Predictive Shipping

A Major Qualifying Project Report

Submitted to the Faculty of Worcester Polytechnic Institute
In Partial Fulfillment of the Requirements for
the Degree of Bachelor of Science

By Matthew Beaulieu, Tiffany Leung, and Derek McMaster

March 3, 2017

Sponsored by
Endicia, Amine Khechfe, Patrick Farry, and Subhasis Saha
Advised by
Professor Mark Claypool

Abstract

United States Postal Service offers several mail classes that vary in shipping time and price. First class is the most economical shipping method, but it does not guarantee a delivery date. Endicia, our sponsor, did not have a tool that could accurately predict when first class packages would deliver. This project produced a validated prototype that forecasts delivery time distributions using conditional probability. The results showed that shipping times highly depend on when a package was originally dropped off. Ultimately, this prototype can help Endicia's business clients greatly optimize their shipping costs.

Acknowledgements

We would like to express gratitude to the individuals and organizations who supported us during our project.

Our Sponsor

Endicia

Endicia Point of Contacts

Aminé Khechfe
Co-founder and General Manager
Subhasis Saha
Vice President R&D
Patrick Farry
Project Manager

Our Advisor

Mark Claypool
Professor, Interactive Media & Game Development
Silicon Valley Project Center

Table of Contents

Table of Figures

Table of Tables

1 Introduction

As society becomes increasingly interconnected, ecommerce continues to gain popularity and market share in the retail space. Projected to reach more than 500 billion dollars by the turn of the decade, e-commerce currently represents a sizeable 373 billion dollars and growing (Vassallo, 2016). While there are numerous attributes affecting e-commerce, shipping is one complex trait that shapes online shopping trends. Delivery time estimations are one factor that contribute to a consumer's decision on whether or not to make a purchase. Recent studies show that 60 percent of those surveyed indicated that estimated or guaranteed delivery dates are important in their purchasing decisions (comScore, 2012). Looking further into the statistics, online shoppers are willing to wait at most 6 days from the purchase date if they paid for shipping, and 8 days if they received shipping for free (comScore, 2015). These statistics become crucial to drive or maintain sales relationships. While larger businesses may have their own statistics and data to offer delivery time estimations, many businesses and consumers rely on estimations given by United States Postal Service (USPS).

Endicia, a company focused on e-commerce shipping solutions, is interested in producing a validated, accurate system that forecasts delivery time statistics for its clients, namely shippers, to use in order to optimize their shipping costs. Endicia had a proof-of-concept prototype and wanted to extend it to create a system that was reliable, extensible, and maintainable.

The goal of our project was to produce a validated prototype that forecasts delivery time distributions to optimize shipping costs. We accomplished this goal by (1) building a new, extensible prototype that handled big data, (2) using optimization tools to elevate performance and capabilities of the prototype to meet Endicia production standards, and (3) validating the probabilistic approach used to predict shipping time distributions.

The results of the predicted shipping time distributions made by the prototype showed that drop-off time has a significant impact on the overall shipping time for a package. We were able to utilize Spark and Neo4j efficiently to process the data from a raw form into an intelligent form that our database understands. However, the predictions of the prototype were not confident enough to be used in production and can be improved by adding more data for more robust calculations and factoring in other variables that affect shipping time.

Chapter 2: Background will introduce the problem that led to the project and the relevant technologies used in development. Chapter 3: Requirements talks about both the technical and business requirements Endicia laid out at the beginning of the project. Chapter 4: Methodology describes the methodological approaches taken to accomplish the project objectives. Chapter 5: Design and Implementation lays out the finer-grain details on how each component of the prototype was implemented. These details include examples and relevant technical details such as important objects and methods. Chapter 6: Results shows the results of the performance of the prototype as well as predictions and accuracy testing. Chapter 7: Discussion is where the discussions of the results are found. Finally, Chapter 8: Conclusions and Future Works summarizes the project and accomplishments made with the prototype, open issues, and recommended improvements.

2 Background

This chapter is divided into three distinct sections: (1) the structure of United States Postal Service (USPS) and Endicia's use cases, (2) Endicia's existing predictive shipping prototype, and (3) the technologies and tools used to build the new predictive shipping prototype. The purpose of this chapter is to provide a background needed to understand the predictive shipping system.

2.1 Company Background

This section details the information about USPS and our company sponsor, Endicia. In addition, it explains how the two companies interact with each other.

2.1.1 United States Postal Service

The United States Postal Service is an independent U.S. Government Agency that provides consumer and commercial postal services. USPS accounts for 47 percent of the world's mail volume, delivering around 154.2 billion pieces of mail per year (USPS, 2016). Of that volume, 4.5 billion account for the total shipping and package volume. USPS states that their scheduled delivery dates are highly affected by origin location, destination location, and handling times during different segments of a package's trip. However, in 2009, the U.S. Government Accountability Office (GAO) conducted an analysis on the delivery operations of USPS facilities and found that delivery efficiency also depended on other factors such as when mail was received from a distribution center and how recently routes were adjusted. Currently, USPS has a Last Mile Technologies experience that has dynamic routing and real-time delivery scans to make delivery more flexible. This solution includes advanced barcode scanning technology, known as Intelligent Mail Postage Barcodes (IMPB) (USPS, 2015), that increases readability of barcodes, as well as a robust tracking system that can rapidly read and track packages. Predictive delivery times are calculated by scanning mail and packages "throughout the network to accurately predict delivery times".

2.1.2 Endicia Online Postage

Founded over three decades ago as PSI Associates, Endicia operates in the e-commerce shipping solutions market (Endicia, 2016). Located in Silicon Valley, California, Endicia provides customizable services to companies and customers across the world. Since its founding, the company has worked with the USPS. This relationship has been the foundation for many of Endicia's the software solutions and technologies. Endica created shipping and postage products, such as *Envelope Manager*, a product that reduced costs associated with undeliverable mail, and *Dial-A-ZIP*, a product that verifies addresses in real-time. Besides software, Endicia also offers APIs and integration to marketplaces and warehouse management systems. Endicia was acquired in late 2015 by Stamps.com (Stamps.com, 2015). Endicia's services and platforms have handled over $14 billion in postage (Endicia, 2017).

Organization of the extensive USPS features is one main selling point for Endicia. For many companies, especially personal business ventures or smaller business lacking shipping resources, Endicia automates and navigates shipping needs. Endicia's relationship with USPS allows maneuverability of different types of shipping, such as expedited shipping methods. Using the aforementioned real-time address verification technology, Endicia helps businesses ship orders faster, which increases the daily order load they can handle. Companies, such as *Olympia Sports*, gain the ability to use one shipping account across many branches giving them the ability to direct orders to a specific store and give company-wide assurance of an orders status (Fitzpatrick, 2017).

Automating repetitive tasks and label printing allow business customers to handle larger volume of parcels. The organizational measures mentioned previously are able to handle large volumes of parcels, encompassing a range of shipping characteristics. Similar to the multiple branch scenario, companies are able to use Endicia to centralize inventory for both shipping and return capabilities at any location or online.

As businesses grow, international customers are a natural market for them to target. Each different country requires a unique set of laws regarding import and export shipping. Endicia allows businesses to use automated, streamlined customs forms and navigate international shipping requirements to handle businesses' intercountry orders and returns. Organizing international shipping potentially saves companies costs of labor and costs associated with incorrectly shipping.

2.2 Problem Background

This sections contains information about the project problem and significance as well as Endicia's previous predictive shipping prototype. It also describes the mathematical theory behind the probabilistic model used to produce the shipping time distributions.

2.2.1 Predicting Package Shipment Time

Predicting a package's shipment time may seem like a straightforward problem at first. Amazon can guarantee Prime members will receive certain packages in two days. However, the question remains as to why predicting package shipping paths through the USPS not as simple as it seems (Amazon.com, 2017). The truth is that there are many variables to consider when predicting the estimated delivery date of a package.

One method of predicting shipping time is taking to the average estimated delivery date. A package estimated to take an average of two days to be delivered could be delivered 9/10 times in one day and 1/10 of the times in 11 days. The average is the same in either case, but 1/10 of the time results in a very unsatisfied customer. This wide deviation from the two day estimate is considered to be unacceptable for Endicia's clients if their use case for the prototype was to save money by shipping packages as first class mail hoping it would deliver in the estimated time predicted by with this averaging method.

Two main problems are the volume of data and scarcity of data. There are approximately 600 million packages mailed each year and assuming each package has 10 tracking scans would mean there could be six billion rows of data in one year. Such a large set of data is hard to manage. According to the USPS, there are over 31,000 Post Service Retail Offices, which means there are approximately 1 billion combinations of shipping pairs to consider. It would be difficult to predict package shipment time when using historic shipping time data between a pair of post offices because with such an extremely large set of combinations, Endicia does not have enough data to accurately model every possible post office to post office combination.

A final issue with predicting package shipment time is the number of variables affecting it. The time of day, day of the week, and season a package is dropped-off at can all affect estimated delivery date and time distributions (USPS.com, 2017). Some USPS offices only pick up packages once per day. If a package arrives after the daily pickup time, an additional day has

to be factored in to the expected delivery date. Also, packages often take longer to deliver in the winter due to inclement weather and seasonal holidays.

These flaws and issues show that predicting package shipment time is not a trivial task. Therefore, being able to accurately predict package shipment time based on historical data would be extremely valuable to shippers.

2.2.2 Current Prototype

The first predictive shipping prototype was created about two years ago by a summer intern at Endicia The goal for this prototype was to calculate a distribution of possible shipping times from a client's business to their customer. To counter for the difficulties discussed in Section 2.2.1, the prototype used the structure of the USPS network to make up for the billions of possible post office combinations. When a package is shipped through the USPS, it starts at a local office and then makes its way to a regional distribution center (RDC). From an RDC, a package is shipped to other RDCs, until it reaches a local RDC and is handed off to the local post office (USPS.com, 2017). Because all USPS shipping routes follow this pattern, Endicia only needed data to predict shipping time between RDCs and from local offices to their respective RDC. Instead of predicting 1 billion possible routes, segmenting a trip meant Endicia only needs to predict package arrival time for 3,000,000 sub trips between RDCs and local offices in the network.

While the above strategy accounts for the data scarcity issue, it does not mitigate the number of variables involved in predicting shipping time. To handle these, the prototype used a probabilistic model. It used Bayes Theorem/total probability formula to look at transit time of a package between facilities as a set of probability distributions conditioned on arrival time. This made intuitive sense based on discrete pick up and delivery events of packages.

$$P(A) = \sum P(A|\, B_i)P(B_i)$$

It then combined all these probabilities for each possible path a package would take through the shipping network. In turn, the problem of predicting a distribution of package shipment times simplified to a problem of mathematical convolution. Assuming each distribution of shipment

times from one node to the next is independent of the other, the solution is simply a sum of random variables.

$$P(T = t) = \sum_{h=0}^{\infty} F_1(h) * F_2(t-h)$$

This equation represents the probability distribution for a package to ship from Facility 1 to Facility 2, and Facility 2 to the next step. Both $F_1(x)$ and $F_2(x)$ are probability density functions indicating how long it is likely to take to ship from one facility to another, where x is the number of transit hours. In this case, *x* is the number of hours a package takes to ship between the two Facilities. An example probability density function would be:

$$F_1(x) = \{0.3\ if\ x = 5, 0.2\ if\ x = 2, 0.5\ if\ x = 10\}$$

In the example function, a package ships between these two facilities in 2 hours 20% of the time, in 5 hours 30% of the time, and in 10 hours 50% of the time.

Predicting a packages shipment time distribution is not as straightforward as the previous examples, however, for two reasons. First, a package can take several paths to ship through the US Postal Network. Therefore, the time a package takes to ship from start to end can be modeled as the distribution of shipping times from start to end for each possible route, weighted by how likely that route is to be taken.

$$P(T = t) = P(R_1) + P(R_2) + \ldots + P(R_3)$$
$$Where\ R_{1\ldots N} = \sum_{h=0}^{\infty} F_1(h) * F_2(t-h)$$

This equation represents the probability distribution of a packages' shipment time duration, given that there are N possible routes, each with its own unique probability distribution of shipment times from the start to end of the route. Each route also has a probability percentage representing how likely that route is to be taken. Consider an example visualized in Figure 1.

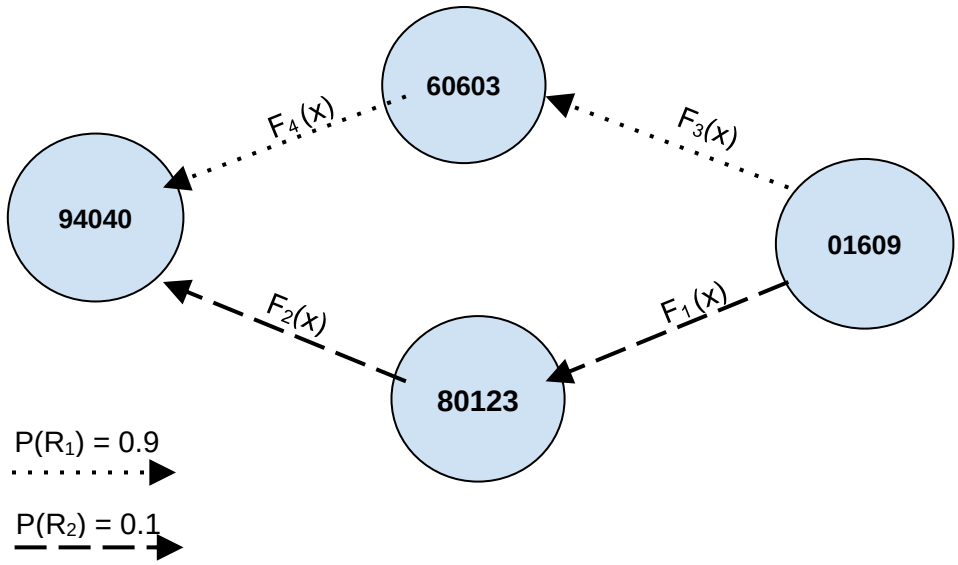

Figure 1: Visualization of example shipping route

In this example, a package shipping from Worcester, Massachusetts (zip code 01609) to Mountain View, California (zip code 94040) takes one of two routes. It will travel through Chicago, Illinois (zip code 60603) 90% of the time based on historical statistical calculation and through Denver, Colorado (zip code 80123) 10% of the time. The time between each facility is modelled by the probability distribution $F_N(x)$. In this case the shipping time distribution would be calculated by the equation:

$$P(T = t) = (0.9 * (\sum_{h=0}^{\infty} F_3(h) * F_4(t-h))) + (0.1 * (\sum_{h=0}^{\infty} F_1(h) * F_2(t-h)))$$

It might seem like these calculations account for all possible paths a package will take to ship through the USPS network and provide a realistic distribution. However, it neglects one critical variable. The hour of day a package arrives at a distribution center or post office can impact the arrival time significantly because some offices only ship at certain times of day. For example, if an RDC has a ship-out time of 4PM but a package did not arrive at the facility until 4:15PM, the package will probably have to wait up to 24 hours before it can continue along its delivery route. Therefore, the equation must account for the shipping time between two facilities given a certain arrival hour. To account for this, the prototype approximates a continuous distribution of arrival time with 24 discrete distributions shipping time distributions depending

on hour of arrival time. When computing $P(T = t)$, the shipping time distributions between two nodes, $F_N(x)$, the time to ship between two facilities, is different depending on the expected arrival time at the start facility. Each arrival hour distribution is then weighted by how likely a package is to arrive at that hour. Consider one route from the earlier example, adjusted to account for the arrival hour variable, shown in Figure 2.

