Scalable Multi Dimensional Threat Analysis

A Major Qualifying Project submitted in partial fulfillment of the requirements for the Degree of Bachelors of Science in Worcester Polytechnic Institute

March 25, 2016

Authors

John Baia

Peter Leondires

Kevin Martin

Dalton Tapply

Faculty Advisor

Professor Elke Rundensteiner

Sponsor Organization

ACI Worldwide

Sponsor Advisor

Eric Gieseke

Abstract

ACI Worldwide provides fraud detection services on high volume transaction data streams. In this project, WPI collaborates with ACI Worldwide to explore how this data can be stored in a graph database and then visualized using open-source big data technologies. Using Titan, transaction data is stored as a graph. The graph is then visualized in several fraud-centric display formats with the Vis JavaScript library to provide fraud analysts a unique way of analyzing transactions. We call this graphical analysis tool GraphView. Achievements of this project include being able to ingest millions of nodes into a graph database and then displaying portions of the graph using innovative visual displays. Optimizations were made by distributing the system over a six node cluster for faster ingestion and graph query support.

Acknowledgements

We would like thank everyone from our sponsor organization, ACI Worldwide, and WPI who helped us make the project a reality. We would like to extend special thanks to our WPI advisor Professor Elke Rundensteiner and our ACI project advisor Eric Gieske. Professor Rundensteiner was always there to advise us on how to make our project professional and complete all requirements needed of us. Mr. Gieseke was kind enough to meet with us every week of our project and advise us the whole way. Eric was always happy to talk problems through and without him the project would be not be where it is today.

We would like to extend thanks to:

Professor Elke Rundensteiner, WPI

Eric Gieseke, ACI Worldwide

Ken Chenis, ACI Worldwide

Jaya Ghosh, ACI Worldwide

Suba Varadarajan, ACI Worldwide

Cleber Martin, ACI Worldwide

Joe Lividini, ACI Worldwide

Sanjay Dodhia, ACI Worldwide

Thea Ghiselli-Crippa, ACI Worldwide

Andrew Morse, ACI Worldwide

Linda Mesinger Nunez, ACI Worldwide

Shahrooz Feizabadi, ACI Worldwide

Table of Contents

1. Introduction

ACI Worldwide is a company that processes electronic transactions for banks, retailers, and financial institutions around the globe. As a company that is responsible for processing its customers' finances, it is critical that ACI provides effective solutions for fraud detection to protect its customers.

Since ACI processes a vast number of transactions daily, transaction throughput is a high priority. Detecting fraud requires analyzing not only the suspect transaction but also investigating previous transactions to make comparisons and determine legitimacy. Looking up historical data, making comparisons, and calculating additional properties on this data can be expensive and may limit the number of transactions that can be processed. The previous MQP with ACI created a horizontally scalable system that could ingest transaction data, perform fraud detection, and do it faster than ACI's current systems[1].

As a next step, our focus now was to create a system that supports new kinds of analysis by feeding the transaction data into a graph database. A graph database is a database in which relationships are stored as a collection of nodes and edges. Building off of the previous MQP team's work, we continued to leverage open source big-data technologies. Using Titan[2] and Cassandra[3], we created a graph that can ingest data from their high-throughput system and then detect fraud within the graph. Organizing data into a graph allows analysts to better discern relations between transactions. Beyond the specific queries that we may leverage using a graph database, the structure also allows the opportunity for visualization. In creating a graphical view

[1]Li, T. & Pham, J. & Zhu, J. 2015. WPI. WPI Major Qualifying Project: Fraud Detection Using Storm.
[2] Apache Cassandra (version 2.0.8). 2015. Apache. http://cassandra.apache.org/
[3] Titan with Hadoop 2 (version 0.5.4). 2014. Aurelius. http://thinkaurelius.github.io/titan/

of transactions and their different attributes we aim to provide a powerful tool for fraud analysts to use in identifying fraud.

2. Background

This section discusses fraud detection, big data, how the two are connected, and what has been done related to these topics in the previous projects working with ACI. We also explore some of the tools we used to accomplish our goals, and our reasoning for using those tools.

2.1 Event Model

We had to consider how data would be handled as it entered our system. Our event model, shown in Figure 2.1, describes the relationships between different parts of our data. When we receive an event from a financial institution, they are classified as either demographic events or transactions. A transaction is any financial exchange a customer makes, while a demographic event occurs when a part of our existing dataset is modified (e.g., a customer changes her email address, a new customer signs up with a bank, etc.). Every event has one or more dimensions associated with it. These dimensions could be accounts, banks, terminals, or customers themselves. A dimension is defined by the metadata. Entities, which could be either an event or a dimension, correspond to the vertices in our graph, while edges represent a relationship between different entities.

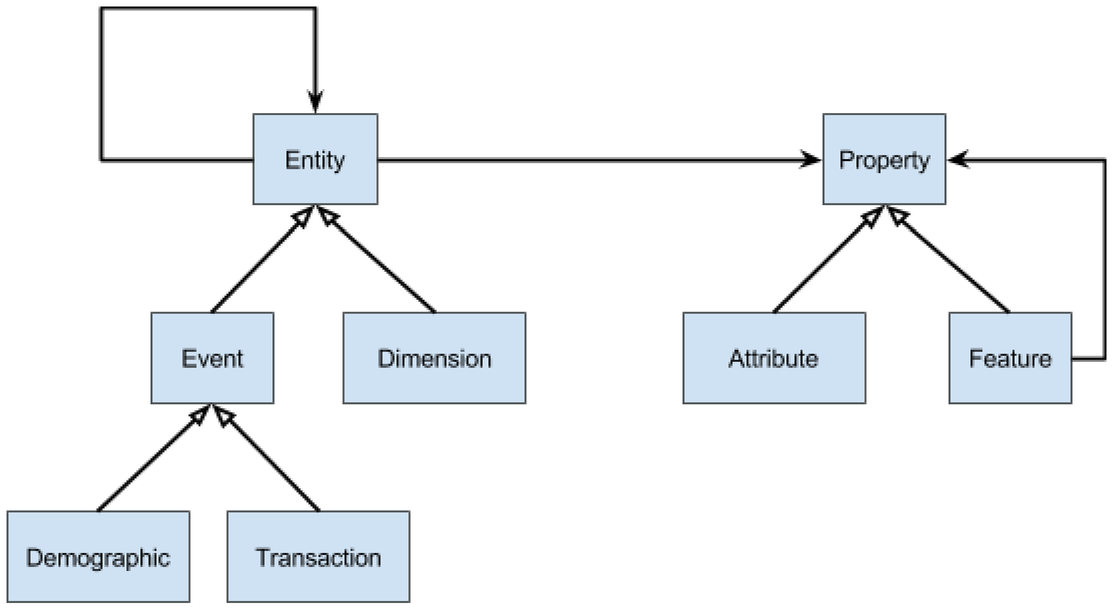


Figure 2.1 Event Model showing the relations between events, dimensions, and their properties

The properties of our entities are either defined as attributes or features. Attributes are information about an entity that is defined before we receive the events. Examples of attributes are a customer's name or date of birth and a terminal's location or unique ID number. Each of these would be defined by the institution that sends us the events. Features differ from attributes because they are calculated after we receive the events. A feature could be the average transaction amount or time since the last transaction. Entities always have at least one property and can potentially have infinitely many. Once our data is defined, we are able to add it to our graph to be stored, queried, and visualized.

2.2 Past MQPs Sponsored By ACI Worldwide

WPI and ACI Worldwide worked together in both 2013 and 2014 to improve fraud detection technology. These projects focused on modeling and feature extraction. They also worked on feature calculations and computation of fraud patterns. The work done by both of these teams helped us better understand the data we were working with.

2.2.1 Complex Event Processing (MQP 2013-2014)

In 2013, the team that worked with ACI set the stage for both the 2014 MQP and this project. They created a Complex Event Processing (CEP) system as a candidate to potentially replace the event ingestion system that ACI used at the time. Using the CEP engine, Esper, they created a real-time, vertically scalable system that replaced the slow SQL queries being used in the old system and translated them into Esper's event processing language. Performance-wise, this system was an improvement compared to ACI's old system, and it gave valuable insight for future performance developments.

2.2.2 Fraud Detection Using Storm (MQP 2014-2015)

Last year's team improved upon the 2013 project by significantly increasing performance. They made use of distributed computing systems, Kafka[4] and Storm[5], to increase the speed of event ingestion and feature computation. They used the stream processing system, Kafka, to ingest a throughput of over 200,000 transactions per second. They also made use of Storm's distributed system to calculate 271 features in 76 milliseconds for each transactions[6]. These features were used to help human analysts determine if there is an instance of fraud. Their entire system is horizontally scalable, so that the performance can increase as the distributed

[4] Apache Kafka. 2015. Apache. http://kafka.apache.org/
[5] Apache Storm. 2015. Apache. http://storm.apache.org/
[6] Li, T. & Pham, J. & Zhu, J. 2015. WPI. WPI Major Qualifying Project: Fraud Detection Using Storm.

cluster size increases. We learned from their usage of Kafka and Storm to maximize the performance of our event ingestion and feature calculation.

2.3 Fraud Detection

According to the Institute of Internal Auditors, fraud is defined as "Any illegal act characterized by deceit, concealment, or violation of trust. These acts are not dependent upon the threat of violence or physical force. Fraud is perpetrated by parties and organizations to obtain money, property, or services; to avoid payment or loss of services; or to secure personal or business advantage"[7]. Fraud can be responsible for not only monetary loss, but can also damage a company's reputation, as well as its relationship with its customers. With an increasing number of fraud related crimes, as shown in Figure 2.2, businesses need to take priority in defending themselves and the customers they serve. The first step in protection against fraud is fraud prevention, but where fraud cannot be prevented, we need methods for detection. Machine learning and statistics serve as effective tools for detecting fraud and are the most common strategies[8].

[7] Institute of Internal Auditors. 2012. Standards. Page 21.
https://na.theiia.org/standards-guidance/Public%20Documents/IPPF%202013%20English.pdf
[8] Bolton & Hand, 2002, Statistical Fraud Detection: A Review, 235-249

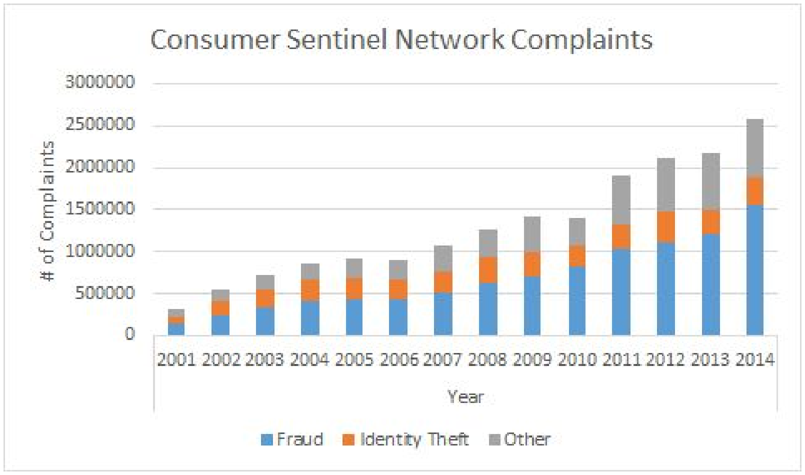


Figure 2.2. Number of complaints received by the Federal Trade Commission's Consumer Sentinel Network[9].

Currently, to detect fraud, ACI's system generates a score that represents the probability of fraud associated with each transaction based on known fraud cases. From there, a rule engine applies a number of rules to this score and the customer's profile to determine whether or not the transaction needs to be flagged. This system is limited in that it only consults the previous 30 days of activity and only uses a fraction of the available features.The current system also lacks a visual approach for the analysts to look through the data. The previous MQP team focused on developing a system to increase the speed of event ingestion and rule calculation, though it is not in use yet.

