulative elapsed time of 10.002ms, which works out to 0.0075ms for one run of that particular search.*

Observation 5: Our application is extremely fast when searching for strings that we knew were not in the text file.

After adding this function, we decided we also needed to improve the precision of our timer because C++'s standard timer is only precise to a millisecond. We did some

digging and found a structure from <Windows.h> called 'QueryPerformanceCounter' that met all of our timing needs. To get the most up-to-date information on this interface go to www.Microsoft.com. We installed and verified the API and implemented it rather easily.

> *Using the <Windows.h> header file, and this timing API, is the only issue preventing our application's code from porting directly to UNIX. A user wanting to port the application will need to change these with suitable replacements.*

Observation 6: The timing API we used was exclusive to Windows.

5.3 Setting up and Testing our Application

Even though we tested our application extensively during its development, we felt it was necessary to perform a final test before beginning our experiment to ensure everything was performing as expected. Our test involved using a text file that was different from our experimental text file, yet had enough variation to assure us we were counting and locating a randomly selected phrase from the file. The two-step process required us to first use a sophisticated text editor to select the phrases, count them, and give us their locations throughout the file. Next, we compared those results against what our application produced. However, before we could use this file we had to first verify that our text editor program was working as expected.

We used EditPad Pro (7.1.2 x64) (www.editpadpro.com) as our text editor. It has all the features we needed to accomplish our objectives for selecting phrases, counting them, and locating them in a file. In a sense, we had to have faith that EditPad Pro was providing us with the correct results to begin with. To verify this we created a two-paragraph text file, did some searching, and verified manually that EditPad Pro did not make any mistakes. Aside from having to read the entire file to manually locate and count the occurrences ourselves, we had no other choice but use this smaller file to verify with a reasonable amount of certainty that EditPad Pro was producing correct results. In

the end, after confirming that our application produced exactly the same counts and locations verified that EditPad Pro was indeed working correctly. We were henceforth confident that we knew exactly how many of our patterns occurred in a text file, and exactly where each occurrence was located.

Getting back to testing our application, we used EditPad Pro to select a set of appropriately sized phrases we would use in our test. The text file we used for testing is *The Works of Mark Twain by Mark Twain*. (Twain, 2009) We obtained this eBook from the Gutenberg Project on the internet at: (http://www.gutenberg.org/). The size of this text file is 20,045,824 bytes (19.1 MB). As its title suggests the contents of this single text file is the complete works of Mark Twain.

We carried out our testing method using the following processes. After performing steps (a) through (d) once for every pattern length, we then used step (e) through (g) on each search pattern and each Index Database. We performed this same procedure in our experiment on the same 396 combinations of parameters that we used in our experiment.

a) Load text file into EditPad.
b) Select a phrase at random making sure it was as long as our selected pattern length for this test. We performed our random selection process manually by blindly moving the cursor horizontally and spinning the mouse down wheel one or more times. While not very scientific, it did not have to be, we just needed to verify our application found the search pattern the same number of times as EditPad Pro found.
c) Run a count in EditPad to get the number of times the selected phrase occurs in the file.
d) Select several occurrences and note their location in the file.

e) Run our application to either build an Index Database (if required), or search an existing Index Database if there is one for a phrase.

f) Verify the total number of occurrences our application produced was the same as the number EditPad produced.

g) Verify the locations obtained from EditPad Pro earlier appeared in our application's output file.

The results of performing the above tests were positive and reassuring.

> *After extensive testing, we found all values we checked matched our expectation, and we did not find any serious flaws or overlooked processes during our testing procedure. The results left us confident our application was doing what we expected it to do and produced results we expected.*

Observation 7: We tested our application with a different text file, and did not find any mistakes.

5.4 Source Computer

All of our work, including testing, building Index Databases, and searching for patterns were all carried out on the same computer. This meant we did not need to do any other forms of analysis to put its results on a level playing field. The following list shows some of the most important hardware and software components possessed by our system and used in performing our experiments.

Manufacturer:	ASUSTeK Computer Inc.
Model:	ASUS Notebook G73jh Series
Processor:	Intel(R) Core(TM) i7 CPU Q720 @ 1.60GHz 1.60 GHz
Installed memory:	8.00 GB

HD Disk 0:	OS (C:); 228,936 Mb; OCZ-AGILITY2 (SSD)
HD Disk 1:	OS (D:); 171,704 Mb; OCZ-AGILITY2 (SSD)
OS:	Windows 7 Ultimate 64-bit Service Pack 1

5.5 Using GREP

Since the purpose of our experiments required comparing our search times against GREP, we begin by reviewing the results obtained from searching with GREP. Even though we are using the 100 Bible text file, the diagram of Figure 3 shows the relative search times for three of our text files: they contain 1, 10, and 100 copies of the Bible. A quick glance at this diagram demonstrates that GREP's time is roughly proportional to the size of its target file with a few minor exceptions.

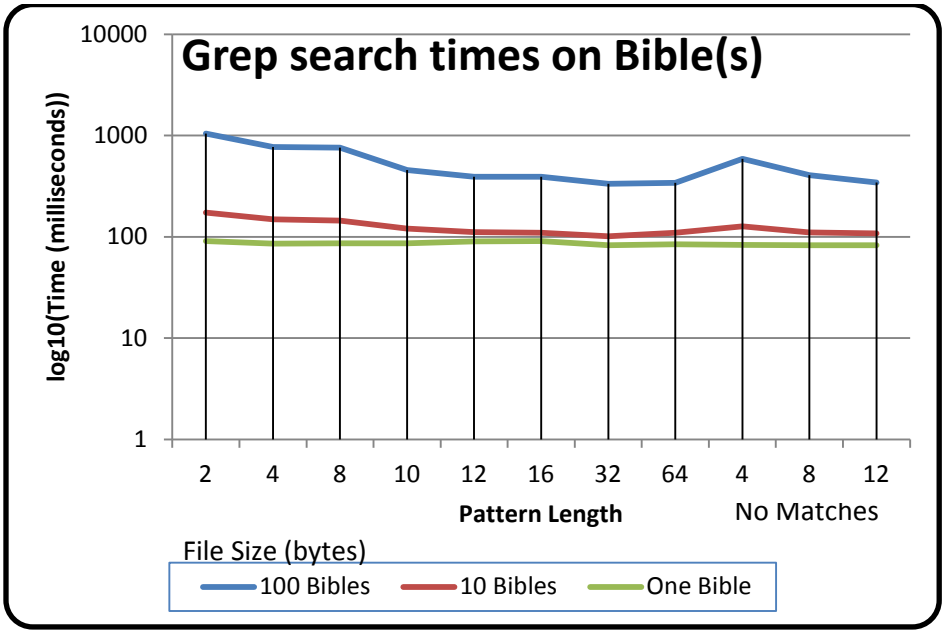

Figure 3: Search times for GREP searching text files containing 1, 10, and 100 copies of The King James Bible.

The first action we performed after running our text file containing 100 copies of the Bible through GREP was comparing the number of occurrences for each of our patterns

with our expected results. We summarized GREP's results in the table below, and used it for this comparison. The table has the following four columns: (1) length of pattern; (2) actual count of occurrences from EditPad; (3) count of occurrences given by GREP, and; (4) time taken by GREP in milliseconds. We highlighted the last three rows to remind readers those searches use patterns whose strings for sure did not exist in the file.

Pattern	Actual	GREP	GREP Time
2	83,400	82,200	1042.28
4	1,216,800	1,146,400	771.47
8	212,100	208,800	758.15
10	39,600	39,500	457.87
12	39,600	39,500	390.88
16	1,300	1,300	390.68
32	100	100	333.43
64	100	100	341.77
4	0	0	587.39
8	0	0	407.24
12	0	0	343.35

Table 5: Results from GREP searching the text file containing 100 copies of Bible.

This shaded portion illustrates an important observation about GREP's times. Since GREP is an on-line search tool, it processes an entire text file for every search. We can therefore expect there to be a time associated with looking for a string that does not exist. In fact, we see times comparable to or even slower than times using a search pattern that not only exists, but also, occurs more than 39,000 times. Another interesting point about the bottom three rows is that the shorter the search string, the longer it took GREP to search a file.

GREP takes roughly the same amount of time to count occurrences of strings that exist in the text file as it does searching for strings that do not exist in the text file. In addition, shorter strings take longer to search for than longer ones, which happens whether the string exists or does not exist.

Observation 8: GREP has roughly the same performance looking for strings not in a file as it does finding occurrences of a string.

One of the most striking observations in this table is how far apart GREP's counts are from the actual count. At first, we thought there was a mistake somewhere in our processes that would explain the discrepancies. After an exhaustive search we were confident our numbers were correct. That is when we discovered the following observation.

GREP does not count the actual number of times a string occurs in a file, it only counts the number of different lines containing at least one copy of that string.

Observation 9: GREP counts lines containing a search string.

This means, once GREP discovers an occurrence on a line, it can skip the remainder of that line; thereby saving big chunks of time. This bias gives GREP an edge over our application that does not even 'know' that lines exist because GREP could end up skipping a large part of a file; especially when lines are long (like paragraphs) and the search string occurs nearer to its start than its end. It also highlights a difference in outputs between the two applications. Whereas GREP reports the line number for an occurrence, our application reports the exact location in the file for that same occurrence.

We explored several methods to make the results more compatible with one another. Modifying our application to act the same as GREP by reporting only line numbers and skipping to the next line once an occurrence is discovered was one such consideration. We rejected this because it restricted our application's ability a great deal. A second

consideration was to pipe the line number from GREP through another program that can actually count occurrences and report locations for each line. We also rejected this because of the overhead required to pipe the results and start this other program would very difficult to capture accurately. In the end, we decided to leave both applications as they were and recognize that with all other things held equal, GREP will report a time based on doing much less work than our application. In other words, we have no way of capturing the difference it would make in time if we could somehow force GREP to count occurrences instead of lines with an occurrence. We therefore must accept this bias with the knowledge that if times are the same for both applications, GREP is definitely the slower one.

5.6 Preliminary Comparison

GREP is a very popular search tool on UNIX operating systems. It will be good to contrast GREP with our approach, which is a command-line program for Windows. Before we present all of our results we must layout our tool and our experimental procedure. We do this in the following sections. This section summarizes a results demonstrate we are heading in a profitable direction regarding our thesis.

Figure 4 shows a summary of the performance we achieved by aggregating roughly 360 or so experiments. The lighter line shows GREP's times, while the darker line shows our best times. Since GREP has no parameters, its time is the same for every run of a particular search pattern length. In contrast, our application has several parameters affecting search time. These parameters yield different search times for each search pattern length; we plotted the best of those times.

This graph confirms that our application shows faster times for experiments involving all search pattern lengths except 4 and 8 bytes. This evidence supports our thesis; thereby proving that not only we can build an application, but also, in some cases that new application outperforms GREP. With more resources to continue studying this problem area we could produce some very fast search times. In the end, this mechanism could be an extremely fast way to index and recall information using a Rabin-Karp fingerprint.

We made this graph by aggregating the same data that was used to create all the graphs shown in Appendix F: Line Graphs of Performance (ms). Appendix F contains one graph for each combination of Radix and Modulus using the same two axes as Figure. The main difference between the following graph and those in Appendix F is that later graphs show performance for each window length in our analysis. Remember Pattern Length and Window Length are different.

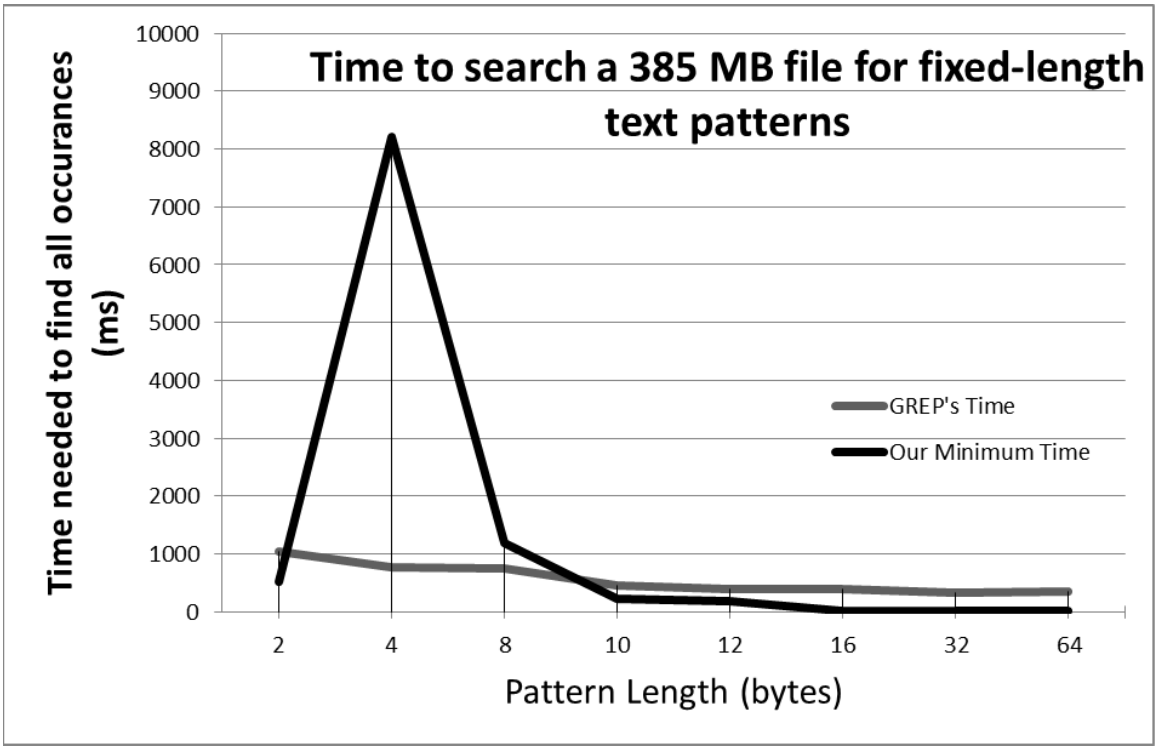

Figure 4: An early look at the outcome of our experiment showing how our application beats GREP times for long pattern lengths

5.7 Creating Index Databases in LevelDB

Recalling that our application consisted of two-step process, we now examine results from step one of that process: creating Index Databases. This procedure involves examining the text file one byte at a time, calculating a fingerprint, flipping the byte

arrays to Big Endian format, and writing that fingerprint and its associated file position to the Index Database. The mechanisms just described used in this procedure are the same ones we described earlier when discussing our Index-Building Algorithm (see Algorithm 2.)

Once created, the Index Database is available for any number of searches. As described earlier, our experiments require us to create 36 different Index Databases. That is, one database for each combination of radix/modulus, and window length. Figure 5 shows the sizes for each of the 36 Index Databases. These results lead to the following observation:

> *Most of the resulting Index Database sizes are roughly ten times as large as the original text file containing 100 Bibles that has a size of 404,742,144 bytes (385 MB).*

Observation 10: The Index Databases for most criteria are approximately ten times larger than the original text file.

The several exceptions to this observation are Index Databases for a radix of four. We will have more to say about a radix of four later. The other exception happens only when the window length is eight, the radix is 128, and the modulo are primes close to 2^{30} and 2^{31}. For some reason, these two instances look totally out of place. Their size is about three or four hundred megabytes larger than all other databases with a window length of eight. They are also about two hundred megabytes bigger than the corresponding databases with window lengths of 10 and 12. We took the same database size data and produced a line graph in Figure 6 below showing exactly the same information as the bar graph. The spike in size is also visible on this graph for the two databases in question. Also, we put the size of the Index Databases beside each respective marker for the smallest radix of 4, a radix of 8, and the largest radix of 128, which produce the smallest, medium and largest sized database sizes respectively.

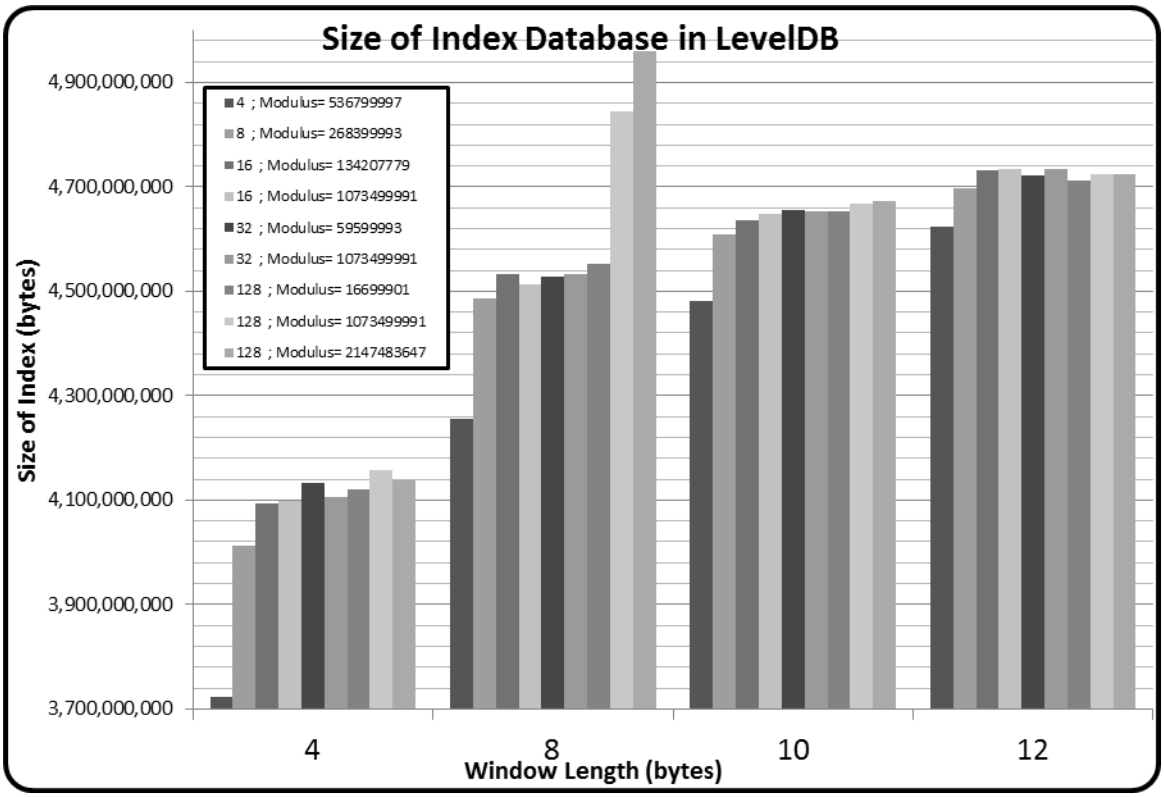

Figure 5: Bar Chart showing the sizes of the Index Databases created for our experiment.

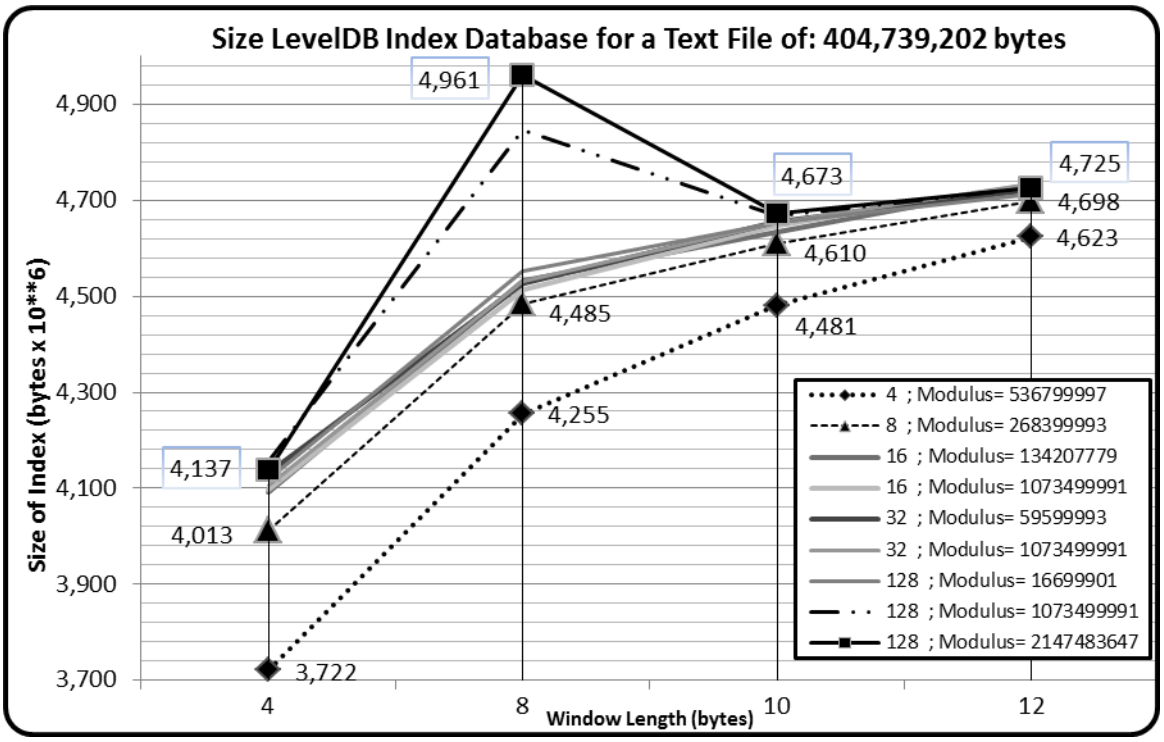

Figure 6: Line Graph showing the sizes of the Index Databases created for our experiment.

The next logical piece of information to examine is the time it takes to build these Index Databases. To that end, Figure 7 below illustrates the times taken to build Index Databases by window length and by radix/modulus. This particular chart cannot help in finding specific times for specific parameters. Instead, it shows how all the times are roughly the same being between 5,000 and 6,000 seconds for all but the same two databases that were the largest above.