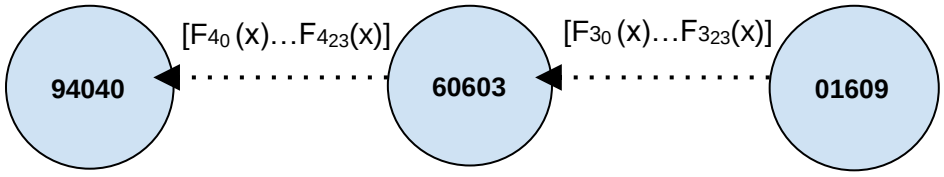

Figure 2: Visualization of more detailed example shipping route

If a package arrived in Worcester (01609) at 8:00AM, what's the shipping time distribution for when it arrives in Mountain View (94040)?

$$\text{If } F_{3-8}(x) = \{0.5\ if\ x = 5, 0.5\ if\ x = 2\}$$

The answer would weigh the distributions for package arriving in Denver at 10AM (8AM + 2 hours) and 1PM (8AM + 5 hours) equally and be represented as,

$$P(T = t) = (0.5 * (\sum_{h=0}^{\infty} F_{4-13}(x))) + (0.5 * (\sum_{x=0}^{\infty} F_{4-10}(x)))$$

This calculation is then done for each possible route, and possible arrival hour at a facility, to get a shipping time distribution. This accounts for both data scarcity and arrival time cutoffs by using delivery network routing. Another benefit is that it accounts for outlier scenarios by weighting them with the same probability of them occurring as they have actually occurred, based on historical data. Thus, the predictions for total shipping time are more accurate.

In implementation, the current prototype has a basic webpage projecting a map, the route input by the customer, and a shipping time distribution histogram on output. This is less important for the working of the project but influential to present to stakeholders to show the non-technical overview of what is going on behind-the-scenes.

2.3 Technology Background

This sections describes the technology and tools used to develop the **new** prototype as shown in Figure 4 in Section 5.1.1. These include SQL Server, Apache Spark, Neo4j, REST API, and Apache CXF and Embedded Jetty. The prototype was tested using JUnit testing, load testing, Kolmogorov Smirnov Test, and cross validation testing and deployed using Apache Maven.

2.3.1 Data Processing Tools

USPS events are stored in a separate SQL server database by Endicia. USPS parcel scans from USPS's database are synchronized with Endicia's database multiple times a day in a format that is not immediately usable by Neo4j. In order to convert tracking events into sensible aggregated information for Neo4j, the prototype uses Apache Spark.

Apache Spark

Apache Spark is an open-sourced tool "built around speed, ease of use, and sophisticated analytics." (Apache Spark, 2017). It is a data-processing engine that is compatible with big data platforms to prepare massive amounts of data in parallel for development and analysis by businesses and engineers. It has a Hadoop infrastructure, which gives it the ability to access data in a Hadoop Distributed File System (HDFS). It also has built-in libraries for database connectivity, machine learning, and graph computation.

Endicia's previous predictive shipping prototype could only calculate roughly 300 million rows of data in a 24 hour time period with an excessive amount of overhead. It was inefficient at communicating with the database with numerous unnecessary open and close connections and there was a memory problem that forced the prototype to store temporary tables in the SQL server. The advantage of using Spark to process the USPS data is that it can transform and reduce millions of rows of data in parallel by "exploiting in memory computing and other optimizations" (Apache Spark, 2017). It can run programs up to "100x faster than Hadoop MapReduce for large scale data processing" (Xin, 2014).

In addition, the core Spark API has an extension called Spark Streaming which enables "scalable, high-throughput, fault tolerant stream processing of live data streams"(Apache Spark,

2017). Spark Streaming will allow Endicia to implement live updates to the prototype as new data is uploaded into their USPS database.

2.3.2 Datastore Tools

Neo4j was the database used for the prototype because of its ability to naturally model the USPS shipping network and meet performance requirements. Neo Fast Import Tool was used to import million rows of processed data by Spark.

Neo4j

Neo4j is an open source graph database. Instead of storing data as a tabular data structure in a relational database, a graph database stores nodes and relationships. Both the nodes and the relationships can be built as custom objects with varying amounts of complexity. Neo4j has built in drivers to connect and communicate with different languages and frameworks, including Java. Figure 3 shows the representation of a Neo4j graph. Circles are nodes and arrows are relationships. Nodes and relationships both can contain additional information in a key-value structure. An example would be the ACTED_IN relationship between Tom Hanks and The Da Vinci Code nodes contains a property Role:Dr. Robert Langdon.

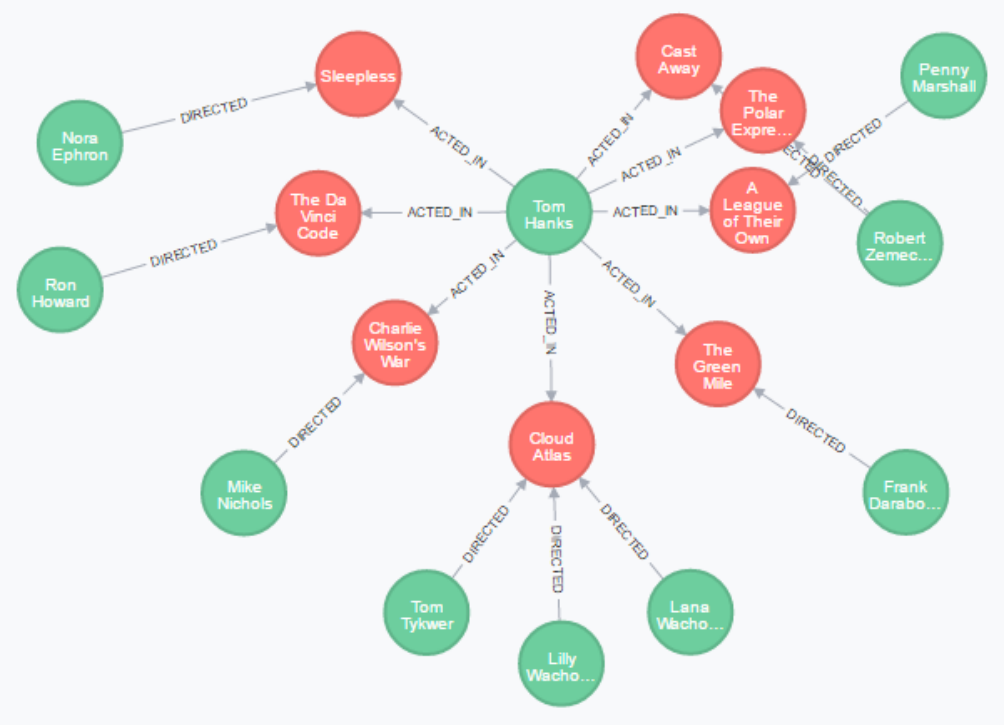

Figure 3: Example of Neo4j graph database representation

USPS packages travel in a path from facility to facility. Neo4j leverages this by being able to represent these paths as relationships between facilities. The databases' ability to handle large amounts of data in both quantity and speed also makes it extremely advantageous. In 2015, 4.5 billion packages were shipped and tracked through USPS. With Neo4j, it is possible to efficiently and effectively import and query our requirements of 75,000 Post Office Facilities, and 2.5 million relationships.

Neo4j Fast Import Tool

Neo4j Fast Import Tool is a feature of Neo4j that allows a new database to be populated with data in CSV files (neo4j.com, 2017). The tool can import data in extremely fast amounts of time. For example, it can import 100 million Stack Overflow questions in under three minutes with nodes for posts and users and relationships for answers, tags, and users. A main reason why it can achieve such high upload speeds is because the tool imports into an empty database, which guarantees that it does not need to overwrite or combine any existing nodes or relationships, and can save all of the information in one commit (neo4j.com, 2017).

2.3.3 HTTP Server and Business Layer Tools

The HTTP server of the prototype was a RESTful API that could allow customers and other web services to access information about shipping predictions. The business layer of the prototype received the request, handled the computation, and returned the output to the server, which is then presented as a response to the user. This section describes the tools used to develop the HTTP server and business layer.

Apache CXF and Embedded Jetty

The HTTP server was developed using Apache CXF, a web service framework, and embedded Jetty, an HTTP server. Apache CXF is an open source services framework that allows developers to create services that communicate over the most common web protocols and transport protocols. These protocols include Representational State Transfer (REST), Simple Object Access Protocol (SOAP), Extensible Markup Language (XML), and Hypertext Transfer Protocol (HTTP) (Apache, 2017). CFX directly integrates Java web services libraries and APIs and is licensed under Apache Software Foundation (Apache, 2017).

With CXF handling the communication, Jetty provides the server framework for CXF and allows the web service to communicate with users or other services. Like CXF, Jetty is Java under-the-hood and is easily associable with CXF. Jetty creates and sustains the server instance used for the prototype Embedded Jetty also means that the prototype can be deployed as a standalone WAR file without an additional web server. It also integrates easily with common production web servers such as Apache or Nginx.

REST and REST APIs

In order to create a system that would be easily accessible, the HTTP server of the prototype was developed as a RESTful API. REST "is an idealized model of the interactions within an overall web-application" (Jakl, 2008). The REST protocol enables caching and reuse of interactions, components, and other useful information making it ideal for distributed systems (Jakl, 2008). The key concept of a REST interaction is that each communication contains every piece of information necessary for the receiver of the message to provide a complete and correct response back to the person requesting resources REST relies on different methods of

communication such as GET, PUT, POST, and DELETE requests which allows developers to build API's around (Jakl, 2008).

A RESTful API is a set of actions based on resource path and REST method that can provide any number of services. REST APIs can be both internal or external from engineering teams within a company to developers hoping to take advantage of open source projects. This benefit will give Endicia's employees access the prototype and receive shipping predictions without having to extensively understand the source code.

2.3.4 Testing and Validation Tools

JUnit

JUnit is a Java-based, simple framework for developing unit tests and testing suites for Java projects (JUnit, 2017). In the Maven architecture, any JUnit test files located at a specific file path in the project will be run every time the project is built. The Maven build will fail if any of the JUnit tests included in the specific directory fail. This feature allowed us to build regression tests directly into our project as we developed different systems and assure that each time we built the project with updates that it was still successfully passing the tests we had written for the last build.

JUnit provides many possible testing tools and outcomes. Not only can it test for boolean values but it can include expected outcomes such as purposely throwing exceptions or breaking your code.

Load Testing

Load testing provides assurance that all of the components interacted dependably and the system could handle the amount of requests it could potentially see in production, which was also one of the technical requirements. Load testing is a black-box style of testing, meaning there is no need to access to the source code to effectively load test an application or platform (Beizer, 1995). The purpose of load testing, which parallels the purpose we used this type of testing, "involves applying ordinary stress to a software application or IT system to see if it can perform as intended under normal conditions" (SmartBear, 2017).

Kolmogorov Smirnov Test

The Kolmogorov-Smirnov Test, K-S Test, is used to test for distributional accuracy between two distributions. The K-S test can determine whether a sample "comes from a population with a specific distribution" (Nist.gov, 2012). To test the accuracy of the prototype, it could be used to determine if the prediction from the system comes from a population with the tested distribution. The K-S test does not depend on underlying cumulative distribution function being tested and can be used to find differences in both continuous and discrete distributions (Nist.gov, 2012). For validating the prototype, both of these features are necessary because the testing distributions are made from the data, function is not known, and the prototype treats each shipment hour as a discrete possibility.

K-Fold Cross Validation Testing

Cross Validation is a model evaluation method used to test predictions based on a set of data (Schneider, 1997). Cross validation generally involves using a subset of the total data to train a model on and using the remaining set of data to validate the accuracy of the predictions made by the model. K-fold is a type of cross validation that involves dividing the total data in *k* subsets. The testing is then run *k* total times each time using the one of the *k* partitions as the subset that gets validated after training the model on the total set of data without the *k* partition used in that round of testing (Schneider, 1997). This testing approach becomes increasingly more accurate with the size of the *k* partition.

2.3.5 Deployment Tools

Apache Maven was the tool used to deploy each component of the prototype. This section describes basic information about Maven.

Apache Maven

Apache Maven is a software project management and comprehension tool for Java-based projects (Apache, 2017). It is a project framework that allows the centralization of every part of the project. A Maven project is centered around a project object model (POM) which specifies

build characteristics, project dependencies, and important project logistical information among other things. Integrating in a new plugin or dependency can be done by adding the dependency to the POM file and it will be dynamically installed when the project is built. A maven build encompasses the whole software development lifecycle. It initially compiles the source code, runs the integration and unit tests specified by the developers, and packages the project for deployment.

2.3.6 Apache JMeter

JMeter is an open source, fully java application used to load test and measure performance of web servers (Halili, 2008). It gives a developer the ability to automate testing while being able to measure resources such as CPU load, memory usage, and response times (Halili, 2008). The application is so widely used it developed a community around it and now has many plugins and extensible features for thorough testing. With JMeter, the user can create and store "Test Plans" to recursively run the same testing conditions or access the cached results after running a Plan. With JMeter's plugins, the results from the testing plans can be sent directly into graphs and visual representations of the testing. The prototype was tested after concluding implementation to measure response times for calculations as well as testing with variable amounts of requests per time interval.

3 Requirements

Shipping delivery estimations are currently calculated using basic averages in accrued, historical statistics. While USPS gives delivery date guarantees for some classes of parcels, it does not offer any for less expensive classes. Endicia currently provides delivery estimates taken directly from the date estimated by USPS using the historical averages. Illuminating delivery times for first class parcels will provide greater value to Endicia's customers.

3.1 Business Requirements

For this project, there are various business goals which must be considered in design and implementation of the prototype to satisfy non-technical stakeholders.

The first business requirement was to mathematically validate the shipping time distributions the prototype. The ultimate goal of our prototype was to offer the results of the system's calculations as a service to customers. In order to offer this as a product, the delivery time distributions produced by the prototype had to be accurate and thoroughly tested and validated.

The second business requirement was to show a visualization of the data being calculated, including a histogram of potential arrival times and an overlay of the paths a package might take on a map. To satisfy this requirement an interactive webpage was built for the purpose of demonstration.

3.2 Technical Requirements

Endicia ultimately wanted to offer shipping delivery date predictions as a web service to their business customers. In light of this goal, there were technical requirements and restraints to meet this goal while using Endicia's local machine Dell Latitude E6430 with 8.00 gigabytes of Random Access Memory (RAM) and an Intel Core i5 central processing unit (CPU) at 2.70 Gigahertz (GHz) and an Intel Core i7 CPU at 2.90 GHz running a Windows 10 Operating System.

Firstly, Endicia wanted our prototype to surpass the performance of the previous prototype. The previous prototype processed 1.2 billion rows in 24 hours. However, Endicia wanted to process this quantity of data in under an hour.

