USING HADOOP, MAPREDUCE, AND CLOUD SERVICES TO FIND FACTORS OF RSA PRIMES

by

CHRISTOPHER C. BOWDEN

A RESEARCH PAPER

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in Computer Science
in the Graduate School of
Troy University

DECEMBER 2009

USING HADOOP, MAPREDUCE, AND CLOUD SERVICES TO FIND

FACTORS OF RSA PRIMES

Submitted by Christopher C. Bowden in partial fulfillment
of the requirements
for the degree of Master of Science
in Computer Science
in the Graduate School of
Troy University

Accepted on behalf of the Faculty of the Graduate School by
the thesis committee:

_____ _____
Matthew Mariano, Ph.D. Date
Chair

_____
Sunil Das, Ph.D.

_____
Emrah Orhun, Ph.D

_____
Irem Ozkaharan, Ph.D.

_____ _____
Frederick M. Beatty, Ph.D. Date
Associate Dean, College of Arts & Sciences

_____ _____
Hal Fulmer, Ph.D. Date
Dean, College of Arts and Sciences

DECEMBER 2009

ABSTRACT

USING HADOOP, MAPREDUCE, AND CLOUD SERVICES TO FIND FACTORS OF RSA PRIMES

Christopher C. Bowden

Database systems have evolved in many ways since their initial conception as Management Information Systems in the 1960's. Today's web-enabled applications have outgrown the notions of strictly centralized data stores and users have come to expect near real-time response to application requests - in addition to high availability. To meet these new technical challenges and adapt to potentially explosive user base growth, distributed database systems have been pushed to the front lines of application architectures that must plan for extreme scalability and graceful fault-tolerance. This paper will focus on discussing the shifting requirements that have led to distributed database systems as we know them today. In addition, the paper will discuss an application of Hadoop and MapReduce through Amazon's cloud computing services. The cloud system will be used to find prime factors of large numbers in a manner that reflects the RSA encryption scheme's private and public key calculations.

Human or Animal Subjects Review

for

Christopher C. Bowden

USING HADOOP, MAPREDUCE, AND AMAZON CLOUD SERVICES TO FIND FACTORS OF RSA PRIMES

This research project has been reviewed by the Research Review Board and approved as follows (the appropriate block must be checked by either the Thesis chair or the Chair of the Research Review Board):

X **Neither humans nor animals** will be used and this research is certified exempt from Research Review Board review by the thesis committee chair.

☐ **Human participants** will be used and this research is certified exempt from Research Review Board review by the thesis committee chair.

☐ **Human participants** will be used and this research was reviewed and is approved by the Research Review Board.

☐ **Animal participants** will be used and this research was reviewed and is approved by the Animal Research Review Board.

_____ _____

Signature of Thesis Committee Chair Date Approved or Rejected

_____ _____

Signature of Chair of Research Review Board Date Approved or Rejected

DEDICATION

To my wonderful wife, whose patience with my musings and frustrations is forever appreciated.

ACKNOWLEDGMENTS

Many thanks are given to Dr. Mariano and the entire Computer Science faculty of Troy University's Montgomery Campus. Without their support, encouragement, teaching, and wisdom this paper would not be possible.

Table of Contents

List of Tables

List of Figures

Key to Symbols or Abbreviations

AMD - Advanced Micro Devices, Inc.

AWS - Amazon Web Services

CPU - Central Processing Unit

DB - Database

DDBMS - Distributed Database Management System

EC2 - Elastic Compute Cloud

FLOPS - Floating-point Operations Per Second

GB - Gigabyte

GCD - Greatest Common Divisor

GHz - Gigahertz

HDFS - Hadoop File System

MIS - Management Information System

MIT - Massachusetts Institute of Technology

MS DTC - Microsoft Distributed Transaction Controller

RDBMS - Relational Database Management System

RSA - Rivest-Shamir-Adleman encryption algorithm

RSA-100 - a particular 100-bit product of primes

S3 - Amazon's Simple Storage Service

SAGE - Semi Automatic Ground Environment

SQL - Structured Query Language

stdin - Standard Input

stdout - Standard Output

WAN - Wide Area Network

CHAPTER ONE – INTRODUCTION

A BRIEF HISTORY OF DATABASE TECHNOLOGY

Software development discussions in academia often focus largely on algorithms, data structures, electronic circuits, and project management. As development topics increase in scale, another subject must be considered for study – data storage and retrieval. Historically, computer scientists relied on basic file system functions and developer, (or, at best, organizational) conventions to create basic systems that would store information on magnetic tape without any relationships beyond the order of the data as it was stored. These file system paradigms quickly became an imposing task to organize as organizations grew to multiple sites, employed multiple systems with incompatible file systems, and functional areas became increasingly dependent on sharing knowledge and data in an understandable format. In addition to these obstacles for file systems, complex real-time systems such as the famous military anti-aircraft system, SAGE, could not possibly handle receiving, analyzing, and transporting results across file systems without an efficient method of data retrieval (Haigh, 2006).

Throughout the 1960s and into the 1970s the idea of "Management Information Systems" or MIS were being touted as the "wave of the future" that would propel businesses to new heights through the use of centralized data stores called databases. The idea largely hinged upon the introduction of the concept that would become relational database management systems, RDBMSs. Relational databases are essentially spreadsheet representations of data in row and column format. In addition to rows, called tuples, and columns, called attributes, primary keys are used to uniquely identify rows. Data in one table can be linked to other tables through the use of foreign keys (Stallings & Brown, 2008). Tabular representations of sets of data are particularly effective in conveying the structure of the data and provide decent retrieval performance of particular data for analysis. RDBMS technology has undergone fine tuning and feature additions to satisfy requirements that could have barely been conceived in 1970. Today's relational databases must not only store and retrieve tabular data but also provide high availability clustering failover to respond gracefully to hardware failure, deliver real-time data analysis tools to drive business success automatically, and offer comprehensive maintenance

solutions to ensure data is backed up and restored in the case of an outage. No longer are databases used merely as file system replacements – today they range from specialized stand alone products to distinct tiers in multi-tier architecture programs to entire enterprise systems in themselves. Figure 1.1 illustrates the use of an RDBMS as the database driving the data layer of a typical n-tiered application.