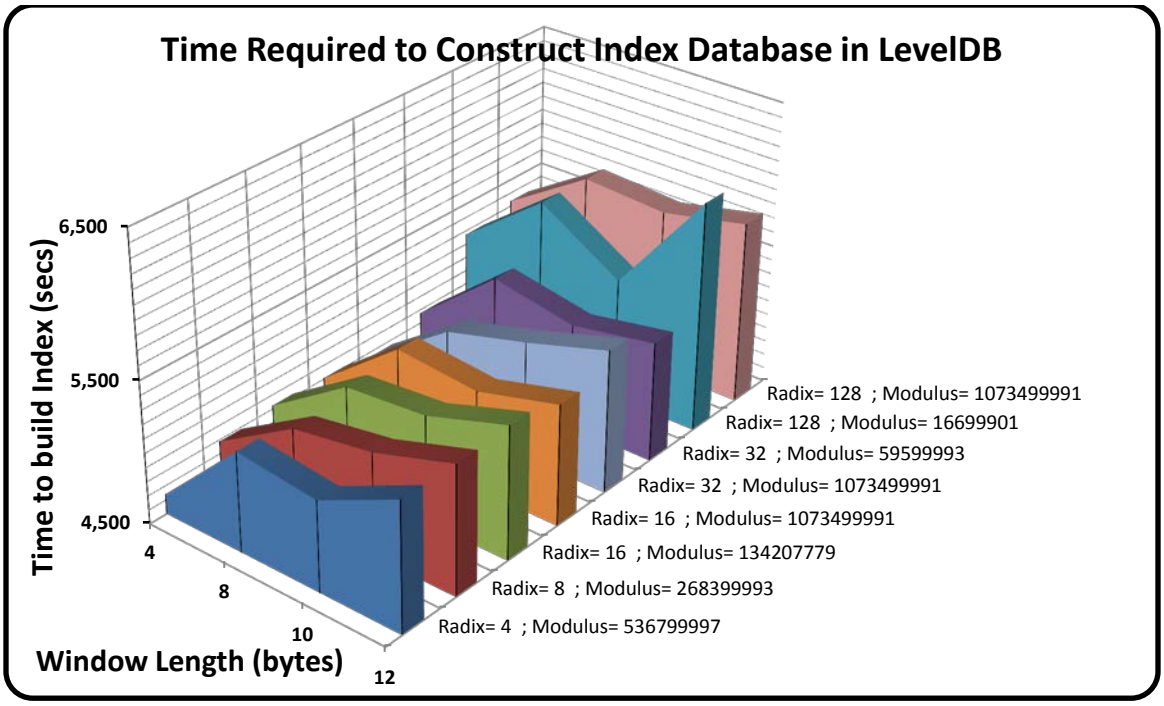

Figure 7: Area graph showing time required to build each of the Index Databases in our experiment.

Figure 6 and Figure 7 illustrate several other observations and conclusions. For instance, the transition from one window length to the next on both graphs is not smooth. Both the times and the sizes jump around without any apparent reason, making it difficult to draw any conclusions with regard to size or to time. However, from a general point of view, the following observation about these results is worth highlighting:

> *In a general sense, both the size and the time required to build an Index Database increase with increasing radix and increasing window length.*

Observation 11: Generally, radix and window length are two variables that have an effect on the time and size of an Index Database.

At this point, we must discuss how LevelDB, despite producing large files, is actually compressing the data a great deal. Recall that the database key consists of the fingerprint

concatenated with a file position making it 16 bytes long. In addition, the number we put into the database value field was file position, consisting of another 8 bytes. This implies that each record in our Index Databases is 16 bytes long. If we assume there is a record in an Index Database for every byte of its respective text file, we can calculate its theoretical size. In this case, we have a 24-byte record and a 404,742,144-byte text file. The product of these two values is 9,713,811,456 bytes. So, one would expect an Index Database of approximately this size. Yet, Figure 6 shows that the maximum size for all of our Index Databases is 4,961,000,000 bytes, roughly half the theoretical value.

According to its documentation, LevelDB automatically compresses data using another application from GOOGLE code called Snappy compression library. (Google Inc., 2012) Snappy is similar to LevelDB in that it is a C++ API that is easy to use. In fact, a typical user would not even know LevelDB had a copy of Snappy embedded in it. According to Snappy's Web Site (https://ccp.cloudera.com/display/CDHDOC/Snappy+Installation)

"It [Snappy] does not aim for maximum compression, or compatibility with any other compression library; instead, it aims for very high speeds and reasonable compression."

Finally, our original experimental design involved including a radix of two. In retrospect, it was not a good idea. Since its alphabet would only consist of two characters, it is perhaps understandable why it took so long building an Index Database. Recall earlier we mentioned that after about 10 or 12 hours of building, we stopped the process. We subsequently aborted our plan of including a radix of two in our experiments. It encourages hash collisions because the resulting number system is so close to binary.

> *We rejected using a radix of two in our experiment. The reasons are the extremely large amount of time it took to build a partial Index Database, likely caused from having an alphabet with only two characters.*

Observation 12: We dropped a Radix of two from our analysis

In any event, both Index Database size and the time required to build one are large but nowhere near unmanageable. These results look as if they could be trimmed adjusting page size, buffer size, and/ or block size in LevelDB. In addition, we did not use LevelDB's function for doing batch writes; which would increase performance. LevelDB's overheads make the time and size at least five orders of magnitude bigger than our algorithm running on its own. To verify this difference, we clocked our program running alone without writing to the database; just calculating the fingerprint for each byte in the file. We recorded a time of less than one hundred milliseconds do this for the 100-Bible text file

5.8 Performing Searches

With such disappointing results from step-one, a salvation of our approach will have to come from searches. We created the following search strategy to test this answer. Table 6 shows the numbers supporting our strategy, which were included in a table shown earlier when we discussed GREP. We used eight pattern lengths containing strings that we confirmed exist in the Bible. Since, we know each string appears at least once for every Bible, we also know that they occur several megabytes apart from one another. The number of occurrences of the string for each pattern length appears in the table's second column. We also included three pattern lengths (highlighted at the bottom of the table) containing strings that do not appear in the text file. The three non-existent strings are substrings of "ZZZZZZZZZZZZ" with the respective pattern lengths.

Pattern	Actual
2	83,400
4	1,216,800
8	212,100
10	39,600
12	39,600
16	1,300
32	100
64	100
4	0
8	0
12	0

Table 6: Pattern length and a count of the actual number of occurrences in a 100-Bible text file. The bottom thre rows are highlighted because they contain patterns that do not occur in the text file.

Our search strategy required us to perform a search for each of the eleven strings on each of our 36 Index Databases. Tracking a search's results was easy because of our option to append results data from each search to the end of an existing results file. That meant when all the processing was finished we had one text file containing all information describing the results.

5.9 Performance Results

After completing all of the above searches, we had a file filled with numbers that we present in this section. Before showing graphs and summaries, we feel it is telling to show a table of those numbers.

5.9.1 Table of Results

We took the results file and imported its data into Excel so we could manipulate it into a presentable summary format containing number of hits, number of spurious hits and total time for each radix/modulus, each window length, and each pattern length. In addition,

having this table provides a mechanism for a reader to verify or double-check values presented later, or to see a context within which a particular result may reside.

Cell Colours
Pattern To Short
PL Shorter than WL
PL Same Length as WL
PL Longer than WL
Pattern does not Exist

Aside from each table showing the radix, modulus, and window length against pattern length, the cells are also colour coded to indicate the relationship between PL and WL. As shown here, the top colour indicates where the difference between pattern length and window length is too large for a given radix and modulus. Next, we have three colours showing the different search scenarios from earlier: PL<WL, PL==WL, and PL>WL. Finally, the last colour shows the case where the pattern is not in the text file.

Pattern Length	Radix = 4 and Modulus = 536,799,997											
	Window Length = 4			Window Length = 8			Window Length = 10			Window Length = 12		
	Matches	Spurious	Time (ms)	Matches		Time (ms)	Matches	Spurious	Time (ms)	Matches	Spurious	Time (ms)
2	83,400	9,970,800	46,862.69	83,400	9,970,800	58,537.84	83,400	9,970,800	45,182.79	83,400	9,970,800	67,604.62
4	1,216,800	8,677,500	35,002.80	1,216,800	8,677,500	32,829.61	1,216,800	8,677,500	42,112.85	1,216,800	8,677,500	50,995.52
8	212,100	614,200	17,457.52	212,100	614,200	2,815.77	212,100	614,200	3,055.83	212,100	614,200	3,900.43
10	39,600	68,200	11,567.70	39,600	68,200	1,132.16	39,600	68,200	428.99	39,600	68,200	546.42
12	39,600	21,700	13,753.54	39,600	21,700	1,179.04	39,600	21,700	373.68	39,600	21,700	326.59
16	1,300	1,000	12,639.82	1,300	1,000	404.36	1,300	1,000	68.45	1,300	1,000	95.06
32	100	0	17,298.16	100	0	78.15	100	0	27.90	100	0	52.16
64	100	0	39,309.76	100	0	1,399.93	100	0	123.99	100	0	37.97
4	0	0	0.01	0	0	0.01	0	0	0.01	0	0	0.01
8	0	0	0.01	0	0	0.01	0	0	0.01	0	0	0.01
12	0	0	0.01	0	0	0.01	0	0	0.01	0	0	0.01
TOTAL	1,593,000	19,353,400	193,892.03	1,593,000	19,353,400	98,376.89	1,593,000	19,353,400	91,374.50	1,593,000	19,353,400	123,558.80
	Radix = 8 and Modulus = 268,399,993											
2	83,400	5,108,600	23,069.93	83,400	5,108,600	23,910.44	83,400	26,210,000	121,066.33	0	0	0.00
4	1,216,800	4,355,600	24,581.35	1,216,800	4,355,600	26,014.16	1,216,800	5,120,300	26,436.77	1,216,800	59,357,997	252,949.10
8	212,100	407,500	13,222.46	212,100	407,500	2,833.25	212,100	407,500	2,191.66	212,100	418,700	2,216.99
10	39,600	27,300	7,870.47	39,600	27,300	741.49	39,600	27,300	266.15	39,600	27,400	276.71
12	39,600	17,500	7,980.07	39,600	17,500	943.78	39,600	17,500	279.11	39,600	17,500	226.30
16	1,300	600	7,937.63	1,300	600	360.02	1,300	600	53.04	1,300	600	50.80
32	100	0	4,741.35	100	0	18.36	100	0	10.72	100	0	10.88
64	100	0	16,648.56	100	0	1,025.15	100	0	35.41	100	0	12.20
4	0	0	0.01	0	0	0.01	0	3,000	23.83	0	17,219,215	62,878.35
8	0	0	0.01	0	0	0.01	0	0	0.01	0	800	3.76
12	0	0	0.01	0	0	0.01	0	0	0.01	0	0	0.01
TOTAL	1,593,000	9,917,100	106,051.85	1,593,000	9,917,100	55,846.69	1,593,000	31,786,200	150,363.03	1,509,600	77,042,212	318,625.08
	Radix = 16 and Modulus = 134,207,779											
2	83,400	1,331,200	6,398.80	0	0	0.00	0	0	0.00	0	0	0.00
4	1,216,800	4,096,000	23,075.13	1,216,800	4,163,200	23,885.04	1,216,800	64,964,997	268,775.27	0	0	0.00
8	212,100	397,100	13,592.47	212,100	397,100	2,797.85	212,100	398,100	2,138.01	212,100	591,000	3,900.05
10	39,600	18,700	7,983.78	39,600	18,700	627.50	39,600	18,700	239.67	39,600	18,800	342.99
12	39,600	16,900	8,607.45	39,600	16,900	931.75	39,600	16,900	272.20	39,600	16,900	328.24
16	1,300	600	7,852.56	1,300	600	358.92	1,300	600	47.95	1,300	600	80.06
32	100	0	2,384.50	100	0	13.15	100	0	10.36	100	0	10.44
64	100	0	9,926.06	100	0	465.86	100	0	43.02	100	0	15.48
4	0	0	0.01	0	300	2.61	0	46,828,818	152,172.48	0	0	0.01
8	0	0	0.01	0	0	0.01	0	300	2.59	0	185,000	903.73
12	0	0	0.01	0	0	0.01	0	0	0.01	0	0	0.01
TOTAL	1,593,000	5,860,500	79,820.79	1,509,600	4,596,800	29,082.70	1,509,600	112,228,415	423,701.56	292,800	812,300	5,581.01

Table 7: Search results for Radix equal to 4, 8, and 16 (with small modulus) showing the number of matches and spurious hits as well as the time required for each search.

Pattern Length	Radix = 16 and Modulus = 1,073,499,991											
	Window Length = 4			Window Length = 8			Window Length = 10			Window Length = 12		
	Matches	Spurious	Time (ms)	Matches	Spurious	Time (ms)	Matches	Spurious	Time (ms)	Matches	Spurious	Time (ms)
2	83,400	1,331,200	6,929.20	83,400	5,068,200	42,972.12	0	0	0.00	0	0	0.00
4	1,216,800	4,096,000	25,429.05	1,216,800	4,124,500	24,998.77	1,216,800	12,727,900	84,896.66	0	0	0.00
8	212,100	397,100	13,170.74	212,100	397,100	2,217.73	212,100	397,400	3,084.95	212,100	406,200	14,823.52
10	39,600	18,700	7,175.75	39,600	18,700	543.69	39,600	18,700	307.08	39,600	18,700	321.77
12	39,600	16,900	5,952.68	39,600	16,900	987.48	39,600	16,900	387.78	39,600	16,900	323.96
16	1,300	600	5,801.42	1,300	600	301.45	1,300	600	69.84	1,300	600	80.14
32	100	0	2,161.91	100	0	12.52	100	0	13.16	100	0	40.23
64	100	0	9,069.78	100	0	497.28	100	0	24.36	100	0	15.86
4	0	0	0.01	0	0	0.01	0	2,815,208	12,937.62	0	0	0.01
8	0	0	0.01	0	0	0.01	0	0	0.01	0	7,000	159.99
12	0	0	0.01	0	0	0.01	0	0	0.01	0	0	0.01
TOTAL	1,593,000	5,860,500	75,690.56	1,593,000	9,626,000	72,531.08	1,509,600	15,976,708	101,721.48	292,800	449,400	15,765.50
	Radix = 32 and Modulus = 59,599,993											
2	83,400	1,304,800	6,211.09	0	0	0.00	0	0	0.00	0	0	0.00
4	1,216,800	4,016,900	22,757.56	1,216,800	11,824,300	61,275.58	0	0	0.00	0	0	0.00
8	212,100	396,800	10,978.55	212,100	396,800	2,075.84	212,100	409,600	3,032.00	212,100	7,275,300	34,387.04
10	39,600	18,700	6,102.47	39,600	18,700	465.20	39,600	18,700	317.85	39,600	22,500	351.25
12	39,600	16,900	6,234.25	39,600	16,900	670.15	39,600	16,900	377.88	39,600	16,900	363.80
16	1,300	600	4,935.72	1,300	600	255.94	1,300	600	74.05	1,300	600	66.91
32	100	0	1,781.10	100	0	11.89	100	0	11.43	100	0	10.50
64	100	0	7,208.96	100	0	281.77	100	0	34.36	100	0	16.60
4	0	0	0.01	0	5,422,600	17,529.07	0	0	0.01	0	0	0.01
8	0	0	0.01	0	0	0.01	0	1,500	12.52	0	6,377,301	29,172.08
12	0	0	0.01	0	0	0.01	0	0	0.01	0	0	0.01
TOTAL	1,593,000	5,754,700	66,209.73	1,509,600	17,679,900	82,565.46	292,800	447,300	3,860.10	292,800	13,692,601	64,368.19
	Radix = 32 and Modulus = 1,073,499,991											
2	83,400	1,304,800	6,580.30	0	0	0.00	0	0	0.00	0	0	0.00
4	1,216,800	4,016,900	24,121.48	1,216,800	4,381,500	36,694.85	0	0	0.00	0	0	0.00
8	212,100	396,800	13,485.66	212,100	396,800	2,642.78	212,100	397,000	13,215.78	212,100	769,900	16,947.70
10	39,600	18,700	7,778.40	39,600	18,700	532.98	39,600	18,700	343.88	39,600	18,700	359.20
12	39,600	16,900	6,335.18	39,600	16,900	809.63	39,600	16,900	368.78	39,600	16,900	343.16
16	1,300	600	6,200.93	1,300	600	330.79	1,300	600	78.19	1,300	600	71.58
32	100	0	2,262.87	100	0	11.46	100	0	10.36	100	0	14.77
64	100	0	9,337.43	100	0	358.17	100	0	28.30	100	0	25.30
4	0	0	0.01	0	593,500	2,366.77	0	0	0.01	0	0	0.01
8	0	0	0.01	0	0	0.01	0	0	0.01	0	312,800	1,715.91
12	0	0	0.01	0	0	0.01	0	0	0.01	0	0	0.01
TOTAL	1,593,000	5,754,700	76,102.29	1,509,600	5,408,000	43,747.45	292,800	433,200	14,045.32	292,800	1,118,900	19,477.63

Table 8: Search results for Radix equal to 16 (with large modulus), and 32 (with both modulus) showing the number of matches and spurious hits as well as the time required for each search.

Pattern Length	Radix = 128 and Modulus = 16,699,901											
	Window Length = 4			Window Length = 8			Window Length = 10			Window Length = 12		
	Matches	Spurious	Time (ms)	Matches	Spurious	Time (ms)	Matches	Spurious	Time (ms)	Matches	Spurious	Time (ms)
2	83,400	109,500	7,457.05	0	0	0.00	0	0	0.00	0	0	0.00
4	1,216,800	0	8,218.98	0	0	0.00	0	0	0.00	0	0	0.00
8	212,100	0	9,915.62	212,100	0	15,208.48	212,100	360,301	14,368.35	0	0	0.00
10	39,600	0	5,020.73	39,600	0	433.37	39,600	400	242.10	39,255	408,331	19,923.22
12	39,600	0	5,334.82	39,600	0	666.62	39,600	0	272.08	39,255	0	257.72
16	1,300	0	4,219.03	1,300	0	204.65	1,300	0	42.83	1,288	0	65.88
32	100	0	2,880.74	100	0	32.15	100	0	10.11	99	0	10.81
64	100	0	7,537.11	100	0	360.05	100	0	24.36	99	0	250.03
4	0	0	0.01	0	0	0.01	0	0	0.01	0	0	0.01
8	0	0	0.01	0	0	0.01	0	342,500	1,768.52	0	0	0.01
12	0	0	0.01	0	0	0.01	0	0	0.01	0	297	5.32
TOTAL	1,593,000	109,500	50,584.11	292,800	0	16,905.36	292,800	703,201	16,728.38	79,996	408,628	20,513.01
	Radix = 128 and Modulus = 1,073,499,991											
2	83,400	0	6,376.74	0	0	0.00	0	0	0.00	0	0	0.00
4	1,216,800	0	11,282.69	1,216,800	98,045,116	429,285.83	0	0	0.00	0	0	0.00
8	212,100	0	9,351.60	212,100	0	1,187.13	212,100	1,600	9,512.54	212,100	102,888,908	447,346.41
10	39,600	0	4,313.34	39,600	0	367.80	39,600	0	245.40	39,600	4,800	221.09
12	39,600	0	4,683.17	39,600	0	611.88	39,600	0	292.71	39,600	0	190.28
16	1,300	0	4,651.45	1,300	0	143.97	1,300	0	36.78	1,300	0	25.65
32	100	0	4,320.12	100	0	47.37	100	0	24.05	100	0	29.30
64	100	0	7,955.65	100	0	227.82	100	0	66.79	100	0	58.44
4	0	0	0.01	0	91,505,516	335,442.46	0	0	0.01	0	0	0.01
8	0	0	0.01	0	0	0.01	0	2,200	117.98	0	0	0.01
12	0	0	0.01	0	0	0.01	0	0	0.01	0	0	0.01
TOTAL	1,593,000	0	52,934.80	1,509,600	189,550,632	767,314.28	292,800	3,800	10,296.27	292,800	102,893,708	447,871.19
	Radix = 128 and Modulus = 2,147,483,647											
2	83,400	0	9,959.80	0	0	0.00	0	0	0.00	0	0	0.00
4	1,216,800	0	14,461.22	1,216,800	106,923,398	576,194.72	0	0	0.00	0	0	0.00
8	212,100	0	14,144.82	212,100	0	1,198.87	212,100	442,800	19,995.07	212,100	60,166,400	318,647.61
10	39,600	0	7,145.12	39,600	0	363.99	39,600	0	297.86	39,600	600	250.99
12	39,600	0	7,211.02	39,600	0	660.82	39,600	0	372.64	39,600	0	235.89
16	1,300	0	6,985.85	1,300	0	160.98	1,300	0	78.27	1,300	0	38.68
32	100	0	5,077.56	100	0	54.39	100	0	35.57	100	0	40.12
64	100	0	9,940.52	100	0	224.53	100	0	106.11	100	0	94.95
4	0	0	0.01	0	26,573,497	136,529.25	0	0	0.01	0	0	0.01
8	0	0	0.01	0	0	0.01	0	1,000	10.50	0	45,666,775	228,391.15
12	0	0	0.01	0	0	0.01	0	0	0.01	0	0	0.01
TOTAL	1,593,000	0	74,925.93	1,509,600	133,496,895	715,387.57	292,800	443,800	20,896.03	292,800	105,833,775	547,699.41

Table 9: Search results for Radix equal to 128 (with three different modulo) showing the number of matches and spurious hits as well as the time required for each search.

One can observe the most interesting situation from these tables by scrutinizing the bottom three rows in each table. There are spurious hits listed in several of these cells where we know the search pattern does not exist in the text file. They should not be caused by duplicate hash values simply because there seem to be too many (91 million for a radix of 128 and a modulus of 1,073,499,991) to have such a simple cause as hash bucket collisions. We have yet to decode the meaning of these because there does not appear to be any obvious reason for them appearing where they do; and no obvious

pattern as to where they occur and how many there are of them. They appear for all radix/modulus combinations except a radix of four. They also do not occur when the window length is four. Other than those two situations, they appear at least once in every radix/modulus combination. At the same time, they do not seem to favour a particular cell for any combination. Its looks as if they appear at random with no particular rhyme or reason. Nevertheless, we do not regard these as errors because all of them correctly show that our application did not find any occurrences, and correctly reported them as being spurious hits. Since we could uncover no reason from our analysis, finding one is a definite recommendation we will make later.