Because Endicia wanted a prototype that met production standards, it was a critical design requirement to consider load capacity. Not only would the prototype have to work quickly but it would have to withstand a high volume of requests. Endicia set the goal for each request taking around 50 milliseconds each. With this performance goal, the server would be able to handle about 20 requests per second on average.

A final technical requirement was to create a RESTful API to be able to interface the prototype with current Endicia platforms or to offer the API to customers who already interfaced their applications with Endicia's other APIs.

4 Methodology

The goal of our project was to produce a validated prototype that forecasts delivery time distributions to optimize shipping costs. Our methodology was divided into three objectives, (1) building a new, extensible prototype that can handle big data, (2) optimizing the prototype in performance and capabilities in order to meet Endicia production standards, and (3) validating the probabilistic approach used to predict shipping time distributions. This chapter details to our methodology.

4.1 Design and Implement a Predictive Shipping Prototype

The first objective was to design and implement a prototype that could handle billions rows of data, process and load those rows into another database, and query the database for shipping information. Our prototype was a full-stack system that encompassed several components, which were divided into (1) Endicia's Virtual Post Office (VPO) database, (2) Spark engine, (3) Neo4j database, (4) calculator and business layer, (5) HTTP server, and (6) front end user interface.

The HTTP server, business layer, and Spark engine were written in Java and built with Apache Maven. The design goals of the system included modularity, extensibility, and simplicity. Designing and implementing the prototype with these goals allowed for easy optimization and integration for deployment.

4.2 Update and Optimize the Prototype

The second objective was to optimize the prototype to meet Endicia production standards. In order to do this, we used the computing power of Spark and uploading speed of Neo4j. These tools are specifically designed to handle millions of rows of data. To completely optimize the upload into Neo4j, we needed to ensure that each row of data was unique so Neo4j was not wasting time checking if a relationship existed, updating it if did, or creating it if it did not. Optimizing the system also required some research into how a Neo4j query called the database, as well as appropriate (but not unnecessary) pruning of improbable events, such as outlier packages stopping at many facilities between two zip codes.

4.3 Validate the Predictive Shipping Mathematical Model

The third objective was to validate the mathematical approach used to produce the shipping time distributions. There were many variables that can be taken into consideration when calculating the time distributions. Our prototype took into consideration five variables which are (1) parcel's origin location, (2) parcel's end delivery destination, (3) the parcel's USPS Mail Class designation, (4) the day of the week which the package was inducted into the USPS system, and (5) the hour of the day which the package was shipped. Other possible variables include seasonality, and weekend delivery, among others.

After researching and designing the predictive mathematical approach we heavily tested and validated the algorithm by using a spectrum of testing methodologies in both calculated and real-world situations. The most extensive testing philosophy we used was regression testing. Regression testing is effectively used to "test a modified program to gain confidence that recent changes have not adversely affected existing features" (Wong et al, 1997). Regression testing provides not only validation but a baseline for future optimizations or changes to the system. This testing approach provided us with the most useful results towards the validation of our model because we had the ability to run our calculations against millions of parcels in the USPS database whose robust data contained the information we attempted to compute. The real-world data provided a strong foundation to promote our prototype upon, depending on the testing results with this data.

5 Design and Implementation

This chapter contains implementation details that describe how the methodology was completed. These include specifics about the architecture, algorithm, and testing and validation of the new prototype.

5.1 Architecture

This section focuses on the complete architecture of the prototype, then proceeds into the details of each component of the prototype including the system, HTTP server, database, algorithm that calculated the time distributions of the shipping predictions, and Spark engine.

5.1.1 System Design

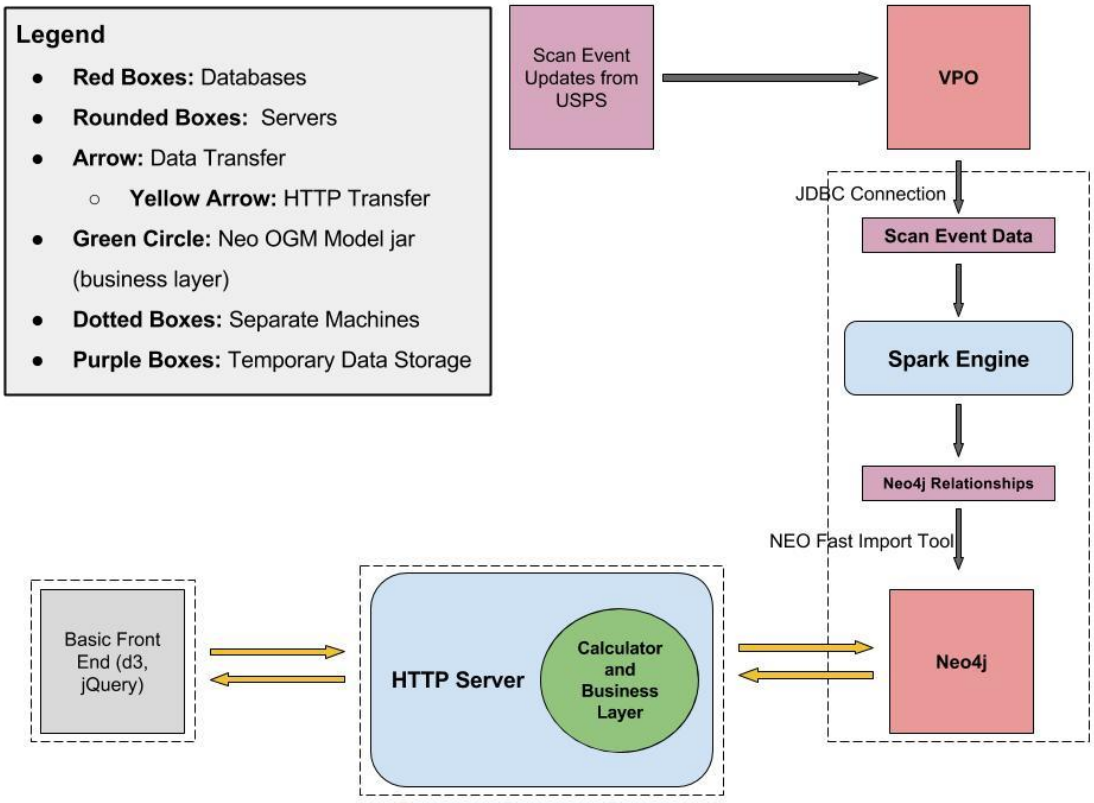

Figure 4: System architecture for prototype

As shown in Figure 4, the new prototype is divided into several main components. There are a VPO database, Spark engine, Neo4j database, calculator and business layer, HTTP server, and front end UI. The frontend UI, HTTP server, and Neo4j communicate through HTTP. The VPO SQL server is populated with scan event data from USPS. The Spark engine uses JDBC connections to pull data from the VPO in the form of CSV files. Spark processes the scan event data and returns Neo4j relationships that can be directly imported into Neo4j with the Neo Fast Import Tool.

5.1.2 VPO Query Design

The rows of scan event data in VPO were queried by joining four tables to accumulate data about the package's tracking id, date and time the package arrived at the facility, the facility's five zip code, the facility's three zip code, the package's mail class, the package's final destination five zip, and the scan event code. For our prototype, the data was only from packages shipped via First Class. Scan events with code 748, 745, and 1080 were not included because they were respectively "Electronic Shipping Info Received", "Delivery Status Not Updated", and "Shipping Label Created" and did not contribute to transit time of the package. Only tracking ids that contained the delivery event code, 760, were used. Facility zip codes that included APO, FPO, and DPO were not included. All scan events were ordered by tracking id and date time. Table 1 shows a sample row of data that the SQL query returns, which were then imported into a CSV file for Spark to read.

Table 1: Example of one row of Scan Event data from VPO

Tracking Id	2660000005
Event Date and Time	2015-03-12 22:27:00
Facility Zip Code (5 digit)	04742
Facility Zip Code (3 digit)	047
Mail Class	First Class
End Destination Area Code (3 digit)	04345
Scan Event Code	750

5.1.3 Spark Engine Design

The Spark engine was divided into four parts: (1) Initial setup, (2) Round one, (3) Round two, and (3) Final rows.

Table 2: A final row produced by the Spark engine

Start Id	96820
End Id	66106
From Facility (5 digit)	96820
To Facility (5 digit)	66106
List of End Destinations	[967, 645, 667, 648, 646, 653, 650]
List of End Destination Occurrences	[2, 7, 3, 5, 10, 3, 6]
List of Time Distributions based on Arrival Hour	{125=1}:{}:{77=1}:{49=2|48=2|75=1|77=2}:{49=6|48=9|46=9|45=1|52=2}:{}:
Type of Neo4j Relationship	ShipsTo

Table 2 shows a row of a Neo4j relationship that Spark produces using the scan event data in Table 1. Each row contains the start node id, end node id, from-facility five zip code, to-facility five zip code, list of final end destination three zip codes, the corresponding number of occurrences per three zip code, list of time distributions and their respective occurrences indexed by arrival time at the from facility, and ShipsTo relationship type specifically for Neo4j. For example, as seen in Table 2, {125=1} is the first group in the list, which means a package that arrived at the facility in 96820 at midnight (0th hour) took 125 hours to travel from 96820 to 66106, and that occurred once from the data processed. These rows are unique per every from-facility and to-facility pair and can be directly loaded into Neo4j using the Neo Fast Import Tool.

The initial setup involved converting local arrival times of the tracking events into Coordinated Universal Time (UTC) so that it was possible to compute the transit time between consecutive facilities. Spark loaded in a CSV containing the zip code and its UTC offset, as seen in Table 3, and iteratively put them into a HashMap. Five digit zip codes 68701 and 68776 belong in the same three digit zip code 687 general zone. Therefore, it was possible to assume

that 68701 and 68776 were in the same time zone. However, as shown in Table 3, some general zones, like 690, were split between two time zones and those were stored as five digit zip codes in order to distinguish the difference. As a result, the total number of key values needed to be stored was reduced to around 1000 values instead of approximately 43,000 five digit zip codes.

Table 3: Example of zip codes and their UTC offset

Zip Code	UTC Offset
687	-6
688	-6
689	-6
69001	-6
69020	-6
69021	-7
69022	-6
69023	-7

Spark requires that each row of data, or tuple, be independent of the others. Once zipcodes and offsets were accessible, it was necessary to group all events by their tracking id in order of when each of its scan event occurred. Together they hold information about the package's path that would not be apparent if the scan events were viewed independently.

The scan event data in Table 2 was converted into tuples with tracking id as the key value and an ordered list of ScanInfo objects that contained the event date time, facility 5-zip, facility 3-zip, mail class, destination 3-zip, and event code per scan event. For example, if tracking id had six scan events, those six rows of data would be converted into ('2660000005', [$ScanInfo_1$,...,$ScanInfo_6$]).

From there, each tuple's list of ScanInfo objects created GoesTo and TimeRecord objects by iteratively by taking the current ScanInfo and succeeding ScanInfo. GoesTo contained a from-facility 5-zip, to-facility 5-zip, and final end destination 3-zip. TimeRecord contained a from-

facility 5-zip, to-facility 5-zip, arrival hour based off the event's data time, and transit time in hours from the from-facility to the to-facility.

Once the data was converted into these independent tuples, Round One and Round Two were the stages where VPO scan event data was transformed and reduced. The local machine could only handle around two million rows of VPO data before it ran into an out of memory heap space problem. To counter this problem, it was necessary to perform the transformations and reductions in two rounds.

Table 4: Round One and Round Two output for GoesTo by Spark

From Facility	To Facility	End Destination	Occurrences
33480	33054	374	1
92025	92029	859	1
99224	98903	989	56
03063	15086	256	1
92199	92121	953	1

Table 5: Round One and Round Two output for TimeRecord by Spark

From Facility	To Facility	Arrival Hour	Transit Time	Occurrence
77315	67276	6	93	1
07699	33605	2	45	4
32824	33054	23	39	2
89510	89701	9	3	2
90052	02904	2	94	1

In Round One, Spark would read in a CSV file with scan event data as shown in Table 1, that contained approximately two million rows. Then, Spark would map the GoesTo object to a key-pair tuple where the key was (fromFacility, toFacility, endDest) and the value was 1 for the initial occurrence. Once all GoesTo objects were mapped, Spark would reduce the tuples by key and write them to a CSV as shown in Table 4 (99224, 98903, 989, 56) indicates that the key (99224, 98903, 989) was seen 56 times in the data given by a CSV file. The same method was

used for TimeRecord objects but the key was (fromFacility, toFacility, arrivalHour, transitTime) as shown in Table 5. Each CSV file produced approximately 275,000 rows of GoesTo reductions and 380,000 rows of TimeRecord reductions. One group contained 31 CSV files, and this group's results, as partially shown in Table 4 and Table 5 would later be used in Round Two for another reduction with other groups to further reduce the number of total rows.

Reducing the rows allowed for two things: (1) aggregation of occurrences for a particular path, and (2) condensation of total rows of data per CSV file. As stated earlier, there was a memory limit to how much data the local machine could handle at a time so reducing the rows again meant overall Spark could process more VPO data. Round Two would take the 8,525,000 rows of a Round One GoesTo group and reduce it to about 3,193,000 rows. 11,780,000 rows of a Round One TimeRecord group would be reduced to 4,712,000.

Once Round Two was complete, Spark would read in all Round Two groups, GoesTo and TimeRecord. These tuples were joined by a (fromFacility, toFacility) key and mapped to the final rows like Table 2, which were readily usable by Neo4j.