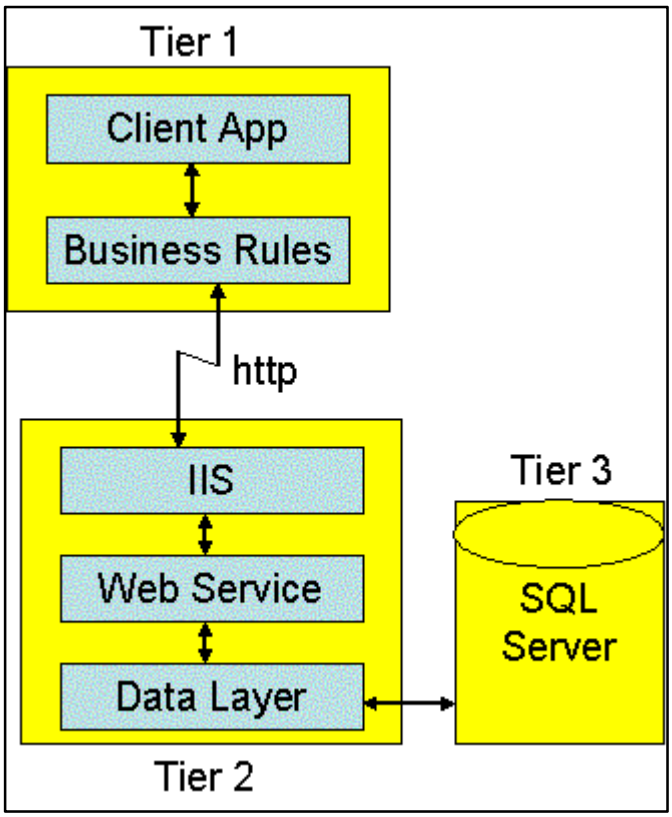

Figure 1.1. Typical n-tier application architecture with data layer driven by RDBMS. From (Sheriff, 2002).

SHIFTING DATABASE REQUIREMENTS

The inception and widespread embrace of the Internet has forever changed the landscape of Information Technology in general, and database systems particularly. Businesses transferred data storage and retrieval systems to corporate intranets and wide-area networks, WANs. Personal computer users began utilizing the Internet to share ideas, memories, and relationships with one another. Developers adapted to these changing requirements by relying on efficient data storage and retrieval. Today's database systems are called on to serve content ranging from text to interactive games to audiences of hundreds, thousands, or millions per hour. Database systems are increasingly called on not only to supply the data needed to direct programmatic flow and content, but also to store that content within the database structures themselves. With an increased focus on the role of the database in the application's architecture, increased demand for data and increased richness of media, databases today are tasked more heavily than ever. Yet the principal concepts at the core of RDBMSs remain unchanged – to be relational, even modern databases must adhere to tabular conventions which may not be best suited to the application.

One problem with relational databases as they exist today is found in the method of accessing data from the database using Structured Query Language, SQL. SQL is a language used to report data from a database structure such as a table or view (Powell, 2006). The query language generally processes one command at a time and returns the results in a tabular format. As a non-procedural language, SQL does not allow the declaration of procedures, but most versions allow blocks of instructions to be sent to the engine in sequence. The report and result nature of SQL is both its strength and weakness: developing queries to return results is fairly straightforward, but the depth and power of procedures and object-oriented paradigms is lost to the desired simplicity. Some RDBMS companies like Microsoft have made great strides in implementing more powerful programmatic principles such as TRY-CATCH blocks, conditional statements, and loop structures, but these logical devices remain outside the core structure of the database (Nielsen, White, & Parui, 2009). Despite these shortcomings, RDBMSs and SQL provide an easy to conceptualize, powerful data storage and retrieval solution.

Web search behemoth Google, social internet king Facebook, and common household name Amazon all have one thing in common: hundreds of millions of users worldwide making thousands of requests every minute every day, creating just as much transaction data to be logged by the applications. In 2008 Google was estimated to have fulfilled 120,000 searches within a fraction of a second (Vise & Malseed, 2008). Facebook has ridden the social networking craze like a champion surfer – their user base has grown from a college fad for ~5 million students in 2005 to a worldwide phenomenon of 250 million users in July, 2009 (Shuen, 2008) (Zuckerberg, 2009). Finally, Amazon, which began as a humble online book ordering application, was estimated to process $1.35 million worth of transactions per *hour* during the last quarter of 2005 (Hennessy, Patterson, & Arpaci-Dusseau, 2007). Traditional, antiquated RDBMSs could be utilized in a large array of servers to handle Google and Amazon's sizable requests, but would fail miserably at scaling to keep up with Facebook's enormous growth. Applications of the class of Google, Amazon, and Facebook are truly beyond the limits of commercial offerings, and would have to be tailored into a

custom RDBMS to handle all the aspects of these modern systems.

HARDWARE SOLUTIONS

Computer hardware manufacturers alleviated some of the strain that modern applications put on database servers by researching ways to pack more processing power into smaller spaces. The introduction of multi-processor motherboards, which allow the operating system access to more than one Central Processing Unit, CPU, was a space and cost-effective means to increase floating point operations per second, FLOPS. However, multi-processor configurations required both an awareness of hardware specifications and also an operating system capable of utilizing both CPUs. Without being able to segment program operations into threads of instructions, single-threaded systems were limited to utilizing each CPU for one process. With the advent of multi-threaded operating systems such as Windows NT and UNIX variants, applications were written to capitalize on the ability of multi-processor hardware to efficiently utilize resources (Flynn & McHoes, 1997). Further advances were made by placing multiple processing cores on a single CPU chip and allowing the motherboard to

schedule threading, rather than relying on the operating system and developers to define the multi-threading. Despite major advances in multi-threaded application development and multi-core hardware implementation, modern applications require more than hardware to meet demand.