2.4 Detecting Fraud with Graphs

Fraud analysts can be faced with overwhelming amounts of data at once. Visa reports having a system capable of handling 56,000 transaction messages per second[10]. Currently the

[9] Federal Trade Commission. 2015, February. Consumer Sentinel Network Data Book for January - December 2014.
[10] Visa. 2015. Visa Inc. Reports Fiscal Second Quarter 2015 Net Income of $1.6 billion or $0.63 per Diluted Share. http://investor.visa.com/news/news-details/2015/Visa-Inc-Reports-Fiscal-Second-Quarter-2015-Net-Income-of-16-billion-or-063-per-Diluted-Share/default.aspx

data exists in relational databases. In addition to the data, there are a number of calculations, called features, which are available to the analysts. Analysts are responsible for examining the data and the calculated features to determine if a transaction is fraudulent. The current methods involve manually sifting through the data that is most likely to be fraudulent. There is too much data to process for an analyst to read through all of it, so it has to be reduced as much as possible.

Combinatorics is the study of counting and arranging objects with specific attributes, such as arranging data by similar properties[11]. If the transaction data could be arranged in a graph, using this theory, it could reveal patterns in the data that could not otherwise be seen. We have successfully created a graph database to store transactions that allows analysts to more easily find patterns in the data. By using graphs to group these patterns in data, analysts are able to find any new transactions that are outside of the graph's existing trends[12]. For example, if a customer typically made transactions of small amounts in the United States and suddenly made a large transaction in another country, this transaction instance could be seen as breaking the customer's existing pattern and could be potentially fraudulent.

2.5 Tools

For this project we needed a variety of tools to help us accomplish our goal, namely Titan, Cassandra, Rexster, Elasticsearch and Vis.js. Each tool needs to work with distributed computing systems, so that we can process large numbers of events quickly. First we need a graph database that is scalable, since we will have millions of transactions displayed in our

[11] Agarwal, Udit & Singh, Umesh Pal. 2009. Graph theory. http://common.books24x7.com.ezproxy.wpi.edu/toc.aspx?bookid=34046.
[12] Brath, R., & Jonker, D. 2015.In Graph analysis and visualization discovering business opportunity in linked data. http://proquest.safaribooksonline.com.ezproxy.wpi.edu/book/databases/business-intelligence/9781118845875/chapter-1-why-graphs/h1_845844c01_0002_html?uicode=wpi

graph. For the graph database we use Titan, which uses Cassandra as its backend. We need a way to access the graph remotely, for which we use Rexster. To use Titan effectively we need an index backend, for which we use Elasticsearch. Finally, we need a visualization tool so that analysts have a convenient way to work with the data. For visualization we use the JavaScript library Vis.js.

2.5.1 Titan/Cassandra

Titan[13] is a distributed graph database optimized for storing and querying graphs. Titan relies on a separate storage backend, for which we used Cassandra[14], a distributed database manager. Titan implements Blueprints, which is a set of test suites and interfaces specifically for graph models. A number of graph databases are Blueprints compatible and use the Blueprints suite.

In *An Empirical Comparison of Graph Databases*[15], researchers Jouili & Vansteenberghe compared multiple graph databases across different workloads and conditions[16]. Some of Titan-Cassandra's alternatives that Jouili & Vansteenberghe tested were OrientDB, SparkSee (DEX), Neo4J, and Titan with BerkeleyDB as a backend instead of Cassandra. Each of these different tools have their own strengths and weaknesses. For graph traversals, the authors found that Neo4J is indisputably better than the others stating "it outperforms all the other candidates, regardless of the workload or the parameters used". To compare graph traversals the authors tested neighborhood breadth first exploration for the different tools. The test randomly selected a

[13] Titan with Hadoop 2 (version 0.5.4). 2014. Aurelius. http://thinkaurelius.github.io/titan/
[14] Apache Cassandra (version 2.0.8). 2015. Apache. http://cassandra.apache.org/
[15] Jouili & Vansteenberghe, 2013, An empirical comparison of graph databases, http://www.odbms.org/wp-content/uploads/2014/05/an-empirical-comparison-of-graph-databases.pdf
[16] Ibid.

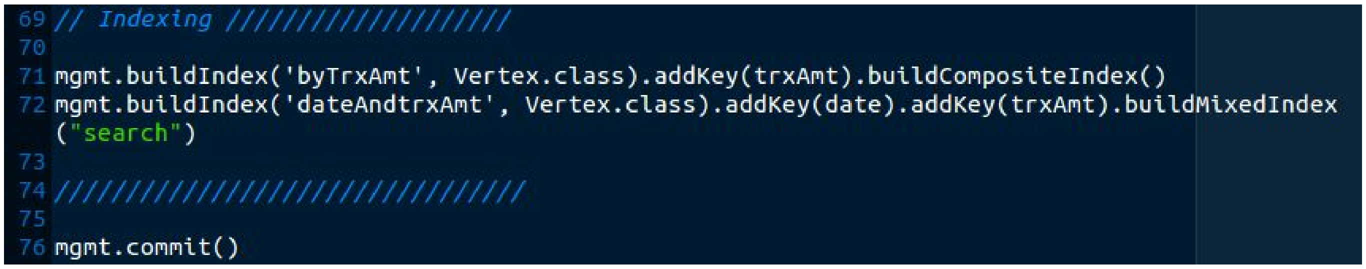
vertex from the graph, and then recorded the time it took to retrieve all of the vertices that were a certain number of hops away from the original vertex. For this test Titan with Cassandra performed the worst as its retrieval times increased much more than those of other graph databases as the number of hops increased.

When we consider workloads where clients are making concurrent requests to the graph, Jouili and Vansteenberghe's tests show that Titan with Cassandra performs faster than Neo4J and the other graph databases[17]. When adding edges or reading and updating vertices, Titan and DEX (Sparksee) significantly outperform the other tools, each performing the tasks in a second or less, compared to Neo4J which took as long as 20 seconds to add edges, or 60 seconds to read and update vertices.[18]. When concerned with read-only workloads Titan-Cassandra fails to deliver, but for read-write workloads Titan-Cassandra and DEX outperform the other databases[19] . For our purposes we needed a tool that could deliver performance for both read and write operations and that scales well with concurrent access. Beyond the comparisons Jouili and Vansteenberghe did, we wanted a distributed system to leverage the previous team's work, so Titan-Cassandra was the obvious choice for us since it scales well with multiple clients, delivers read-write performance, and makes use of distributed systems.

[17] Jouili & Vansteenberghe, 2013, An empirical comparison of graph databases, http://www.odbms.org/wp-content/uploads/2014/05/an-empirical-comparison-of-graph-databases.pdf.
[18] Ibid.
[19] Ibid.

2.5.2 Elasticsearch

Elasticsearch[20] is a distributed and scalable index search backend that can run side by side with Titan[21] and Cassandra[22]. With an Elasticsearch backend, Titan is able to create multi-key indices on graph data, making it possible to query separate objects in a database based on multiple conditions. In Titan, the multi-key indices are called mixed indices, which can be created via the Gremlin console. Below is an example of how to create a mixed index over the date and transaction amount vertex properties.



```
// Indexing ///////////////////

mgmt.buildIndex('byTrxAmt', Vertex.class).addKey(trxAmt).buildCompositeIndex()
mgmt.buildIndex('dateAndtrxAmt', Vertex.class).addKey(date).addKey(trxAmt).buildMixedIndex
("search")

///////////////////////////////

mgmt.commit()
```

Figure 2.6 Gremlin code for indexing

Elasticsearch not only provides the capability to produce multi-key indices, but it can also be distributed across a cluster to improve performance. Elasticsearch operates as a self-contained cluster. This is one of the key reasons we chose Elasticsearch over its competitor, Apache Solr[23], another index backend Titan supports, which depends on a ZooKeeper[24] server to function.

[20] Elasticsearch (version 1.2.1). 2015. Elasticsearch. https://www.elastic.co/
[21] Titan with Hadoop 2 (version 0.5.4). 2014. Aurelius. http://thinkaurelius.github.io/titan/
[22] Apache Cassandra (version 2.0.8). 2015. Apache. http://cassandra.apache.org/
[23] Apache Lucene. 2015. Apache. http://lucene.apache.org/solr/
[24] Apache ZooKeeper. 2016. Apache. https://zookeeper.apache.org/

2.5.3 Rexster

Rexster[25] is a graph server used to allow access to a Titan graph through a REST API as well as Rexster's custom binary protocol RexPro. Rexster is a necessary component to be able to access the graph database remotely. Rexster provides a number of configuration options that allow each instance to be easily customized to suit the intended user. For example, one Rexster instance could allow only read only access, whereas another could permit read and write access, yet both are using the same graph. Rexster can also be configured to ensure high-availability, guaranteeing that the graph can still be accessed remotely even in the event that a node fails.

2.5.4 Vis.js

The tool we used to visualize our graph was a browser based JavaScript library called Vis.js[26]. Vis allowed us to display our graph in a browser. It also provides tools to manipulate the graph's nodes and edges. We were able to change the size of nodes and their color using Vis's built-in functions. Vis holds the information for the nodes and edges such as the node properties. This allowed us to dynamically populate a website based on what was currently in the graph.

One alternative visualization tool that we considered was D3, another JavaScript library. D3 was equally capable of visualization as Vis is, but where D3 fell short was how it placed nodes on the screen. The way D3 and Vis arrange their nodes is based on physics built into each of the libraries. We found that D3's built in physics did not lead to coherent layouts like Vis's engine did. Vis also provides tools to zoom in and out, center our graph, and drag and drop our nodes.

[25] Rexster (version 2.6.0) 2014. Tinkerpop. https://github.com/tinkerpop/rexster/wiki/Downloads
[26] Vis.js (version 4.15.1). 2015. Vis.js. http://visjs.org/

2.6 Terminology

Event

An event is any kind of transaction or demographic event. A demographic event is an update to a dimension. An example demographic event would be a customer changing their name.

Dimension

Dimensions are the entities associated with a transaction. A graph could be populated with dimensions such as retailers, location, time, or customers.

Properties

Properties are either values that describe an event or are computed from existing values. These values represent information about the different dimensions. They can be divided into two groups, attributes or features.

Attributes

Attributes are properties of an event that are found in the original data.

Features

Features are properties of an event that are computed from attributes or other features. An example of a feature would be the average transaction amount for a customer for the last 30 days.

Semantic Tag

A semantic tag is a naming convention for specifying dimensions and properties within the graph. Data from different types of transactions will have their own field names and formats. Those field names are mapped to semantic tags, a definition for fields across datasets, so that different data formats may feed into the same graph.

Synthetic Nodes

A synthetic node is a node that is generated based on the properties of different events or dimensions. These nodes are created after the ingestion of data, and are used to connect existing nodes which share the assigned properties.

3. System Methodology

This section details our implementation plans and the design of the GraphView architecture. Next we explain each component, and how they connect with one another.

3.1 Motivations

The goal of this project was to provide a tool to fraud analysts to improve their ability to find patterns in transaction data. Since the analysts currently use a non visual approach to analyze the data, displaying the data as a graph would allow the user to find patterns that they might otherwise miss. We wanted to leverage open-source big-data technologies as well as use a horizontally-scaling, distributed computing system in order to introduce these technologies to the company. Open source software is cheaper than proprietary software and usually has many different users committing to the project. The benefit of having many committers is that many of the bugs are likely to have been worked out. Using big data technologies allows us to store vast amounts of information and compute information about it. The company was interested in big data technology and wanted to use our project as a test run for future projects. In order to store large amounts of data, we needed a distributed graph backend, so that we could spread the data storage and computation load onto multiple systems. Overall, we expect that our GraphView system will increase the accuracy and performance of fraud detection at a lower cost.