5.1.4 Database Design

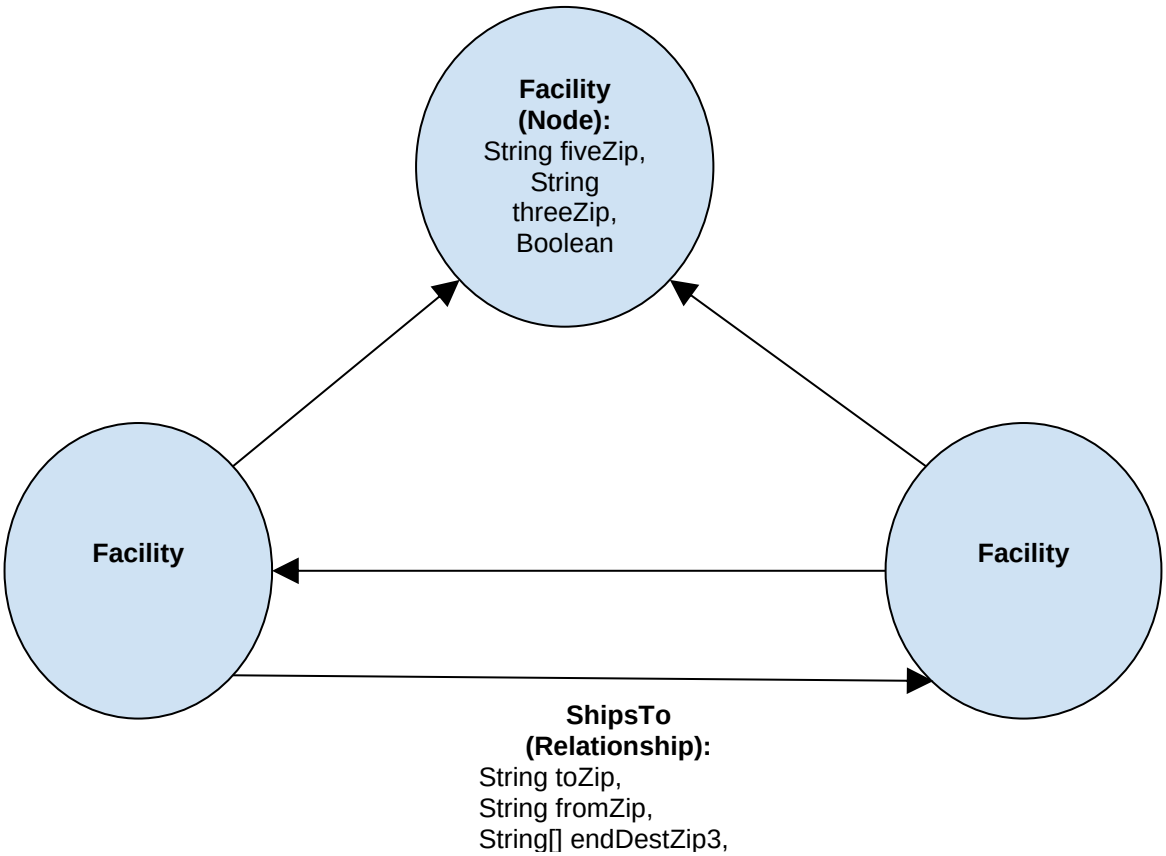

Figure 5: Neo4j Schema Model

Once the CSV files were processed by Spark, the rows of data were loaded into Neo4j using the Neo Fast Import Tool. As shown in Figure 5, in the Neo4j schema, nodes are Facilities, each representing a post office or distribution center. A facility contains its five digit zip code, three digit zip code, and a boolean for whether or not it has been visited. Five digit zip codes are unique for each facility in the USPS network and can be used as a primary key. The visited boolean is used in shipping probability calculation to remove graph cycles. To increase query efficiency, an index on the facility's five digit zip code was added to the database. Aside from the facility zip code information, all other information need for shipping time prediction was stored on the relationships connecting facilities. Each relationship, called ShipsTo relationships, provided a direct connection between two Facility nodes. They each have a five digit string for its from-zip code, five digit string for its to-zip code, an array of three digit zip code strings that each represent the final end destination zip codes a package has traveled with that corresponding

from-zip and to-zip facility zip codes, an array of integers of occurrences that directly correspond to the end destinations in the previous array, and a string of transit time occurrences indexed by arrival hour at the from-facility. The importance of having this data for each arrival hour was discussed in Section 2.2.2. The data needed to be saved as a string because Neo4j did not support an array containing anything other than primitive data types. Figure 6 and Figure 7 respectively show the information stored in a Facility node and a ShipsTo relationship in Neo4j. As explained in Section 5.1.3, in Figure 7, the listOfProbs signifies that based on historical data, one package traveling from node 01609 to node 01605 with final end destination of 016 arrived at node 01609 at the 22nd hour of the day and took 65 hours to travel.

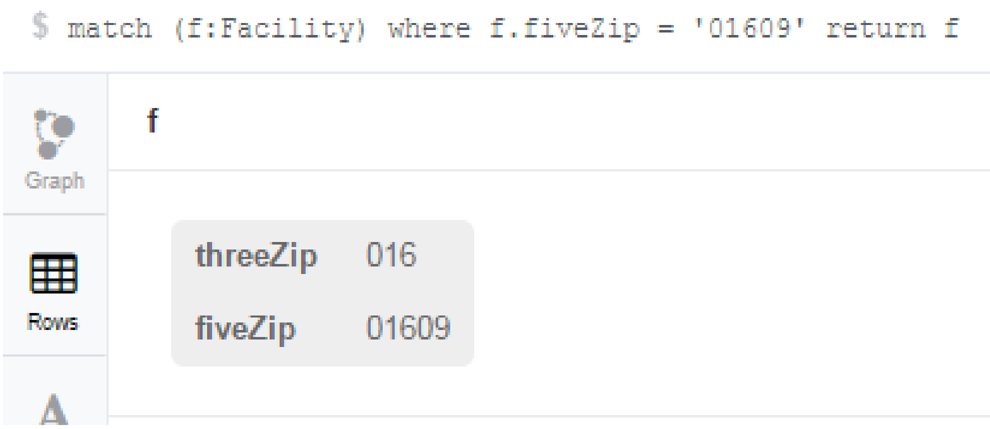

Figure 6: Facility node in Neo4j

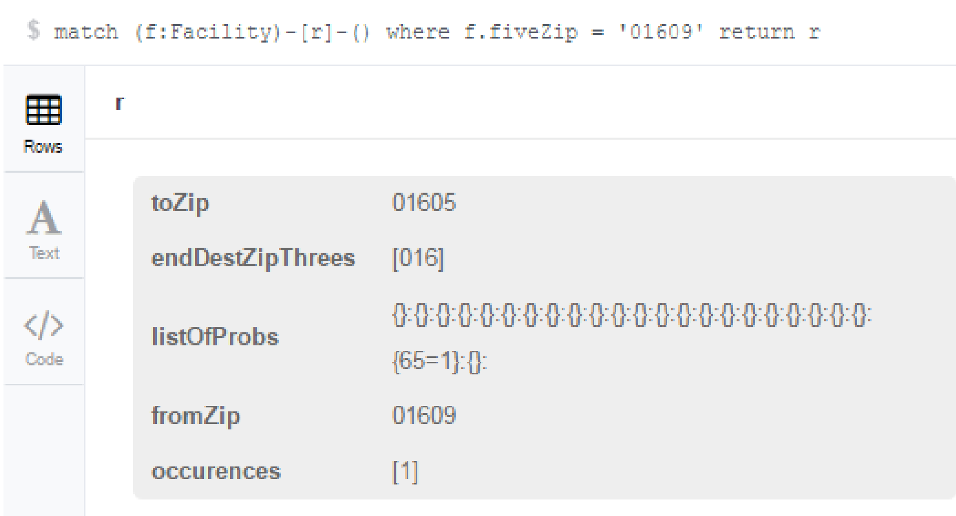

Figure 7: A simple ShipsTo relationship in Neo4j

5.1.5 Business Layer

The business layer of the prototype had two purposes: (1) querying for path information regarding specific Facility nodes and ShipsTo relationships from Neo4j in a hashmap, and (2) calculating time distributions using data stored in the hashmap and the mathematical formulas described in Section 2.2.2. The business layer was packaged into a .jar dependency and imported by the HTTP Server to fetch time distributions based on user input.

The first purpose of the business layer was to be to connect with Neo4j and pull data based on the user input from the HTTP server. To do this, Neo OGM, an object graph mapping library, was used. In order to calculate the time distribution, all Facility nodes and ShipsTo relationships connecting the drop-off start zip code to the final end destination zip code were queried and stored them in a *Graph* object.

The first version of the business layer pulled down routes, the relationship and node connections between zip codes, of any size in one Neo4j query. However, with only a subset of about 70,000 Facility nodes and 300,000 ShipsTo relationships, this query ran out of memory on the heap to store the results. When the heap size was increased, the query took over a minute to finish, which was unacceptable based on the Technical Requirements defined in Section 3.2. The first step in optimizing the time and memory used to find routes in Neo4j was limiting the length of potential routes Neo4j would return. To find an appropriate maximum query length, we examined how many stops, or jumps, a each package made when traveling through the USPS network. Figure 8 shows a histogram of the number of jumps a package made on four months of USPS data. Based on the data, 99% of packages are delivered within 7 stops or less. However, there were outlier packages that traveled to more than 7 facilities. A maximum of 0.005% of packages could be disregarded from the path search. This corresponded to paths of up to a length of 11 stops. Therefore, the Neo4j query only looked for paths with lengths no more than 11. Despite these optimizations, the query still took 12-15 seconds to return results, which was considered too slow. In addition, running the updated query on the a fully populated database of 70,000 Facility nodes and 3,500,000 ShipsTo relationships resulted in the same memory and speed problems discussed earlier.

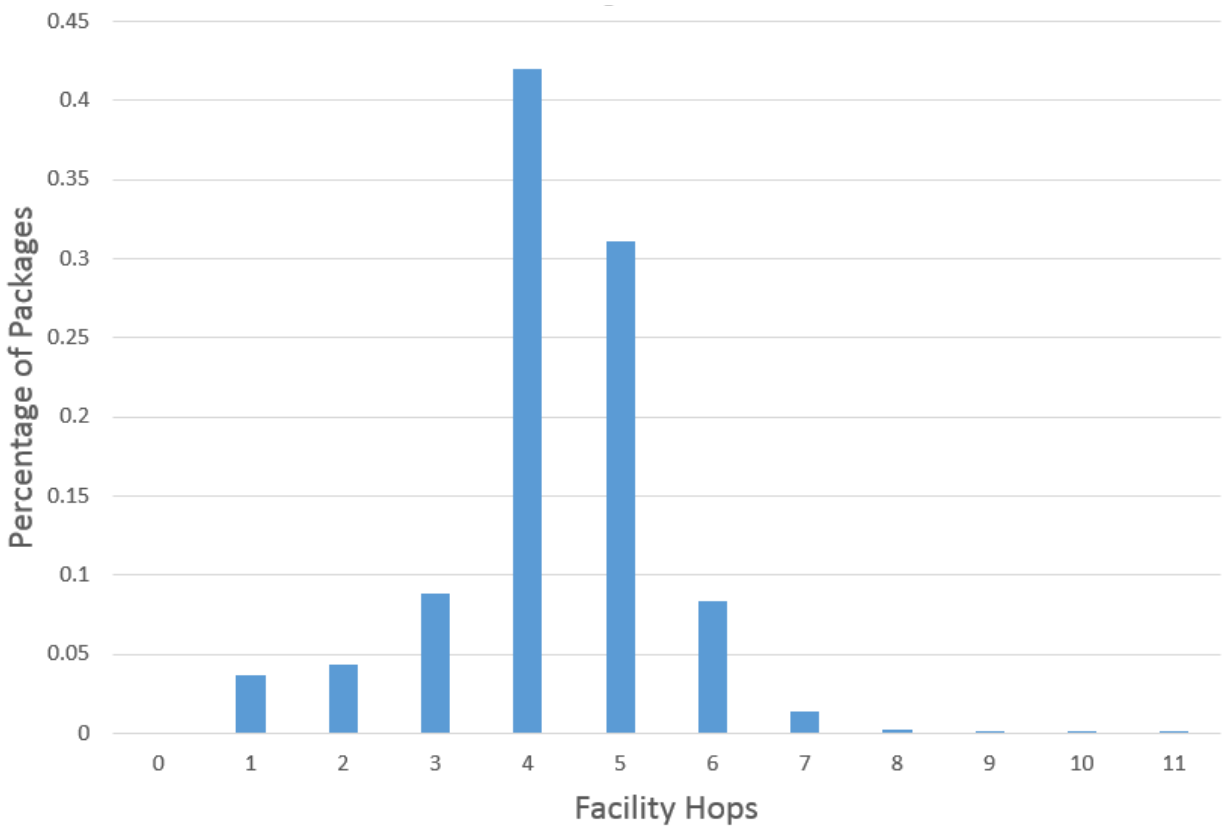

Figure 8: A histogram of facilities a package stops at

To resolve the issues, we used a profile tool to determine the bottlenecks in the Neo4j query. Figure 9 shows the breakdown of the Neo4j query using the profile tool.

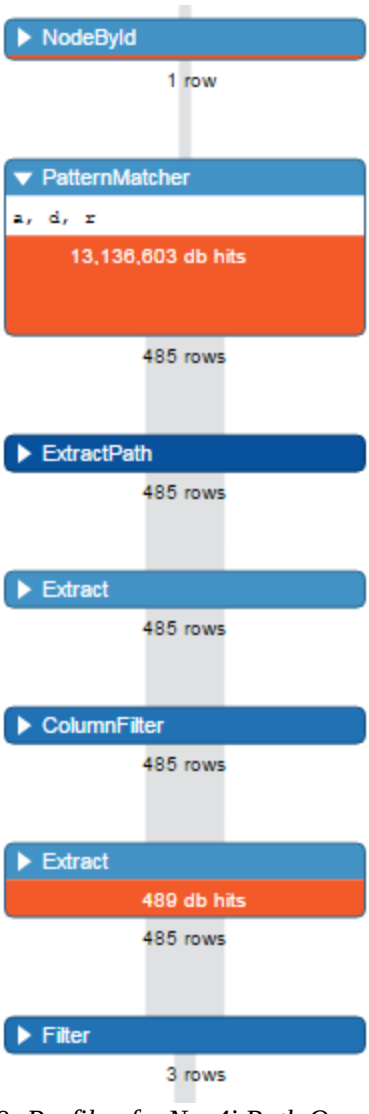

Figure 9: Profile of a Neo4j Path Query

The profiler shows that the PatternMatcher, which gets the routes between the initial start Facility node and final end destination Facility node, hits the database the most. The query extracted 485 rows (nodes and relationships combined) but only returned 3 rows after filtering. This was because the query was designed to get all routes between start and end nodes before filtering these routes to return only those shipping to the specified end destination zip three region. If the query were to filter out routes as it searched during the PatternMatcher step, instead of after, it could eliminate a huge portion of the database hits, which in query shown in Figure 9 was over 99% of them. The final version of the Neo4j query filtered paths as it searched, improving the speed of the query to return in < 1 second on most queries.

Once the query returned the desired paths, it stored the ShipsTo relationships and Facility nodes in a *Graph* object, which was then used by the *TripCalculator* to perform the convolution math described in Section 2.2.2. *TripCalculator's* most important methods were *pathSearch()* and *timeSearch()*, which performed the convolution for the next facility a package would jump to and the hour it would arrive, respectively. Pseudocode for these functions was as follows:

pathSearch Method

```
pathSearch(Facility currentFacility, int arrivalHour, double probability,
           int transitHours){
    ...
    if (currentFacility is endFacility)
        addTransitHoursAndProbabilityToAnswer(transitHours, probability);

    else
        for (Path nextPath in  currentFacility.getNextPaths()){
            probability_next_step = occurrences_of_nextPath / total_occurrences;
            timeSearch(nextPath, arrivalHour,
                      probability_next_step * probability, transitHours);
        }
}
```

timeSearch Method

```
timeSearch(Path nextPath, int arrivalHour, double probability,
           int transitHours){
    ...
    get_time_distribution_for_arrival_hour();

    for (int shipTime in time_distribution_fo_ arrival_hour){
        probability_arrival_hour = occurrences_of_shipTime / total_occurrences;
        pathSearch(nextPath.toFacility(), (arrivalHour + shipTime) % 24),
                   Probability * probability_arrival_hour,
                   transitHours + shipTime);
    }
}
```

These functions recursively called each other, accounting for every possible path at every possible arrival hour of a package and weighting the probability appropriately at each step. Only when *pathSearch()* reached the *endFacility* of the *Graph* was the probability and transit hours added to the distribution that was sent to the HTTP server. While the pseudocode covered the essential logic in calculating shipping time probabilities, for simplicity's sake, it omitted the steps to determine whether a graph is in a cycle. In that case, the probability of a cycle was added to

an “unknown time” field. The pseudocode also omitted cases where no time distribution exists for packages arriving at a particular arrival hour. In this case, the adjacent arrival hour distributions was combined and used instead. These techniques were used to account for rare edge cases and data scarcity issues. Once the time distributions were calculated by the business model, they were returned to the HTTP server.