Creating large arrays of multi-core servers might be one approach to solve the problems that modern application usage generates. However, from a database management perspective, a new set of problems are introduced revolving around real-time transaction logging. Server arrays are not inherently aware of each other's objects on disk and in memory, so additional resources must be consumed by inter-server communication, disk writing, and memory status logging. In addition to being highly wasteful, this approach is also fault intolerant: a hardware failure in one server would potentially bring the entire system down in order to prevent disparate system states. Unfortunately this approach was typical to scalability problems associated with the "dot com" era of computing – explosive, unplanned growth and large amounts of investment led to firms amassing vast server farms that were inefficient and poorly thought out. Hardware failures led to loss of fickle investor confidence and much finger-pointing between

software developers, network engineers, and hardware manufacturers (Luetkehoelter, 2008).

DISTRIBUTED DATABASE MANAGEMENT SYSTEMS (DDBMS)

With the enormous success and infamous failures of the dot com boom in mind, Internet application giants of today have poured vast resources into avoiding the catastrophic repercussions of uncontrolled scalability. Even today's marketplace is volatile and subject to the interest of users. However, one technology that has made companies like Google, Facebook, and Amazon successful is distributed database management systems (DDBMSs). Distributed databases are simply databases whose physical resources are spread across a series of servers. Despite decentralizing resources, DDBMSs are designed to appear as a traditional, single database (Rob, Coronel, & Crockett, 2008). DDBMSs often comprise a series of computers called nodes with a central node serving as a master server, scheduling work to be broken into pieces and then handed out for each node to process and return results. A Microsoft SQL Server 2008 implementation of an application that uses multiple DDBMSs each with multiple SQL Server 2008 databases to accept distributed transactions is illustrated in Figure 1.2.

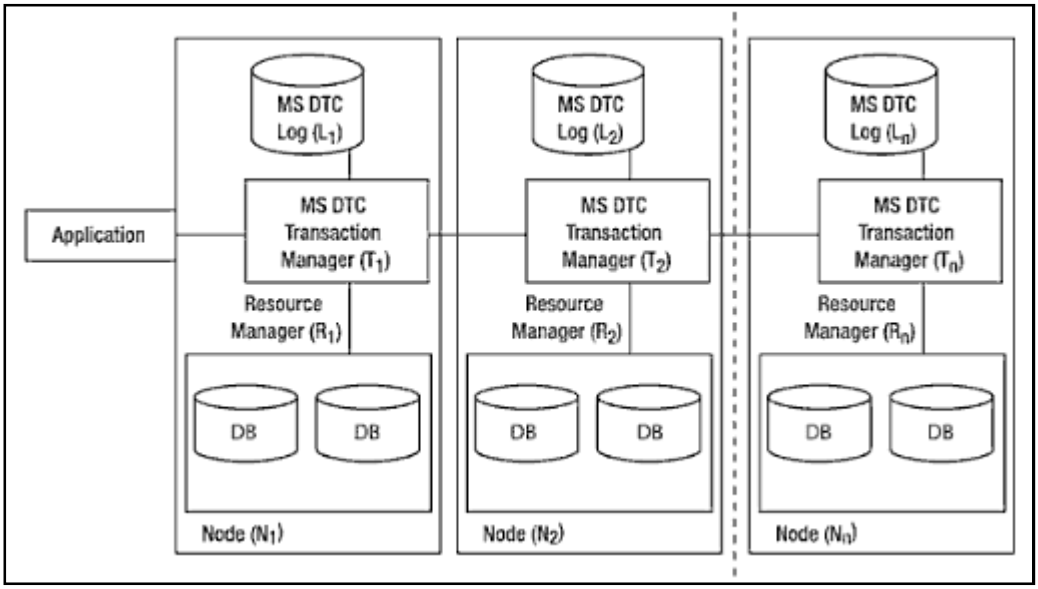

Figure 1.2. SQL Server 2008 Distributed Architecture. From (Paul, 2009).

Distributed databases are ideal for modern internet-enabled application architectures because they are designed to provide seamless scalability – network hardware is the only major technical limitation of robust distributed systems. In addition to handling scalability with ease, distributed database systems typically handle failure gracefully. When a node reports failure to the job scheduler for a task assigned, the scheduler simply sends the job to another available node for completion. By breaking tasks into small segments, the impact of the failure is localized and the scope contained, if not completely mitigated, by the system. Third, distributed systems allow businesses to reuse old hardware in addition

to purchasing new hardware. No longer do businesses necessarily have to keep up with frantic hardware buys when new technology emerges; instead, cheaper, more cost-effective hardware can be purchased to augment systems already in place. Finally, the attitude of reuse and efficient data center management is essential to many businesses' commitment to ecological responsibility.

Google, Inc. has used distributed databases at the core of their organization's "green" initiatives. Being the most used search engine worldwide necessitates vast server clusters to handle hundreds of thousands of requests per minute in under a second each. By optimizing their hardware and leveraging distributed systems to place data centers in a network-efficient manner, Google has been able to reduce their carbon footprint to around 0.007 ounces of carbon dioxide per query (Google, Inc., 2009). To put the number in perspective, see Table 1.1.