3.2 GraphView Architecture

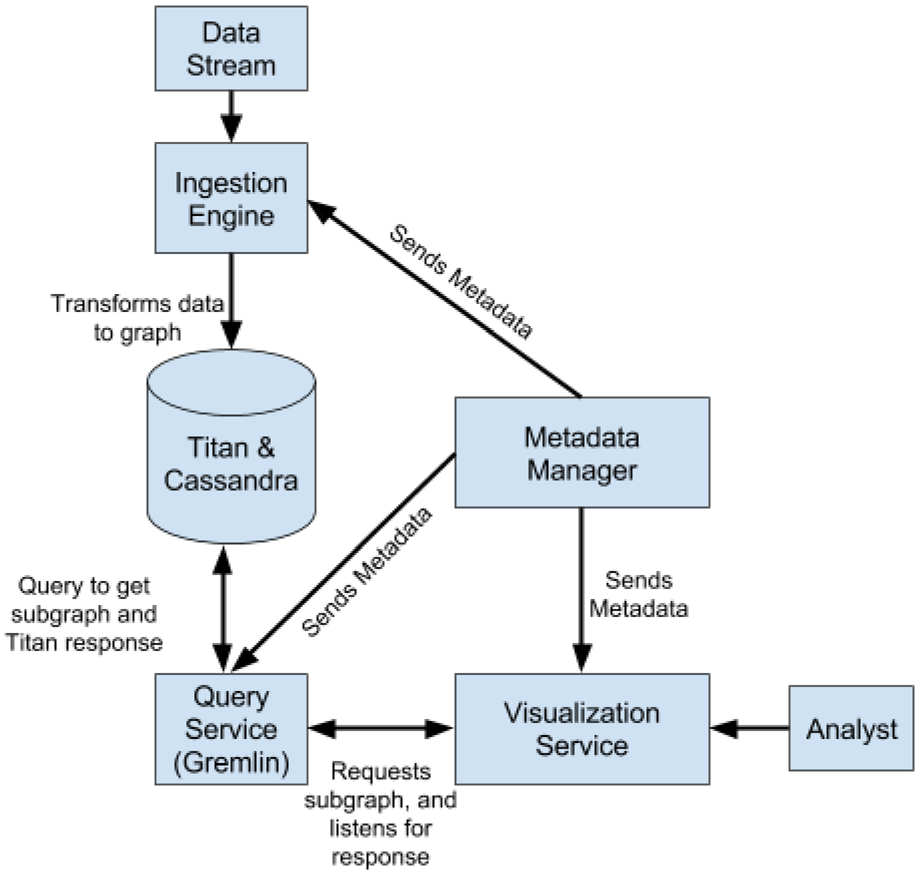


Figure 3.1 GraphView architecture

The architecture for the GraphView system consists of the entities in Figure 3.1, starting with the Ingestion Engine. The Ingestion Engine is responsible for taking in events, and inserting them into our graph database. We use Titan for our graph database, which uses Cassandra as its storage backend. The graph can then be queried using the Query Service, which uses an implementation of a Rexster Client to run Gremlin queries on Titan.

The Query Service is responsible for querying Titan for a subset of the graph, and outputting Vis compatible JavaScript to the Visualization Service. The Visualization Service represents the user interface that a fraud analyst will use to interact with the graph. Requests made through the Visualization Service will iterate through the architecture chain to generate the appropriate response.

3.3 Metadata Manager

Metadata, in the form of JSON, is stored in the Metadata Manager. The Metadata Manager is an HTTP Server wrapper around the Metadata Manager, allowing anyone to make HTTP calls for specific metadata at request. The goal is to allow any application to easily access the most up-to-date metadata.

There are two types of metadata, generic and decoration. Generic metadata contains any information relating to transaction file format types such as transaction field offsets, lengths, names, semantic tags, etc. This is the information stored in file descriptors for each specific transaction format type. Using this metadata, systems would be able to parse out any specific transaction fields from a transaction tuple.

Decoration metadata contains information relating to a specific system, which in our case is our graph metadata. For instance, our graph decoration metadata holds semantic tags, dimension types, vertex and edge information, etc. This metadata is used to define how the event data should be transformed into nodes and edges by the ingestion engine. For example, using the metadata we tell the ingestion engine to create edges between customer vertices and their account vertices.

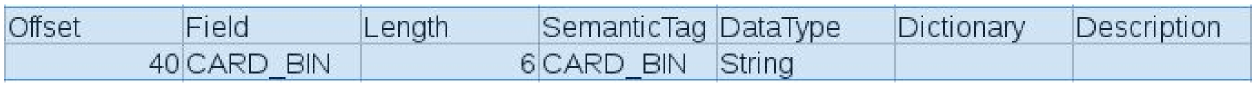
The Metadata Manager also has the functionality to join general and decoration metadata by Semantic Tag. When general and decoration metadata is requested, a joined version is returned. Decoration metadata fields that don't have a matching Semantic Tag field, in the general metadata to be joined, get dropped. Whereas general metadata fields get added to the joined metadata regardless.

3.3.2 Metadata

As explained above, there are two categories of metadata, generic and decoration. In our case we use graph decoration metadata and a generic metadata set. The generic metadata describes general information about the data files. The graph metadata maps those generic entries to the graph specific information of each. All metadata is stored in CSVs for easy modification and are converted to JSON during runtime of the metadata manager.

3.3.3 Generic Metadata

Each entry in the generic metadata has a semantic tag, field name, offset, length, data type, description, and dictionary. Semantic tags are used to map any field names specific to that file into properties that can be shared across different data sets. The field name represents the original field name from the data set. The offset is the byte offset for the beginning of the field for each line in the data set. Length is the total length of the field. Using the offset and length the actual value of a field is determined during ingestion. The data type defines what class to parse the field value as. Description is an optional descriptor for the field name, and dictionary represents a set of possible values if the field only accepts certain values. An example entry can be found in Figure 3.2.

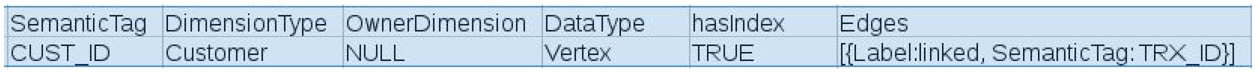


Offset	Field	Length	SemanticTag	DataType	Dictionary	Description
40	CARD_BIN	6	CARD_BIN	String		

Figure 3.2 General metadata example entry for CARD_BIN field

3.3.3 Graph Metadata

Each entry in the decoration metadata has a semantic tag, dimension type, owner dimension, data type, index boolean, and an array of edges. Like in the general metadata, a semantic tag defines a field that is common across different data set formats. In this case the semantic tag is used to connect the graph and general metadata. Dimension type represents the kind of dimension that the entry belongs to. The owner dimension specifies the semantic tag of the dimension that the entry should be applied to as a property. If the owner dimension is null, this indicates that the entry represents a vertex. The data type is used as in the general metadata to specify class types in order to correctly generate properties and parse values. We use the vertex keyword to signify that the entry is a dimension. The hasIndex column is a boolean value to define whether or not this field should be indexed or not. This allows customizability of the index configuration of the graph. The edges column is used for any vertex to define the outgoing edges from it. It exists in a JSON array style format where each edge in the array is defined with a label for the edge, and the semantic tag of the vertex to attach to. The format is such that an edge is created from the entry it's declared in outgoing to the semantic tag specified. An example entry can be found below in Figure 3.3.



SemanticTag	DimensionType	OwnerDimension	DataType	hasIndex	Edges
CUST_ID	Customer	NULL	Vertex	TRUE	[{Label:linked, SemanticTag:TRX_ID}]

Figure 3.3 Graph metadata example entry for CUST_ID semantic tag

3.4 Ingestion Engine

The ingestion engine is responsible for translating data into nodes, edges, and their properties within the graph. The ingestion relies heavily on the Metadata Manager which provides all of the necessary information to properly parse the data.

Given a set of events, the engine first retrieves the metadata associated with these data. This includes both the general metadata about how to parse the incoming event, as well as the graph metadata describing how each field gets translated into the graph. Any metadata must be created prior to ingestion as it describes how the data is processed.

Once the metadata is acquired the graph metadata is used to determine the property keys, indexes, vertex labels and edge labels of the graph, all part of the graph's schema. For each field in the metadata a property key is generated based on the given data type and defined such that each node with the property can only have one instance, one value of that property. Any field that needs to be indexed has a composite index created for it based on that field's property key. If the field is a vertex identifier we specify uniqueness using the index builder. This helps enforce that vertices cannot have any duplicates in the graph. So if for example an Account vertex is based on some account number, that value for the account number property must be unique throughout the entire graph. Finally to finish preparing the schema we create vertex and edge labels based on the metadata.

Once the schema has been created the actual data ingestion takes place. Each event is either a demographic event or a transaction. Parsing the data sets is dependent on the provided general metadata. From the metadata the Ingestion Engine determines the length and offset associated with a field and can parse the specific data out from the data file. In the datasets used, an event is terminated with a carriage return, so the data can be easily parsed considering each line as a separate event.

As has been described before, there are two kinds of events to consider: Demographic events and Transactions. Demographic events are focused on updating existing nodes or edges in the graph. The engine's process is to find the specific node (or edge) to be changed, and then update the specified properties. In this case a node would be found based on its semantic tag. So to change a customer's email for example, a customer id would be necessary to identify the correct node. Then the new email value as well as the emails property key name would be used to change the property.

Transaction events are much more involved as each transaction can have a large number of vertices and edges to create each with their own properties. The metadata is sorted to ensure that Vertex fields are processed first. This ensures that we have obtained/created all of the vertices related to the current transaction, meaning we can easily add the remaining properties, and create the edges. As the vertices are found or created for the transaction, they're stored in a Hashmap to use for property and edge creation. This limits the number of reads to the graph, which can be very expensive relative to the other ingestion operations. The updates to the graph are committed after each transaction.

The ingestion engine exists as a java application. By replicating it across different machines, multiple instances can be loading data into the same graph simultaneously. Data files are split evenly based on the number of nodes, and then those files are distributed among the nodes. Once data is distributed, one machine is used to setup the graph. In this stage the graph may be shut down and cleared of any previous data if necessary, and it will have its schema defined based on the metadata. Once the schema has been created from each machine an instance of the Ingestion Engine is executed, simultaneously loading the separate data sets into Titan. A diagram of this process is seen below in Figure 3.3.

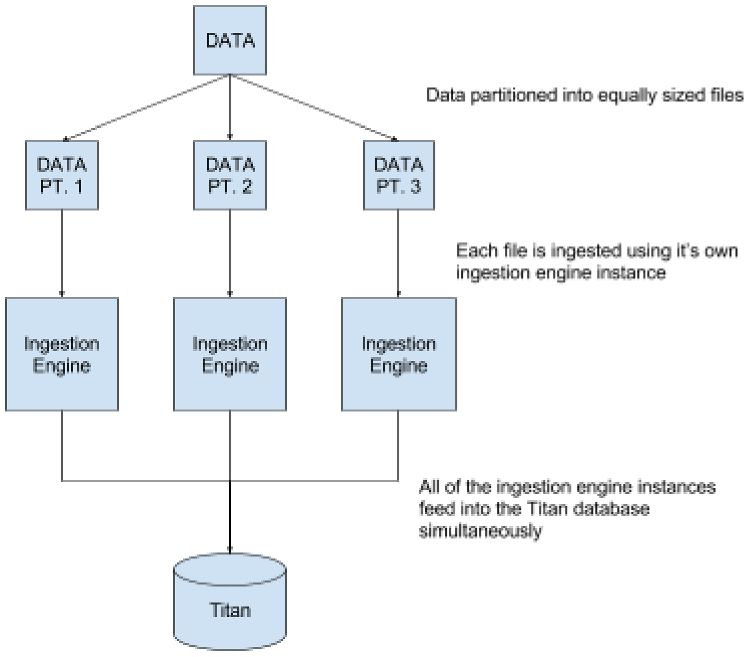

DATA
Data partitioned into equally sized files
DATA PT. 1
DATA PT. 2
DATA PT. 3
Each file is ingested using it's own ingestion engine instance
Ingestion Engine
Ingestion Engine
Ingestion Engine
All of the ingestion engine instances feed into the Titan database simultaneously
Titan

Figure 3.3 Diagram of simultaneous ingestion process

3.5 Query Service

The Query Service is primarily responsible for taking a graph filter query, obtaining a subgraph from Titan, and transforming that subgraph into valid Vis JavaScript. The process starts with the Query Service getting an HTTP request from the Visualization Service to obtain the subgraph for a specified query. The HTTP request contains a JSON formatted query which is converted into a Gremlin query and then executed using an implementation of a Rexster Client. The returned edges and vertices are ingested into our custom subgraph class where we can manipulate and convert them into Vis JavaScript.