5.1.6 HTTP Server and API Design

The HTTP server of the predictive shipping prototype was built using the RESTful architectural style. The server instance was run through Embedded Jetty and overlaid with Apache CXF to handle RESTful web services.

The server’s base Uniform Resource Locator (URL) was hosted at

```
http://localhost:{port number}/predict/v1/
```

and users could access the prototype’s frontend, demo User Interface here as well.

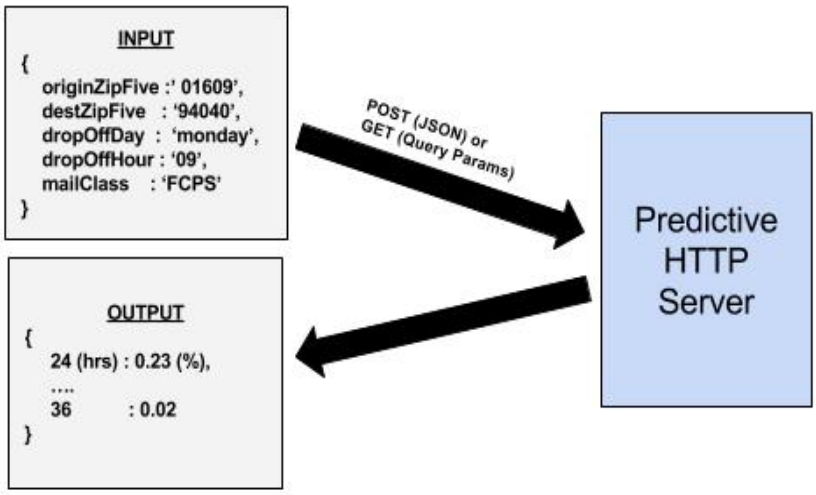

Figure 10: Diagram of a request to and a response from HTTP server

The server allowed users to make requests to the prototype at a GET endpoint and a POST endpoint. As shown in Figure 10, the server parsed query parameters for GET requests and JSON objects for POST requests. Once the request was processed in the business layer, the

server returned a JSON object with the delivery hours to probability percentage distributions. Users made requests at the following URL:

```
http://localhost:{port number}/predict/v1/query
```

1. `POST` request with parameters required by the API, formatted in a JSON object
2. `GET` request with parameters required by the API, formatted as query parameters in the URL

Table 6 shows the parameter names, values, and descriptions for the GET and POST requests. Example requests and user documentation can be found in Appendix A.

Table 6: Parameters of JSON object for POST

Parameter Name	Parameter Type	Restrictions/ Requirements	Description
originZipFive	String	Five numerical digits	Five digit zip code of origin facility
destZipFive	String	Five numerical digits	Five digit zip code of end destination
dropOffDay	String	Full name of day, case insensitive	Day of week (also accepts weekend days)
dropOffHour	String	Two digits for parsing	Military format between 00-23
mailClass	String	FCPS (First Class Parcel Service) - currently only has FCPS data in database	The mail class of the parcel

When a request was made to the HTTP Server, there were two main classes which were vital to the handling of the input. Upon starting the server, *ServerStart*, which contained the main class to instantiate the server, loaded into memory a hashmap named *zipOffsetMap*. This Map contained a mapping of every valid, five-digit USPS zip code and its associated

Coordinated Universal Time (UTC) offset. This was extremely important for the handling of the input because the time distributions stored in the system's database were stored in UTC time. UTC time is considered "the world's standard time" and was the basis that the 24 hour time zone was centered around (UTC, 2017).

The second integral class in the server was *MainPostQuery*. Main post query contained all of the REST API calls supported by the server. Each separate RESTful function was denoted with the "Path" parameter with associated path from the top level of the server. The function following the Path parameter contained the tasks executed when a correct request was made to that path in the server. Also inside the *MainPostQuery* class were helper functions which convert the hours from the aforementioned *zipOffsetMap*.

Upon receiving the results back from the server, *MainPostQuery* also contained two helper functions, *convertMapToJson()* and *convertToFrontEndJson()*. These functions converted the return results to JSON objects that were passed back to the user and were translated for various business objectives. The latter of the two functions is solely used to convert the results to a JSON object that is interpreted by the demo user interface.

```
38: "0.2"
39: "0.2"
42: "0.1"
55: "0.4"
57: "0.1"
-1: "0.0"
```

Figure 11: Sample time distribution JSON Object output by the server

Once the request is made to the server and the prototype has completed the probabilistic computation, the server outputs a JSON Object containing the time distributions with their associated probability. Figure 11 above shows an example of an output JSON sent back after a request from the server. The package has a 20 percent chance, 0.2 probability out of 1, that the parcel will get from its origin to destination location in 38 hours. The -1 hour is the margin of error. To be as accurate as possible, instead of ignoring outliers or pumping up other probabilities when there is a data scarcity, we send those percentages to the error variable -1 to

not skew the results. There was enough data for that path where the error variable was unnecessary and therefore has a value of 0.

5.2 Testing Design

This section focuses on the testing procedures performed to ensure the quality and correctness of the implementation. These included JUnit testing, load balancing, and accuracy validation.

5.2.1 JUnit Testing

The Spark engine, business layer, and HTTP server were each unit tested thoroughly to ensure that each component was implemented correctly and producing the results that were expected.

Spark Engine

Since the final results produced by Spark highly depended on the original full paths of packages, it was necessary to ensure that events were ordered by date time and belonged to their respective tracking id. If the engine did not pass this test, then there could be tuples that had incorrect information of from-facility and to-facility as well as total transit time. Scan events were tracked in local time of the facility's time zone. Tests were run to check that date times were being properly matched with their correct time zone and converted accordingly. Also, there were tests to see that a time was correctly converted to its hour. For example, 12:01 would return 12 and 11:59 would return 11. If these tests failed, then again, transit times would be inaccurate. Finally, unit tests were written to ensure that total occurrences per (fromFacility, toFacility, endDest) and (fromFacility, toFacility, arrivalHour, transitTime) were being aggregated and counted correctly.

Business Layer and Calculator

Unit testing of the business layer was split into two testing classes, *TestTripCalculator* and *TestNeoConnector*. *TestTripCalculator* ran a variety of tests that generated a fake graph of

paths between start and end Facility nodes and calculated the time distributions to ensure the math returned the expected values. It also tested cases that had sparse time data and cycles in the graph to validate that those are handled correctly as well.

The second testing class, *TestNeoConnector*, populated the Neo4j database with test ShipsTo relationships and Facility nodes and pulled queried paths between them. It also performed a distribution calculation to ensure that storing data and pulling it out of Neo4j does not have any unexpected results.

HTTP Server

Unit testing for the HTTP server was focused on utilities as the server's capabilities were thoroughly tested during load testing. The bulk of the server's unit testing surrounded the testing of input to the server to make sure that bad input could not make it to the model and, by extension, divert resources that could be dedicated to other requests. There were also unit tests that checked the sanitation of the drop off hour with its associated UTC offset based on the drop off zip code. These were to make sure correct data was passed to the business layer, which were directly used in the computation of shipping time distributions. The REST capabilities were tested during the load testing

5.2.2 Load Testing

The results of a load test were response times from server requests based on the amount of requests being submitted over a period of time. Continuing to increase the amount of requests could also be used to determine breaking points for a system or application. Our purpose for load testing was to determine average response rates under different amounts of stress and compare those results to the goal from the technical requirements of around 50 milliseconds.

5.2.3 Accuracy Validation

Once the prototype was completely implemented and unit tested, it was necessary to test the accuracy of the probabilities returned by the trip calculator in the business layer. Because the prototype gave probabilities of different shipping times instead of an overall average time, the accuracy validation of the model needed to be more complex than a linear regression test with expected ship time and average actual ship time. There were two main demands of the accuracy

validation of the system: (1) determine whether a distribution calculation was accurate, and (2) assure the system could give accurate answers for queries from actual users, not just historical test data. These were accounted for using Kolmogorov Smirnov Test and Cross Validation Testing.

Kolmogorov Smirnov Test

As stated in the Background chapter, the K-S Test does not depend on the underlying function of the distributions it is comparing, and can be used to find differences in discrete distributions as well. The K-S Test was used to test the prototype and determine how "correct" the distributions produced by the prototype were versus distributions of actual package shipments. Python's Scipy package has built in functionality for running this test, so all accuracy testing scripts was written in Python. To produce distributions and compare against those of the prototype, the test used popular zip-to-zip pairs with multiple arrival hour and transit time combinations, by querying Endicia's Virtual Post Office Database. One shortcoming of this method it could not test every possible pair of zip codes entered. However, with enough zip-to-zip combinations to model packages shipping over most of the country, correctly predicting those gave confidence that the methodology as a whole was correct as well.

Cross Validation Testing

While the K-S Test can check the differences between distributions, it does not give confidence that the prototype will predict real scenarios well. To account for this, the system was cross validated on an independent data sets from the one it was built with. The particular cross validation technique used on the system was K-Fold validation, where a the USPS scan event data was split into K sets and the model was trained on K-1 sets and tested against the remaining set, K different times. Figure 12 shows a representation of K-Fold testing, where the gray box is the test data set and the green boxes are the train set.

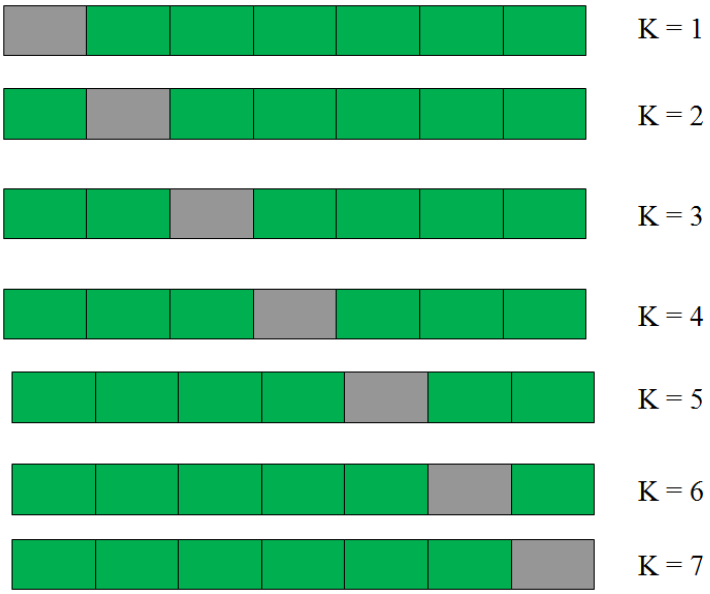

Figure 12: K-Fold Test Representation

6 Results

In this chapter, we present the data and graphs generated by performance tests of each component of the system and also the accuracy tests described in Section 5.2. This includes memory and processing time results from each round of Spark processing, speed testing from various iterations of the Neo4j query, the results of load testing the HTTP server. Finally, the chapter shows the predictions and accuracy results for the mathematical approach used to calculate the shipping prediction time distributions.

6.1 Spark Engine

Endicia's VPO database contains billions of rows of USPS scan event data. Spark was able to process 15.5 million tracking ids, which is about 62 million rows of scan events, in memory at a time on our local machine. A local machine for this project is a Dell Latitude E6430 with 8.00 gigabytes (GB) of Random Access Memory (RAM) and an Intel Core i5 central processing unit (CPU) at 2.70 Gigahertz (GHz) and an Intel Core i7 CPU at 2.90 GHz running a Windows 10 Operating System. The total average time it took Spark to process this amount of data into Neo4j relationships was on average 37 minutes. Figure 13 shows the amount of time Spark took to process various amounts of scan event data. The relationship is linearly dependent on the number of VPO data rows with an R^2 with 0.993.

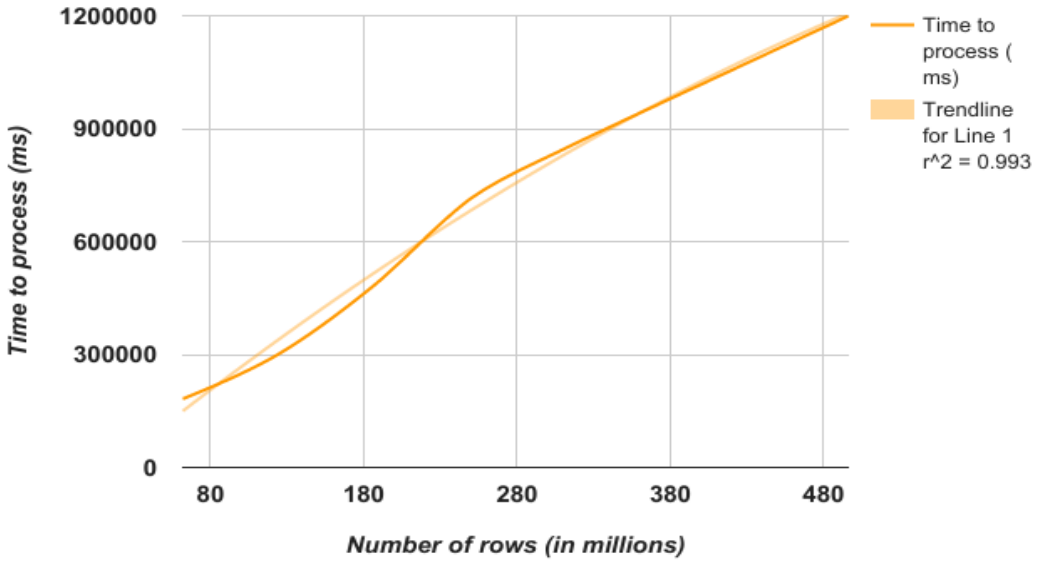

Figure 13: Processing time versus the number of VPO data

In Round One, Spark reduced the 62 million rows to approximately 21.3 million tuples. In Round Two, Spark reduced the 21.3 million tuples by 61.9%, or about 13.2 million tuples.

For this subset of 62 million VPO rows, in the end, Spark produced approximately 582,800 Neo4j relationships. Every relationship is unique by from-facility and to-facility. Figure 14 shows that the number of VPO rows versus the unique Neo4j relationships grows linearly with an R^2 of 0.993.