Common Activities	**Carbon Dioxide Equivalent Google searches**
Carbon dioxide emissions of an average daily newspaper (100% recycled paper)	850
A glass of orange juice	1,050
One load of dishes in an Energy Star dishwasher	5,100
A five mile trip in the average U.S. automobile	10,000
A cheeseburger	15,000
Electricity consumed by the average US household in one month	3,100,000

Table 1.1. Google searches per activity as measured in carbon dioxide emissions. Adapted from (Google, Inc., 2009).

Distributing data centers allows Google to achieve extremely low network latency, or delay, between a client and data centers, thus allowing Google's servers to run slower than max or to be put on lower powered, more energy-efficient hardware. DDBMSs within each data center also decrease overall server idle times while allowing servers to be turned off completely during periods of low usage demand (Ghemawat, Gobioff, & Leung, 2009).

HADOOP, A POPULAR OPEN-SOURCE DDBMS

In addition to being truly gargantuan Internet entities, Google, Facebook, and Amazon, (and Yahoo!), also all contribute to, and benefit from an open-source DDBMS

project organized by The Apache Foundation called Hadoop. Hadoop's origins are owed to the Nutch project headed by Doug Cutting in 2002. The project grew and MapReduce papers were released, solving some of the difficulties of Nutch. Mr. Cutting began working for Yahoo! and Apache to bring Hadoop up to a web scale, which was achieved in 2008 (Cutting, 2009). As the Apache Foundation describes, "[Hadoop] provides a distributed file system (HDFS) that can store data across thousands of servers, and a means of running work (Map/Reduce jobs) across those machines, running the work near the data" (Apache Foundation, 2009). In general terms, Hadoop uses MapReduce algorithms with a job scheduler to allow queries on a Hadoop File System to be split into jobs and assigned to the many nodes of a distributed system. Hadoop takes advantage of MapReduce's highly distributable framework. MapReduce works in two steps, first by creating maps between nodes and secondly processing those maps with user-defined functions in the "reduce" step. Through the use of a master, "name node" server, MapReduce tasks can be sent to "data nodes" for processing in parallel. If a node fails, the name node schedules another data node to perform the failed task,

resulting in quick turnaround and limited impact to the end user (Apache Foundation, 2009).

Hadoop and MapReduce's popularity has pushed both technologies into new areas of application. In fact, Hadoop now supports the use of Amazon's cloud Simple Storage Service, S3. Cloud computing is an architectural idea that major datacenters, combined with broadband internet access to and from users' access points, can effectively replace functions previously held by the workstation (Greenwald, Stackowiak, & Stern, 2004). From secure, encrypted online storage to web-based applications that allow real-time collaboration to entire operating systems residing within the browser, cloud computing has the potential to change the way we use computers forever. Despite the frequency of grandiose visions that appear in the Information Technology industry, cloud computing does deliver real, powerful technological gains when combined with DDBMSs. One success story is documented by *New York Times* who used Amazon's S3 and Elastic MapReduce services to convert millions of archives from scanned images to text documents. The project was expected to take months. After completing the image recognition script development, *New York Times* was able to perform initial analysis on four servers and then scale the

project up to 100 (Jordan, 2009) servers in order to complete processing in 24 hours (Gottfrid, 2007). The computing price is estimated to be less than $300 – an extremely minute sum compared to the enormous amount of labor the distributed system performed, flawlessly. While time will tell how quickly the era of cloud computing makes a grand entrance in the consumer world, the results and power of the cloud combined with smart technologies like Hadoop is being realized and capitalized by forward-thinking businesses today.

CHAPTER TWO – HADOOP IN ACTION: RSA PRIME FACTOR SEARCH

METHOD OVERVIEW

In order to discuss how Hadoop and MapReduce work together in cloud environments we will demonstrate a simple example. One of the most widely used encryption schemes is RSA (Stallings & Brown, 2008). RSA works on a public and private key exchange and an algorithm that hinges upon the factoring of very large numbers that have only two prime factors. Two very large primes are chosen and used to determine public and private keys. The keys are then used to generate encrypted messages. RSA's encryption strength lies in the fact that factoring very large numbers is computationally inefficient with a brute-force approach on a single machine.

However, the power of distributed systems is their large levels of parallelism. By using a cluster, the number of primes that can be searched by 4, 8, and 16 nodes in an hour will be determined. Searching primes by applying a brute-force method is not particularly useful in practice, but is merely intended to show the flexibility and power of cloud computing. The task is also well-suited to Hadoop and MapReduce: a list of primes will be generated as the

dataset, mapping will be performed, and a simple script to determine whether each prime is a factor or not will be processed. Specifically, each data node will be sent a section of prime numbers and a value to test each prime against. The value to be tested against will be the product of two large prime numbers, and will be called R. Prime number set partitioning is handled by Hadoop and the job scheduler. Each data node will then run a reducing script to determine whether the dataset contains a factor of the number R, and return the value of any found factors.

A BRIEF RSA OVERVIEW

The RSA algorithm is an encryption method that uses asymmetric key pairs to digitally sign, encrypt, and secure plaintext transmissions. RSA was developed in 1977 by Ron Rivest, Adi Shamir, and Len Adleman at MIT (Stallings & Brown, 2008). Since then, RSA has remained adequate and has been adopted into the security practices of many electronic transactions that deal with personal, sensitive, or classified information. RSA's strength rests in using two very large primes' (1024 bit, or approximately 300 numerals) product as the encryption mechanism. These primes are chosen in such a way that the key pairs are

multiplicative inverses modulo $\varphi(n)$, where $\varphi(n)$ is the Euler totient of *n*. The Euler totient of *n* is the number of positive integers relatively prime to *n* (Stallings & Brown, 2008). In other words, the following relationship must be met by an encryption key, *e*, decryption key, *d*, and message, *M*, as seen in Equation 1:

$$M^{ed} \bmod n = M \tag{1}$$

The standard RSA encryption algorithm works as follows (Stinson, 2006):

1. Select two prime numbers, *p* and *q*.
2. Calculate *n = p * q*.
3. Calculate $\varphi(n)$ = *(p – 1) * (q – 1)*.
4. Select positive integer, *e* such that *e* is relatively prime to $\varphi(n)$, and less than $\varphi(n)$.
5. Determine *d* such that *d*e* mod $\varphi(n)$ = 1 and *d* < $\varphi(n)$.