Another observation worth noting from these tables concerns the speed with which our application determines no match is found (see cells in bottom three lines that do not have those mysterious reportings of spurious hits.)

> *Our application can determine when a search pattern does not exist in less than 0.01 milliseconds, which we used as a default for faster times because our significant digits left us with a limit of two decimal places.*

Observation 13: Our application is blazing fast at determining when a string does not exist in a text file.

5.9.2 False Positives by Category

At the top end of the tables, where PL is shorter than WL analyses occur, we also find some large numbers for spurious hits. These are much more understandable because of the search mechanism we use to find matches in this area; see Algorithm 3 on page 72. We search all fingerprints that fall between the minimum and maximum fingerprint values calculated by our application using Equation 9 and Equation 10. Figure 8 below shows the distribution of all spurious hits for all combinations of radix/modulus and category.

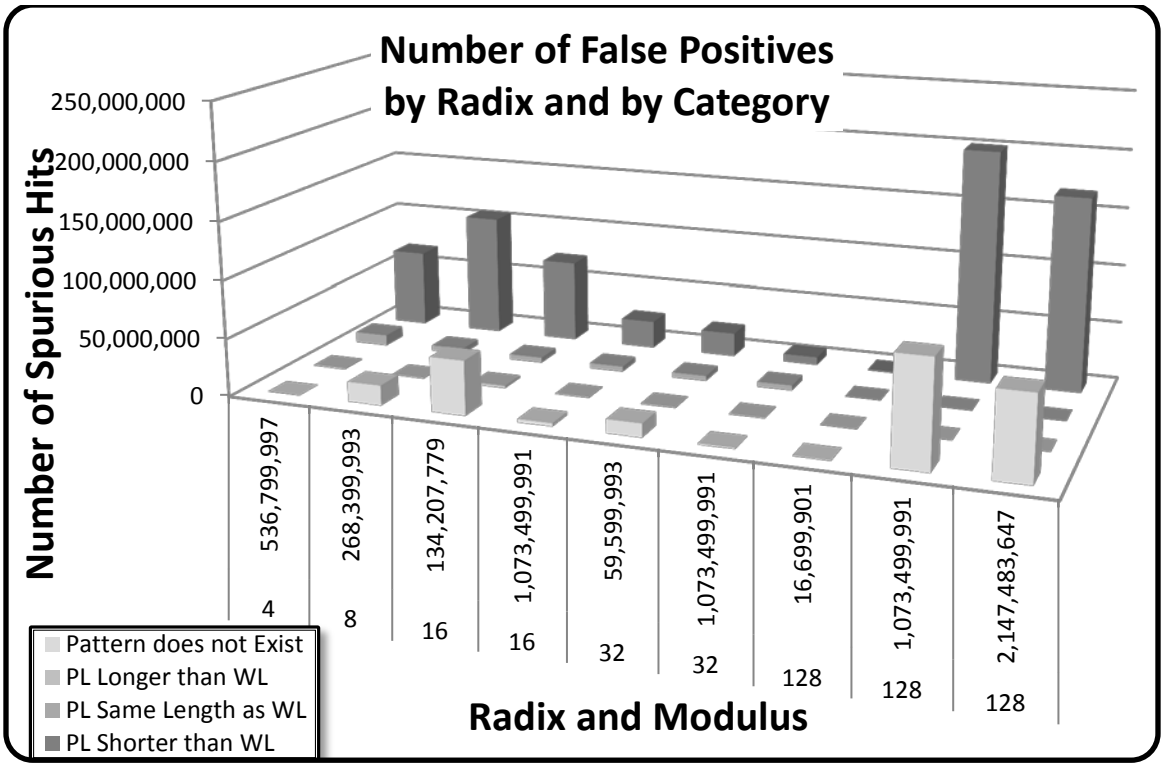

Figure 8: Number of spurious hits for each radix/modulus combinations.

The diagram shows how the spurious hits are most prominent in the back row "PL shorter than WL" and the front row "Pattern does not Exist" categories. In both cases, we find the largest values in the categories having a radix of 128 and the 2 largest modulo. Once again we have no explanation for why the non-existent patterns are even there, let alone so large. However, we can explain why the radix 128 values for large modulo exist in the "PL shorter than WL." The distance between the minimum and maximum fingerprints, we calculate using Equation 9 and Equation 10 on page 63 can be very large, thereby including all records with fingerprints between those two values. In contrast, the diagram shows radix 128 with the lowest modulus as the only combination having very few if any spurious hits in all categories.

False positives occur mainly in the PL<WL category and in the "Pattern does not Exist" category.

Observation 14: The categories with the most false positives.

5.9.3 Average Rate of Processing by Category

Figure 9 below shows the rate (in terms of hits per millisecond) at which processes run in each of the categories for each of the radix/modulus combinations. We decided to use hit/millisecond because the term milliseconds/hit produced some rather small numbers. When using this 'hits per millisecond' measure, bigger is better. The term hit refers to both actual hits and spurious hits for each category.

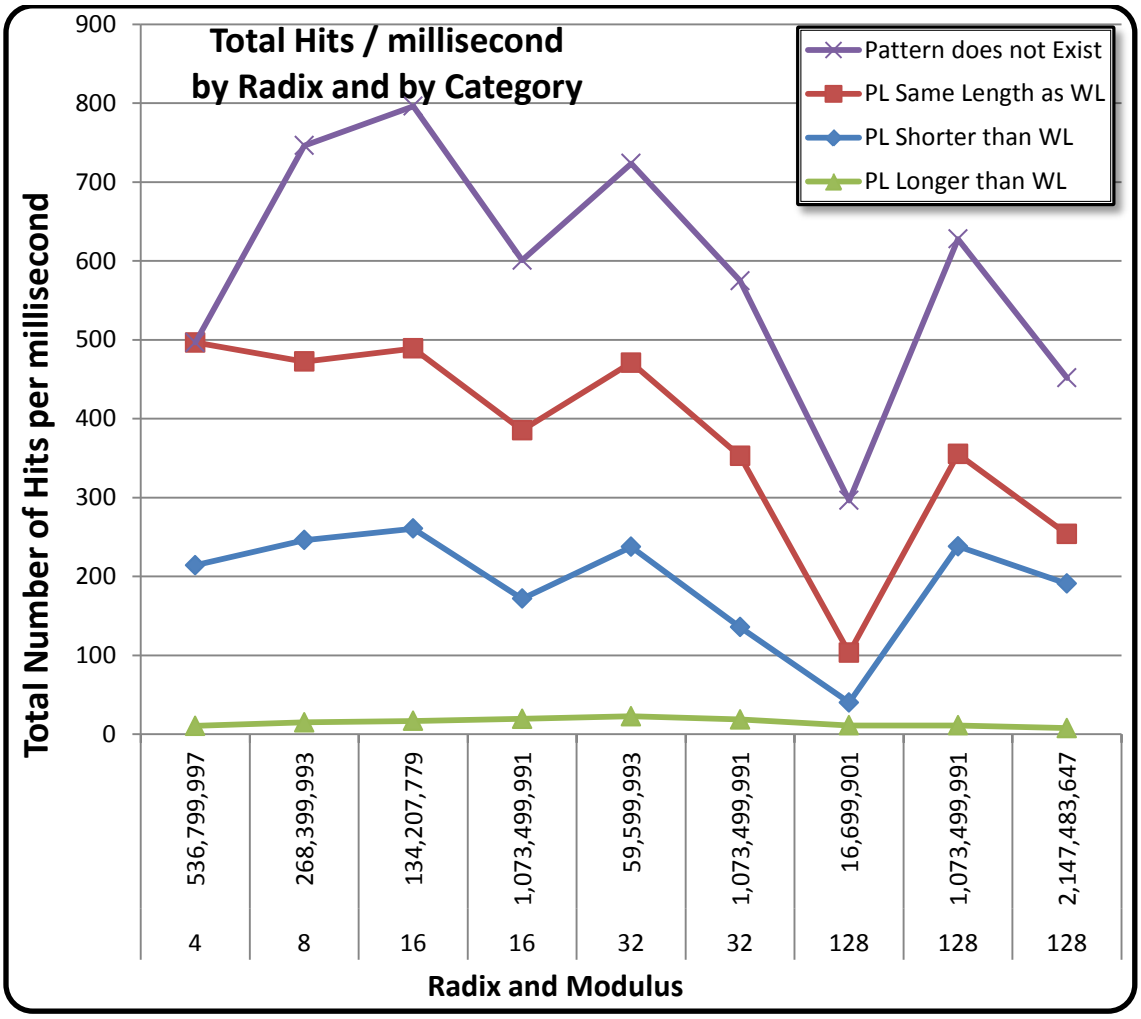

Figure 9: Time to process each category of data for each radix/modulus combination.

As one can see from this graph, the best times occur for the "Pattern does not Exist" category in each radix/modulus combination, which is the same conclusion we reached earlier. This processes flies along at rates in the order of 600 to 800 hits per millisecond. It makes sense that this is the fastest category because there would most likely be only one or two characters required to validate it as a false positive in the text file. Recall that these patterns are substrings of "ZZZZZZZZZZZZ" and, the Bible does not have any occurrences of "ZZ." Similarly, the longest times occur for the "PL longer than WL" category. Once again, this makes sense since our approach must validate a set of different WL-sized substrings, which can take longer to verifying than any of the other categories. The slowest radix/modulus arrangement was the 128/16,699,901 combination, while the fastest was 16/134,207,779 combination.

Recalling that the graph's vertical axis is 'hits per millisecond' means the lower the value the worse the time. There is a remarkable dip in all categories for the 128/16,699,901 combination. One may think this appears odd because the previous chart showed the same combination had the lowest number of false positives. However, it does make sense because this combination is only processing actual hits, each of which requires validation to check every character; whereas the other averages are "washed down" by fast false positive validations that usually finish after only a few characters.

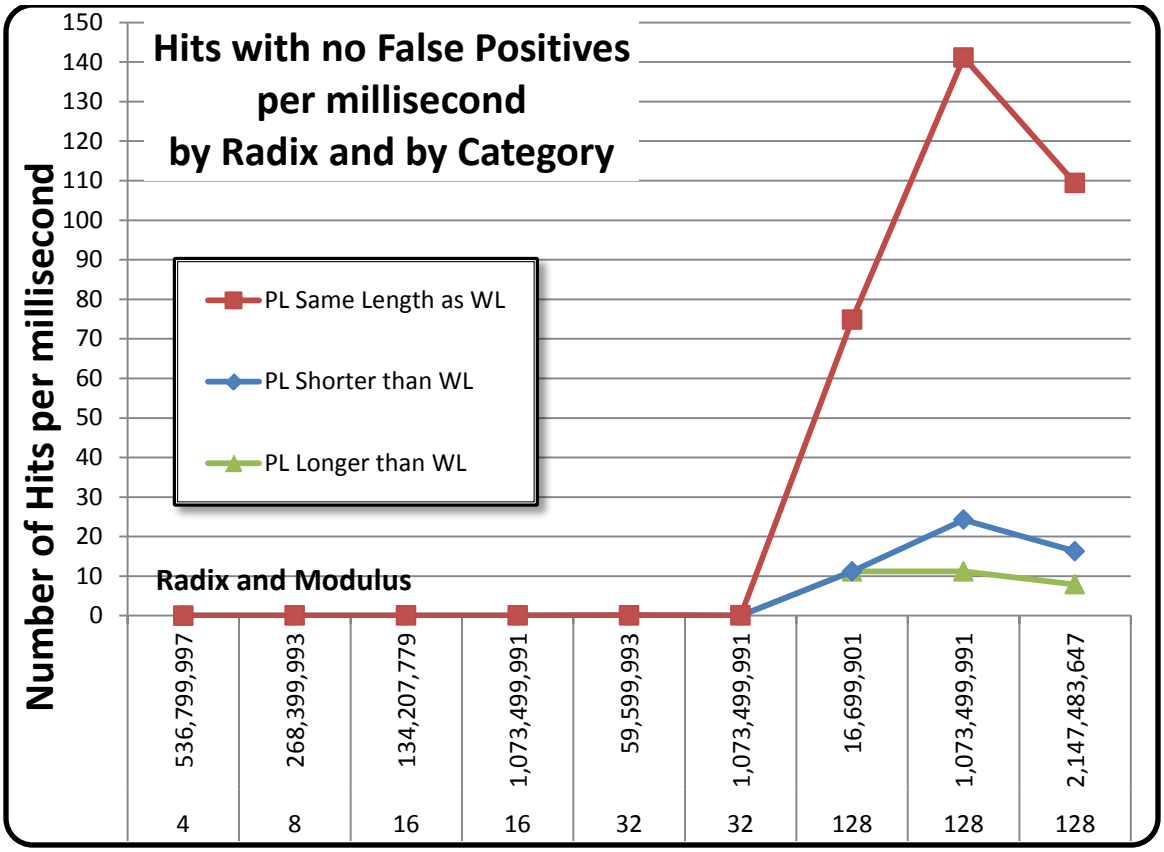

Figure 10: Performance of our approach for situations where no false positives occurred.

Figure 10 is the same graph but only shows analyses that did not have any false positives. In other words, these are the only numbers we have that demonstrate our application's speed without needing to separate false positive's time. As can be seen in this graph, these only occur for a radix of 128. The graph also shows the category where WL equals PL exhibited the fastest times. The markers that fall on the horizontal axis represent analyses that had false positives and were therefore not eligible for this graph since it is trying to capture the raw times each of the categories takes to process actual hits.

5.9.4 Search Time Performance

The following diagram shows an overview of the search times for all combinations of radix/modulus and window length. We include this figure to give a bird's-eye-view of the overall results of our experiment.

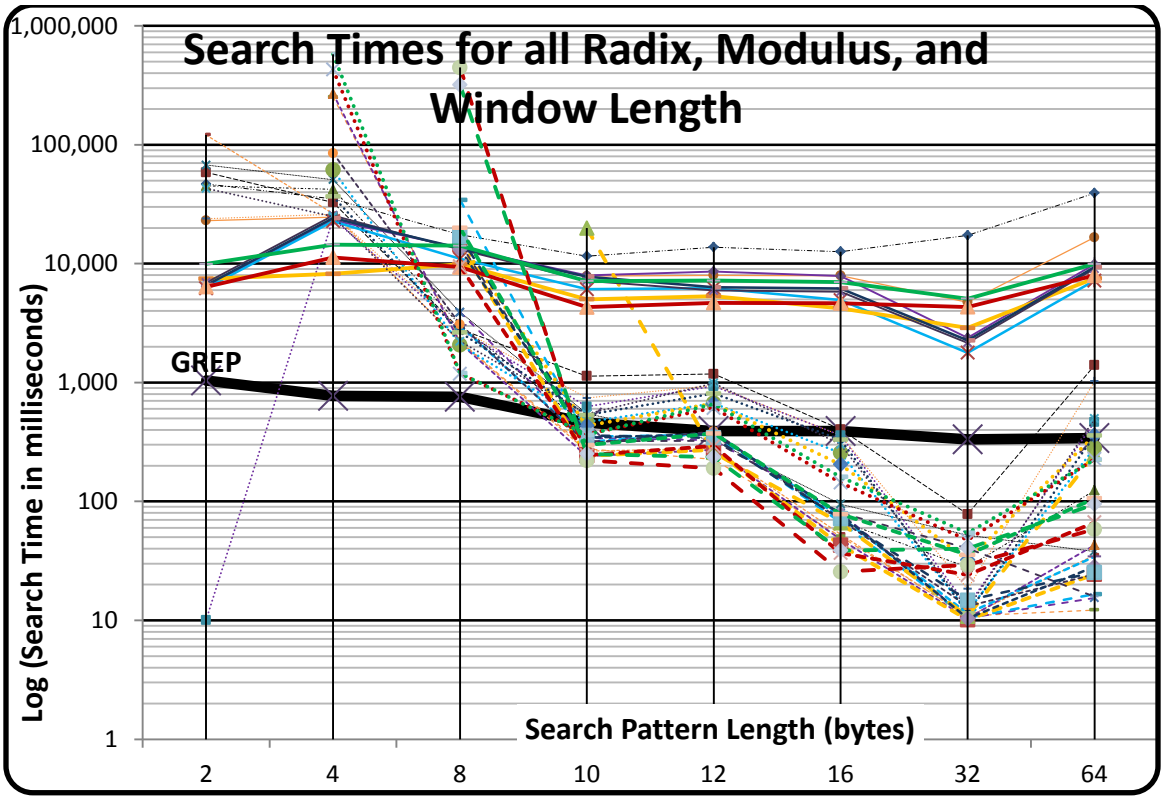

Figure 11: Search times for all combinations of radix/modulus and window length.

We included this diagram to demonstrate how a group of combinations does not run faster than GREP, while a good number of them do. To avoid cluttering up this part of the report, we break this diagram into its constituent radixes and include the individual graphs for each radix in Appendix D, since there are so many of them.

> *The analyses where our application runs faster than GREP involve larger pattern lengths, starting around 10 bytes. The groups that stay above the GREP line for all pattern lengths are all associated with a window length of four.*

Observation 15: Our application runs faster than GREP when pattern lengths are larger, but never beats GREP when the Index Database has a window length of four.

We also took the time results and did some cleaning of the data. In other words, we removed (a) the three "Pattern does not exist" rows from our results, and (b) the best and worst times from our results and from GREP. We also sorted the average times from smallest to largest. Finally, we plotted the average times for our application, the average time for GREP, and the minimum times for our application on the graph shown in Figure 12 below. The horizontal axis labels on this graph also show the radix/modulus/window length combinations in order of fastest to slowest.

> *Each radix/modulus/window length combination has eleven different pattern lengths to analyze one after the other. On an individual basis, the fastest time for the eleven analyses is always faster than GREP's average time, with the exception of combinations having a window length of four. Collectively, when we average the times for the eleven searches, most of these averages times are faster than GREP's average.*

Observation 16: The fastest time of the eleven pattern length searches in each radix/modulus/window length combination was faster than GREP except for searches involving a window length of four.

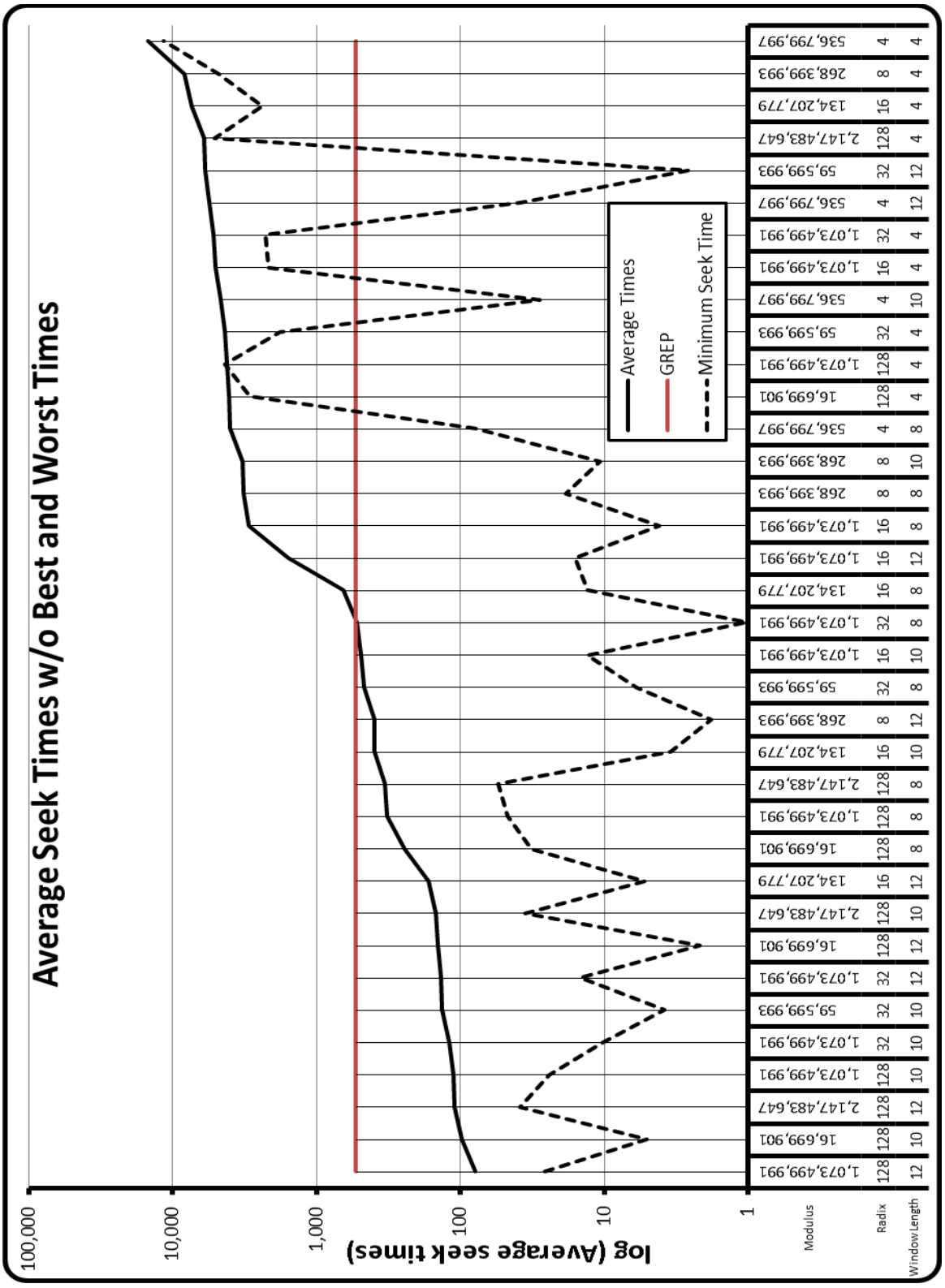

Figure 12: Average and lowest times for our application versus average GREP times

5.9.5 Comparing Actual and Spurious Hits

In the same manner as earlier, we placed another group of diagrams in Appendix E to avoid cluttering up this part of the report. We show a sample of these graphs in Figure 13 below. It consists of a bar chart showing the positive (actual) hits in white and negative (spurious) hits in black for each pattern length and each window length.