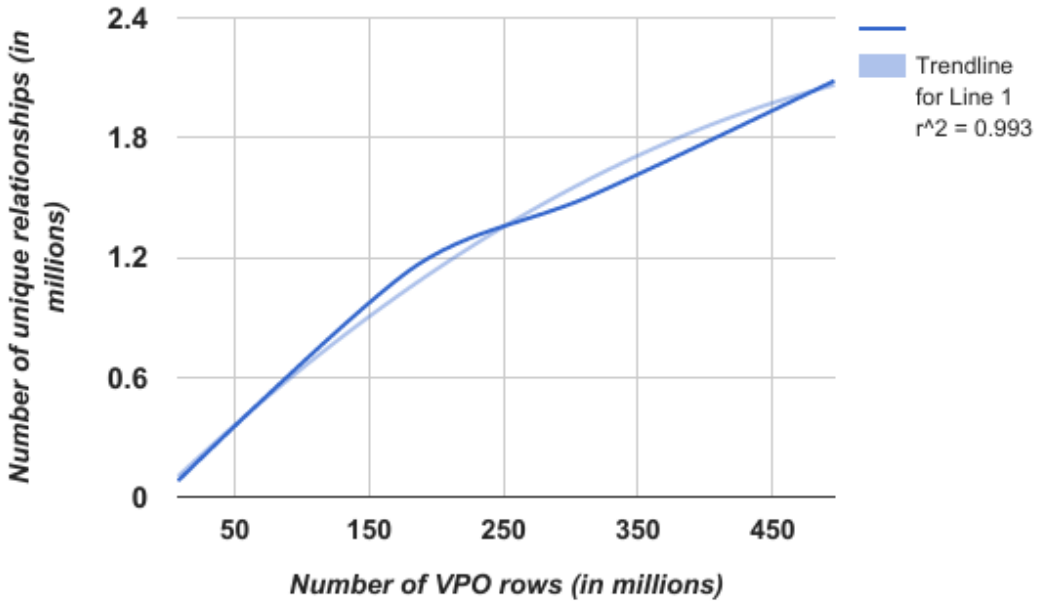

Figure 14: Number of unique relationships versus number of VPO data rows

6.2 Neo4j

As discussed in Section 5.1.5, there were several iterations of querying Neo4j before a fully populated database returned results quickly enough for the demands of the system. This section shows the results querying a path between two zip codes in Neo4j for each optimization step of the data base, for test data, a partially populated database, and fully populated database.

6.2.1 Test Data

Table 7 shows the time (in milliseconds) that a NeoPath query takes on unit test data. More complicated paths take longer to load, so for this example the query loaded a path containing 4 nodes and 6 relationships. The unit test dataset contained 15 nodes and 30 relationships. Figure 15 shows an area graph of these results, showing the differences in query

time when an index was added to a Facility's five digit zip code. With no optimization, queries were averaging 37.6 milliseconds. With an index on a primary key, queries were averaging 13.8 milliseconds, which is 63.3% faster than the initial non-indexed query.

Table 7: Time in milliseconds for Neo Path queries on Test Data

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average
No Optimization	43	40	29	38	38	37.6
Index Added	18	13	15	9	14	13.8

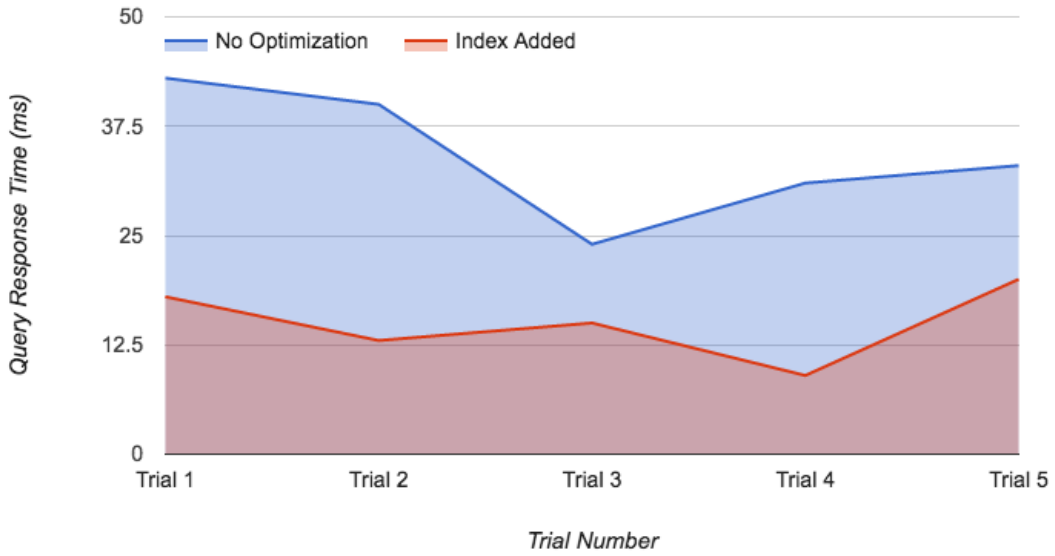

Figure 15: Neo path query time with test data

6.2.2 Subset of USPS Data

Table 8 shows the time (in milliseconds) that a NeoPath query takes on a subset of the full US Postal Network, 70,000 facilities and 300,000 relationships. For the next two experiments, the query is loading a path that returns 15 nodes and 23 relationships. In this experiment, running a Neo4j Path query without a maximum length, index or non-indexed, both ran out memory. With a maximum path length of 11, the query was averaging at 13, 310.2 milliseconds. To further reduce the amount of time for a query to complete, a filter was used on the paths, which resulted in an average of 339.4 milliseconds. Figure 16 shows an area graph of these results, showing the difference filtering paths during a query causes.

Table 8: Time in milliseconds for Neo Path queries on a subset of USPS Data

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Avg
No Optimization	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	
Index Added	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	
Max Path Length	13046	14135	12886	13550	12934	13304.2
Paths Filtered	231	435	412	287	332	339

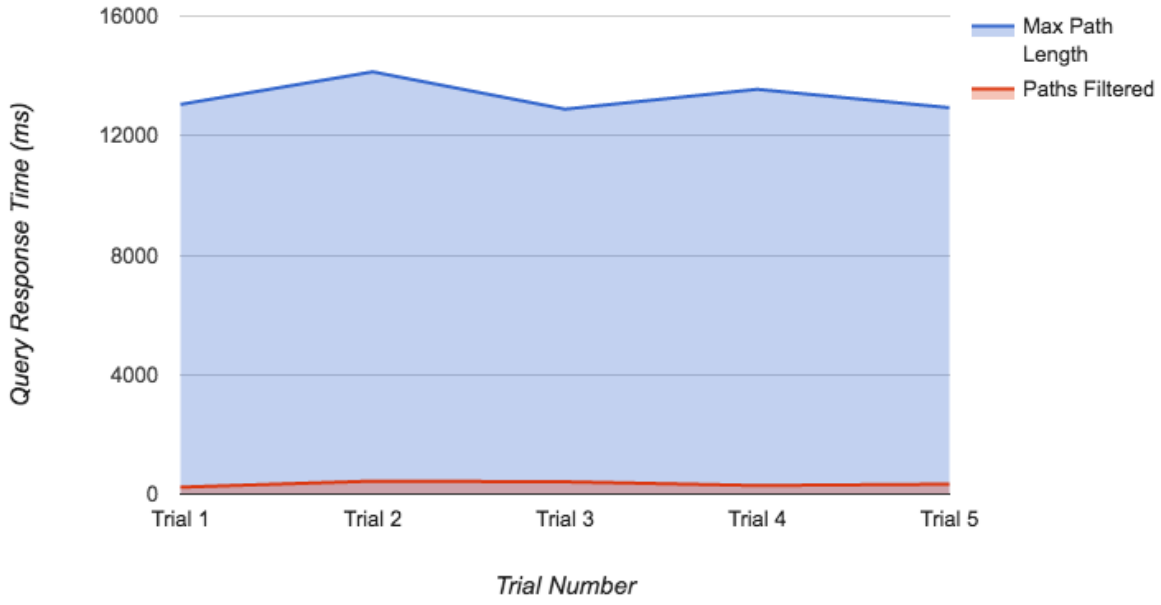

Figure 16: Neo query time with a subset of USPS data

6.2.3 Four Months of USPS Data

Table 9 shows the time (in milliseconds) that a NeoPath query takes on 2.1 million relationships, which was accrued from 500 million scan events from US Postal Network. In this experiment, running a Neo4j Path query without a maximum length, index or not, both ran out memory. However, when paths were filtered, the query completed with an average of 866 milliseconds. Figure 17 shows an area graph of these results, showing the difference filtering paths during a query causes.

Table 9: Time in milliseconds for Neo Path query on 2.1 million relationships

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Avg
No Optimization	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	
Index Added	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	
Max Path Length	Out of Memory	Out of Memory	Out of Memory	Out of Memory	Out of Memory	
Paths Filtered	833	821	859	915	902	866

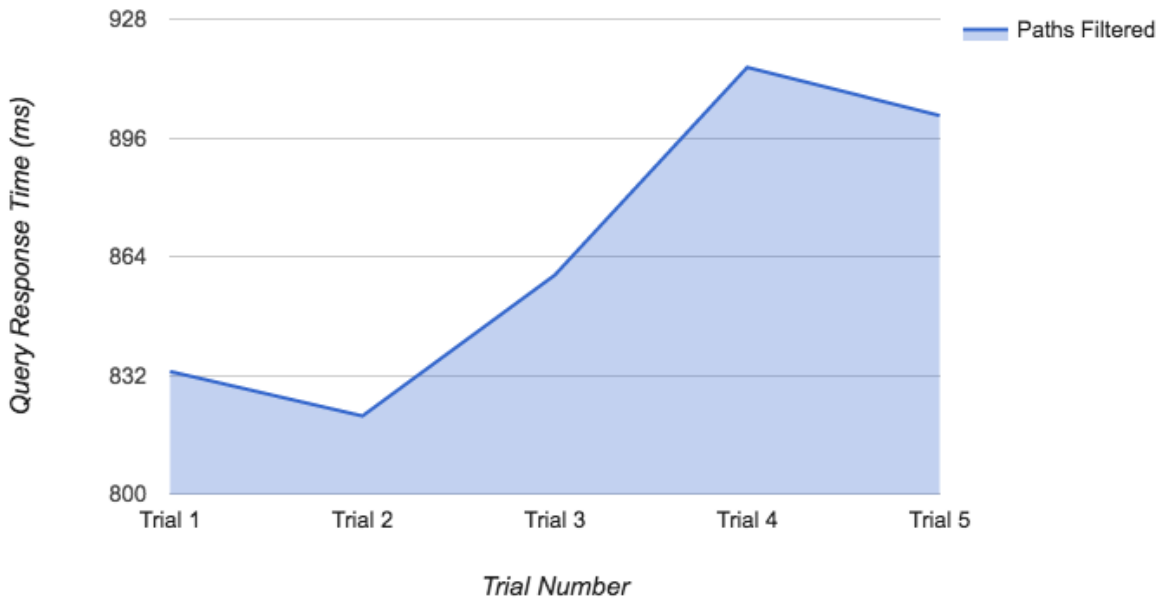

Figure 17: Neo query time with a subset of USPS data

6.3 HTTP Server Load Testing

The HTTP Server was load tested in 3 trials with requests of varying complexity. The request's complexity was based on the number of nodes incorporated in the network between the origin and destination zip codes. After testing of the Neo4j server, the average number of nodes in a network was 4. Using this statistic, we developed an extremely complex query with 28 nodes in the network, a medium complexity request with just over the average at 6 nodes, and a simple request with 3 nodes in the network. Each trial, no matter the complexity or amount of

requests, was run with a clean start of the server. This means that after every trial the server was stopped and restarted. Further discussion about the results of the load testing can be found in section 7.3.

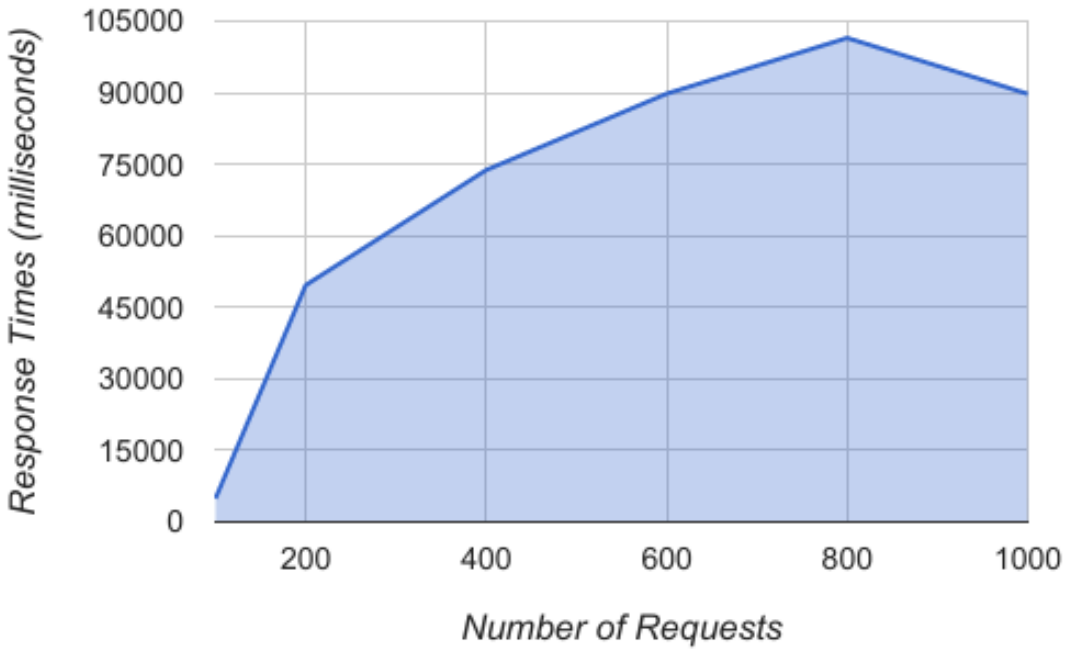

Figure 18: Server response times for a very complex query

Starting out with the most complex requests in the system, a complex query was run in load testing capacity at variable number of requests sent consecutively over a period of 1 minute. Figure 18 shows the results of a complex request to the system over a variable number of requests. This complex-level request went through 28 nodes with 56 relationships which is much higher than the average nodes a package passes through of 4 nodes. This request averaged well over our goal of 50 milliseconds but interestingly peaked at around 800 requests. A request with these many nodes is highly unlikely to be requested but was useful to see how our system would hold up under extreme loads. On the last trial of 1000 requests, 106 of them were dropped by the server when the load reached capacity but were then rerun to make the last trial 1106 total requests. Only the response times for the successfully handled requests were taken into account.