When selected using these mathematical properties, a message, *M*, can be encrypted by the sender using the receiver's public key, *e*, illustrated in Equation 2:

$$M^{e} \bmod n = C \tag{2}$$

The resulting cipher-text, *C*, can then be securely sent to the receiver for decryption with the receiver's private key, *d*, as shown in Equation 3:

$$C^d \bmod n = M \tag{3}$$

The strength of the RSA algorithm is relative to the size of the chosen primes, *p* and *q*, and the astronomical computational cost to factor *n*. A theoretical attacker named Eve, who accesses the publicly-known information, *e* and *n*, and also factors *n*, could then calculate *p*, *q*, and $\varphi(n)$. Eve may then trivially determine the private key, *d*, using the relationship *d* * *e* mod $\varphi(n)$ = 1. With *d* known, Eve now not only decrypts secure transmissions, but also creates new messages to impersonate the private key's owner, breaking the non-repudiation feature of RSA.

By selecting primes of sufficient length, factoring by naïve brute-force methods quickly approaches thousands of MIPS-years (million-instructions-per-second) (Stallings & Brown, 2008). Despite the staggering number of calculations needed to factor practical keys of 512-bit and larger sizes, new hardware inventions are making significant strides toward enabling naïve approaches to key discovery.

Table 2.1 illustrates the rapid rate of RSA key discovery on a sampling of RSA numbers from 100 to 155 digits (332 to 512 bit keys).

Number of Decimal Digits in RSA Number	Approximate Number of Bits in RSA Number	Date Factorization Achieved	MIPS-Years Required
100	332	April 1991	7
110	365	April 1992	75
120	398	June 1993	830
129	428	April 1994	5000
130	431	April 1996	1000
140	465	February 1999	2000
155	512	August 1999	8000

Table 2.1. Progress in factoring RSA numbers as measured in million-instructions-per-second years (MIPS-Years). From (Stallings & Brown, 2008), 639.

Due to considerable progress in computer architecture and optimized factoring with the use of sieves, keys of 1024 and 2048-bit strength are recommended (Stinson, 2006). In addition to using extremely large keys, using one-time pads, frequent key regeneration, and other methods of inserting randomness into the system are often recommended to extend the strength and life of RSA security schemes (Stinson, 2006).

FACTORING PROBLEM

Many complex and efficient methods exist to factor numbers. Elliptic curves, numbers sieves, and exploitation of number theoretic properties of classes of numbers can

all drastically reduce the computational expense of factoring. In addition, recent advances in quantum computing have allowed quantum algorithms to be developed that are able to solve discrete logarithm problems and integer factorization problems in less than polynomial time (Jordan, 2009). Other than quantum algorithms, all of the above methods are in essence brute force attacks, requiring more computations than the size of the number in question. In contrast, the most naïve of methods to factor a product of two large primes, n by brute force would consist of an algorithm to test all odd numbers less than or equal to the next odd greater than the ceiling of the square root of n.

In order to utilize cloud computing and specifically Hadoop with MapReduce, a non-traditional and resource-intensive method of factoring will be employed. The distributed nature of Hadoop will be used to partition a superset of the possible factors that we are interested in discovering, namely the set of all odd integers and the only even prime, 2. While number theoretic approaches focus on using elegant mathematical techniques, cloud computing harnesses the power of cheap, highly scalable hardware architecture.

To exploit the available hardware and computational scheme of MapReduce, primes must be chosen that will be big enough to create measurable run times, but small enough for each data node to complete processing in a few hours. To determine ideal primes to use for *p* and *q*, algorithm run times were recorded for the product *p*q=n*. Rather than attempting to factor *n* by looping through all odd numbers, the program will use our knowledge of the factors of *n* to reduce the numbers to test from *n-1* down to $\sqrt{n}$. Since *n* has exactly two factors, *p* and *q*, we know that the lesser factor must be found after $\sqrt{n}$ tests. In order to test each number, the list of odd numbers and 2 will be generated and uploaded to Amazon's Simple Storage Service. By creating a dataset of possible factors of *n*, MapReduce will be able to distribute partitions of the data to various data nodes for processing, thus reducing the overall processing time required to factor *n.*

Another vector of the system's performance is the method of determining whether the dataset elements are factors of *n* or not. Rather than using complex factoring methods, the a priori knowledge of *n* allows the use of number theoretic techniques such as modular arithmetic to determine whether a number between 2 and *n-1* is a factor of

n. Since *n* is the product of exactly two prime numbers, *p* and *q*, we know that the greatest common denominator (GCD) of *n* is the greater of *p* or *q*. Once we know the GCD of *n*, finding the second factor is trivial – simply divide *n* by the GCD. One effective algorithm for determining GCD is Euclid's Algorithm (Stinson, 2006), which is rendered recursively in the language Python for Figure 2.1.

```
def euclids_gcd(a,b):
    if (b==0):
        return a
    return gcd(b, a%b)
```

Figure 2.1. Euclid's GCD algorithm implemented in Python.

Many more techniques, ranging from simple to advanced can be employed to increase the efficiency and breadth of factoring, as well as to reduce the number set that will be searched (Stinson, 2006). For the purposes of this paper, however, a level of inefficiency is desired in order to both effectively convey the topic and more accurately predict running times (and therefore reduce service costs). Preliminary running time testing of ideal prime size was performed on a 3 GHz Intel Core 2 Duo processor using the "DatasetGenerator.py" script (see Appendix). RSA-100, a 100 digit number that is the product of two 50 digit primes was

initially desired but proved to be computationally infeasible to factor within minute or hour scales.