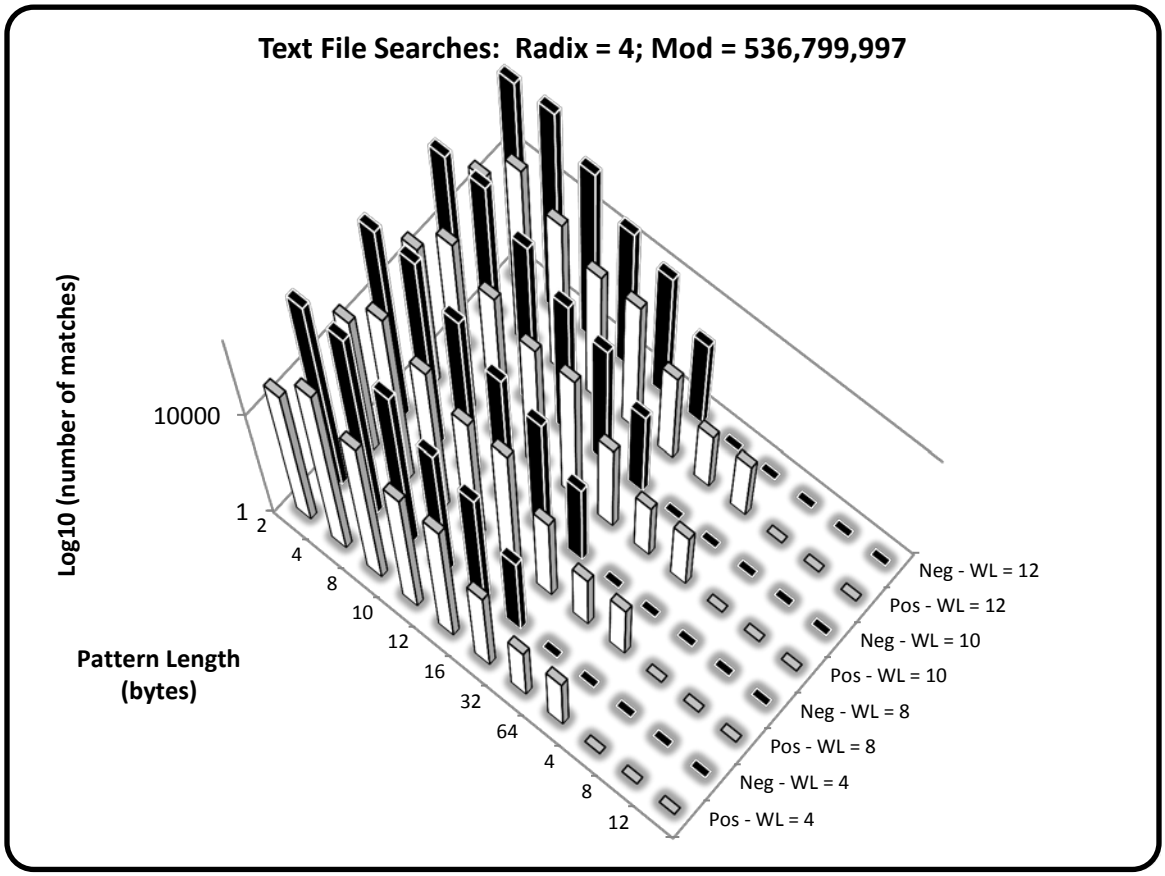

Figure 13: Actual and spurious hits for Radix = 4.

The chart shows that for all but the 32 and 64 byte search strings have about as many positive hits as negative (spurious hits) for this particular radix/modulus combination. It also shows how this combination does not have any strange false positives in the last three positions where no pattern exists.

Chapter 6: Conclusion & Future Work

The work required for the information presented in this report has covered a great deal since it began. Aside from a few setbacks here and there, there is little doubt it was a success. As its title alludes, we present this final chapter in two parts: Conclusions and Future Work. Conclusion looks back at our motivation and thesis from the first chapter, briefly reviews not only the application we built but also the results of experiments we ran using it. Future Work covers all those tasks we need to look at to make this application more useable such as reducing the database size and improving several parts of our algorithm.

6.1 Conclusions

We begin the Conclusion by reflecting on our original motivation and Thesis. Following that, we will briefly describe both steps. Next, we will examine the experimental results we achieved running our system and comparing our results to GREP. We remind avid readers to review the "Observations" boxes above to refresh some of the many of observations we made during the experiment and while writing the respective part of this report. In short, there is enough positive evidence to not only support the original thesis, but also to merit putting more resources into it; perhaps even for smart devices.

6.1.1 Original Motivation

Way back in the Introduction we began our motivation discussion describing one-step search engines and talked about their popularity for on-line searching. We then introduced a two-step method; and more or less described it as follows:

Few, if any, local text search applications in widespread use have a two-step approach. Most current local text search applications, like GREP, run on-line. This means that work done in the current search is independent of, any and all other work already done in a previous search. A two-step-text-search-engine builds a database in the first step; thereby allowing users to take advantage of doing many kinds of pre-calculations once, and subsequently using their results many times for any and all searches ever needing to

be run. One-step applications like *GREP* have no memory. Instead, they perform all required calculation each time they perform a search.

Building a two-step local search engine was our motivation.

6.1.2 Thesis

The above motivation steered us to the following Thesis.

> ***We can modify the Rabin-Karp Algorithm, and configure the LevelDB database, to create a two-step-text-search-engine algorithm that will outperform the one-step-text-search-engine GREP in finding search patterns within local text files.***

In a general sense, the two-step application we built does beat GREP more than half the time. It also is four orders of magnitude faster determining when a string does not exist. It also seems feasible to apply a list of tasks that will make it beat GREP every time. We will present that list of tasks later.

6.1.3 Building the Application

We used C++ to build our application so anyone could compile and use in either Windows or UNIX. There was one feature we needed a more precise timer function we obtained from <Windows.h> that invalidates this assumption. However, comparable timer function APIs exist for UNIX that can easily replace the one we used from Windows.

We also downloaded and linked LevelDB into our application. Since LevelDB is a background kind of API, it had no bearing on the fact that we built our application to run in a command-line environment. It has about a dozen or so possible parameters explained in Appendix C. Since we used Windows, we also used Visual Studio 10 as our IDE, and ran our app from a DOS window. During testing, we would ran our application directly from a DOS window. For our experiments, we used a batch file to sequence

several analyses together. This was especially helpful for writing the entire set of 100-Bible Index Databases whose batch file ran overnight.

In the end, our two-step application worked as follows. A preprocessing step using a modified Rabin-Karp algorithm to create fingerprints that we save in a hash file using LevelDB, which we usually refer to as an "Index Database." After building an Index Database, our application used it and a set of parameters to find the positions of all occurrences of a search string quite quickly. To get our application in a position to perform both steps we needed to write a great deal of C++ code.

Earlier, we described several techniques from the Rabin-Karp algorithm we used to carry information from one fingerprint calculation forward to the subsequent fingerprint calculations while building the hash file. We also built look-up tables and several other techniques we thought would speed everything up. While we were building our application, we were thinking these kinds of improvements would significantly reduce the time required to build an Index Database. Unfortunately, even though this part of the processing had better performance, it only characterizes about 0.1% of the total time actually required to build a Database Index; LevelDB used the other 99.9%.

We made several other substantial improvements in the Rabin-Karp Algorithm worth mentioning. First, we built the functionality to perform searches for five different circumstances: (1) (PL - WL) is too big, (2) PL<WL, (3) PL=WL (this is the only function the original Rabin-Karp algorithm covered), (4) PL>WL, and (5) search Pattern does not exist. Second, we had to build the functionality to understand the architecture type, decide whether integers were Little Endian or not, and, if so, switch the integer's byte arrays into Big Endian format for use in LevelDB.

Finally, we had to not only test our application, but also design an experiment to help produce evidence for our thesis. We did both. Earlier we described how our testing went smoothly, demonstrating only minor issues that we repaired easily compared to the above work. The following section describes our observations during the experimental portion of this work.

6.1.4 Experiment and Results

After we constructed and tested our application, the next part of our work involved us designing and running an experiment that would help decide how our application faired against GREP. We had a rather lengthy, but familiar, list of parameters to accommodate in our experiment. These were: (1) radix, (2) modulus, (3) window length, and (4) search pattern length.

Our Index Database structure incorporated the first three of these parameters. We used four radixes: 4, 16, 32, and 128. Next, first we used one modulus for each radix based on not allowing any calculation to overflow a signed 32-bit integer. Recalling a modulus must be a prime number, for these modulo we found the largest prime number smaller than (2^{31}/radix). This approach resulted in four experiments, to which we added four more modulo; one for radix 16 and 32, and two for radix 128. This makes eight combinations so far. For each of these combinations of radix and modulus we built four Index Databases, one for each of our four window lengths: 4, 8, 10, and 12. In the end, we ran 36 step-one processes to begin our experiment that resulted in 36 different Index Databases.

Next, we needed to test these databases against a range of pattern lengths. We used two types of search patterns to accomplish this. First, we used eight patterns whose strings we knew existed in the text file, and whose lengths were 4, 8, 12, 16, 32, and 64 bytes. Second, we used three patterns we knew did not occur in the text file whose lengths were 4, 8, and 12. This gave us eleven Patterns to run through each of the 36 Index Databases resulting in 396 different runs.

By far the fastest of these runs were for patterns that we knew did not occur in the text file. All of these, with the exception of a few surprising runs having false positives, ran at speeds that were faster than 0.01 milliseconds. In fact, the fastest of these searches occurred for a radix/modulus combination of 128/1,073,499,991; a window length of 4; and a pattern length of 4. This search took a trifling 0.0075ms. At the same time, our slowest search occurred for a radix/modulus combination of 128/ 2,147,483,647, a

window length of 8, and a pattern length of 4. This search took an astounding 576,194.72ms to find 1,216,800 occurrences. The reason this search took so long was the time needed to rule out the huge number of false positives, 106,923,398 of them to be exact.

During our analysis of the results, we noted that in cases like above where the pattern is smaller than the window length, we use a 'from fingerprint' and a 'to fingerprint' to find all locations that may contain the search pattern. In such situations, especially with a large radix, the distance between the 'from' and the 'to' fingerprints is so large our application must sift through millions of spurious fingerprints looking for the search pattern. The reason for the slowness for these types of searches is a practical matter of volume of work rather than some sophisticated theoretical justification.

Naturally, therefore, GREP was faster than our application for most searches where the pattern length was less than the window length. However, this was the only category where GREP was a clear winner. In all other categories, especially, when searching for a string we know does not exist, our application was faster than GREP with a minor exception here and there.

6.1.5 Final Conclusion

Now that we have conducted all the experiments and examined the evidence from several interesting perspectives, we submit that we should look at the evidence from two points of view, holistically and individually. From a holistic standpoint there is enough positive evidence to not only support our original thesis, but also to merit putting more resources into the application, perhaps with an eye towards smart devices. From a more individual viewpoint, our experimental results did not help us make statements one way or another about a combination of values for any of the four parameters (radix, modulus, window length, and pattern length) that was the best; or, that was even better than most. We have a few indications about the worst, like window lengths of 2 and 4; but not best. It is probably more descriptive to say that our application is not yet mature enough to reveal sufficient evidence to support these kinds of decisions.

We will address all of these issues in our recommendations that follow.

6.2 Recommendations and Future Work

Our experiments show that this new approach has a great deal of potential to help improve the performance of local searches. Moreover, although we did not even look at wider searches, our application could help in searching enterprise data stores as well. However, before we can even consider performing these kinds of search, we must ensure we make several improvements our experiments brought to light. These are as follows.

6.2.1 Find out why False Positives Appeared in Non-existent Strings

One of the most frustrating parts of our results was the existence of false positives for strings we know did not exist in the file. We need to investigate why and how strings like "ZZZZ" and "ZZZZZZZZ" and "ZZZZZZZZZZZZ" could produce fingerprints that existed in an Index Database; in one case 91,505,516 times. We checked the source file and verified the strings themselves did not occur. Therefore, we must do some research to find out what is happening in these cases.

6.2.2 LevelDB Tuning

We mentioned earlier in the report that LevelDB has several parameters we can use to tune the database. Given the fact that building the Index Database is slow and end-up being huge, we must investigate them and adjust them to make our step-one perform as nicely as step-two does.

6.2.3 Improve Approach to Small Search Strings

Searches whose pattern length (PL) is less that the window length (WL) are difficult and much more susceptible to performance lags due to a wide search area. In fact, this formulation and approach cannot beat GREP's performance for any combination of parameter values. As we just pointed out our range of fingerprints is so wide, we find that most of the time spent searching is in eliminating false positives. We need to find a way to tighten up the way we handle these types of searches.

6.2.4 Remove GREP's Advantage

Throughout the report, we have reminded readers about GREP counting lines only. By far the most time consuming part of our algorithm is verifying character strings in the text file match the search string. Therefore, when a line has more than one occurrence of a search pattern our application must verify them all; whereas GREP only need look at the first occurrence. This means our comparisons are biased toward GREP. It is essential we find a mechanism to correct this biased.

6.2.5 More Documents

Currently we have our application configured to only work with one text file at a time. There is no reason why the application can produce one index file for a set of documents.

6.2.6 Find Parameters

A theme of our experiment was to vary the parameters one at a time and compare the results in terms of performance, which was successful. However, we had trouble identifying a set of parameters that stood out as the best. This leaves our basic question as to, "what are the best parameters?" unanswered. We need another experiment designed without any GREP comparison. This experiment will have an objective to identify the best set of parameters.

Bibliography

Anonymous. (1989, Aug 1). *The Bible, Old and New Testaments, King James Version*. Retrieved December 12, 2012, from Project Gutenberg: http://www.gutenberg.org/ebooks/10

Baeza-Yates, R., & Ribeiro-Neto, B. (1999). *Modern Information Retrieval* . New York: ACM Press.

Beaza-Yates, R., & Ribeiro-Neto, B. (1999). *Modern Information Retrieval.* New York: ACM Press.

Bentley, J. L., & McLiroy, M. D. (2003). *Patent No. US 6,611,213 B1*. United States of America.

Bille, P., Gørtz, I., Sach, B., Vildhøj, H., Kärkkäinen, J., & Stoye, J. (2012). Time-Space Trade-Offs for Longest Common Extensions. *Lecture Notes in Computer Science-Combinatorial Pattern Matching, 7354*, 293-305.

Corman, T. H., Leiserson, C., Rivest, R., & Stein, C. (2001). *Introduction to Algorithms.* Cambridge: The MIT Press.

Crochemore, M., C., H., & Lecroq, T. (2009). *Alorithms on Strings.* Cambridge: Cambridge University Press.

Dean, J., & Ghemawat, S. (n.d.). *Leveldb*. Retrieved 01 12, 2012, from Google Project Hosting: http://leveldb.googlecode.com/svn/trunk/doc/index.html

Fuyao, Z. (2009). A String Matching Algorithm Based on Efficient Hash Function. *Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on* (pp. 1-4). Wuhan, China: IEEE eXpress Conference Publishing.

Gnu. (2009, Feb 13). *Grep for Windows*. Retrieved Feb 10, 2012, from Sourceforge.net: http://gnuwin32.sourceforge.net/packages/grep.htm

Google. (2011). *leveldb (A fast and lightweight key/value database library by Google)*. Retrieved 4 10, 2011, from Google Project Hosting: http://code.google.com/p/leveldb/

Google Inc. (2012, 05 30). *leveldb*. Retrieved 06 01, 2012, from Google Project Hosting: http://code.google.com/p/leveldb/

Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences.* Cambridge: Cambridge University Press.

Intel Corporation. (2004, 11 15). *Endianness White Paper*. Retrieved 12 2, 2011, from intel.com: http://www.intel.com/design/intarch/papers/endian.pdf

Karp, R., & Rabin, M. (1987). Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development*, 249-260.

Kijewski, P. (2006). Automated Extraction of Threat Signatures from Network Flows. *18th Annual FIRST Conference.* Baltimore: CERT Polska/NASK.

NASK (Research and Academic Computer Network), P. (2012, August 09). *Home Page*. Retrieved 09 08, 2012, from ARAKIS, a CERT Polska (NASK) project: http://www.arakis.pl/en/index.html

Navarro, G., & Raffinot, M. (2007). *Fexible Pattern Matching in Strings.* Cambridge: Cambridge University Press.

Prata, S. (2005). *C++ Primer Plus (Fifth Edition).* Indianapolis: Sams Publishing.

RapidTables.com. (2011). *Little and Big Endian*. Retrieved 12 02, 2011, from RapidTables.com: http://www.rapidtables.com/prog/endianess.htm

The Large Canterbury Corpus. (2001, 11 20). *English Natural Language Corpus*. Retrieved 10 17, 2011, from The king James version of the bible: http://corpus.canterbury.ac.nz/

Twain, M. (2009, May 14). *The Works of Mark Twain by Mark Twain*. Retrieved December 12, 2011, from Project Gutenberg: http://www.gutenberg.org/ebooks/28803

Appendix A: Formalizing Characters, Strings, and Search

A.1: Introduction

This chapter covers the following topics. To begin, we briefly repeat a point from the Introduction concerning the strings involved in our work, to emphasize their importance. Next, we commence our formal system discussing a string's most important element, a character, (or symbol.) We discuss the difference between the two and describe some of their important properties. We then use that formalism to move the discussion into strings, their characteristics, and their operators and functions. The discussion is impressively both broad and deep, due to the importance of string analysis in science. Finally, we once again address our favourite topic, the set of strings involved in this study. This time, we dedicate a whole section to each of them discussing their role in the analysis and a set of useful properties and characteristics. Finally, to make or formal definitions stand out we use a simple bullet list format (NOTE: We do not use bullets or bullet lists anywhere else throughout the text except for this.):

The Chapters that follow make use of a string. In fact, we have made it abundantly clear strings are the cornerstone of this work. We therefore begin our discussion providing a rigorous formalism to strings, their elements, and a few characteristics. This formalism will provide mathematical and notational mechanisms needed to include strings, their elements, and their characteristics in subsequent formal discussion. Doing this, forces us to begin with a string's atomic unit, a character (or symbol.)

- A text string (**S**) is a contiguous set of characters.
- **S** has a length (**SL**) equal to the number of characters it has (i.e., **SL = |S|**).
- Every character in a text string occurs at a unique position (**i**) within a string. The first character in a string occurs at position **i = 0**. The last character in a string occurs at position **i = SL - 1**.

An easy way to think of a text string is as an array of characters where the array index acts as a position within the string. As above, the array representing the text string **S** has an index (**i**) with a range [**i = 0** to **SL - 1**]. Arrays and their indexes offer the kind of accurateness we need when referring to a character at a particular position. For instance, **S[j]** or $\mathbf{S_j}$, refer to the **jth** element of string/array **S**. Similarly, we use **S[i…j]** or $\mathbf{S_{i\ \ j}}$ when referring to substrings of **S** beginning at position **i** and ending at position **j** (where, **i < j** and **j < SL**.)

A.2 Brute-Force Algorithm

Armed with this information we can now examine a method of finding one string inside another. To that end, we turn to giving a formal description of the brute-force algorithm for text search. We hope this will define a benchmark of sorts on how the most basic form of text searching is accomplished. Before we can give a brute-force solution, we must first clearly define a problem. We have described it in words above. Now we formally defined the *text search problem as* follows.

- Given a pattern **P** of length **PL** and a text file **T** of length **TL** (where PL is much smaller than TL), search all position **l** of **T** and report all occurrences where $\boldsymbol{P_0^{PL-1} = T_l^{l+PL-1}}$ **(l = 0, 1,…, TL-PL).**

We treat the terms Brute-Force and Naïve as being interchangeable. They both describe a procedure that checks the first position in a text file to see if its following characters match every character in the search pattern. It records the results and moves to the second character in the text file to perform the same check. It repeats this process until it has checked each substring in the text file having a length of **PL**.

Earlier we defined Text Search as a process of finding the occurrences of a search pattern **P** of length **PL**, in a text file **T** of length **TL** (where **SL << TL**.) Since Cormen gives

such a succinct description of a naïve (and brute-force) text-search algorithm, we reproduce it here as follows (Corman, Leiserson, Rivest, & Stein, 2001)

```
Naïve-Text-Search(T, P)

2 PL ← Length(P)

1 TL ← Length(T)

3 for  s ← 0 to TL – PL

4    do if P[0 … PL-1] == T[s … s+(PL-1)]

5         then print "Pattern occurs at position" s
```

Algorithm 4: Naive Text-Search

While easy to understand and easy to program, having a worst-case running time of Θ(nm) makes the Brute-Force method unsatisfactory for most applications including ours. Even the practical running time of this method is still too slow for large texts. Consider for instance, when **T** represents one or more web sites, **TL** gets extremely large (10^9 characters is a realistic number in such an application) the shear bulk of characters to be checked makes it obvious that response times would be totally inadequate for modern search applications.

The *worst-case* for this solution operates in time *O*(**PL(TL – PL + 1)**). A *worst-case* scenario exists when both **P** and **T** contain the same repeated character (recall that overlaps are allowed.) In such a case, the number of comparisons made is Θ*(***PLTL***)* because there is an occurrence of **P** at each of the first **TL – PL+1** positions in **T**; and the method performs exactly **PL(TL-PL+1)** comparisons. For instance, if **P** = `aaa` and **T** = `aaaaaaaaaa` (**PL=3, TL=10**,) then the *worst-case* is encountered with exactly 24 comparisons being made.