The JSON query is formatted as the following:

```
QueryObject = {groups:
                    [{dimensionType: “”,
                    properties:
                         [{propertyType: “”,
                         propertyValue: “”,
                         operator: “”
                         }]
                    }],
               edgeLevels: “”,
               vertexLimit: “”}
```

The group list contains a variable number of dimension property comparators. Each group will create a separate Gremlin query, whereby each vertex and edge will be added to the subgraph; a logical “OR”. Each item in the group list has a dimension type and a list of properties. Each item in the property list contains a property type of which it is filtering by, property value of which it is comparing to, and the operator of which it is using to compare. These are the parameters for filtering the Titan graph.

The edge levels parameter is used to obtain outward level edges and vertices, from the initial set of vertices that are obtained from the dimension list filtering. The Query Service will generate a new Gremlin query for each level, run the queries, and add the new vertices and edges to the subgraph. If there are duplicate edges or vertices returned from these queries, the java subgraph class will ensure that they do not get added.

Finally the limit parameter is there as a hard limit for the number of vertices returned in the subgraph. The subgraph class won't allow vertices to be added to it if the size of the list of vertices is equal to the limit. In order to be able to return a limited subgraph that takes into account edge levels requested, there is a process in which the gremlin queries have a gremlin limit tagged onto them. In detail, the queries are all set to return the first, Limit / (NumOfGroups + NumOfLevels). The outcome of this is that the subgraph will contain some vertices from each query group, and vertices and edges from each level, instead of just returning the first Limit number of vertices.

3.6 Visualization Service

The visualization service is a blanket term for the entire web application that we have created. The purpose of the web application is to allow the user to visualize the graph we have created. Upon accessing the web app, the user will be met with a login screen where they can either login with existing credentials or create new ones. The login screen provides two advantages to our system, security and customizability for each user. The security aspect of the page is fairly self-explanatory. The data in the graph that this service visualizes for the user is highly sensitive data and should only be accessible to specific people. We were also interested in giving each user the option to easily save graphs that they use often, and the best way to do this was to create user-specific capabilities.

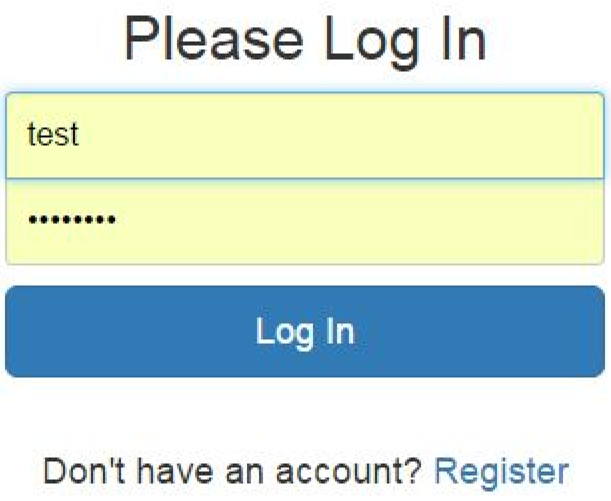
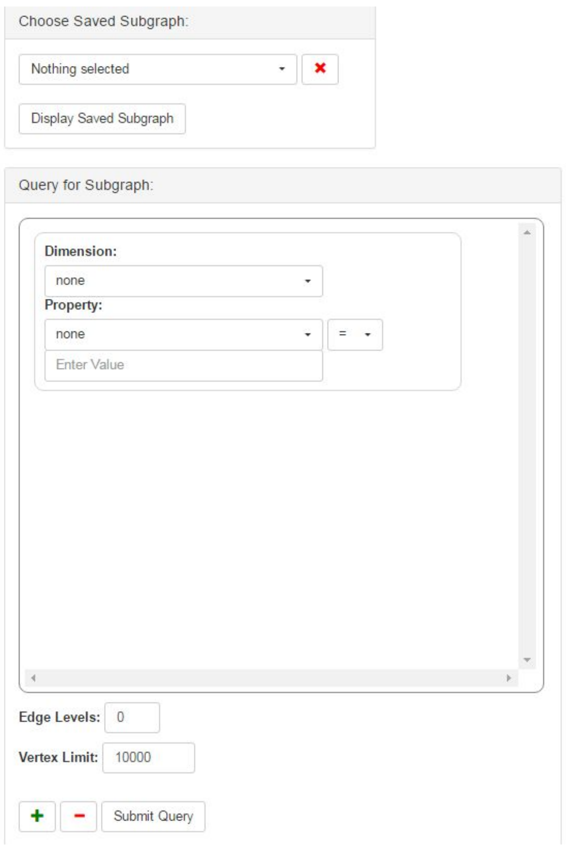


Figure 3.5 Login page

After a user logs in, they are presented with a page that allows them to query the Titan database for a graph. There is no feasible way for the client to render every node in the database, therefore we needed a way to allow the user to decide what nodes they want rendered. This page allows the user to interface with the database without needing to learn the complicated querying language. The user can use this page to either query the database for a new graph, query the database for a saved query, or simply load a saved graph without needing to wait for it to be updated. This page sends a JSON to the Query Service which responds with a list of edges and vertices formatted in JavaScript for our Visualization Service to render.


Choose Saved Subgraph: Nothing selected

Query for Subgraph:

Dimension: none

Property: none =

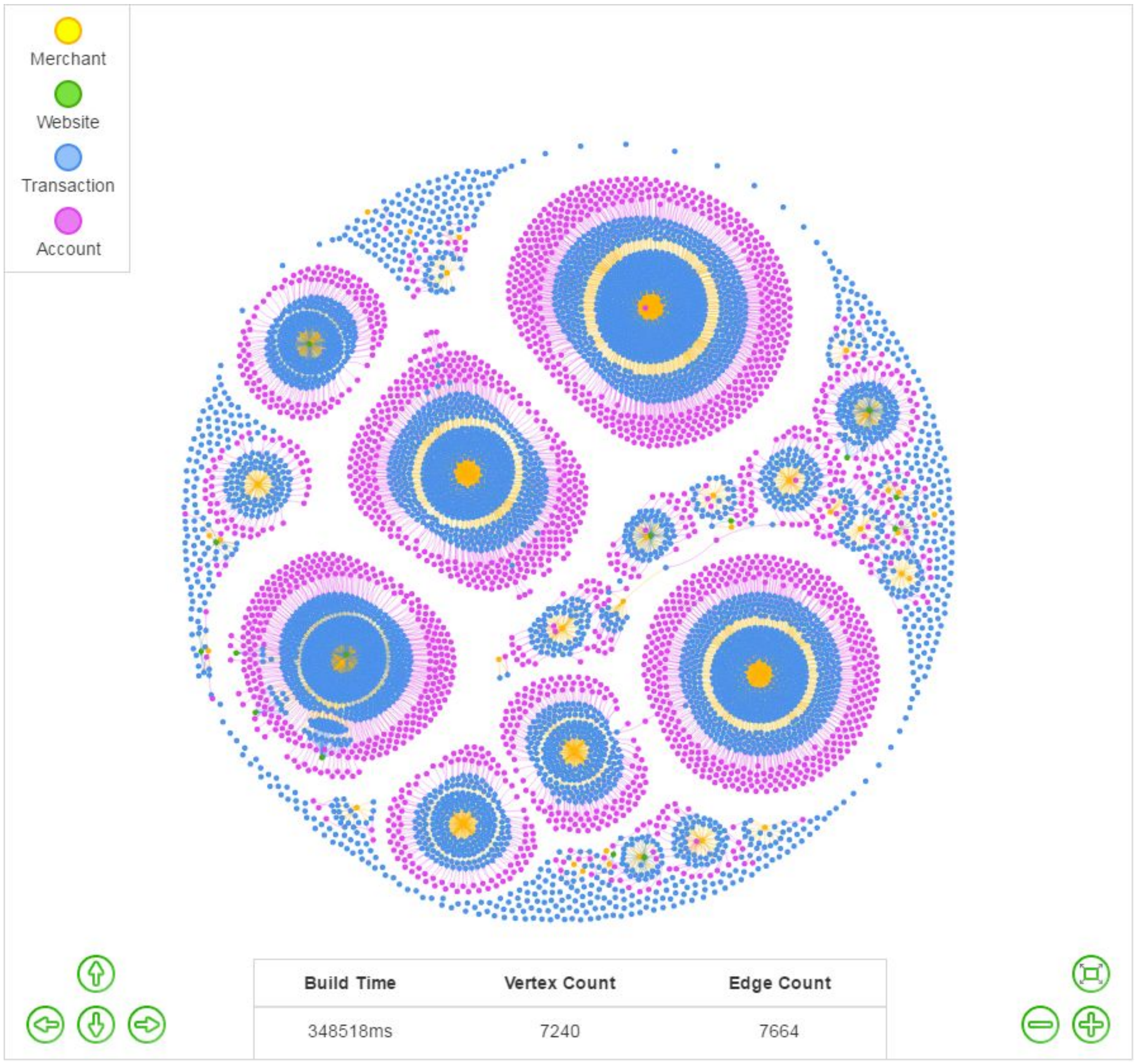
Enter Value

Edge Levels: 0

Vertex Limit: 10000

Figure 3.6 Querying page to get sub- graph based on user input

The Query Service sends this JavaScript to another web page that renders the graph. An example of a rendered graph is seen in Figure 3.7. This page allows you to interact with the graph in various ways. You can filter this graph by sending queries to it, through an interface similar to the previous page. This will highlight all nodes on the graph that meet the designated query. The dimension and property drop down menus used to build the queries are populated based on the graph's metadata which is retrieved from the metadata service. You can also manually select nodes by clicking on them. Data about selected nodes can be found in a panel on the right side of the page. Some of this data is also available when you hover over a node. There are also a variety of options on the page, so that the user can customize the graph in order to find patterns they could not otherwise find. Some of these options include coloring or sizing the nodes by various properties.



Build Time	Vertex Count	Edge Count
348518ms	7240	7664

Figure 3.7 Visualization page example graph

3.7 Titan Cassandra Configuration

Titan is configured using Cassandra as its storage backend, and uses Elasticsearch as its indexing backend. The graph database is an abstraction of the underlying Cassandra database.

The entire configuration is a cluster of nodes, each running Cassandra, Elasticsearch, as well as a Rexster server which is necessary to provide remote access to Titan. The configuration can be seen in Figure 3.8. The reason for hosting a Rexster server on each client is to provide a local connection between the server and the Cassandra database regardless of which node a user is trying to communicate with.

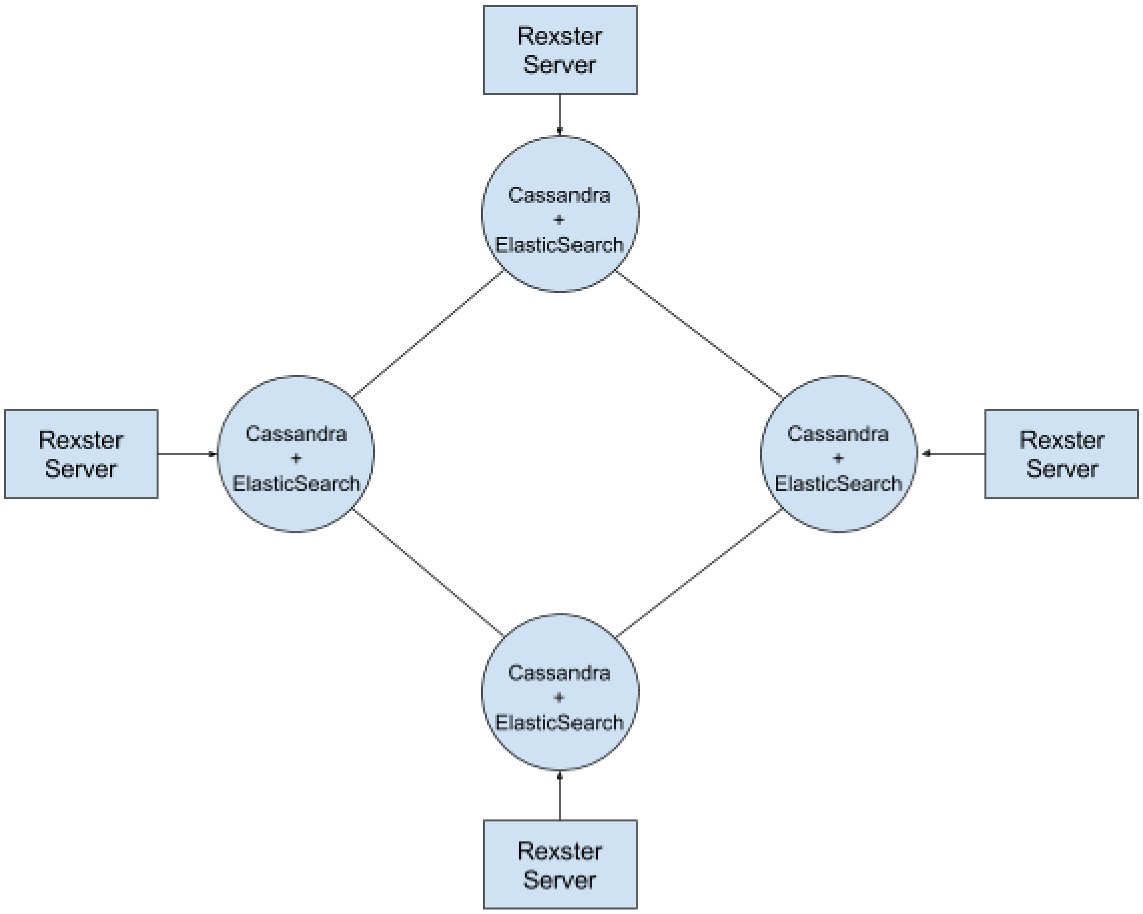

Rexster Server
Cassandra + ElasticSearch
Rexster Server
Cassandra + ElasticSearch
Cassandra + ElasticSearch
Rexster Server
Cassandra + ElasticSearch
Rexster Server