Through trial and error, primes under 50 bits or 10 numerals were found to be sufficiently large to cause the program logic to run for a human-detectable time without taking more than 24 hours. Two 32-bit primes, 2,259,259,231 and 2,259,259,243, were selected and tested for processing suitability. The primes were multiplied together to create *n*, (5,104,252,299,969,822,133), and all the odd numbers between *1* and $\sqrt{n}$, including *2*, were tested using Euclid's GCD algorithm (see Appendix). The entire process ran for over 13 hours and determined that only 2,259,259,231 was a factor of *n* within the numbers tested. In addition to the overall running time, the current elapsed running time was output after every millionth iteration (2 million numbers checked). For these small values, the elapsed time between millions of iterations remained fairly consistent, increasing from approximately 7 seconds per million to 11 seconds per million at the time of determining the first factor.

The results should reflect the general performance of each data node in the cloud. While the smallest tier of servers available for cloud services are listed as having

1.6 GHz AMD Athlon processors, the overall performance per node will be within the same scale as the preliminary testing system. Each data node will be instructed to run Euclid's GCD algorithm against the partition of the dataset that the master node provides to the data node. An example of Amazon's MapReduce architecture is seen in Figure 2.2.

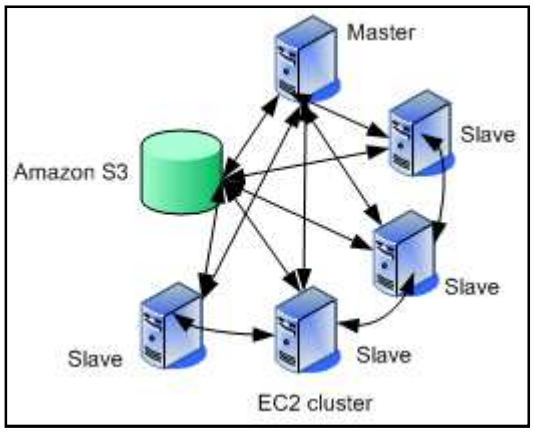

Figure 2.2. Amazon Elastic MapReduce architecture diagram. Data stored in S3 is distributed by the Master node to scalable Slave nodes to form an EC2 cluster. From (Amazon Web Services, LLC, 2009).

Performance gains from distributing the data set across data nodes will be tested on four, eight, and sixteen nodes. For the first 1,129,629,618 odd numbers less than $(\frac{\sqrt{n}}{2} + 1)$ that will be tested, the data node size choices will yield, at most, 283 million, 142 million, and 71 million iterations of Euclid's GCD algorithm, respectively, on each data node.

CHAPTER THREE – RESULTS AND ANALYSIS

ELASTIC MAPREDUCE RUN TIMES

The dataset consisting of the first 1,129,629,618 odd numbers, including 2, was written to local disk at a storage cost of 13 GB. Since Amazon's S3 only accepts files of size 5 GB and below, the 13 GB dataset was split into thirteen 1 GB files before uploading. The mapping and reducing scripts were written in Python and named "gcd_mapper.py" and "gcd_reducer.py" (see Appendix), respectively. The mapping script consists of Euclid's GCD Algorithm applied to the standard input that is piped from S3 into the script. The mapping script prints the results of the GCD function for piping into the reducing script. The reducing script examines the results of the map and returns the values of any dataset element whose value is equal to the returned value of the GCD map. Because of the underlying structure in the chosen n and the elements of the dataset selected, the resulting output therefore contains only the two factors, p and q, of n.

The depth of the factor search is determined by the supplied values of p and q within the code of "gcd_mapper.py". In order to maximize the use of the dataset, the consecutive prime values closest to the last

dataset value are selected as 2,259,259,231 and 2,259,259,243.

In addition to testing each element of the dataset for GCD with *n* on four, eight, and sixteen nodes, two levels of nodes were compared. Amazon Web Service (AWS) provides a number of different tiers of servers, and the least and most powerful were used. On the low powered end, AWS's "m1.small" node tier is equivalent to approximately 1.6 GHz Athlon CPU. At the upper end, the "c1.xlarge" processing power is roughly 8 times that of "m1.small", or a 12.8 GHz Athlon CPU (Amazon Web Services, LLC, 2009). The results of the job runs are detailed in Table 3.1.

Server Tier (# Nodes)	**Elapsed Time**	**Total Instance Hours Consumed (Nodes * Hours)**
m1.small (4)	6 Hours 55 Minutes	28
c1.xlarge (4)	38 Minutes	32
m1.small (8)	2 Hours 42 Minutes	24
c1.xlarge (8)	29 Minutes	64
m1.small (16)	1 Hour 27 Minutes	32
c1.xlarge (16)	30 Minutes	128

Table 3.1. GCD MapReduce running times and overall computational hours consumed.

RESULTS EXPLORATION

While limited in depth, the results support the idea that additional nodes will cause an increased performance,

and thus decreased elapsed run time for the GCD algorithm across the dataset of all odd numbers from *1* to $\sqrt{n}$, including *2*. As nodes scaled from 4 to 8 to 16 on the "m.small" architecture, run times decreased from 415 minutes to 162 to 87. As the dataset was split across more and more nodes, each node's computation expense was decreased, and the length of time needed to complete the task decreased. In terms of the MapReduce system's overall computational expense, however, the results showed that adding more nodes does not necessitate increased processing efficiency.