The algorithm gets a text file **T** and a search pattern **P** as parameters. Notice at line 3 how a *for-loop* covers each position in the text file one position (byte) at a time. At each offset, we align the left end of **P** with the left end of **T** and compare the characters of **P** and **T** left-to-right until a *mismatch* occurs or **P** is exhausted. If a mismatch occurs, we continue below at shifting. If **P** is exhausted without a mismatch, we report an *occurrence* at the position of the left end of **P**. If, however, all characters of **P** *match* all characters of **T**, we report an *occurrence* as above. **P** is then *shifted* one character to the right by returning to line 3, where a string comparison is repeated beginning at the left end of **P**. The for-loop is repeated until the right end of **P** is at location **TL**; which is one character past the right end of **T**. (Corman, Leiserson, Rivest, & Stein, 2001) and (Gusfield, 1997)}

A.3 Character Characteristics

The atomic unit of a string is a *character (or symbol)*. Later in this chapter, we deal with our preference to use the term character over the term symbol because the work we do deals mainly with strings taken from the English language. In that sense, it feels more natural to refer to *English characters* than to refer to *English symbols*. The result is that our preference is to use character. However, we may encounter situations where we find the term symbol more convenient and precise. Whichever sounds better is likely to be our choice. This section formalizes these concepts.

A.3.1 Characters and Symbols

It is difficult to overstate the importance of symbols in general, and characters in particular; their usage after all constitutes visible portion of an alphabet. They form the basic building blocks for some major branches of science that deal with information; library science, information management, communications, and computer science to name a few. They also form the basic building block of this work. Although, our main goal is manipulating strings, we must first understand a string's elements, their

characters, and some of their fundamental properties before we can evaluate them when they occur in strings.

The programming language C illustrates the two-sided property of characters nicely. In C, a **char** type has the unique property that it is both a character type and an integer type simultaneously. In other words using a character's integer representation, we can write its alphabet as a set of integers by, $\mathbf{\Sigma = \{i \mid i = 0 \ldots (|\Sigma|-1)\}}$. We can also write it as a set of characters by, $\mathbf{\Sigma = \{c_i \mid i = 0 \ldots (|\Sigma|-1)\}}$. Looking at ASCII character '**a**' as an example, it has the following bit code: **`0110 0000`**. Interpreting these eight bits as an integer evaluates to a code of $\mathbf{(1*2^5 + 1*2^6) = 96}$. Thus, C regards lines: `char c = 'a';` and `char c = 96;` as both valid and equivalent.

We take advantage of this equivalence throughout our application code. In addition, later in this document, we will treat characters as symbols and integers interchangeably depending on the situation. It is important to notice that in this case we are using zero-based addressing as a convention that is popular in textbooks. We also find it more helpful programming in C. However, one-based addressing is just as common in the literature.

A.3.2 The ASCII Character Set

The most popular encoding for English language texts is ASCII (American Standard Code for Information Interchange) defined by (ANSI13.4-1968. See http://www.research.att.com/~bs/glossary.html#GANSI) We use ASCII in our analysis because we are restricting our study to English language corpuses. Even though ASCII has elements that we typically refer to as symbols, the vast majority of ASCII includes all upper and lower-case English letters, punctuation marks, digits, and arithmetic operators. We will therefore henceforth begin referring to most ASCII symbols as characters.

- ASCII is a simple one-byte character encoding where each element in the alphabet has a unique one-byte bit arrangement. At the same time, the seven

lower order bits from that same byte define the character's code as an integer. As we demonstrated earlier, this means each ASCII character has a natural zero-based code build into its byte.

While it may be tempting to refer to an ASCII code as an "index," we will resist the temptation. The reason for this is that later we use the term "index" to refer to a character's position within a string. It is vital we do not mix-up a character's code (an integer value,) for its index-position within a string-(another integer value.) The two are vastly different concepts.

Alphabet index or ASCII code, it does not matter what name one calls it; there can never be any confusion about its value. We calculate it from the lower seven bits of its byte; using the eighth bit for a sign. Incidentally, we ignore negative ASCII codes. Appendix A shows an ASCII table that demonstrates |ASCII|=128. The codes go from 0 up to 127. Another way of confirming a code's value range is to observe that $2^7 = 128$, then minus one for 0 gives 127 as our upper limit.

A.3.3 String Characteristics

Having introduced the basic atomic elements of strings, we are now in a position to discuss some characteristics of strings. Our analysis later will make use of most of these characteristics. The first point to understand is that we use the term string and sequences synonymously to mean an ordered finite-length set of characters from an alphabet. We use string or sequences depending on the context; but we heavily prefer to use string.

Another point to notice is our approach deals with strings that may or may not have any meaning to us. The issue of meaning does not have any role in our analysis. We simply aim to analyze the juxtaposition of characters in a string and use that result to help us find positions where that string occurs in other files. The features and characteristics of these strings are all the tools we need for our approach.

A.3.4 String Nomenclature and Properties

We begin by formalizing several interrelated terms and concepts needed to describe a string's features and characteristics. They designate not only strings themselves, but also their characteristics, and several operations we perform on them during our approach. The following list terms and concepts have been adopted from (ref=GUSFIELD; Algorithms on Strings, Trees, and Sequences). We repeat earlier definitions of a string for completeness).

- A string **S** is a finite-length set of contiguous characters written from left to right. It is helpful to think of a string as a one-dimensional array of characters.
- The characters contained in a string belong to some finite alphabet **Σ**. Our analysis relies on the ASCII alphabet or some subset of it.
- **Σ*** denotes a set of every finite length string that can be formed using characters from the alphabet **Σ**.
- The empty or NULL string, denoted **ε**, contains no characters. Incidentally, **ε** belongs to **Σ*** as well as every other string having **ε** as a suffix, prefix, or both.
- Each character in a string exists at a location or *position* (as we prefer) within the string.
- The *position* is calculated by counting the number of characters beginning at the first character on the left end of a string; which is assigned a position of 0. The second character has a position 1, and so on. It is especially important to note that we adopt a convention where the first position in a string is zero (not one.) It is not uncommon to encounter in the literature, not only descriptions for string manipulation using positions that are one-based, but also, use several other terms

describing a character's position in a string. For instance, depending on the context, we sometimes conveniently refer to a position as an *offset*; a *location*; or, *shift* from the beginning of a string.

- We have already seen earlier, the *length* of a string **S** is the number of characters it contains, and is denoted **SL**, or mathematically |**S**|. Moreover, using a zero-based approach, the position of the last character in a string is |**S**|**-1** or **SL-1**.
- **S[i]** signifies the *character* of **S** occurring at position **i**. Remember, zero-based positioning means all positions must fall in the range: **0 <= i <=** |**S**|**-1**
- **S[i,…,j]** is a *substring* of **S** that starts at position **i** and ends at position **j** (where **i ≤ j**).
- Sometimes, we shed the square brackets finding it convenient to denote a substring of **S** that begins at location **r** and has a length of **n** using the term: $\boldsymbol{S_r^n}$. Furthermore, in situations where we know **n** by the context (or understand it), we omit **n** and simply use $\mathbf{S_r}$ instead. This nomenclature also helps us reduce some equation's size and clutter when describing string characteristics and properties.
- *Concatenating* two strings **S** and **T** produces a new string, written as **ST**, consisting of the characters from **S** followed by the characters from **T**. The length of this new string is |**S**| + |**T**|.
- The substring **S[0,…,i]** is a *prefix* of **S**, beginning at the first position and ending at location **i**.
- Similarly, the substring **S[i,…,**|**S**|**-1]** is a *suffix* of **S** that begins at position **i** and ends at the last position in **S**.

- Note that **ε** is both a prefix of and a suffix of all strings. Similarly, a string **S** is also both a prefix of and a suffix of itself. To distinguish these trivial cases, the term *proper* is prepended to prefix, suffix, or substring if the prefix, suffix, or substring is (a) not the entire string **S[i,…,j]**, where **i**=**0**, and **j=|S|-1**, and (b) is not the empty string **ε**. For instance, a substring **S[0,…,3]**of **S** is a *proper prefix* of **S**, iff **|S| > 4**.
- It is an error to define a substring **S[0,…,3]**of **S** if **|S| < 4**.
- When comparing a character from one string with a character from another string, they *match* if their characters (and/or codes) are equal; otherwise, they *mismatch*.
- When two strings (say **S** and **R**) of the same length **n** are compared, we say string **S** *matches* string **R** if every character in respective positions from the left to the right match (i.e., they match iff **S[i]=R[i]** for all **i=0,…,n-1**); otherwise, **S** and **R** *mismatch*.
- When two strings have different lengths, we compare characters in the *shorter* string to corresponding characters in a proper prefix of the longer string. If each character in the shorter string matches the corresponding characters in the proper prefix of the longer string, we say that the shorter string *matches* the longer string; otherwise, they *mismatch*.

A.4 The Text Document

Among one of the most important strings we deal with is a *text file*. When a user begins an analysis, they must supply a string **T** called a *text file* of length **TL** as the object they wish to search. Optionally, they can also provide a string **P** called a *search pattern* of

length **PL** they wish to find in **T** (where **PL<<TL**). As described earlier, we can solve this problem by searching for all occurrences of **P** in a string **T**. In a general sense-that is, while not following the earlier algorithms Brute-Force or Rabin-Karp-the results of our analysis amounts to repeating a comparison for all proper substrings of **T** having a length **PL**, from position **0** to position **TL-PL**; and, reporting an *occurrence* at offset **l** whenever $\boldsymbol{P_0^{PL-1} = T_l^{l+PL-1}}$**.** Fortunately, we do not need to perform so many operations to obtain the same results. The point here is that a search for a pattern **P** must report every occurrence of **P** in the text file **T**.

When we open a text file, we do it in *binary mode* so we do not have to worry about the end-of-line issues associated with opening them as text files. Once opened, **T** has *positions* that start at zero for the first character and end at (**TL-1.**) They are the source of values used to satisfy an information need such as when declaring, "a pattern occurrence was discovered at this position in T." Our application provides occurrence positions in a file or on the screen after performing a search process.

A.5 The Search Pattern

A search pattern is another value a user must supply when using step-two the matching step. As we have reiterated many times throughout this report already, a user supplies a string **P** called a *search pattern* of length **PL** they wish to find. The application then returns a file containing the position of each occurrence of that pattern in a text file.

Appendix B: ASCII Table

Dec	Hex	Binary	Character	Description
0	00	00000000	**NUL**	null
1	01	00000001	**SOH**	start of header
2	02	00000010	**STX**	start of text
3	03	00000011	**ETX**	end of text
4	04	00000100	**EOT**	end of transmission
5	05	00000101	**ENQ**	enquiry
6	06	00000110	**ACK**	acknowledge
7	07	00000111	**BEL**	bell
8	08	00001000	**BS**	backspace
9	09	00001001	**HT**	horizontal tab
10	0A	00001010	**LF**	line feed
11	0B	00001011	**VT**	vertical tab
12	0C	00001100	**FF**	form feed
13	0D	00001101	**CR**	enter / carriage return
14	0E	00001110	**SO**	shift out
15	0F	00001111	**SI**	shift in
16	10	00010000	**DLE**	data link escape
17	11	00010001	**DC1**	device control 1
18	12	00010010	**DC2**	device control 2
19	13	00010011	**DC3**	device control 3
20	14	00010100	**DC4**	device control 4
21	15	00010101	**NAK**	negative acknowledge
22	16	00010110	**SYN**	synchronize
23	17	00010111	**ETB**	end of trans. block
24	18	00011000	**CAN**	cancel
25	19	00011001	**EM**	end of medium
26	1A	00011010	**SUB**	substitute
27	1B	00011011	**ESC**	escape
28	1C	00011100	**FS**	file separator
29	1D	00011101	**GS**	group separator
30	1E	00011110	**RS**	record separator
31	1F	00011111	**US**	unit separator
32	20	00100000	**Space**	space
33	21	00100001	**!**	exclamation mark
34	22	00100010	**"**	double quote
35	23	00100011	**#**	number
36	24	00100100	**$**	dollar
37	25	00100101	**%**	percent
38	26	00100110	**&**	ampersand
39	27	00100111	**'**	single quote
40	28	00101000	**(**	left parenthesis
41	29	00101001	**)**	right parenthesis
42	2A	00101010	*****	asterisk
43	2B	00101011	**+**	plus
44	2C	00101100	**,**	comma
45	2D	00101101	**-**	minus
46	2E	00101110	**.**	period
47	2F	00101111	**/**	slash

Dec	Hex	Binary	Character	Description
48	30	00110000	**0**	zero
49	31	00110001	**1**	one
50	32	00110010	**2**	two
51	33	00110011	**3**	three
52	34	00110100	**4**	four
53	35	00110101	**5**	five
54	36	00110110	**6**	six
55	37	00110111	**7**	seven
56	38	00111000	**8**	eight
57	39	00111001	**9**	nine
58	3A	00111010	**:**	colon
59	3B	00111011	**;**	semicolon
60	3C	00111100	**<**	less than
61	3D	00111101	**=**	equality sign
62	3E	00111110	**>**	greater than
63	3F	00111111	**?**	question mark
64	40	01000000	**@**	at sign
65	41	01000001	**A**	
66	42	01000010	**B**	
67	43	01000011	**C**	
68	44	01000100	**D**	
69	45	01000101	**E**	
70	46	01000110	**F**	
71	47	01000111	**G**	
72	48	01001000	**H**	
73	49	01001001	**I**	
74	4A	01001010	**J**	
75	4B	01001011	**K**	
76	4C	01001100	**L**	
77	4D	01001101	**M**	
78	4E	01001110	**N**	
79	4F	01001111	**O**	
80	50	01010000	**P**	
81	51	01010001	**Q**	
82	52	01010010	**R**	
83	53	01010011	**S**	
84	54	01010100	**T**	
85	55	01010101	**U**	
86	56	01010110	**V**	
87	57	01010111	**W**	
88	58	01011000	**X**	
89	59	01011001	**Y**	
90	5A	01011010	**Z**	
91	5B	01011011	**[**	left square bracket
92	5C	01011100	****	backslash
93	5D	01011101	**]**	right square bracket
94	5E	01011110	**^**	caret / circumflex
95	5F	01011111	**_**	underscore
96	60	01100000	**`**	grave / accent
97	61	01100001	**a**	
98	62	01100010	**b**	
99	63	01100011	**c**	
10	64	01100100	**d**	

Dec	Hex	Binary	Character	Description
10	65	01100101	**e**	
10	66	01100110	**f**	
10	67	01100111	**g**	
10	68	01101000	**h**	
10	69	01101001	**i**	
10	6A	01101010	**j**	
10	6B	01101011	**k**	
10	6C	01101100	**l**	
10	6D	01101101	**m**	
11	6E	01101110	**n**	
11	6F	01101111	**o**	
11	70	01110000	**p**	
11	71	01110001	**q**	
11	72	01110010	**r**	
11	73	01110011	**s**	
11	74	01110100	**t**	
11	75	01110101	**u**	
11	76	01110110	**v**	
11	77	01110111	**w**	
12	78	01111000	**x**	
12	79	01111001	**y**	
12	7A	01111010	**z**	
12	7B	01111011	**{**	left curly bracket
12	7C	01111100	**|**	vertical bar
12	7D	01111101	**}**	right curly bracket
12	7E	01111110	**~**	tilde
12	7F	01111111	DEL	delete

Adapted from: http://www.rapidtables.com/prog/ascii_table.htm#space (accessed May 30, 2012.)

Appendix C: Command-Line Interface

C.1 Introduction

We implemented our application using Visual C++ and Microsoft Visual Studio (10) VS. We used this tool not only because it had unsurpassed support in its IDE that includes optimization for very fast performance, but also because we kept it simple enough so it can be compile in UNIX using gcc. gcc is a public domain C compiler for UNIX by GNU. In any case, this section is briefly about the salient points related to being a command line tool. We discuss all available options; what they do; and, how to use them. There are a few new ideas presented as well.

C.2 Application Paths

To run our application the only required path is to the LevelDB library file. Some of the other paths required to be set are a function of how a user wants to run the application. If they want to execute it from a different directory than it resides, then a path to the application's home must be set. Other than that, paths to the text file, to the database file, to the output file, and to the configuration files are application parameters.

C.3 Input Parameters

Our application is an exe called: ***RKLevelDB.exe***. It occupies a mere 58k of disk space, even less memory when running. To use the application a user must be at a command prompt with all paths needed already set. A user must then type in the application name and supply enough *command-line parameters* to satisfy the application's need. We reproduce these command-line parameters in Table 10 on the next page. There, we give a context for not only discussing them, but also for the application's capabilities they produce. All parameters have a *long* and a *short version* that we juxtaposed in the table. We primarily refer to the long version of parameters throughout our discussion as a convenience, but users are free to select either. In the next section, we highlight a few of the important parameters.

Long command	short	Description of command values and purpose
--brief	-b	Flag suppressing verification step in search (T = suppress).
--database-directory=	-d=	Path to index database-DO NOT add final separator in path.
--report-every=	-e=	Mb to report time statistics secs/(value supplied) Mb
--finish-after=	-f=	number of Mb in text file to index (for testing)
--generate-pattern	-g	Flag for selecting pattern from text file randomly.
--help	-h	Flag to show list of command-line arguments.
--window-length=	-l=	Length of window (number of characters) for creating fingerprints.
--modulus=	-m=	Prime number for fingerprint modulo arithmetic.
--database-name=	-n=	name of the index database
--overwrite-database	–o	Flag to replace index database with a new one.
--pattern=	-p=	Pattern to search for.
--radix=	-r=	Radix used as base of number system for characters & fingerprints.
--search	-s	Flag to signal a search is requested (T to include search.)
--path-to-text-file=	-t=	Path (including name) to the text file that is to be built or searched.
--verbose-file=	-v=	Path (including name) of text file to append performance data.
--write	-w	Flag to write to index database (true = write/append if exists)

NOTE-1) Always use '/' as the path separator for both DOS & UNIX; and, always place a '/' at the end of the path (e.g., “C:/dir1/dir2/'').

NOTE-2) All options (except -d & -n) can be set in a configuration text file as follows: the program will search for and use a file with the name: --database-directory+--database-name+.config if it exists. Use only short option names in the file, only ever put one parameter on one line, always include the “=” sign when listing parameters requiring them and, always ensure the “-” sign is in col 0 on a line

NOTE-3) Any parameter given on command-line takes precedence over these options

Table 10: List of command line parameters.

C.3 Technical Parameters

Some parameters are optional, others are mandatory, and some are mandatory depending on other parameter values. First, notice how options requiring a user-defined value have an equal sign on their right hand end; the equal sign is part of the parameter usage. For example the proper use of the 'finish-after' parameter given on a command line is shown between the quotation marks as follows: "--finish-after=1500". In addition, whenever a directory has a space in its name, a user must enclose it within quotation marks. In fact, we recommend as standard practice to use quotation marks when supplying all string parameter values. We also recommend using the UNIX style for path separators, a forward slash (i.e., "/") for directories. This separator works for directories in both Windows and UNIX; making it easy to switch from one operating system to another. Using this separator will also help avoid some confusing situations with escape sequences. Unfortunately, the Windows style backslash separator (i.e., "\") is the same as the symbol used for an escape sequence (e.g., "\0" is the escape sequence for a NULL character). With these preliminaries out of the way, we now turn to describing some details for parameters.

As can be seen in Table 10, we use the same application for both creating/modifying the Index and searching for patterns using the (**--write**) and (**--search**) parameters respectively. A user must supply one of these, but may choose to supply both to perform both steps in one execution. Each of these parameters needs a group of other parameters to supply some of their needed details. For instance, we already discussed above about two of the most important parameters needed for both types of analysis; these are the Index Database directory (**--database-directory=**) and its name (**--database-name=**). Referring to **Error! Reference source not found.**, we can see that (**--database-directory=)** option supplies a path to the target LevelDB database, and by our convention path names must end in a separator. Then, the (**--database-name=)** option supplies the name for the Index Database that is a subdirectory in (**--database-directory=)**. Both of these parameters set a context for the pending action of either reading or writing to and from an Index Database. One part of the application goes out and searches for the --

database-name= subdirectory, using it if available; creating it if not. We will have more say about these particular parameters later. In addition, both require a user to supply the name and location of a text file they want to index or search. They supply this file (and path) through the (--**path-to-text-file=**) parameter.

When the *search* parameter is specified, a user must also supply a search pattern using the (**--pattern=**) parameter. In addition, searching has an optional parameter called (**--brief**) that disables the character-by-character verification that we initiate when a pattern's fingerprint equals a text's fingerprint. Be wary with this command because it essentially says a search's results include false positives. We provide this switch primarily for research related studies.

When a user supplies the write parameter, several optional parameters are available that mostly control the reporting of performance information. Before getting to their description, however, there are several other parameters used primarily for testing that we include for thoroughness. To begin, the (**--overwrite-database**) parameter tells the application to build a new Index Database. Using the database directory and name specified in earlier parameters, the application checks to see if an Index Database exists, and deletes it if it does before building a new one. Another parameter helps with testing the index creation step. A (--**finish-after=**) parameter is supplied to ask the application to stop processing the text file after so many bytes. If a user sets this option, the application will finish building an Index Database when it reaches the number of bytes in the text file supplied as the parameter's value.

Aside from this parameter, the write process has several interrelated parameters controlling the creation and filling of a file containing performance information. A brief description of these parameters follows. To begin with, the (**--verbose-file=**) parameter not only triggers the reporting of performance information, but also names the file where this information will be written. Performance information is two columns, first, the number of bytes processed since previous write, and second, the time taken in milliseconds since the previous performance report. Omitting this parameter turns off

performance reporting altogether. When the parameter is included, the application checks to see if the file named in the parameter exists. If it does not exist, the application creates it. If, on the other hand, the file does exist the application opens it, leaves any existing data in tacked and appends new information at the end. The performance information reported above is controlled by the (--**report-every=**) parameter. This parameter allows a user to specify how many bytes to process between reporting incidents. That is, our application has a clock and a counter of bytes processes in the text file. This option takes advantage of that information by reporting the time it has taken to process that many bytes of the text file.