Figure 3.8 Cluster configuration for Cassandra, Elasticsearch and Rexster

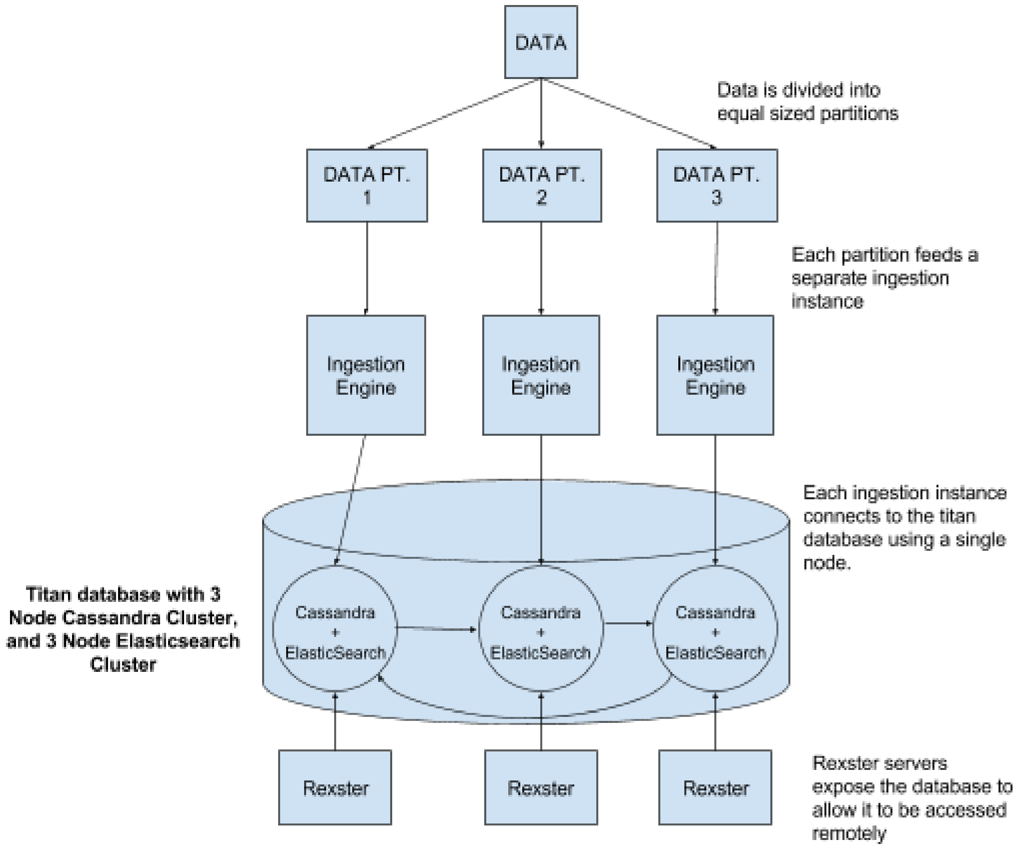

DATA
Data is divided into equal sized partitions
DATA PT. 1
DATA PT. 2
DATA PT. 3
Each partition feeds a separate ingestion instance
Ingestion Engine
Ingestion Engine
Ingestion Engine
Each ingestion instance connects to the titan database using a single node.
Titan database with 3 Node Cassandra Cluster, and 3 Node Elasticsearch Cluster
Cassandra + ElasticSearch
Cassandra + ElasticSearch
Cassandra + ElasticSearch
Rexster
Rexster
Rexster
Rexster servers expose the database to allow it to be accessed remotely

Figure 3.9 Diagram of the simultaneous ingestion process feeding into the Cassandra cluster

As described earlier, during ingestion each node can be configured to load a piece of the data set, allowing for concurrent additions to the graph to improve performance. A diagram of this can be seen above in Figure 3.9. All graph configurations are defined so that each ingestion engine instance will connect to the local Cassandra and Elasticsearch hosts to avoid some network latency.

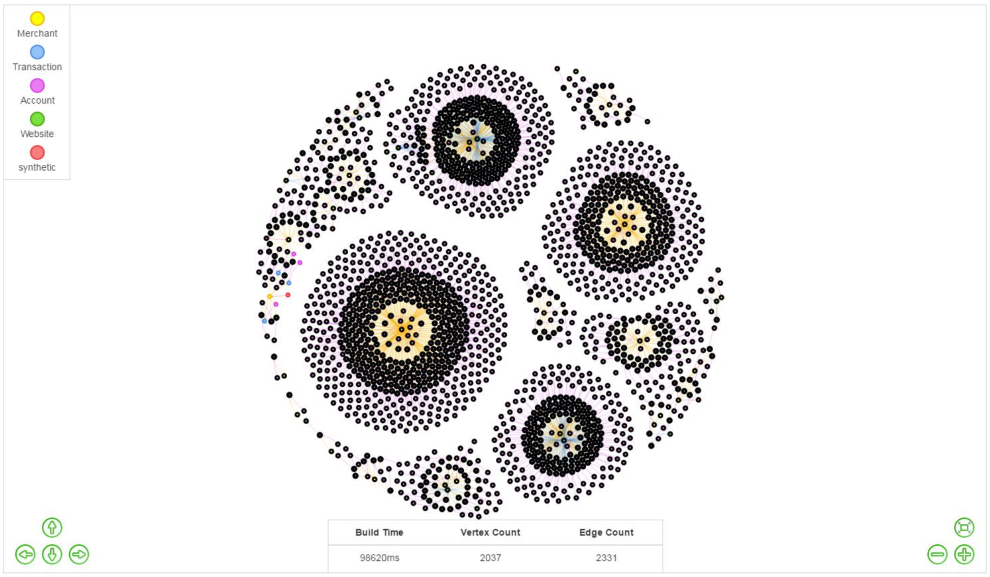
3.8 Titan-Hadoop

Titan includes its own version of Hadoop, Titan-Hadoop, which can be used to run map-reduce functions across the graph. This functionality allows for features to be computed across the graph on a per-node basis. It also means that computations can consider not only a node's properties, but a node's neighbor and its own properties. These functions allow for a wide range of potential manipulations after the data has already been ingested. Two example uses are explained in the following sections.

3.8.1 Fraud Index Calculation

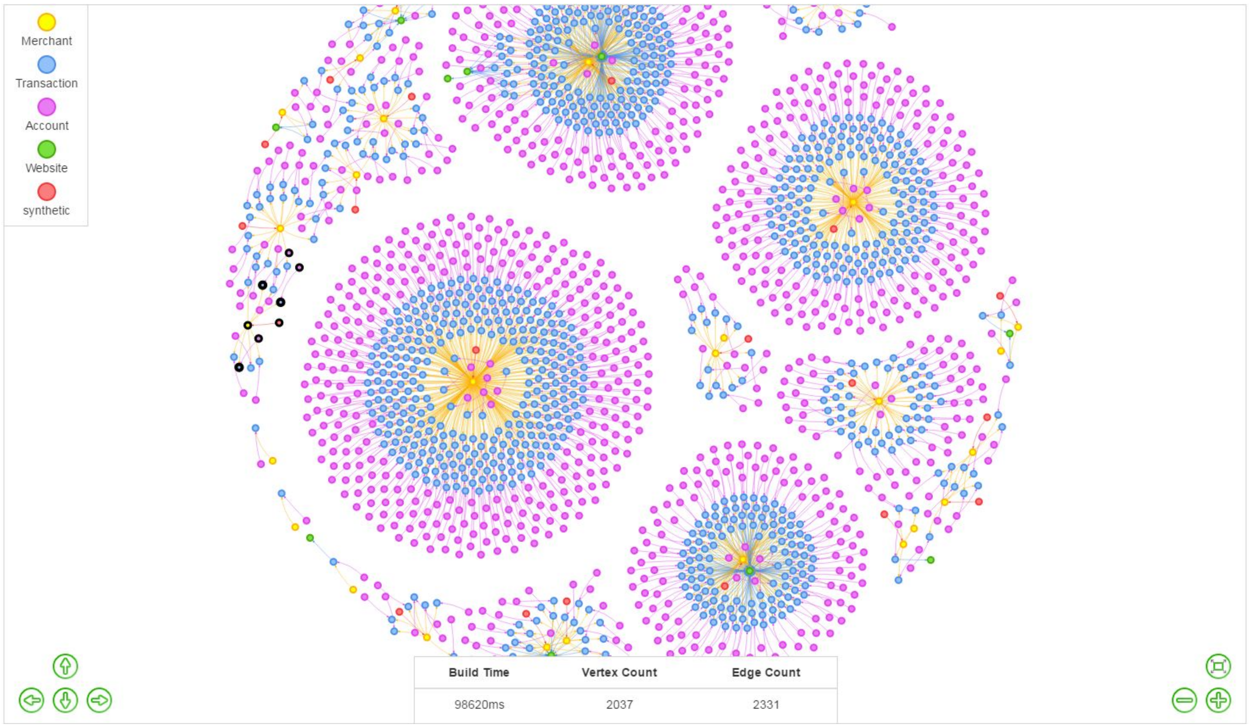
Hadoop scripts were used to determine the fraud index, the likelihood of fraud, for any given node. Using the ReD Shield Data set, a set of sanitized online transactions that were made over a 30 day period, any transaction that was deemed fraudulent has been labeled as such. The labels were as follows, a "B" meant the transaction was fraudulent, and a "G" meant the transaction was legitimate. A mapreduce job was run to analyze all of the transaction dimensions labels and specify their fraud index. Any transaction dimension labeled with a "B" was given a 1.0 fraud index rating, indicating it is guaranteed to be fraudulent. Those transactions that were labeled with a "G" were given a 0.0 fraud index, meaning it is guaranteed to be legitimate. Once all of the transaction dimensions labels have been parsed and the fraud indices specified, another job is used to average the fraud index for a node based on all of its neighbors' fraud indices. Transaction vertices do not have their indexes recalculated as they have already been definitively labeled as being fraudulent or not.

Using the query functionality it's possible to narrow down vertices based on the fraud index to find suspicious activity. Due to the nature of the data and the fraud within it, it can be hard to spot fraudulent activity because it is so infrequent compared to legitimate activity. In several thousand transactions there may be only a handful that end up being fraudulent. To contrast the difference Figures 3.10 and 3.11 show selections of fraud indices less than 0.1, and those greater than 0.1. Figure 3.12 displays a selection of fraud indices greater than 0.5. Comparing Figures 3.11 and 3.12 we lose the highlighting on the computed fraud for the Merchant and synthetic nodes. This is a disadvantage of using an average, however it could be solved by basing computations of ACI's feature computations and rules that they already have in place. Alternative calculations of the fraud index could prevent the index from being averaged down so low for vertices with a large number of edges like Merchants for example. It would provide a better spread for the fraud indices throughout the graph, which could yield more meaningful results and simpler analysis.