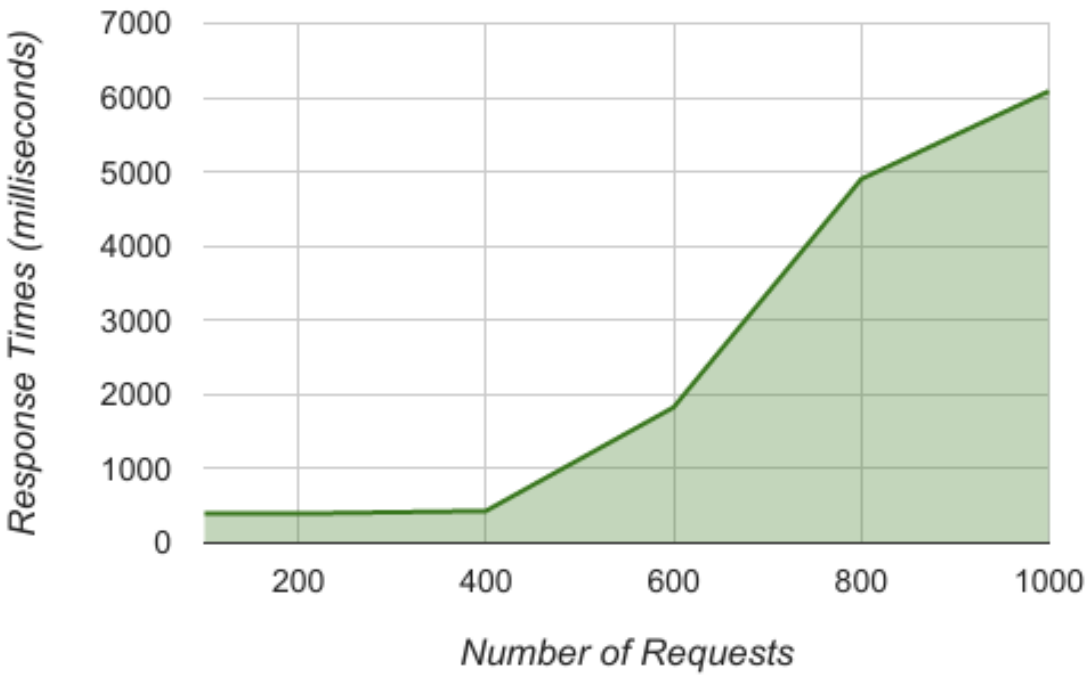

Figure 19: Server response times for an average complexity query

Figure 19 shows the results of load testing the server on a medium complexity request. This request calculated distributions from 6 nodes with 9 total relationships. This was still over the average of 4 but was not an uncommon amount of nodes. This request performed multitudes better than the complex query with the average time for 1000 requests matching up to about 300 requests from the complex query. The averages for the medium complexity query were all over the project goal of 50 milliseconds with even the fewer number trials running around 10 times that figure.

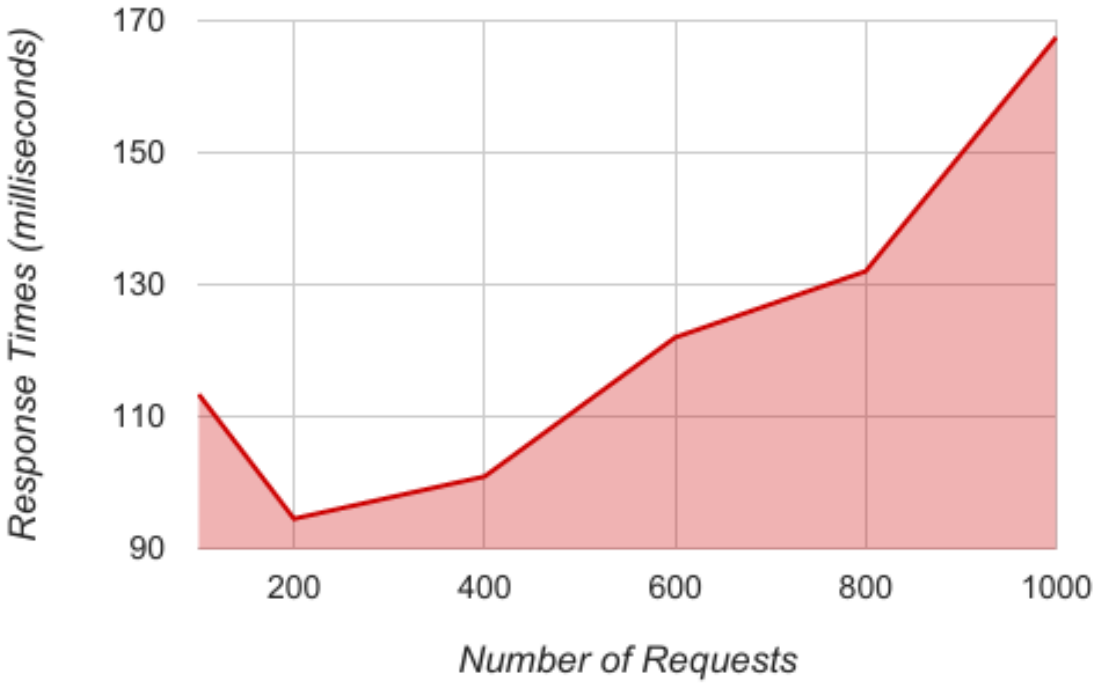

Figure 20: Server response times for a simple query

Figure 20 shows the results of load testing the server on a request of simple complexity. This request contained a path with 3 nodes with 2 relationships which is just under the average of 4 nodes. This request performed much closer to our project goal of 50 milliseconds than the complex or medium complexity requests. Every request in the simple query trials took under 200 milliseconds with the lower numbers even reaching under 100 milliseconds.

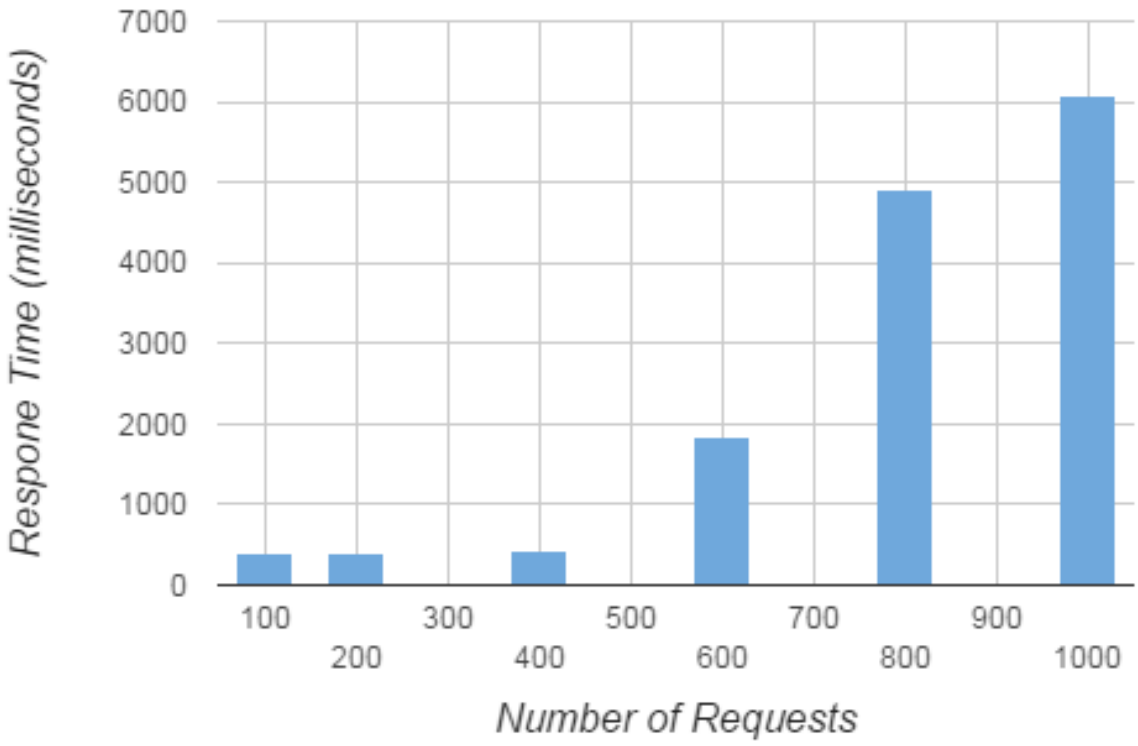

Figure 21: Average Server response times for an average complexity query

Digging deeper into the medium complexity query, which was considered close to average on nodes and relationships in the processed network, Figure 21 shows the results of running multiple trials of a variable number of requests. For each number of requests data point, three trials were run of the respective value using the medium complexity query the server. This was the same request used to run the analysis in Figure 21 described above. None of the trials produced results close to the project goal of 50 milliseconds but the trials with 100, 200, and 400 produced averages of around 400 milliseconds per request which was the closest to 50 but still 8 times larger.

6.4 Predictions and Accuracy Testing

Figure 22 shows two predicted shipping time distributions the prototype produced for a package shipping from 55121 (St. Paul, MN) to 11747 (Melville, NY) based on the hour the package was dropped off. A package that was dropped off at 8AM has essentially a 99% chance that it will arrive to Melville in 31 hours or less. A package that was dropped off at 1PM in St. Paul has about a 25% chance it will arrive in 49-52 hours instead of 31 hours or less.

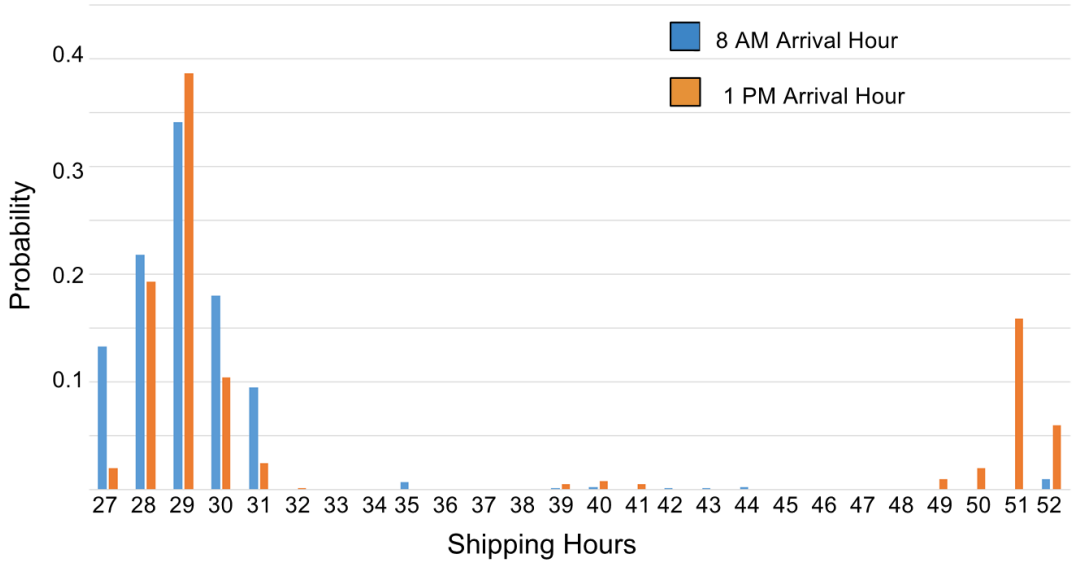

Figure 22: Predicted shipping time distributions for a package shipping from 55121 (St Paul, MN) to 11747 (Melville, NY)

As discussed in Section 5.2.3, we used the K-S test to evaluate the accuracy of the system. To do this, we identified 10 of the most common from-zip, to-zip, and drop off hour tuples from our available data in Neo4j and chose those whose paths covered most of the country. For a tuple to be considered, it had to have happened at least occurred 300 times in the data. Table 10 shows the results for these 10 pairs, as well as their D value from the K-S test. The D value is the max between the cumulative distribution functions of the actual distribution and tested distribution.

Table 10: Results from accuracy testing

From Zip	From Location	To Zip	To Location	Hour of Day	D Value
33065	Pomona Beach, Fl	89135	Las Vegas, NV	22	0.32
55902	Rochester, MN	75001	Dallas, TX	16	0.36
93721	Fresno, CA	80203	Denver, CO	17	0.27
55902	Rochester, MN	85014	Phoenix, AZ	16	0.31
91311	Chatsworth, CA	97204	Portland, OR	19	0.21
19134	Philadelphia, PN	20018	Washington, DC	16	0.26
91311	Chatsworth, CA	98103	Seattle, WA	18	0.18
55902	Rochester, MN	46208	Indianapolis, IN	16	0.21
99219	Spokane, WA	55408	Minneapolis, MN	20	0.26
64030	Kansas City, KS	10010	Manhattan, NY	19	0.13

7 Discussion

This chapter focuses on the analysis and extrapolation of the results shown in the previous chapter. These include discussions for Spark, Neo4j, HTTP server, and predictions and accuracy tests.

7.1 Spark

As mentioned in Section 6.1, Spark was able to process 62 million rows of scan event data to Neo4j relationships in approximately 37 minutes. Seven months of scan event data is an equivalent of 1.2 billion rows. This means that Spark is estimated to process 1.2 billion scan events to Neo4j relationships in 11.8 hours, which is over 50% faster than the previous benchmark of 24 hours. The processing time can be further reduced if the Spark engine can run on multiple cores and have more resources for memory and computing power. This will essentially divide the work among the cores and reduce the time with many cores working.

As shown in Figure 14, the number of VPO scan events versus the number of unique Neo4j relationships appears to be a linear relationship. However, because there is a finite number of Facility nodes in the USPS network, eventually with enough scan events, all relationships can be accounted for. In Figure 23, the X and Y variables signify the total amount of scan event data and the number of unique relationships when it begins to be more constant than linear. After this point, the only memory coming into the database will essentially be additions of the end destination occurrences and time distribution occurrences as the actual information about the end destination and time distribution themselves will most likely already exist in the database. The benefit is that if Endicia can find the memory for the Y relationships, then it will know that they will not face significant memory problems.

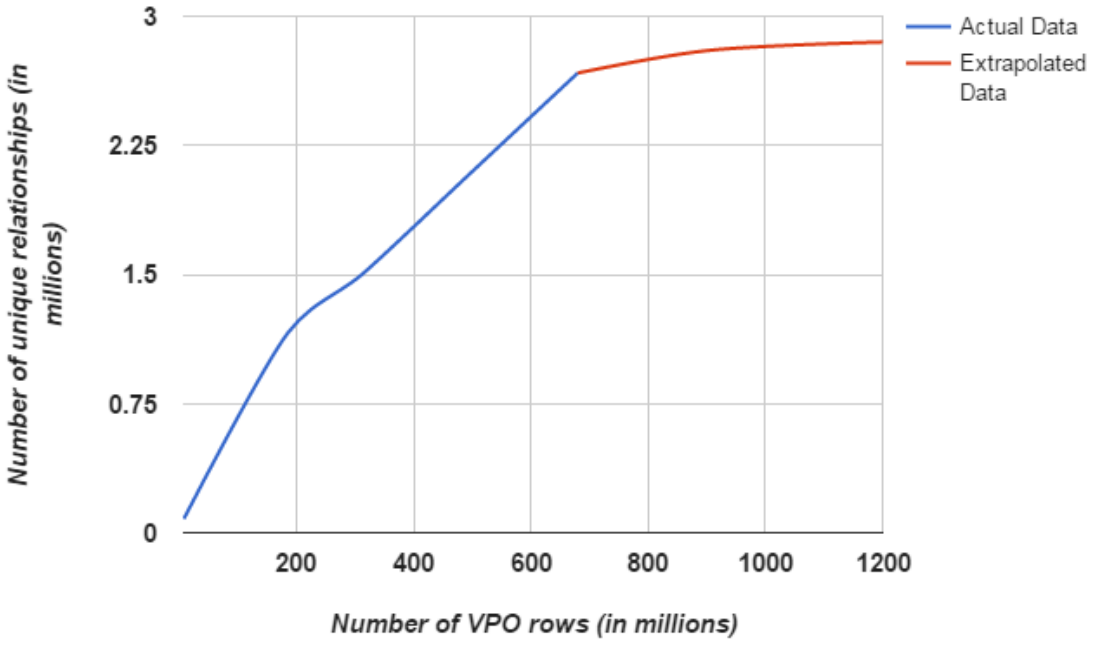

Figure 23: Graph showing the projection of unique Neo4j relationships

7.2 Neo4j

Based on the results presented in Section 6.2, the steps taken to optimize a Neo4j path query to run on a full USPS database. While the first version of the path query was first run on unit test data, its response time was improved by adding an index to each facility. When Neo4j was partially populated with data, the query had to be given a maximum length, which addressed the database running out of memory before completing the query. The last step of optimization of Neo4j path query was filtering unused paths during the query, which greatly improved response time on a sparsely populated database, and was the only version of the query that did provided results in a completely populated database.