C.4 Configuration File

All options (except **--database-directory=** and **--database-name=**) can be set in a configuration text file. We need both "**--database-directory=**" and "**--database-name=**" beforehand through command-line so the application can know where to search for a configuration file. Referring to the above table, we can see that "**--database-directory=**" option supplies a path to the target LevelDB database, and by our convention path names must end in a separator. Then, the "**--database-name=**" option supplies the name for the Index Database that is a subdirectory in **--database-directory=**. Both of these parameters set a context for the pending action of either reading or writing to and from an Index Database. One part of the application goes out and searches for the **--database-name=** subdirectory, using it if available; creating it if not. We will have more say about these particular parameters later.

At the same time, another part of the program automatically searches for a file with a name that is similar to the Index File. Using the long command-line parameter names given above we can see the configuration file name is simply the Index Database directory and name with a ".config" tacked onto its right hand end. In other words, a configuration file has the following name and location, after substituting parameter values for their name below:

--database-directory + --database-name+.config

If the file does not exist, the application continues as normally. If the file does exist however, it will use the parameters specified within it; provided those values have not already been supplied on the command-line. In essence, command-line supplied parameters take precedent over commands in a configuration file.

C.5 Creating a Configuration File

A configuration file is a place to put commonly used parameters that you want associated with a certain database. As demonstrated above, the file lives in the same directory as the database itself so no confusion should arise. Configuration files are optional meant to help researchers configure experiments consistently. The following list gives some of the rules and formats for a configuration file that has a name equal to: "database-directory+database-name+.config"

- Create file with one option per line.
- The order of option names in the file is totally arbitrary regardless of connections
- Use only short option names (i.e., the single letter ones.)
- Do not indent lines, ensuring the "-" is the first char on a line.
- For example, to set the radix to 128 use "–r=128" on a separate line.
- As shown in previous line, the "=" is part of option name and must be included.
- Also, as shown there can be no blanks between any two characters
- It is good practice to wrap all strings including database directory with quotes.
- It is important for us to reiterate the fact that options supplied on the command-line take precedence over options given in a configuration file. In other words, with the example given above for setting radix in a configuration file. A radix of 128 will only be used if neither "-r=x" nor "--radix=x" were included on the command-line.

Appendix D: Seek Times for each Radix/Modulus Combination.

The graphs in this Appendix show how seek times change for each pattern length for our application and for GREP. There is one graph for each radix. Then, within a graph, there is a line for GREP and a line for each window length and Modulus. The vertical axis is the log of seek time in milliseconds; with log required because of the wide spread of time. Later we show graphs of performance in term of hits per millisecond where higher values are better. Unlike that measure of performance, the one used on these graphs is actual time taken to process; meaning that higher is worse. Therefore, we are looking for lines that are lower than the line for GREP. In addition, we mentioned earlier how none of the analyses involving a window length of four was faster than GREP. The line in the graph below runs almost parallel to the GREP line. Finally, the graphs also show how the times for longer patterns (> 10) are almost always better than GREP.

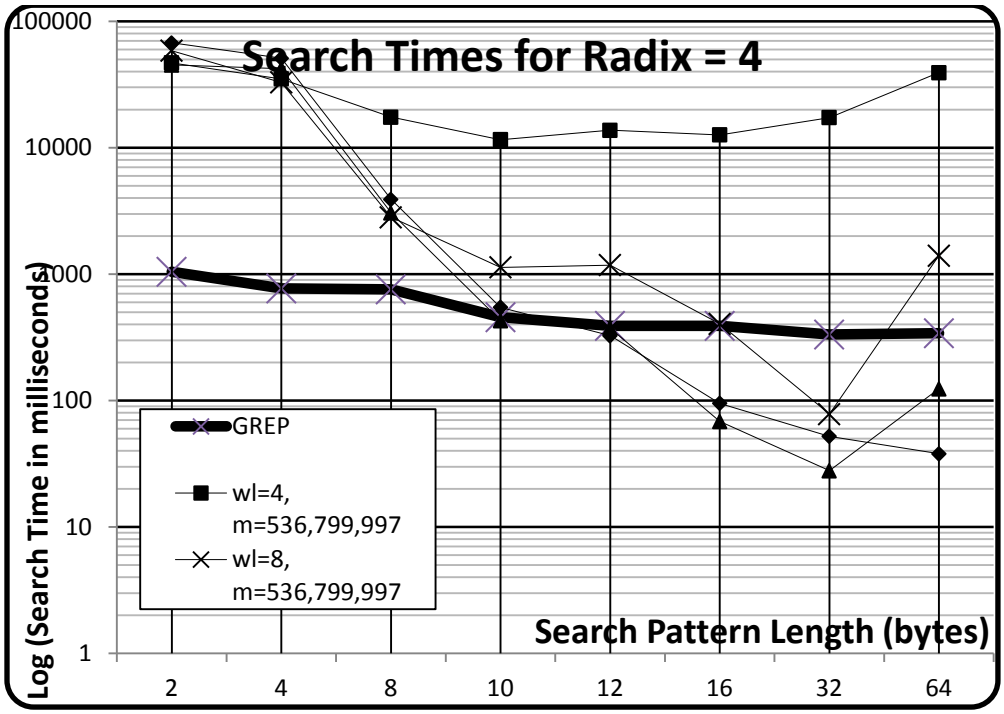

Figure 14: Search Times for Radix = 4

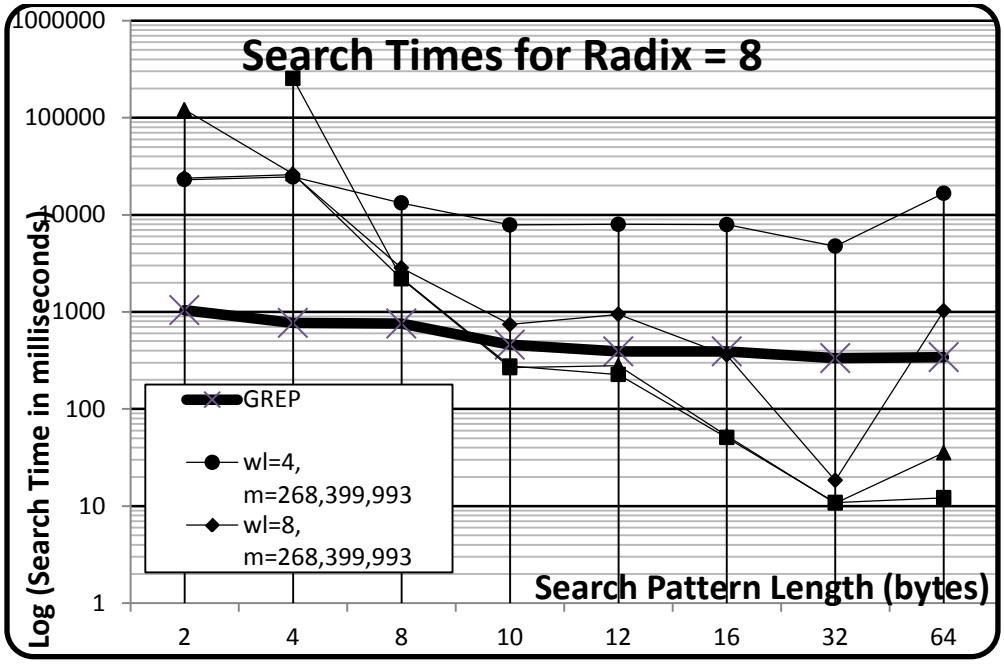

Figure 15: Search Times for Radix = 8

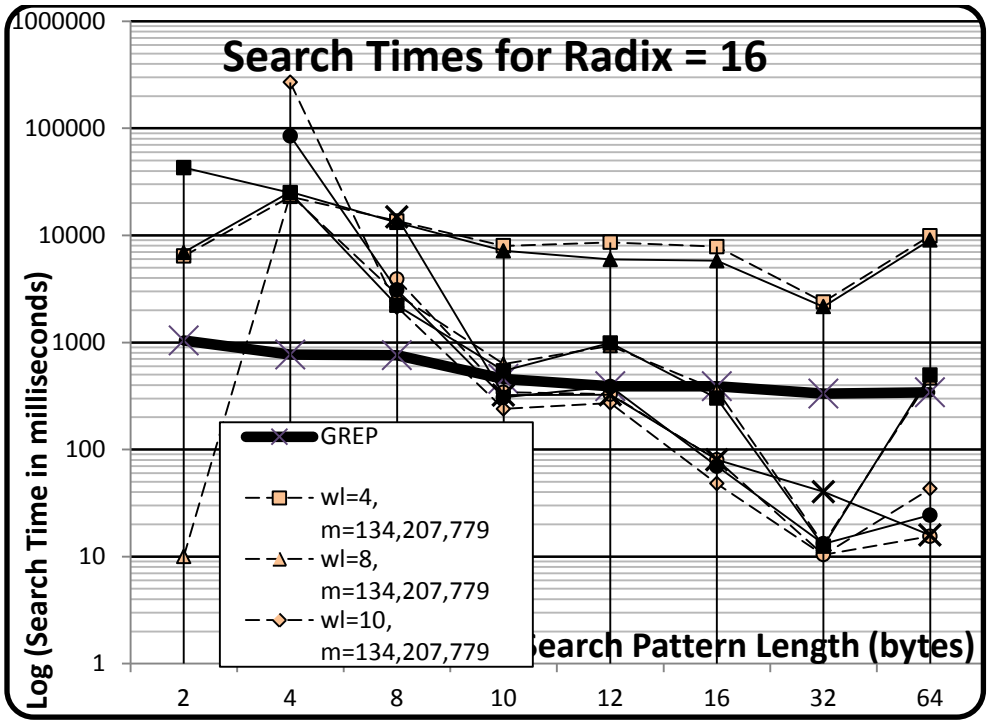

Figure 16: Search Times for Radix = 16

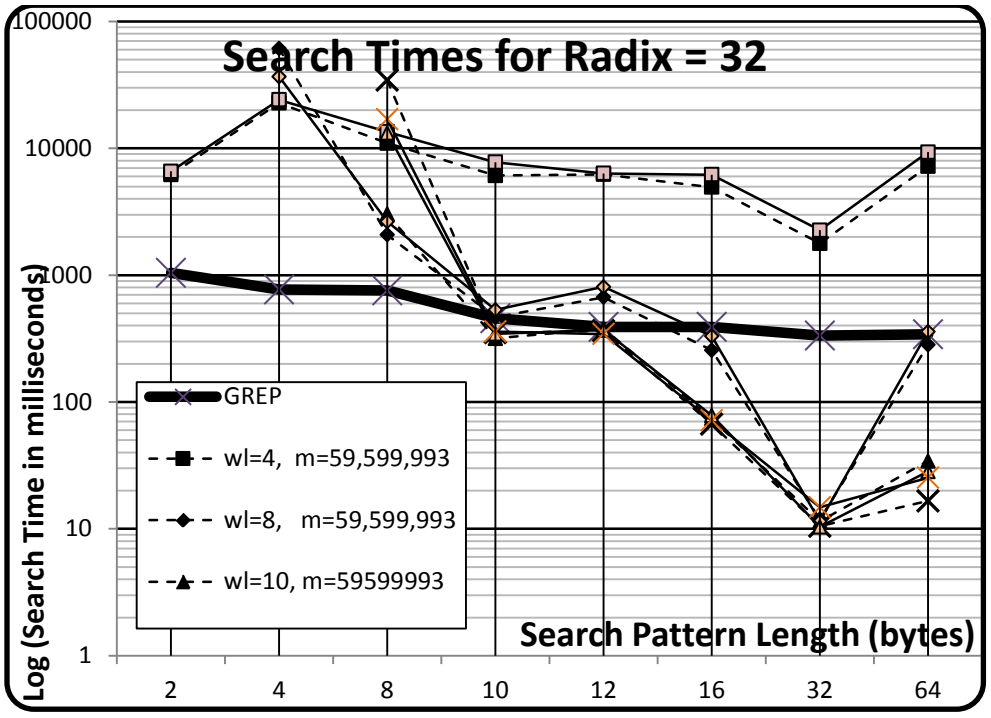

Figure 17: Search Times for Radix = 32

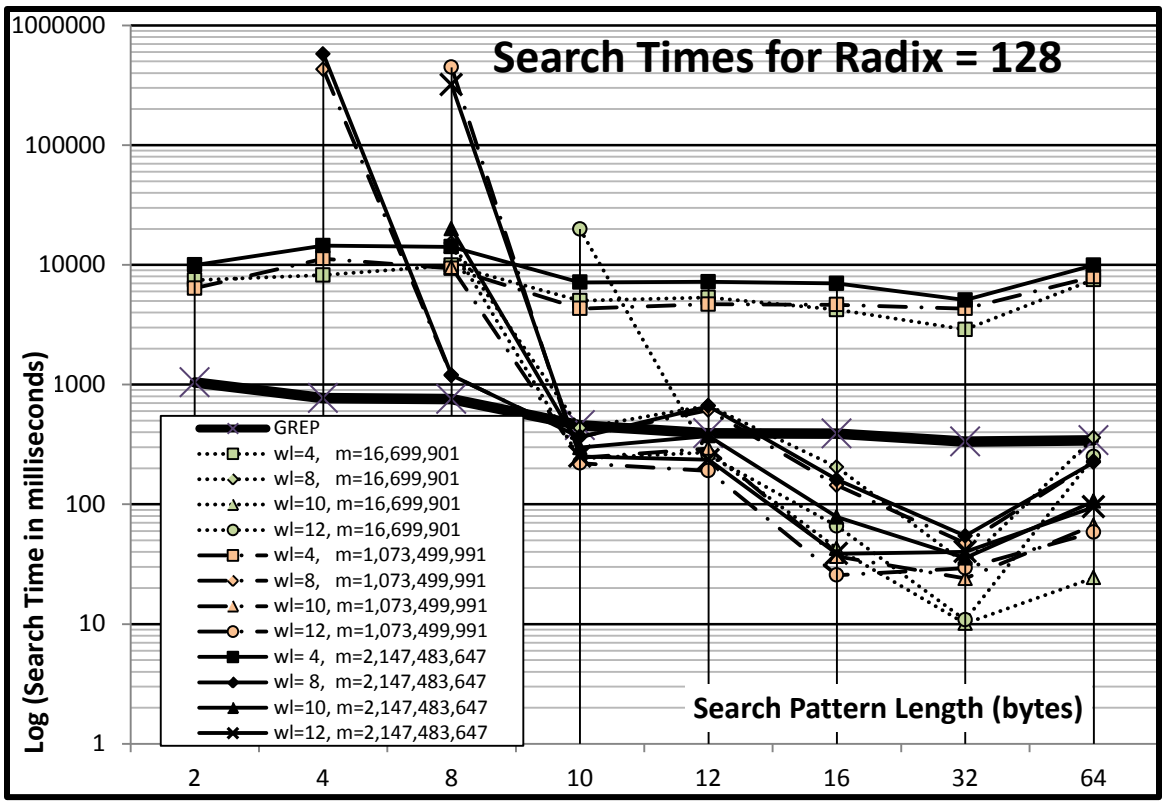

Figure 18: Search Times for Radix = 128

Appendix E: Diagrams of Actual Versus Spurious Hits

The bar graphs shown in this Appendix juxtapose actual hits (white bars labeled "Pos") with false positive hits (dark bars labeled "Neg") by pattern length and window length. There is one graph for each radix/modulus combination. In the main body of the report, we mentioned how pattern lengths of 32 and 64 never have a false positive for any radix. In addition, we also mentioned that the last three categories (where the search pattern did not exist in the file) should not have any false positives. While the graph below demonstrates this condition, later graphs show how false positives appear seemingly at random locations for these categories. Finally, a quick glance at all these graphs illustrates how the false positives