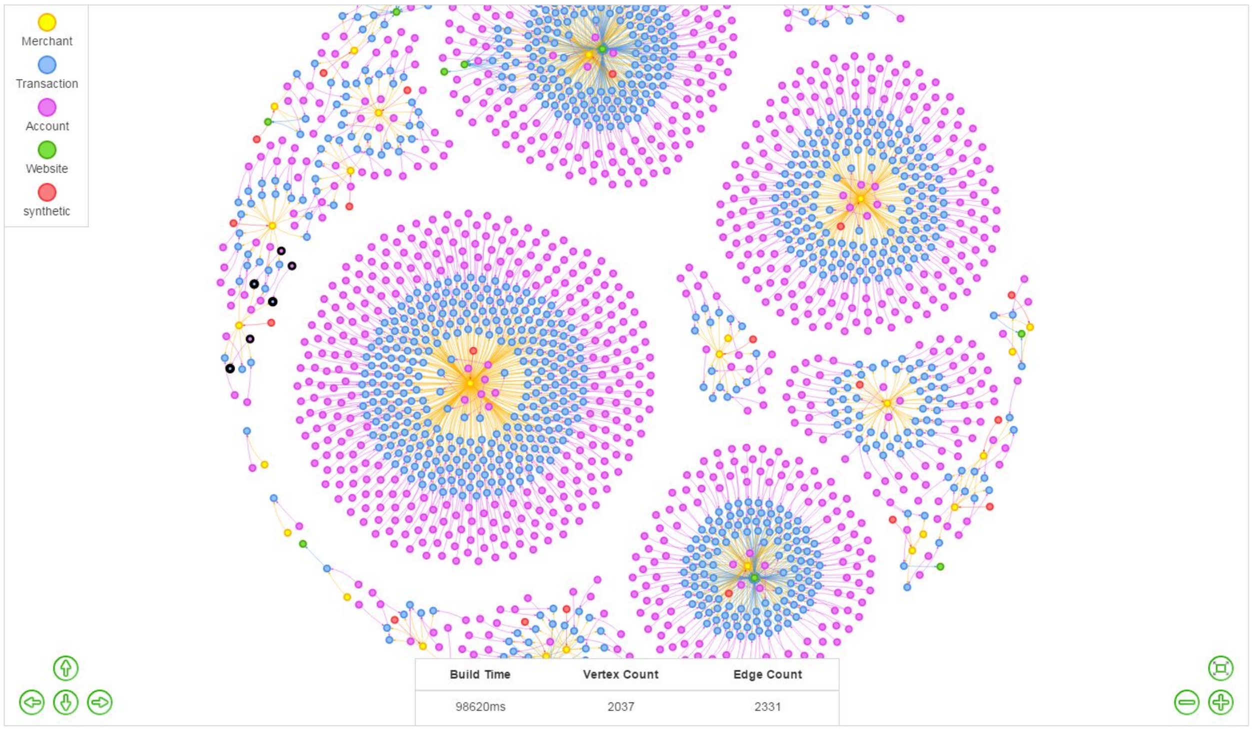
Build Time	Vertex Count	Edge Count
98620ms	2037	2331

Figure 3.10 Graph with all fraud indices < 0.1 selected (black nodes)



Build Time	Vertex Count	Edge Count
98620ms	2037	2331

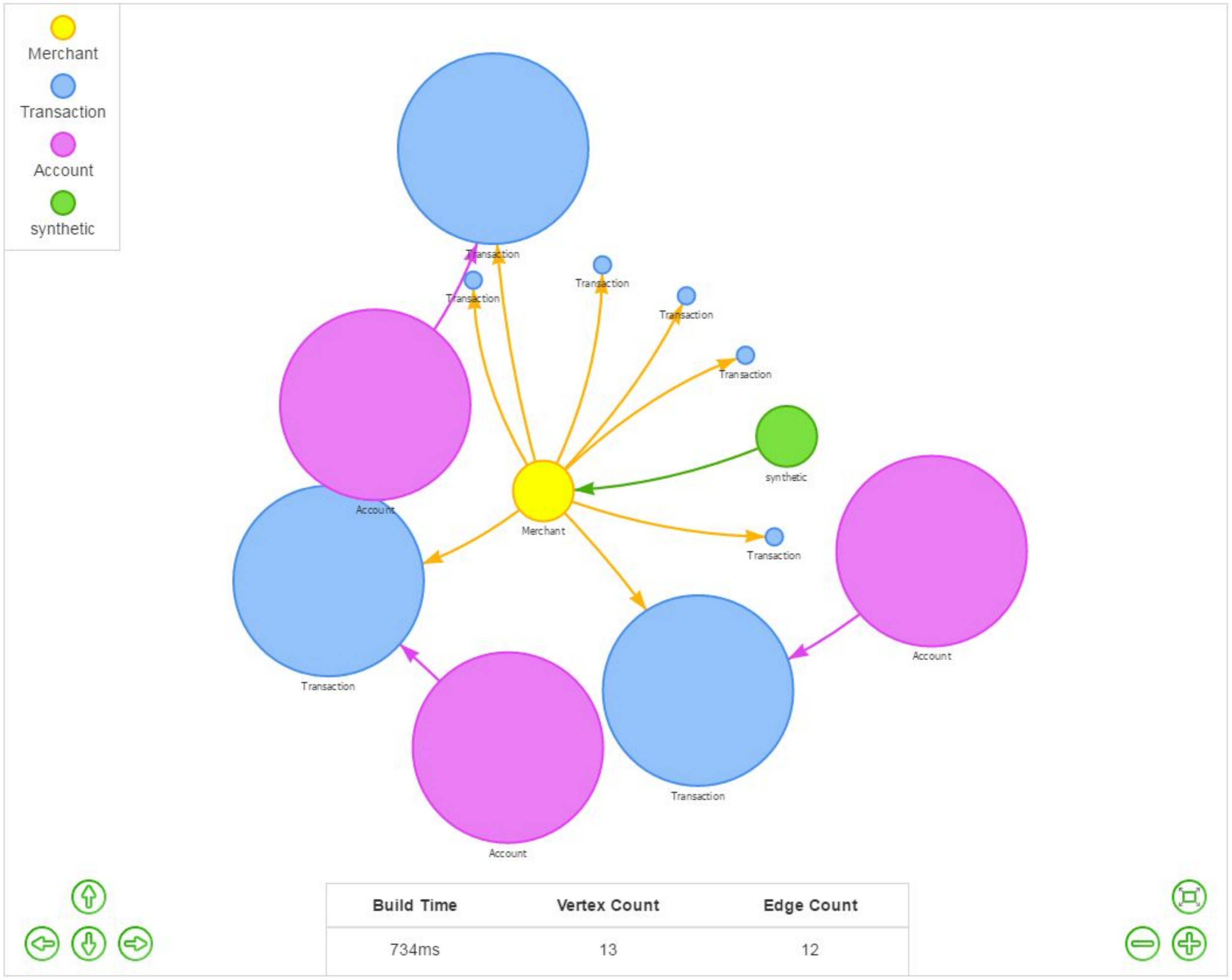
Figure 3.11. Graph with all fraud indices >0.1 selected (black nodes)



Build Time	Vertex Count	Edge Count
98620ms	2037	2331

Figure 3.12 Graph with all fraud indices >0.5 selected (black nodes)

The addition of the fraud index to the vertex properties lead to the ability to alter the graph in order to better see where fraud is occurring. These features are the ability to color all of the vertices based on their fraud index, and the ability to size nodes based on their fraud. For color, red indicates the highest levels of fraud, and green the lowest, while the middle ranges between yellow and orange. The size adjustments increase the size of vertices as their risk level increases. An example of each is below in Figure 3.13 and Figure 3.14. In Figure 3.13 the largest nodes are the most fraudulent. Each large transaction is one with a 1.0 fraud index. The neighboring accounts are also guaranteed fraudulent because they are only associated with a bad transaction. The Merchant is connected to a number of good and bad transactions, so it is not at as high as risk as the Accounts in this case. The same graph can be seen in Figure 3.14, this time with nodes colored based on their fraud. In this example the Merchant is a slightly darker green, and those nodes that are most fraudulent are a bright red to indicate their risk.



Build Time	Vertex Count	Edge Count
734ms	13	12

Figure 3.13 Graph with vertices sized based on their fraud index (larger nodes indicate higher fraud index)

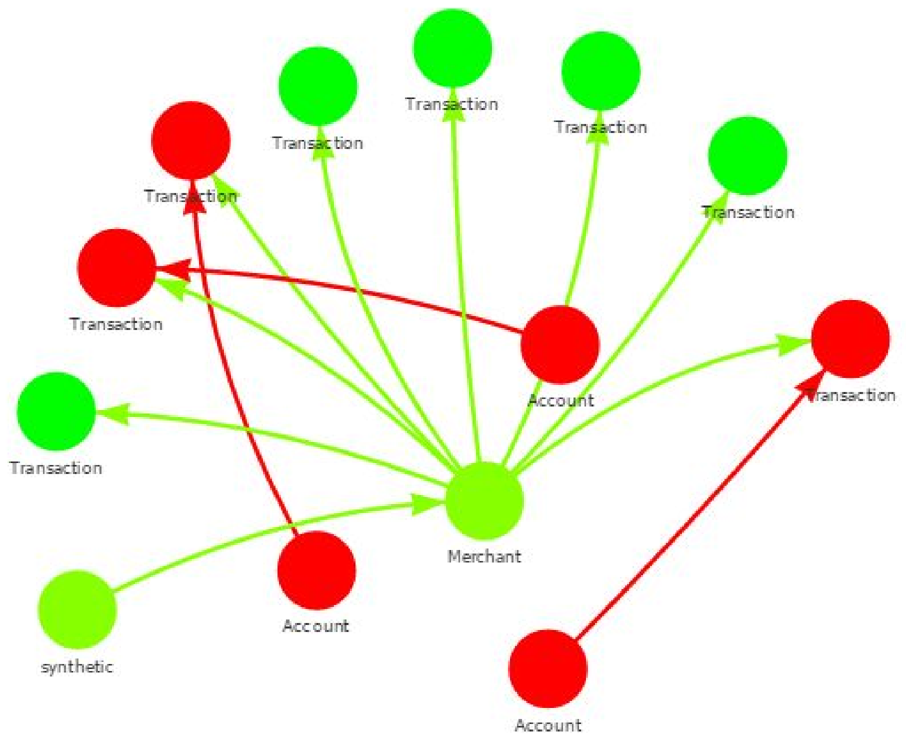
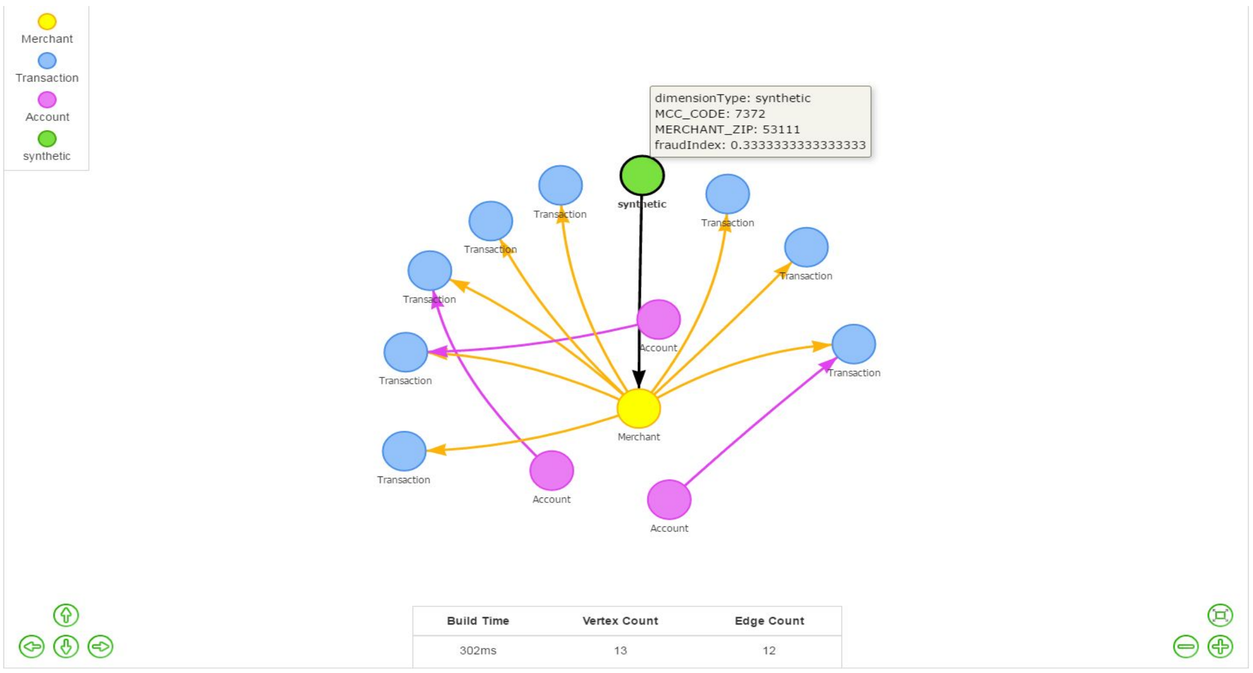