Server Tier (# Nodes)	**Elapsed Time**	**Overall Compute Time (Elapsed Time * Number of Nodes)**
m1.small (4)	6 Hours 55 Minutes	27.7 Hours
m1.small (8)	2 Hours 42 Minutes	**21.6 Hours**
m1.small (16)	1 Hour 27 Minutes	23.2 Hours

Table 3.2. GCD MapReduce overall computation times in compute hours.

While "m1.small" tier results appear conclusive, "m1.xlarge" trials appear to have completed too quickly to form useful results. Inconsistency is particularly noticeable when comparing the run times of eight and sixteen node runs on the "m1.xlarge" architecture. The run times were nearly identical, with eight nodes taking 29 minutes to complete, and sixteen nodes taking 30 minutes –

a result that is counterintuitive. In addition, the normalized instance time for eight nodes was 64 hours, while sixteen nodes consumed 128 instance hours. One explanation of the disparity can be examined in the way that MapReduce's job scheduler works. Before processing can begin, the dataset must be parsed and distributed among nodes. The results of "c1.xlarge" runs support a scenario where the upfront computational expense of parsing the dataset vastly outweigh the CPU time needed to perform the mapping and reducing script work. Another cause of the inconsistent run times could reside in the stdin and stdout passing of data from the dataset parse to the mapping function, from the mapping function to the reducer, and finally from the reducer to the output files.

ECONOMIC FEASIBILITY OF AN ATTACK

Selecting increasingly larger primes to increase the security of RSA cryptosystems will eventually reduce the computational efficiency of the system. To increase the security of an RSA system without causing burdensome computations, one-time and time-based padding information can be applied. By recycling this information on a fixed period, the system can provide defense against brute-force methods. Padding can reduce the vulnerability of attack by

a single user attempting to try all possible keys, but the simple scenario described in this paper can effectively overcome systems that recycle keys less frequently than an hour. The only deterrent an attacker has is cost.

The example's processing time is dependent on the number of elements to be tested, which are dependent on the size of the key selected. The dataset used required AWS to test over one billion (1.1e+10) odd integers generated from using 32-bit keys. Real world applications of RSA are advised to use 1024-bit and 2048-bit modulus values. In contrast, using the 1024-bit modulus provided by RSA's Factoring Challenge yields a dataset of odd numbers consisting of 5.8e+153 numbers (RSA, 2008).

The most efficient architecture from the example was four "c1.xlarge" nodes. The configuration successfully factored the target modulus by testing over 283 million numbers in 38 minutes. In order to obtain similar performance in a 1024-bit modulus system, each "c1.xlarge" node would need to test the same number of elements as the 64-bit modulus system. Dividing the number of elements in the 1024-bit modulus system, 5.8e+153, by the number of elements per node in the 64-bit modulus system indicates

the 1024-bit system would require 5.3e+143 nodes to achieve a comparable running time.

At the time of the example, Amazon charged $0.80 per computing hour for “c1.xlarge” architectures, and an additional $0.015 for each Elastic MapReduce computing hour. Using these figures, a theoretical 1024-bit example would cost $4.32e+143 – an impossibly large amount. Another way to view the results could begin with an attacker with $1,000,000 who wanted to use the system to factor a 1024-bit number. Based on the current AWS prices, the attacker would be able to purchase 1,226,993 computing hours. In order to obtain performance levels of the 64-bit example each node would need to process 4.7e+147 numbers. The amount of work scales by a factor of 1.7e+139 from the 283 million elements per node in the 64-bit system to the 4.7e+147 elements per node in the theoretical 1024-bit scenario.

The proposed dataset of odd numbers less than the square root of the modulus is clearly infeasible. It is unlikely in the foreseeable future that any datacenter could house enough servers to provide 5.3e+143 nodes, entity could spend $4.32e+143, or realistic hardware gains produce a 1.7e+139 scale performance increase. However, the

system proposed in this paper is elementary in scope. Applying more advanced number theory principals to exclude more subsets of numbers from the test subset could provide significant performance gains.

CHAPTER FOUR – CONCLUSION

The example explored in this paper served a two-fold purpose: to show how cloud computing with the MapReduce architecture can satisfy some requirements of modern database systems and to give a brief example of how a very naïve brute-force attack on a weak asymmetric security system based on RSA could utilize cloud computing with little monetary and time expense. As an example of a modern database system, the example showed the ease of which scalability and capacity issues can be handled. Amazon's cloud service scales as easily as adding more nodes when requesting a new job, which is performed through a web browser. Capacity is also specified during the job creation process. Amazon provides five different types of hardware architectures for the compute cloud, allowing a user to fine-tune the most cost-effective method of processing.

As a simple application of MapReduce, factoring a number n into its two prime factors, p and q, was selected due to the relative ease of mathematical concepts and interest as a computer science topic. In reality, a brute force attack is particularly difficult to perform against RSA encryption, even with a cloud computing service. However, the results found that it is certainly possible,

and in some cases feasible, to approach RSA factoring through the use of clustered computers, a large dataset of numbers, and the MapReduce architecture. The example found the factors of a 64-bit sized modulo in times ranging from 29 minutes to 6 hours and 55 minutes, depending on server size and number of nodes employed.

While RSA recommends cryptosystems to use 1024 and 2048-bit keys, the meager 32-bit keys employed are fairly insignificant to an ultimate goal of brute-force cracking a real-world RSA scenario (Stinson, 2006). Despite their small size, some important considerations should be noted: elapsed times were in tens to hundreds of minutes, and the number of nodes employed was small – sixteen nodes at most. By simply scaling the system to hundreds or thousands of nodes and by attacking RSA systems that do not frequently retire keys, a brute-force attack could be viable.