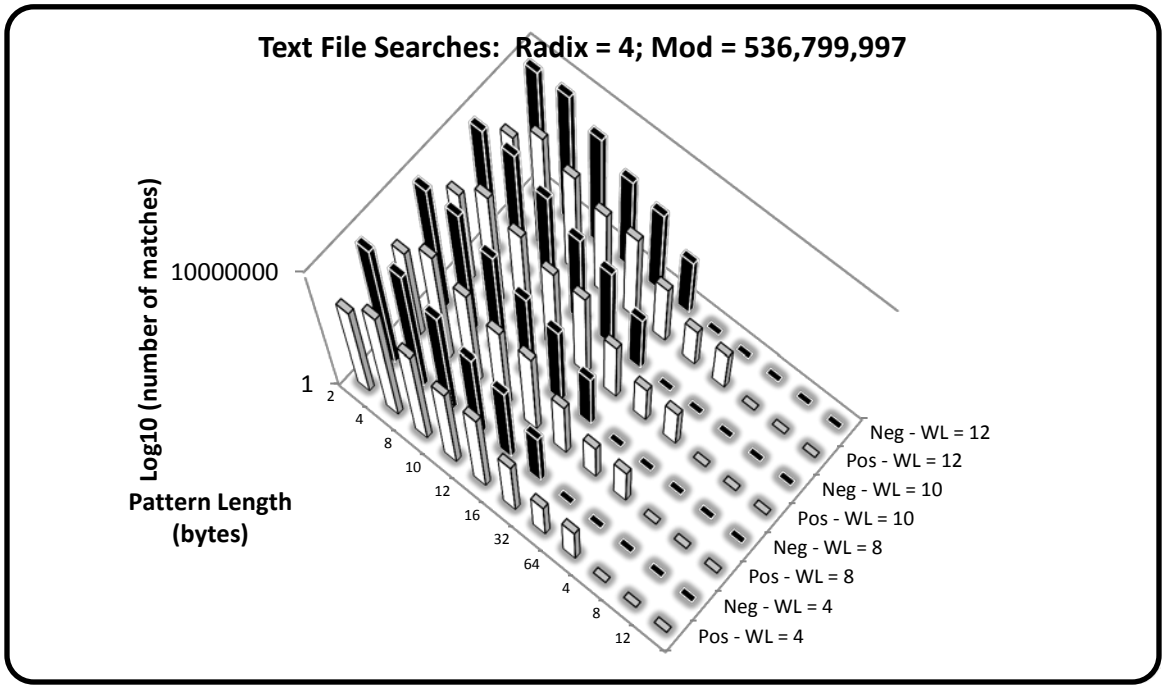

Figure 19: Actual and spurious hits for radix = 4

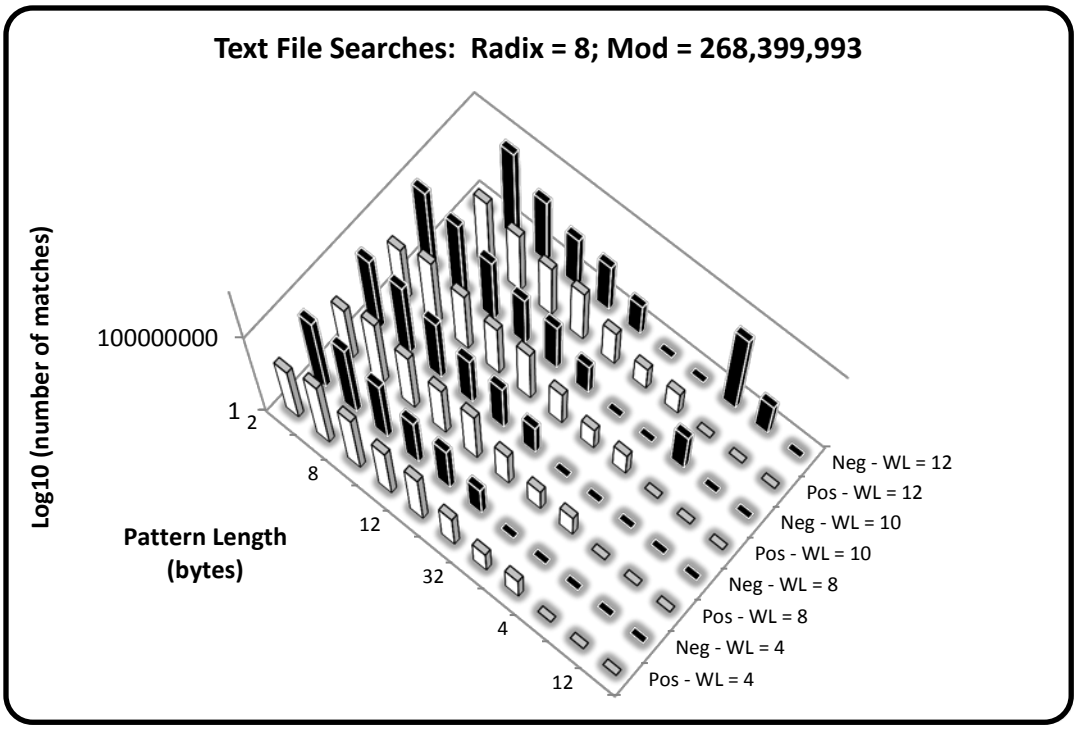

Figure 20: Actual and spurious hits for radix = 8

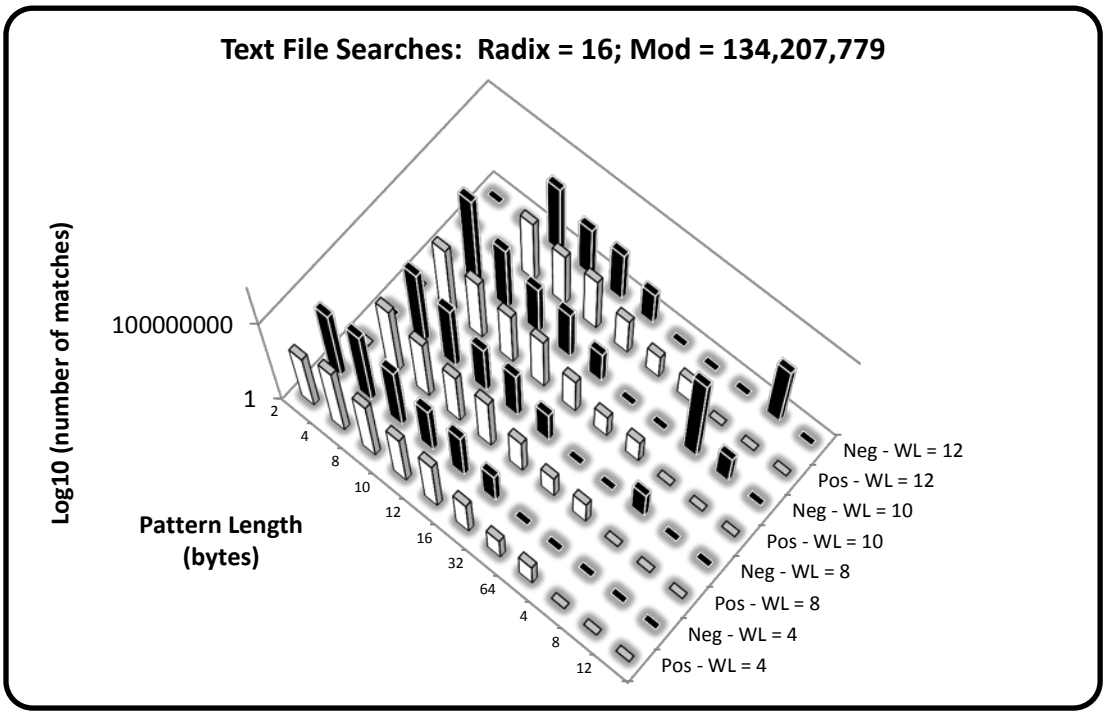

Figure 21: Actual and spurious hits for radix = 16, modulus = 134,207,779

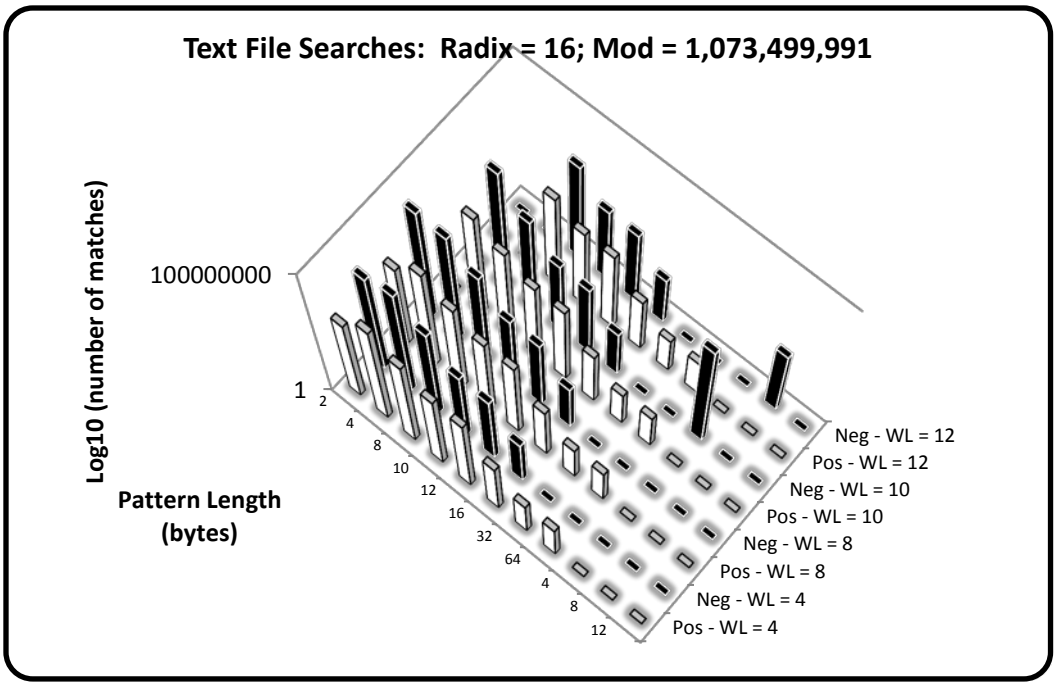

Figure 22: Actual and spurious hits for radix = 16, modulus = 1,073,499,991

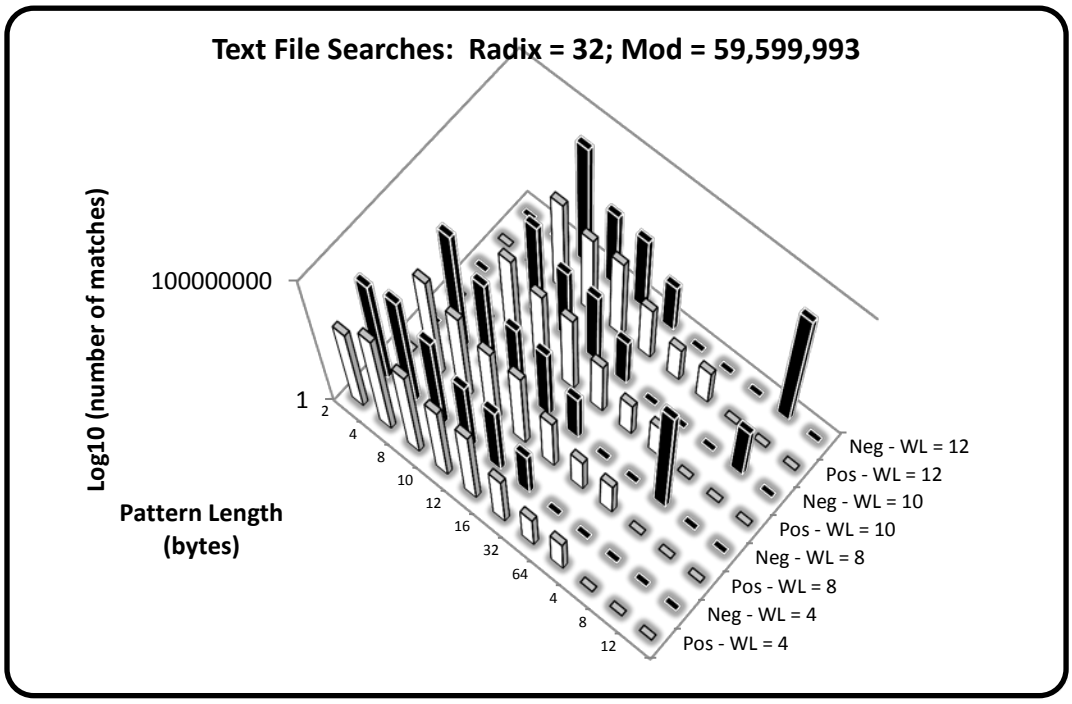

Figure 23: Actual and spurious hits for radix = 32, modulus = 59,599,993

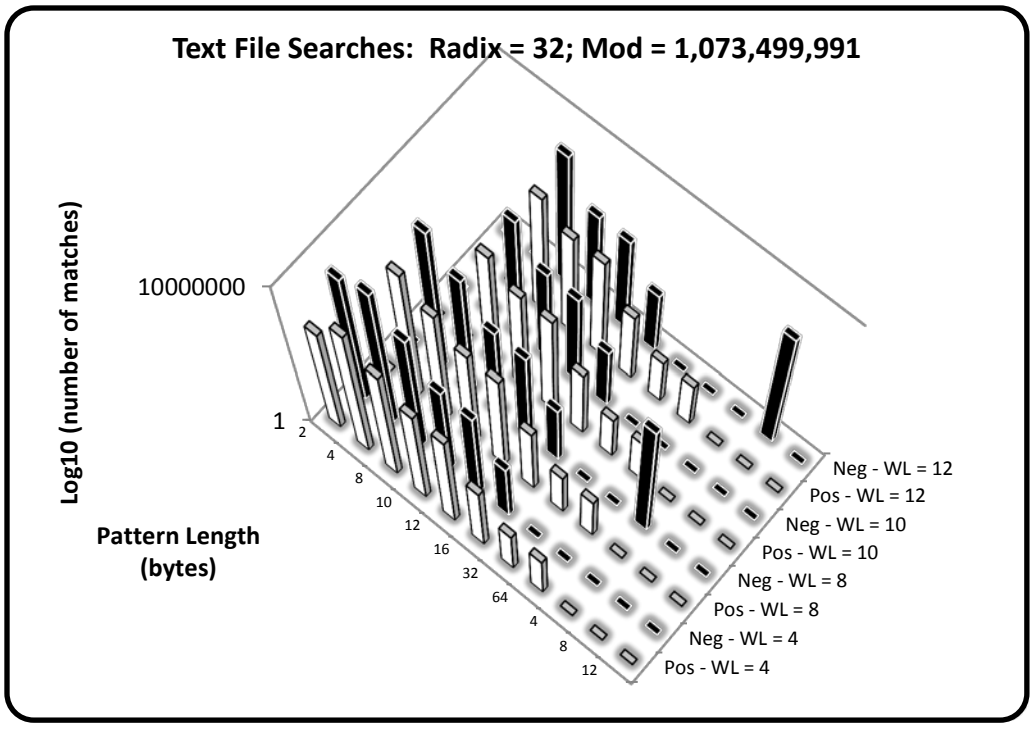

Figure 24: Actual and spurious hits for radix =32, modulus = 1,073,499,991

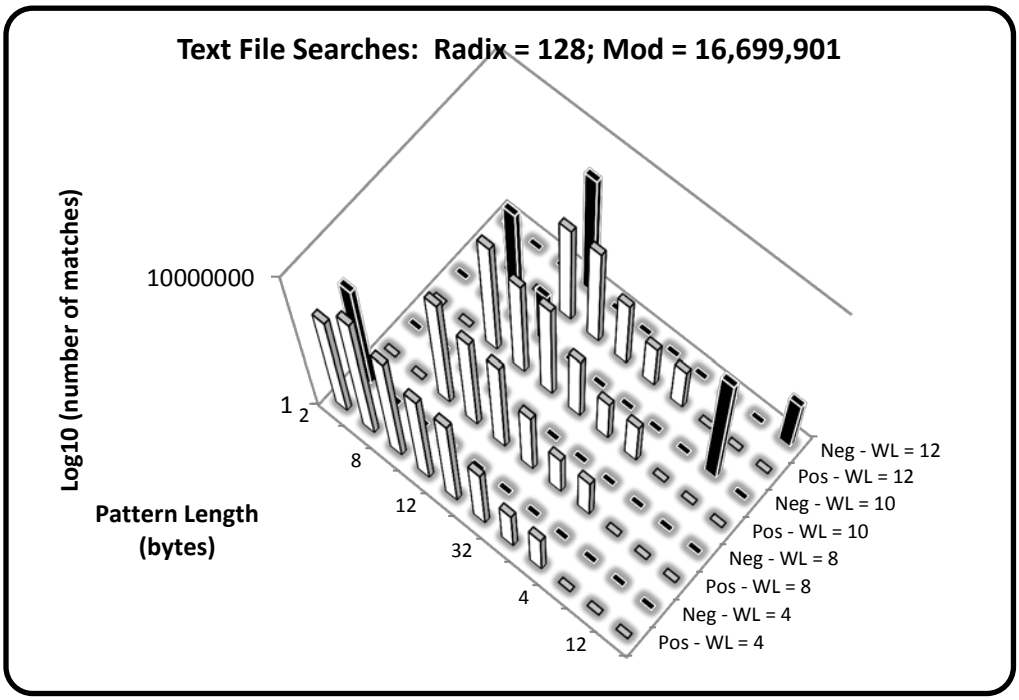

Figure 25: Actual and spurious hits for radix = 128, modulus = 16,699,901

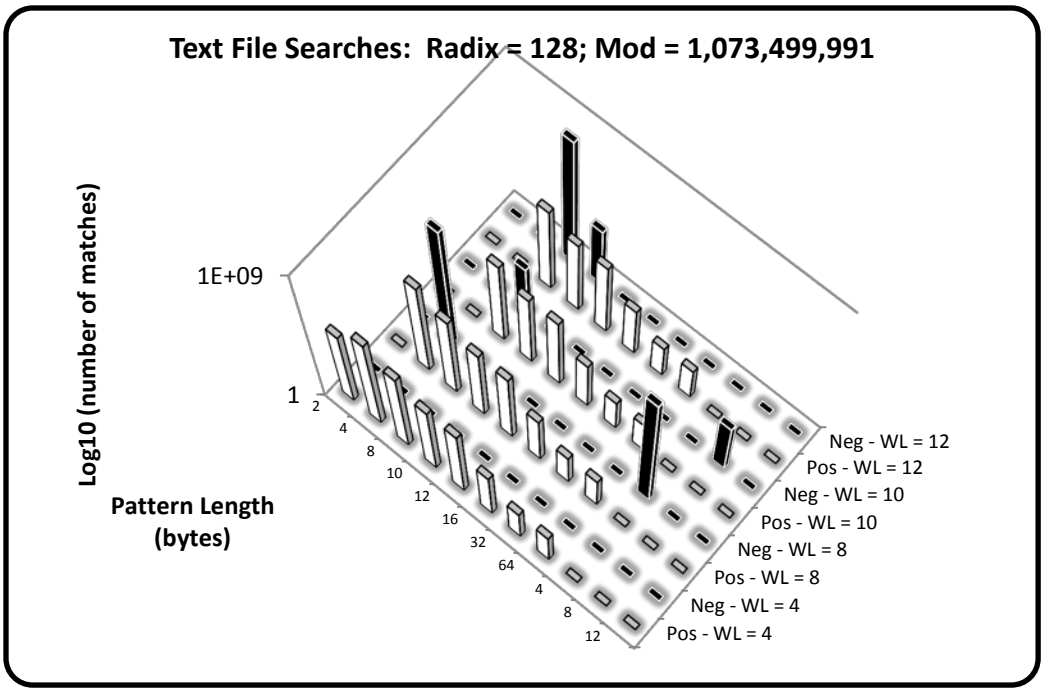

Figure 26: Actual and spurious hits for radix = 128, modulus = 1,073,499,991

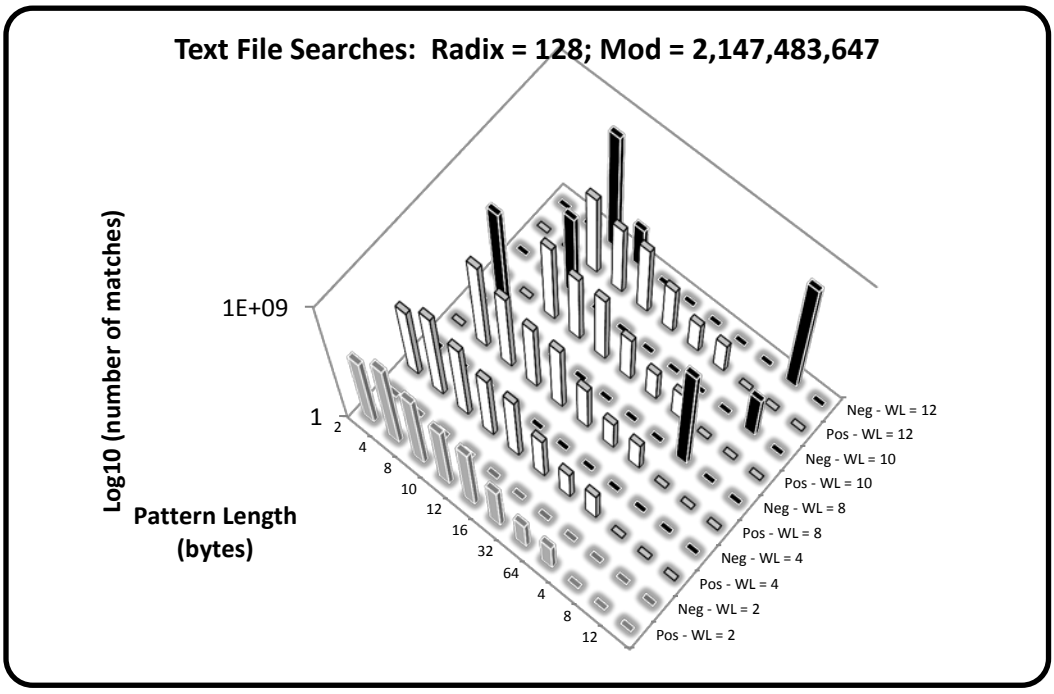

Figure 27: Actual and spurious hits for radix = 128, modulus = 2,147,483,647.

Appendix F: Line Graphs of Performance (hits/ms)

This Appendix contains a set of line graphs showing the rates at which "hits" are processed. A hit is an actual occurrence of a string, or a false positive. The performance measure shown on the graphs is "hits per millisecond," meaning higher numbers show better performance than lower numbers. The first graph shows the rates for GREP; while the remaining graphs show the rates for each radix. These later graphs also superimposed the GREP rates for easy comparisons. The vertical axis shows the log of the rates. Using the log of the rates is due to GREP's huge range from 0.29 through 1486.00 milliseconds. In addition, GREP's incredibly fast processing rate for a pattern length of four its average processing rate really high compared to our application.

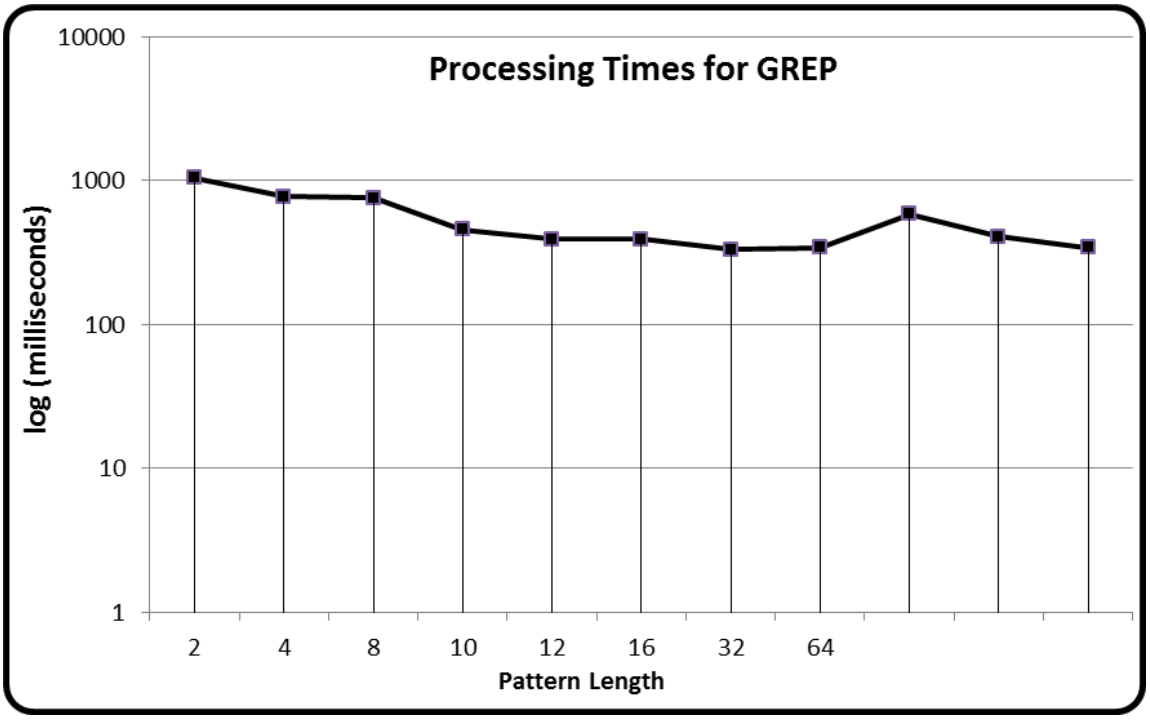

Figure 28: Processing rates (ms) for GREP

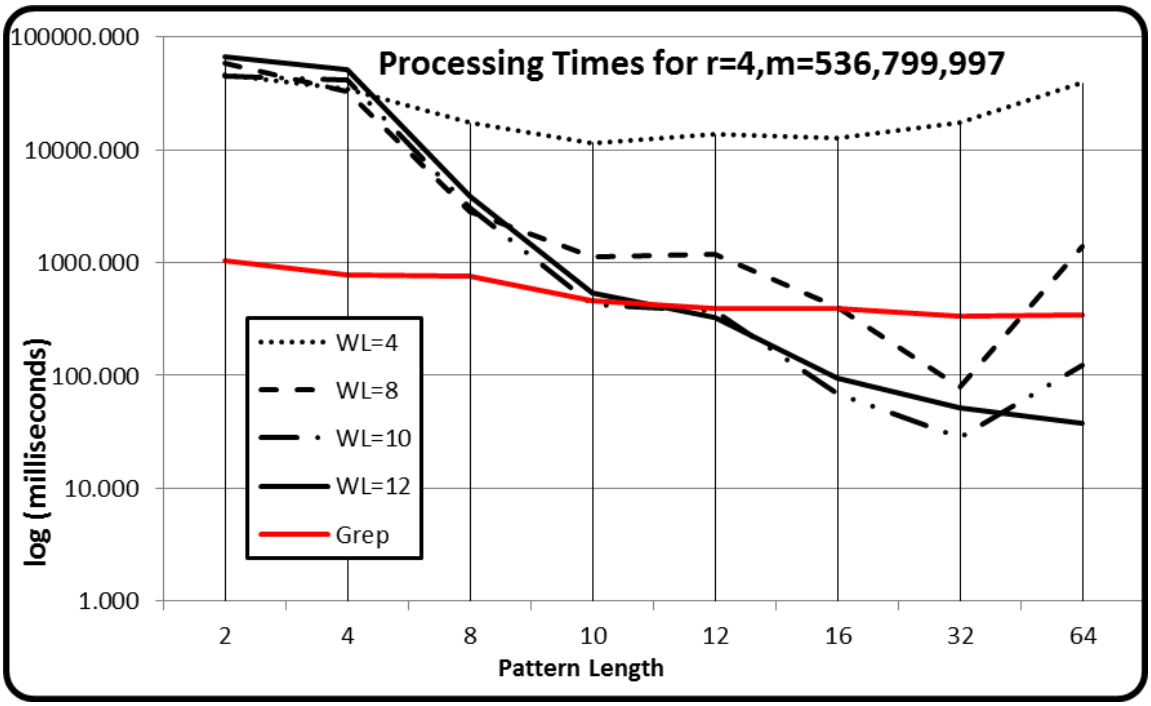

Figure 29: Processing rates (ms) for r=4,m=536,799,997.