Figure 3.14 Graph with vertices colored based on their fraud index

3.8.2 Synthetic Nodes

In addition to the creation and modification of vertex properties, Titan-Hadoop can be used to create new vertices and edges. A common strategy fraud analysts use in identifying fraud is to observe not just single fields of data, but a combination of a number of fields. To implement this idea, new nodes can be created based on certain properties. A common example is to pair the MCC code and zip code and determine which pairs are at risk. In this case the job would create new synthetic nodes for all MCC codes and zip codes, and then connect these new synthetic nodes to the dimensions who own those two properties. Once the edges are created the fraud index is calculated based on each synthetic node's neighbors. From this specific example it could be determined if a specific kind of retailer was being targeted in certain areas. Any number of properties could be combined to create complex synthetic nodes to help understand the relationships within the data, and better identify fraud. A small subgraph displaying a synthetic node made up of MCC_Code and Zip Code connected to fraudulent transactions is shown below in Figure 3.15.

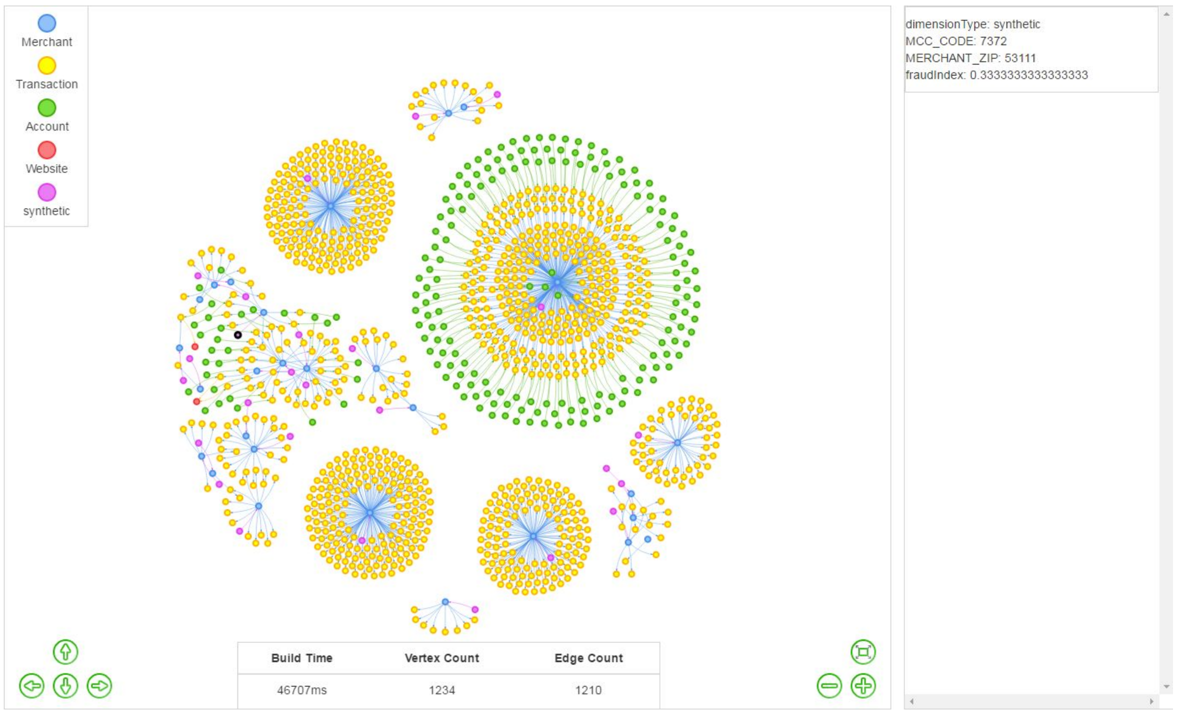


```
dimensionType: synthetic
MCC_CODE: 7372
MERCHANT_ZIP: 53111
fraudIndex: 0.3333333333333333
```

Build Time	Vertex Count	Edge Count
302ms	13	12

Figure 3.15. Graph with a synthetic node before its fraud is calculated

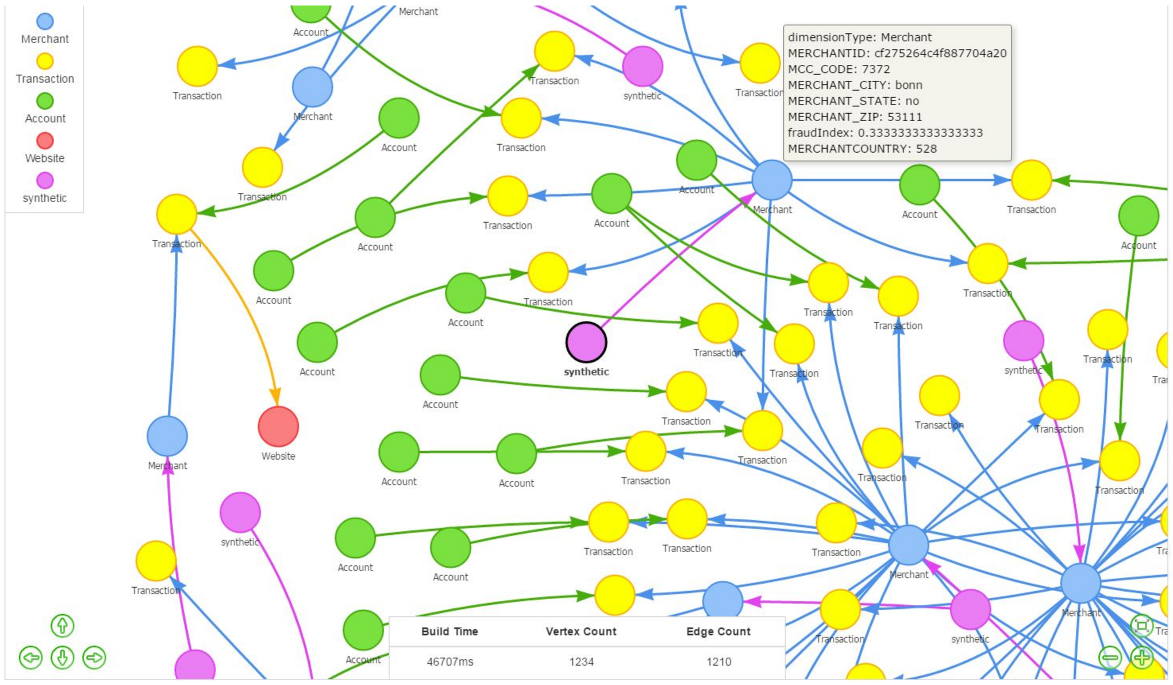
In Figure 3.16 the subgraph above is seen as part of the much larger graph. We query the graph for any synthetic node with a fraud index greater than 0.0 to find all nodes that are potentially fraudulent. We see the at risk node highlighted, and with its properties listed on the right. Figure 3.17 shows an alternate view of the graph, displaying the highlighted synthetic node which connects to a merchant with the same properties (blue node). By focusing on the synthetic node we created, we can see the associated merchant and that merchant's fraud index.



dimensionType: synthetic
MCC_CODE: 7372
MERCHANT_ZIP: 53111
fraudIndex: 0.3333333333333333

Build Time	Vertex Count	Edge Count
46707ms	1234	1210

Figure 3.16. Graph with fraudulent synthetic nodes highlighted



dimensionType: Merchant
MERCHANTID: cf275264c4f887704a20
MCC_CODE: 7372
MERCHANT_CITY: bonn
MERCHANT_STATE: no
MERCHANT_ZIP: 53111
fraudIndex: 0.3333333333333333
MERCHANTCOUNTRY: 528

Build Time	Vertex Count	Edge Count
46707ms	1234	1210

Figure 3.17. Graph with fraudulent synthetic nodes, zoomed in on highlighted node

3.9 Applications and Use Cases

The ability to visualize transaction data allows users to see relationships in data and be able to accurately identify fraud. A user could query our graph database using the web interface to find all customers and see everything that they are connected to. These customers could have relationships with other customers, accounts, transactions, and vendors. Once a user has queried for customers, they will be able to see a visualized graph of all the relationships that the customers have. They could then pinpoint a single customer and see their individual relationships, along with any calculated fraud index that is associated with them.

For example, if a user knew that a customer was fraudulent, they could query the graph database for that customer's name or account number. They could specify how many edge levels out they wanted, in order to see who or what is related to this customer. When this query is visualized, the user would be able to search that graph for the specific customer and also be able to see their relationships. Using the fraud index provided with the transaction data, the system can also modify the color and size of the nodes based on it. These modifications can make it clear to the analyst which elements are fraudulent.

3.9.1 Users

There are three different kinds of users who will be using GraphView: admins, data scientists, and fraud analysts. Admins are responsible for ensuring that the Titan database is functioning properly at all times. They would have the capabilities to amend any issues that may occur during use, specifically during Event Ingestion. Second are data scientists, users that are observing the data but not specifically to analyze fraud. Data scientists would have the ability to improve the system by adding new features, and by creating and applying machine learning models. Data scientists would also be able to view the graph, query it, manage the metadata, and import external data sets. Lastly are the fraud analysts. The analysts will be able to query the graph, view portions of the data as a subgraph, and apply available models to the dataset. See Figure 3.18 for a diagram of the current capabilities each kind of user will have.

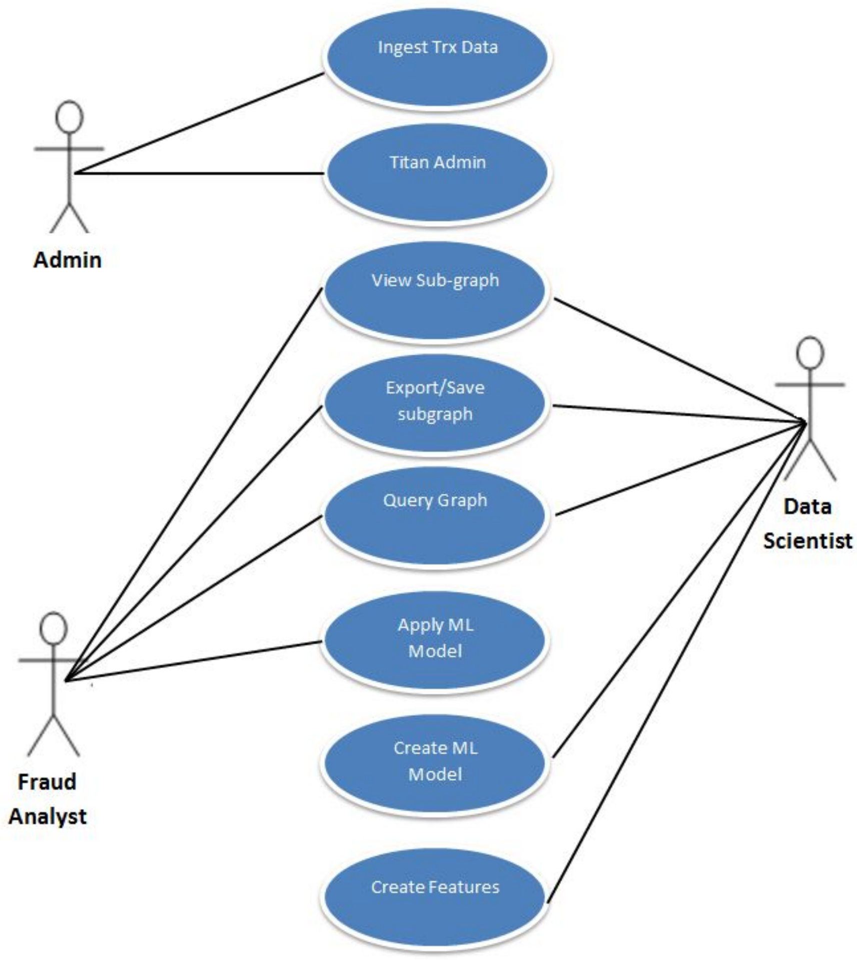


Figure 3.18 User privileges

4. Evaluation

For the project to be successful we need to provide a tool that not only allows for novel analysis of the data, but one that is also accessible and useful to ACI's customers. There are a number of performance factors to address as well. This section addresses our metrics for success, as well as the results of our testing.