Adding hardware and allowing more computation time are simple gains; more extreme performance enhancements could be implemented by reducing the number of elements in the dataset to be searched. The ideal dataset would contain only the primes between *2* and the largest 1024 or 2048-bit primes. In addition, further research and development could improve performance through the use of advanced cloud

programming that would cause the nodes to generate their own datasets and cause the entire system to halt after finding a factor, thus reducing wasted iterations. By exploring and implementing advanced techniques, factors of very large RSA numbers could be within reach of current cloud computing systems.

BIBLIOGRAPHY

Amazon Web Services, LLC. (2009, October 2). *Amazon Web Services*. Retrieved October 2, 2009, from Amazon Web Services: http://aws.amazon.com

Apache Foundation. (2009, July 21). *Hadoop Wiki*. Retrieved September 7, 2009, from Apache.org: http://wiki.apache.org/hadoop/ProjectDescription

Cutting, D. (2009, October 26). *Hadoop: A Brief History.* Retrieved October 26, 2009, from Yahoo!: http://research.yahoo.com/files/cutting.pdf

Flynn, I. M., & McHoes, A. M. (1997). *Understanding Operating Systems.* Boston: PWS Pub.

Ghemawat, S., Gobioff, H., & Leung, S.-T. (2009, January 1). *The Google File System.* Retrieved September 5, 2009, from Google: http://labs.google.com/papers/gfs.html

Google, Inc. (2009, May 8). *Clean Energy Initiatives*. Retrieved September 9, 2009, from Going Green at Google: http://www.google.com/corporate/green/datacenters/

Gottfrid, D. (2007, November 1). *Self-service, Prorated Super Computing Fun!* Retrieved September 9, 2009, from New York Times: http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/

Greenwald, R., Stackowiak, R., & Stern, J. (2004). *Oracle essentials: Oracle database 10g.* Sebastopol: O'Reilly Media, Inc.

Haigh, T. (2006, September 8). A Veritable Bucket of Facts:' Origins of the Data Base Management System. *ACM SIGMOD*, 33-36.

Hennessy, J. L., Patterson, D. A., & Arpaci-Dusseau, A. C. (2007). *Computer Architecture: A Quantitative Approach.* Boston: Morgan Kaufmann.

Jordan, S. P. (2009, September 13). *Quantum Computation Beyond the Circuit Model.* Retrieved November 1, 2009, from arXiv.org: http://arxiv.org/PS_cache/arxiv/pdf/0809/0809.2307v1.pdf

Luetkehoelter, J. (2008). *Pro SQL Server Disaster Recovery.* Berkeley: Apress.

Nielsen, P., White, M., & Parui, U. (2009). *Microsoft SQL Server 2008 Bible.* Indianapolis: John Wiley and Sons.

Paul, S. (2009). *Pro SQL Server 2008 Replication.* Berkeley: Apress.

Powell, G. (2006). *Beginning Database Design.* Indianapolis: John Wiley and Sons.

Rob, P., Coronel, C., & Crockett, K. (2008). *Database Systems.* London: Cengage Learning.

RSA. (2008, August 29). *Challengenumbers*. Retrieved November 16, 2009, from RSA Laboratories Home: RSA Security Research Center: http://www.rsa.com/rsalabs/challenges/factoring/challengenumbers.txt

Sheriff, P. D. (2002, February 1). *Building an N-Tier Application in .NET.* Retrieved October 26, 2009, from MSDN: http://msdn.microsoft.com/en-us/library/ms973279.aspx

Shuen, A. (2008). *Web 2.0: A Strategy Guide.* Sebastopol: O'Reilly Media.

Stallings, W., & Brown, L. (2008). *Computer Security Principals and Practice.* Upper Saddle River: Pearson Education, Inc.

Stinson, D. R. (2006). *Cryptography Theory and Practice.* Boca Raton: Chapman & Hall.

Vise, D. A., & Malseed, M. (2008). *The Google Story: For Google's 10th Birthday.* New York: Delacorte Press.

Zuckerberg, M. (2009, July 15). *Now Connecting 250 Million People.* Retrieved September 6, 2009, from Facebook: http://blog.facebook.com/blog.php?post=106860717130

APPENDIX

Supporting Code

```
#!/usr/bin/python

import sys

p = 2259259231
q = 2259259243
n = p*q

def gcd(a,b):
    if (b==0):
        return a
    return gcd(b, a%b)

#Main program logic:
for element in sys.stdin:
    element = long(element)
print str(element) + "\t" + str(gcd(n,element))
```

Figure A.1. GCD_Mapper.py

```
#!/usr/bin/python
import sys

for line in sys.stdin:
    (element, gcd_result) = line.strip().split()
    if element == gcd_result:
print('Factor found! Value = ' + str(element) + '\n')
```

Figure A.2. GCD_Reducer.py

```
def DecimalToBinary(n):
    bStr = ''

    if n < 0: raise ValueError, "must be a positive integer"
    if n == 0: return '0'
    while n > 0:
        bStr = str(n % 2) + bStr
        n = n >> 1
    return bStr

print(DecimalToBinary(2259259231*2259259243))
print(len(DecimalToBinary(2259259231*2259259243)))

#10001101101010111110010011011000101111110101011111111110110110101
#63
```

Figure A.3. DecimalToBinary.py

```
import time
from math import sqrt

start = time.time()

p = 9990454949
q = 9990454939
n = p*q
factors = [ ]

def gcd(a,b):
    if (b==0):
        return a
    return gcd(b, a%b)

#print(gcd(n,p))
i = 2
if(gcd(n,i) == i):
    print("A factor is: " + i)
else:
    i = 3

while(i < sqrt(n)+1):
    if(i%1000001==0):
        print(i)
        print("Current Elapsed time: " + str(time.time()-start))
    if(gcd(n,i) == i):
        factors.append(i)
    else:
        i += 2

print("The factors of " + n + " are: " + factors)
print("Total Elapsed time: " + str(time.time()-start))
```

Figure A.4. DatasetGenerator.py