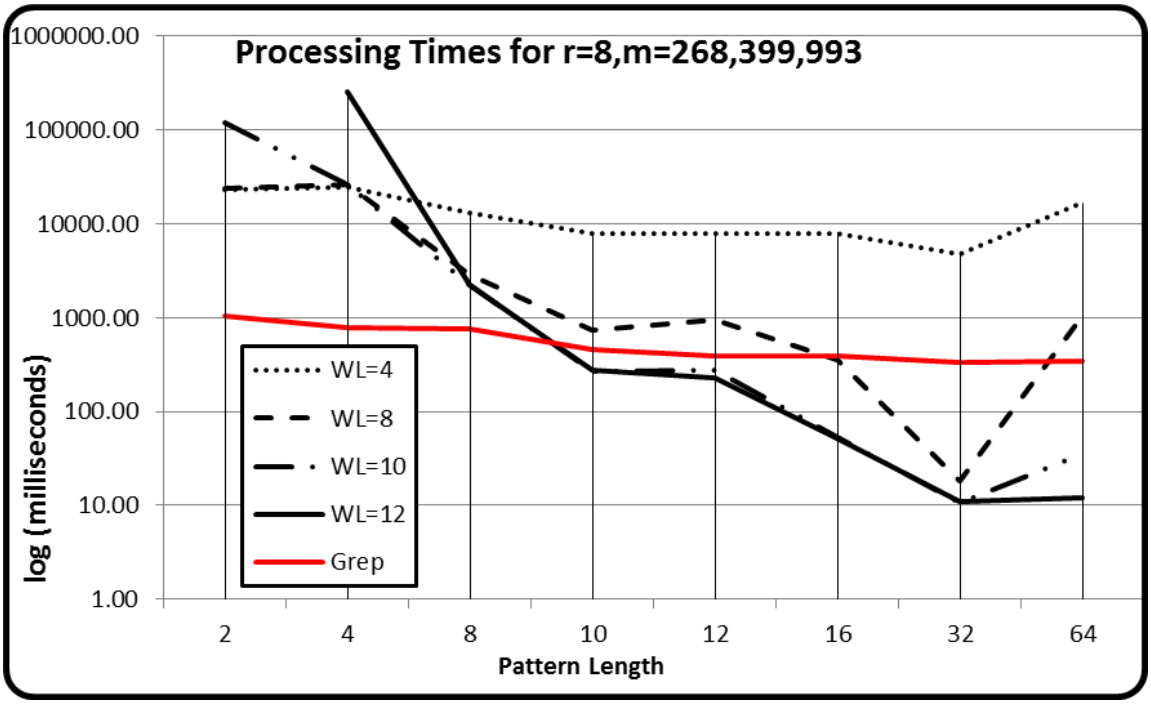

Figure 30: Processing rates (ms) for r=8,m=268,399,993.

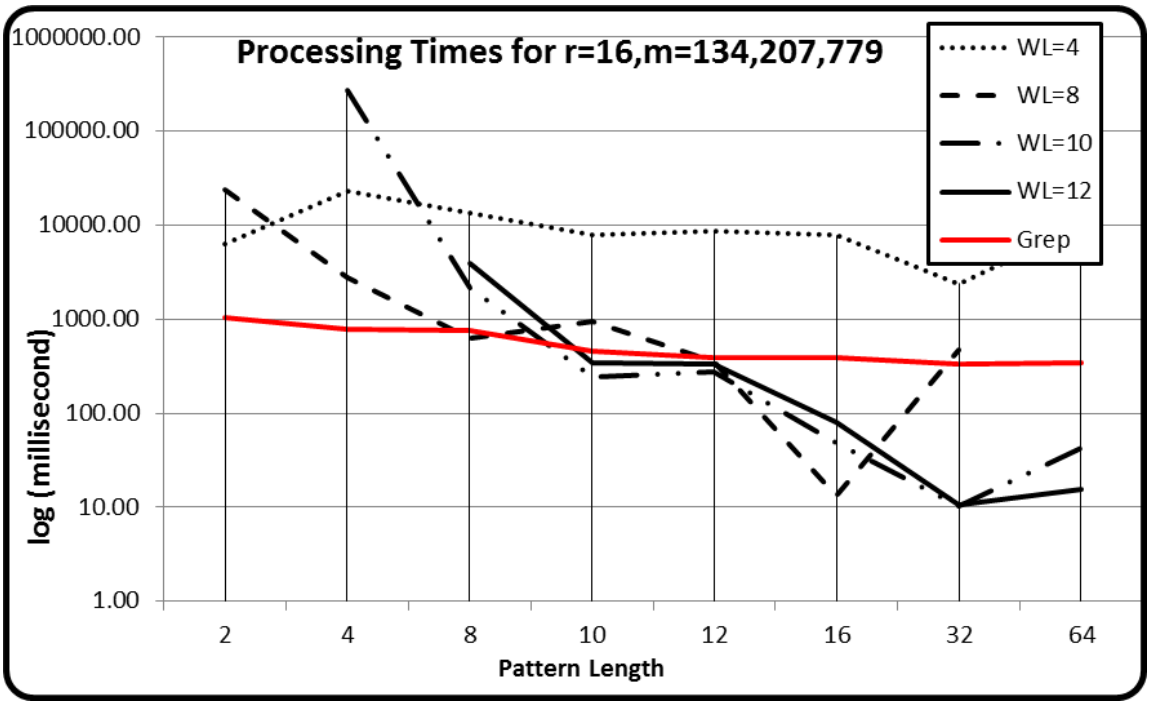

Figure 31: Processing rates for r=16,m=134,207,779

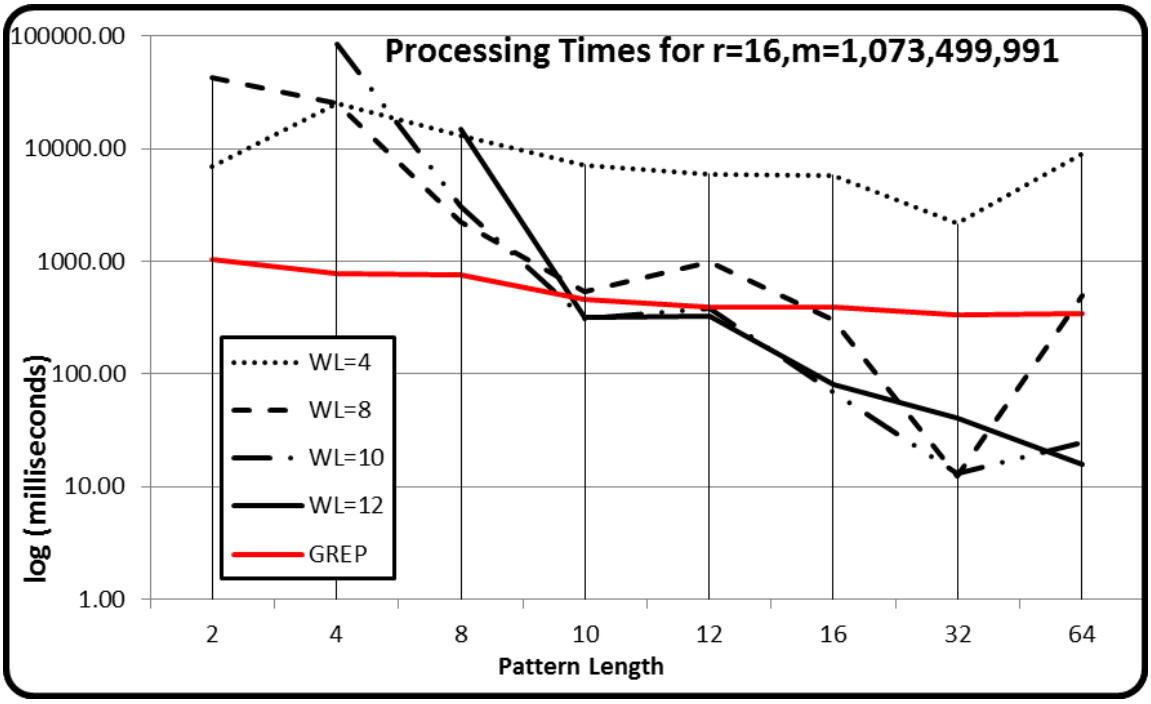

Figure 32: Processing rates for r=16,m=1,073,499,991

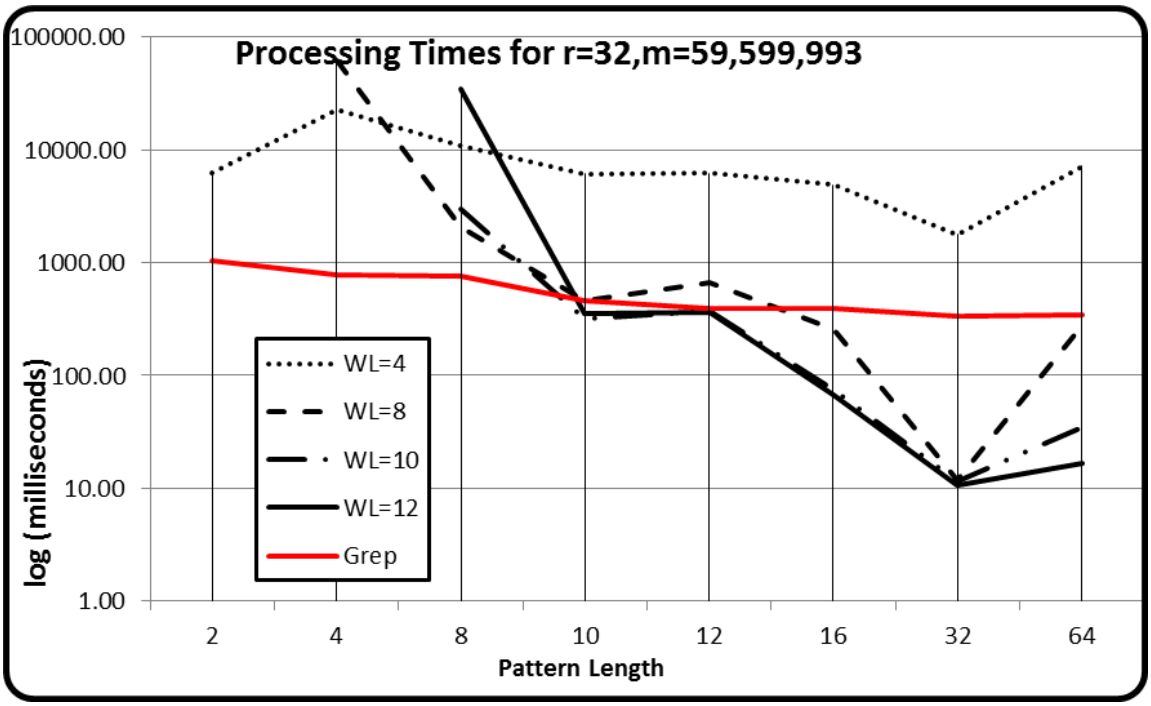

Figure 33: Processing rates for r=32,m=59,599,993

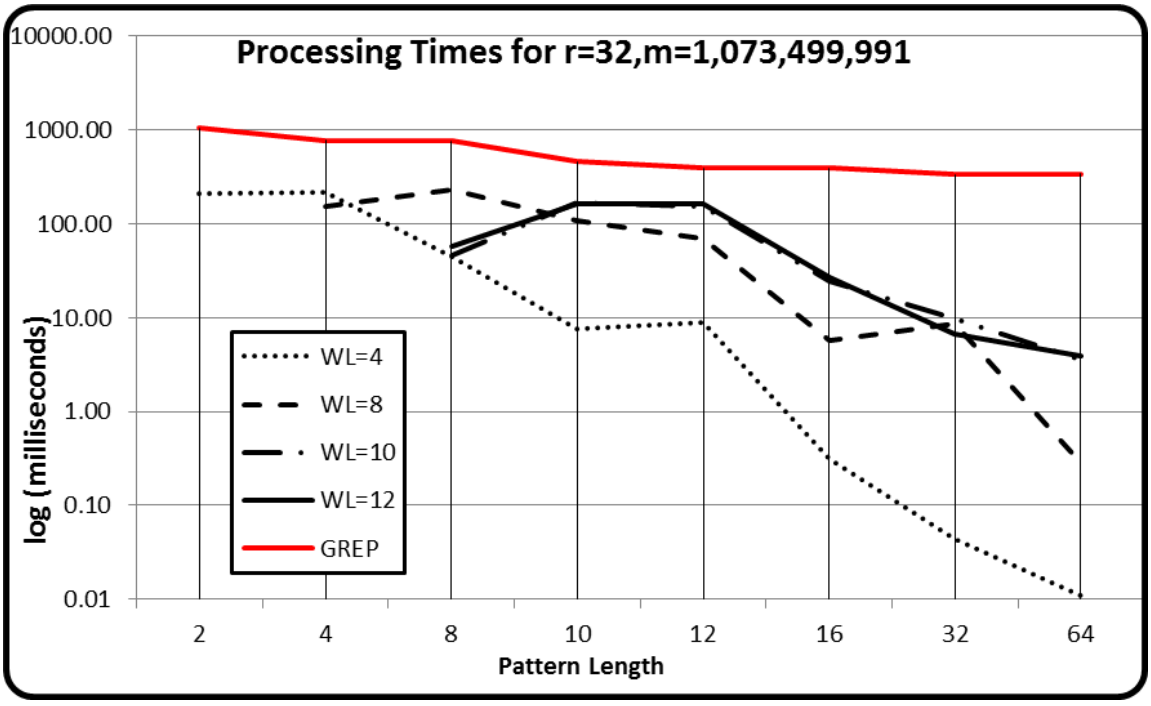

Figure 34: Processing rates for r=32,m=1,073,499,991

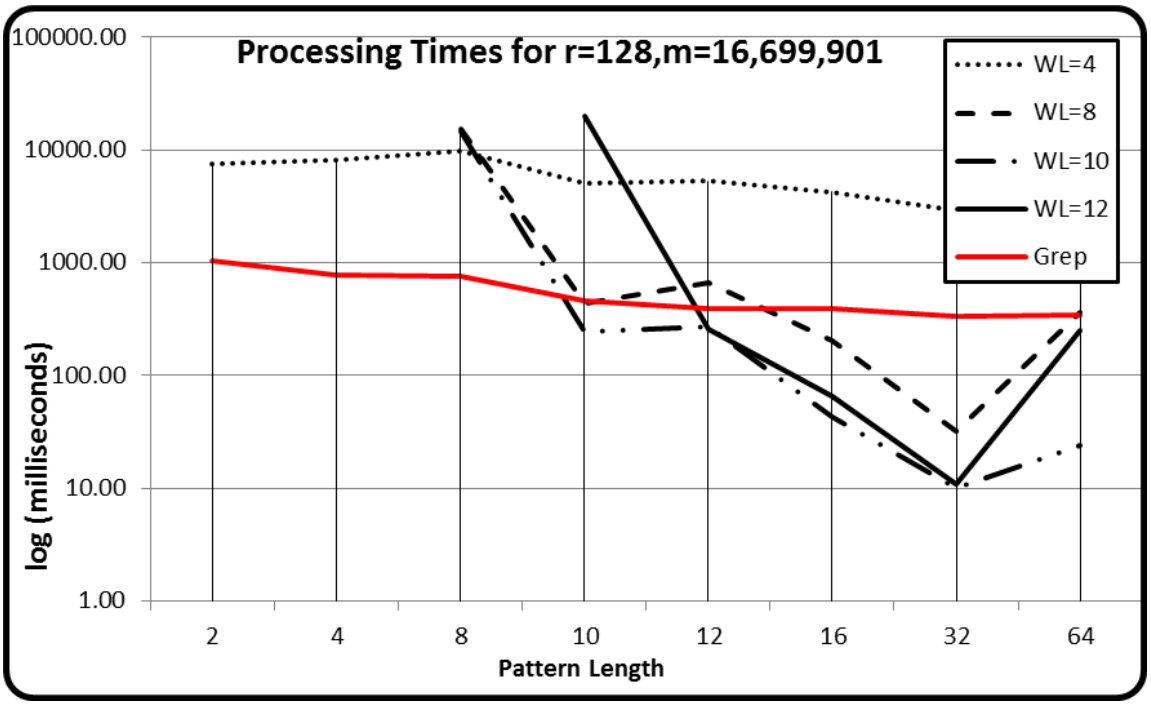

Figure 35: Processing rates for r=128,m=16,699,901

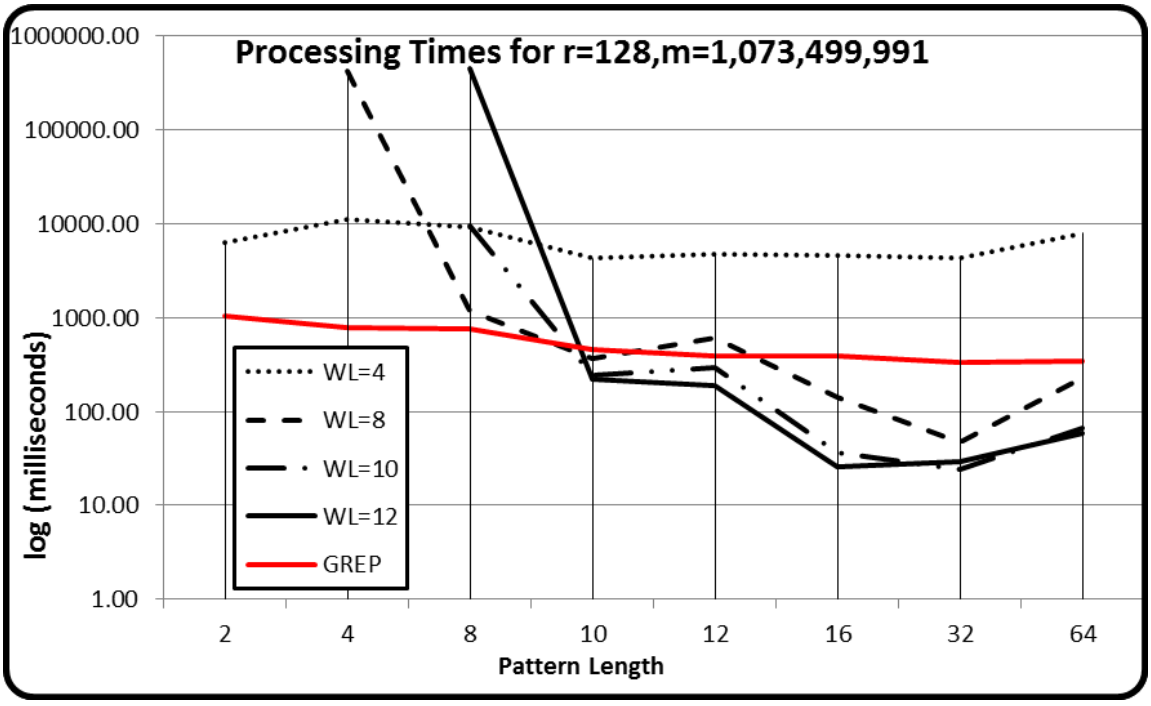

Figure 36: Processing rates for r=128,m=1,073,499,991

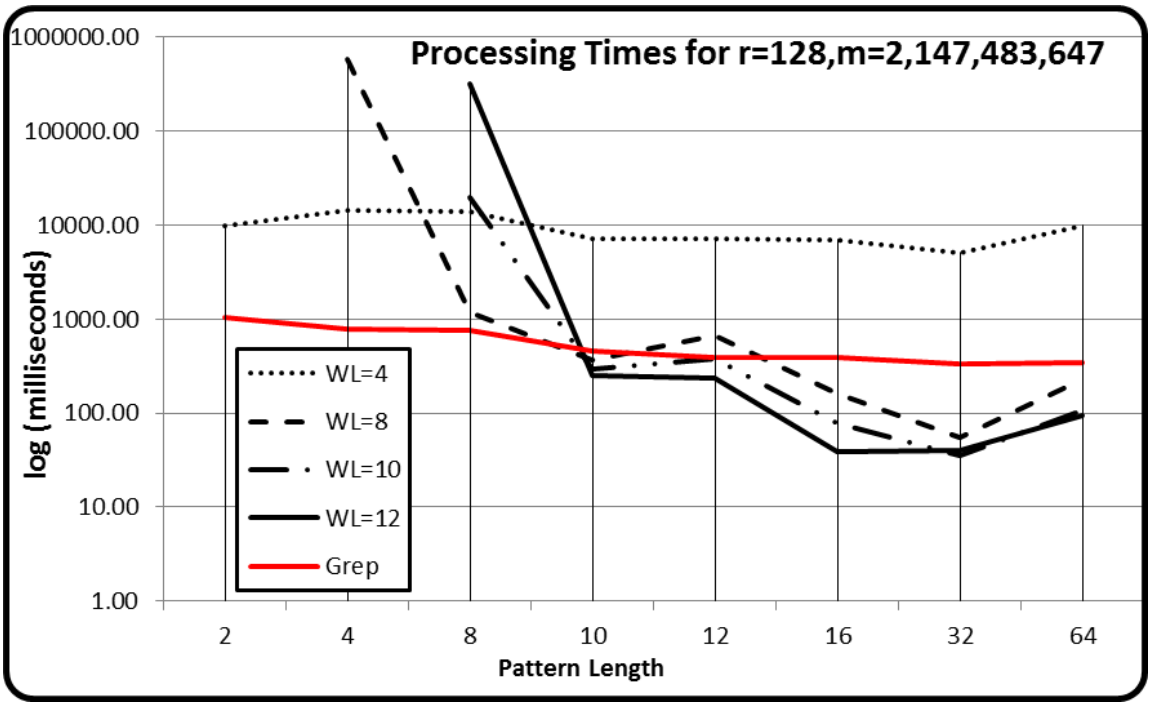

Figure 37: Processing rates for r=128,m=2,147,483,647