7.3 HTTP Server

The load testing of the HTTP Server produced results that were consistently over the project goal of 50 milliseconds. There were some interesting trends in the results as well as some observations and ideas which are relevant when considering why the results did not meet the goal.

The first interesting trend was that the trials with 100 requests were usually similar or greater than the next trial with double the number at 200 requests. This was against the trend of the other data points on the graphs which typically followed a linear increase as the number of requests increased. The second data point that bucked the linearly increasing trend was the final round of testing on the most complex query with 1000 requests. When running this specific trial, a chunk of each trial were dropped due to oversaturation of the server. These response times were not taken into account in the graphs and data [presented in the results section but it is important to note that each time the trial was run it occurred. Subsequently after noticing this trend, it could not be definitively concluded whether the portion of time when the server was dropping requests had an effect on the remaining requests that were rerun to reach the 1000 mark.

A second interesting factor when considering the results is the number of nodes and relationships in the requests used for testing. The average number of nodes and relationships contained a high degree of standard deviation and become statistically unimportant when considering the average number of nodes was 4 but there are paths with as many as 30 nodes. While we could compute the average nodes and relationships, we were unable to effectively test whether there was a relationship between the number of nodes or the number of relationships affecting response time. Further, we were not able to thoroughly test whether increasing one or both of these variables was the driver behind response time. We were also unable to definitively determine whether those two variables were affecting response time over other variables such as less relationships but with more dense time distributions along those connections. For these reasons it was difficult to choose zip codes and times to use in the load testing because we wanted to try requests of varying complexities to make sure our testing was thorough. The zip codes we ended up testing were for saturation of relationships which accounts for the big differences in number of nodes and relationships over the three requests.

7.4 Predictions and Accuracy Testing

Figure 22 was a good example that shows how drop off time for package can affect the probability that it will arrive at its destination in a specific time. If packages ship from St. Paul to Melville are dropped early at around 8AM, then it will almost always arrive in 31 hours or less

as opposed to sometimes arriving in 49-52 hours. If a business wants its packages to arrive to its customers in under 48 hours, based on these predictions, they can use first class and be confident that 99% of the time the package will arrive on time if they drop packages off at 8AM instead of 1PM.

Based on the results in Table 10, the D values given by the K-S test for the 10 distributions tested ranged from 0.13 to 0.36. For context, two distributions that are entirely different would have a D value of 1.0 and distributions that have a 50% chance they came from the same sample have a D value of 0.5. Results like Minnesota to Dallas example with a D value of 0.32 signify that there's a 68% confidence that the predicted time distribution matches the actual time distribution. However, the Manhattan to Kansas City example, with a D value of 0.13, shows that in some cases the prototype gives accurate predictions with a 87% confidence. Figure 24 is a histogram of the time distributions predicted and generated between these two facilities.

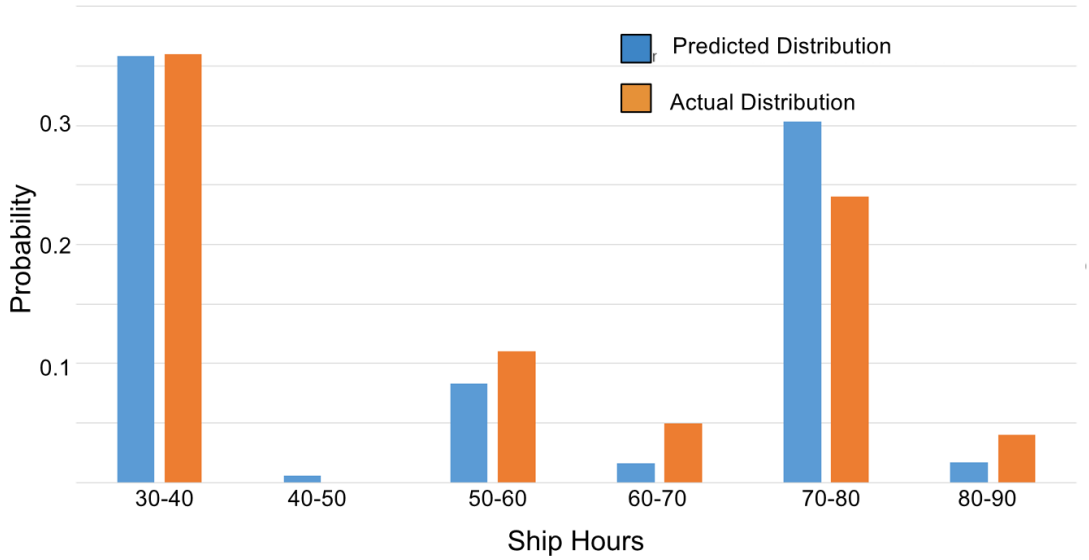

Figure 24: Graph showing of predicted vs actual time shipping from Kansas City to Manhattan

Ultimately, the accuracy testing that was performed gives close, but not perfect predictions. Further research needs to be done to isolate poor results and determine if any are shortcomings results of testing techniques. Due to time constraints, the K Fold testing discussed in methodology was not performed, and the system was only tested with a set of training data

from 2015, and a set of test data from 2014. This data is not random, and the time difference between the data the distributions were being built on could contribute to higher D values.

A second factor to take into consideration about whether the prototype produced accurate enough results is determining the definition of accurate. While the K-S test can compare discrete distributions, and can determine which predictions are better than other predictions, there is no threshold for what is good enough to show customers. Also, K-S test examines whether two distributions came from the same larger distribution. The prototype is producing a distribution that should mirror the actual one, but they do not come from the same dataset, so the expectations are not the same. For this reason, a higher D value might be deemed appropriate.

8 Conclusions and Future Work

The goal of this project was to was to produce a validated prototype that forecasts delivery time distributions to optimize shipping costs. We needed to be able to process million rows of scan event data, load millions of relationships into the Neo4j database, and query the database. We achieved this accomplishing our three objectives: (1) designed and implemented a predictive shipping prototype, (2) optimized the prototype to production standards, and (3) validated the mathematical approach used to calculate the shipping time distributions.

At the conclusion of our project, we had a full working prototype, used Spark to efficiently process the data, modeled the USPS network in Neo4j, and produced somewhat accurate predictions with query speeds averaging under two seconds. Based on our prototype's predictions, drop off time has a significant impact on the overall shipping time for a package.

A summary of the notable performances by Spark, Neo4j, and HTTP server as well as accuracy testing on our prototype is as follows,

- Spark was able to process VPO scan event data efficiently. With about 558 million rows, our prototype was able to process them into 2.32 million Neo4j relationships in 1462427 ms, or about 24.4 minutes, on a single local machine. With multiple machines, it would be possible to reduce the processing time because of Spark's ability to parallelize. Regardless, Endicia would essentially only need to perform processing of this magnitude once because it is the initial loading of the database.
- The prototype's Neo4j query was able to return results for facilities for even some of the most complex paths in the database by filtering the paths to return only those shipping to the specified end destination zip three region. For instance, paths that had around 30 facility hops still on average returned under 2 seconds.
- The HTTP Server was able to successfully communicate with all of the components in the system and effectively pass information back and forth between itself and the calculator and business layer. It was able to handle average complexity requests in a

reasonable amount of time and, even flooded with very complex requests, was able to return correct and accurate shipping time distributions.

- Based on the D values of the K-S test, the distributions produced by the prototype do not give enough confidence to show the results to customers of Endicia. The results can likely be improved with more data and adding other factors to the model, such as seasonality. More research also needs to be done to determine what maximum D value is appropriate to show customers.

Our prototype is very adaptable and maintainable because of the way the system architecture was designed. However, there are a lot of potential areas for extension and improvements.

- To further validate the accuracy of the shipping predictions, the prototype can undergo cross validation testing.
- All the results came from running the prototype on a local machine. The prototype deployed into the cloud using Microsoft Azure, Amazon Web Services, or Google Cloud may improve performance. These services provide a greater number of resources in both memory storage and processing power, which can increase the performance of both Spark and Neo4j. Spark can utilize the multiple cores in parallel to drastically improve the time it takes to process the VPO scan events into Neo4j relationships. The local machine could not handle querying Neo4j when there were two million relationships so another advantage of the cloud is to be able to handle the amount of memory needed to handle the greater number of relationships.
- Another improvement is live updating the Neo4j database. Endicia's VPO database gets updated with new USPS scan event data every 15 minutes. Spark has a tool called Spark Streaming, which allows an input data stream to get batched and sent to the Spark engine to be processed. The benefit of streaming new data into Neo4j would mean the model is always up to date.
- A final improvement is extending the prototype to account for other variables that affect shipping time, such as seasonality, day of week the package was dropped off, and mail class. The prototype's system architecture and design allows for minimum refactoring of

the source code to factor in these variables. For instance, a Priority mail class can be easily modeled by simply giving Spark scan event data mailed via Priority.

References

About the United States Postal Service. (2016). Retrieved March 3, 2017 from https://about.usps.com/who-we-are/postal-facts/size-scope.htm

Amazon.com, Inc. (2017). Guaranteed accelerated delivery fine print. Retrieved March 3, 2017 from https://www.amazon.com/gp/help/customer/display.html?nodeId=201117450

Apache Foundation. (2017a). Apache CXF: An open-source services framework; Retrieved March 3, 2017 from http://cxf.apache.org/

Apache Foundation. (2017b). Licensing of distributions. Retrieved March 3, 2017 from http://www.apache.org/licenses/

Apache Foundation. (2017c). Welcome to Apache Maven. Retrieved March 3, 2017 from https://maven.apache.org/

Apache Spark. (2017). Retrieved March 3, 2017 from https://databricks.com/spark/about

Apache Spark. (2017). Spark streaming programming guide. Retrieved March 3, 2017 from http://spark.apache.org/docs/latest/streaming-programming-guide.html#spark-streaming-programming-guide

Beizer, B. (1995). *Black-box testing*. New York: Wiley.

comScore Inc. (2012). *Online shopping customer experience study*.

comScore Inc. (2015). *2015 UPS pulse of the online shopper.* UPS.

Delivering solutions to the last mile. (2013). Retrieved March 3, 2017 from https://www.usps.com/lastmile/

Donahue, P. (2015). *Postal facts.* ().United States Postal Service.

Endicia. (2016). About us. Retrieved March 3, 2017 from http://www.endicia.com/about-us/company-history

Endicia Solutions. (2017). Retrieved March 3, 2017 from http://www.endicia.com/segments

Fitzpatrick, P. (2017). Customer success stories - Olympia. Retrieved March 3, 2017 from http://www.endicia.com/why-us/success-stories/olympia-sports

Genisoft. (2017). Microsoft technology stack. Retrieved March 3, 2017 from https://www.genisoft.eu/microsoft-technology-stack

Halili, E. H. (2008). *Apache JMeter*. Birmingham: Packt Publishing.

Intelligent mail package barcode frequently asked questions; (2015). Retrieved March 3, 2017 from https://ribbs.usps.gov/intelligentmail_package/documents/tech_guides/IMPB_FAQsFeb2015.pdf

Jakl, M. (2008). *REST representational state transfer*.

JUnit. (2017). Retrieved March 3, 2017 from http://junit.org/junit4/

Mail and shipping services. (2016). Retrieved March 3, 2017 from https://www.usps.com/ship/mail-shipping-services.htm

Needham, M., & Hunger, M. (2016). Effective bulk data import into Neo4j. Retrieved March 3, 2017 from https://neo4j.com/blog/bulk-data-import-neo4j-3-0/

NIST. (2013). *Engineering statistics handbook.* U.S. Department of Commerce.

Our Future Network. USPS (2015). Retrieved March 3, 2017 from https://about.usps.com/news/electronic-press-kits/our-future-network/ofn-usps-delivery-standards-and-statistics-fact-sheet.htm

Schneider, J. (1997). Cross validation. Retrieved March 3, 2017 from https://www.cs.cmu.edu/~schneide/tut5/node42.html

Shear, W. B. (2009). *U. S. postal service: Mail delivery efficiency has improvedbut additional actions needed to achieve further gains* DIANE Publishing.

SmartBear Software. (2017). What is load testing? Retrieved March 3, 2017 from https://smartbear.com/learn/performance-testing/what-is-load-testing/

Stamps.com. (2015, Nov 19,). Stamps.com announces the completion of the Endicia acquisition from Newell-Rubbermaid. *News Bites - Computing & Information* Retrieved March 3, 2017 from http://search.proquest.com/docview/1734039671

USPS. (2016). *Postal facts 2016*.

UTC – the world's time standard. (2017). Retrieved March 3, 2017 from https://www.timeanddate.com/time/aboututc.html

Vassallo, J. (2016). Forrester forecasts US online retail to top $500B by 2020. Retrieved March 3, 2017 from https://www.forrester.com/Forrester+Forecasts+US+Online+Retail+To+Top+500B+By+2020/-/E-PRE9146

Wong, W. E., Horgan, J. R., London, S., & Agrawal, H. (1997). A study of effective regression testing in practice. 264-274.

Xin, R. (2014). Apache Spark officially sets a new record in large-scale sorting. Retrieved March 3, 2017 from https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

Appendix A - Server Documentation, Version 1

Input

The Server currently operates on localhost HTTP ports. It is hardcoded to port 9090 but allows for the user to pass in a port number when executing the jar. The base path of the server after building and running the server is:

```
http://localhost:{port number}/predictive/v1/
```

The Predictive HTTP Server currently supports two different REST calls to query the model. The path to query the model, relevant to the base path is "/query". Written out in its entirety, the path for a model query request is:

```
http://localhost:{port number}/predictive/v1/query
```

The two relevant REST calls which will query the model are:

1) **`POST`** request which takes in a JSON object
2) **`GET`** request which takes in query parameters

Both of the requests to query the model take in the same parameters and will return the same JSON Object.

Parameter Name	Parameter Type	Restrictions/ Requirements	Description
originZipFive	String	Five numerical digits	Five digit zip code of origin facility
destZipFive	String	Five numerical digits	Five digit zip code of end destination

dropOffDay	String	Full name of day, case insensitive	Day of week (also accepts weekend days)
dropOffHour	String	Two digits for parsing	Military format between 00-23
mailClass	String	hard coded to FCPS	The USPS mail class of the parcel

Output

Upon sending a request to the server with the input parameters in the preceding table, the server will query the model and return back to the requesting entity a JSON object. The JSON object contains a distribution of hours to deliver the package with a corresponding probability. An example of the output JSON is below and can be read as the number of hours on the left side of the colon and the probability (out of 1.0) on the right.

```
38: “0.2”
39: “0.2”
42: “0.1”
55: “0.4”
57: “0.1”
-1: “0.0”
```