To analyze the results we test the different components using varying data subsets from the ReD Shield Transaction data set. The Ingestion Engine, Query Service, and Visualization Service are each tested separately to evaluate their performance and provide insight into how the cluster sizes, number of transactions, and indices affect the results.

4.1 Metrics

The Ingestion Engine, Query Service, and Visualization Service are the components that require performance testing.

4.1.1 Ingestion Engine

For the ingestion engine the primary concern is the time it takes to ingest a specific dataset. To prove that the system is horizontally scalable it is necessary to display some performance increase as cluster sizes are scaled up. It is also necessary to show that the system is capable of ingesting large datasets. To confirm these, the system will be evaluated based on different sized data sets and different sized clusters.

4.1.2 Query Service

For the query service the most important factor is how long it takes for the query to complete. So what the total time is for the service to submit the query and receive a response from the database, the application latency. These tests are completed on different sized clusters and different graph sizes to examine the performance differences. As the cluster size increases the performance is expected to improve as well.

4.1.3 Visualization Service

The visualization service has a performance metric in terms of how quickly the graph can be displayed once a query has been submitted. The most import metric for the visualization however is based on testimonials from potential end users. Fraud analysts, modelers, and other potential users already have some tools and practices in place to analyze transaction data and detect fraud. Their opinion is important in determining if this tool would make identifying fraud or analyzing transactions in general easier, or if it would extend their abilities in any way.

4.2 Setup

For the complete evaluation the system was tested using a sanitized data set from ReD Shield consisting of labeled transaction data. The system was deployed on clusters of varying sizes on ACI's machines. The ingestion process was timed under different combinations of cluster size, and number of transactions. Once each ingestion test was complete the performance of the Query Service and Visualization Service was tested on the ingested graph. This evaluated the performance of each piece of the pipeline for different cluster configurations, and number of transactions.

4.2.1 ReD Shield Data and Metadata Configuration

The ReD shield data is a set of online transactions made over a 30 day period. The data was sanitized of any personally identifiable information to protect the customers. The data is formatted such that transactions are separated by line, and that fields within each transaction are specified by a certain offset within the line, and a length.

The metadata was defined such that there are five dimensions (vertices) and five edges associated with each transaction. Figure 4.1 below displays the configuration of edges and vertices for a single transaction. For each transaction, there is some account making the purchase, signified by the edge directed from the Account to the Transaction. The account contains both account and customer information.. Finally edges are created from the transaction to the merchant where the purchase was made, and the specific website where the purchase was made. Each dimension has a number of additional properties that provide more useful information. In total the metadata converts 33 fields from the transaction data into nodes, edges and their properties. Table 1 in Appendix C describes some of the more important fields that are considered during ingestion and specifies which dimension they belong to.

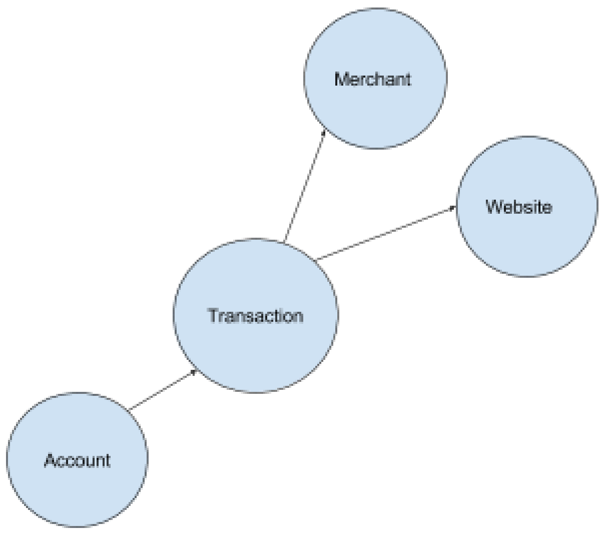


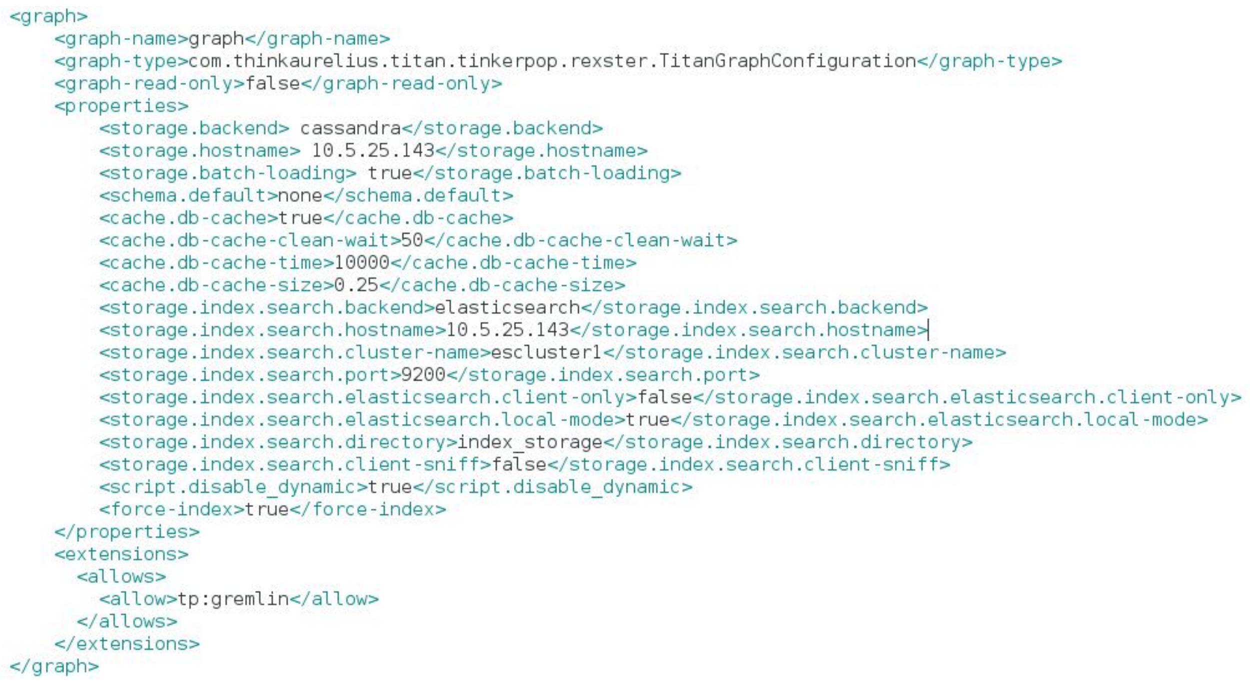
Figure 4.1 Diagram of vertex and edge configuration for a single transaction (directed edges indicate from->to relationship)

4.2.2 Cluster Configuration

The evaluation was performed on three different sized clusters consisting of one, two, and six nodes. Each node was configured to run Cassandra, Elasticsearch, and Rexster.

For each Cassandra cluster each node was allotted the recommended 256 tokens, and each cluster was given one seed node. Paired with each Cassandra node was an Elasticsearch instance. All of the Elasticsearch instances were then clustered together. The Elasticsearch configuration and Cassandra configurations can be found in Appendix B.

Multiple Rexster servers were hosted per cluster to not only provide more availability, but to ensure local host access from any Rexster instance to the database. The specific graph configuration can be found below, and the full configuration file can be found in Appendix B.



```
<graph>
    <graph-name>graph</graph-name>
    <graph-type>com.thinkaurelius.titan.tinkerpop.rexster.TitanGraphConfiguration</graph-type>
    <graph-read-only>false</graph-read-only>
    <properties>
        <storage.backend> cassandra</storage.backend>
        <storage.hostname> 10.5.25.143</storage.hostname>
        <storage.batch-loading> true</storage.batch-loading>
        <schema.default>none</schema.default>
        <cache.db-cache>true</cache.db-cache>
        <cache.db-cache-clean-wait>50</cache.db-cache-clean-wait>
        <cache.db-cache-time>10000</cache.db-cache-time>
        <cache.db-cache-size>0.25</cache.db-cache-size>
        <storage.index.search.backend>elasticsearch</storage.index.search.backend>
        <storage.index.search.hostname>10.5.25.143</storage.index.search.hostname>
        <storage.index.search.cluster-name>escluster1</storage.index.search.cluster-name>
        <storage.index.search.port>9200</storage.index.search.port>
        <storage.index.search.elasticsearch.client-only>false</storage.index.search.elasticsearch.client-only>
        <storage.index.search.elasticsearch.local-mode>true</storage.index.search.elasticsearch.local-mode>
        <storage.index.search.directory>index_storage</storage.index.search.directory>
        <storage.index.search.client-sniff>false</storage.index.search.client-sniff>
        <script.disable_dynamic>true</script.disable_dynamic>
        <force-index>true</force-index>
    </properties>
    <extensions>
      <allows>
        <allow>tp:gremlin</allow>
      </allows>
    </extensions>
</graph>
```

Figure 4.2 Rexster graph configuration

4.2.3 Graph Configuration

The graph configuration defines storage, index, and schema specifications, and is used to tell the ingestion engine how to open the correct graph. Graph configurations are also how graphs can be accessed locally using the gremlin shell. Each graph is configured the same, varying only the hostnames for the indexing backend and storage backend to connect to the correct cluster.

4.2.4 Software Versions

Software	**Version Number**	**License**
Titan	0.5.4	Apache 2.0
Cassandra	2.0.8	Apache 2.0
Elasticsearch	1.2.1	Apache 2.0
Rexster	2.6.0	Apache 2.0
Vis.js	4.15.1	Apache 2.0 and MIT

4.3 Results

All of the results are based on the setup and configurations described above. Ingestion Engine and Query Service testing was done to evaluate performance times for a number of different configurations. The visualization service was tested using graphs of varying vertex and edge sizes.

It is important to note that the ACI machines use a Storage Area Network (SAN). SANs are not recommended for use with Cassandra as they create a single point of failure, and the overall performance suffers[27]. Unfortunately there was no alternative for deploying a cluster so the following performances are not ideal.

[27] Apache Cassandra, Anti-Patterns in Cassandra, https://docs.datastax.com/en/cassandra/2.1/cassandra/planning/architecturePlanningAntiPatterns_c.html

4.3.1 Ingestion Engine Performance Testing

In the table below are the results of the ingestion performance testing. The ingestion testing proves the hypothesis that the larger the cluster, the faster data can be ingested. The cluster is used to accomplish simultaneous loading which shows significant performance increases as the cluster size grows. Looking at the one million transaction ingestions, we have ingestion times of 28,753 seconds, 17,505 seconds, and 6,314 seconds for the one, two, and six node clusters respectively. Comparing these results, the six node cluster is over 4.5 times faster than the single node cluster, and over 2.7 times faster than the two node cluster. This is important to show that the system is indeed horizontally scalable and that ingestion can be accomplished in a reasonable timeframe even for large datasets. It's important to note that these time increases are in spite of the SAN that the clusters are built on. Deploying Cassandra on a cluster where each node can have a separate, local disk would surely provide even greater performance because the locality of data removes the need for large volumes of data to be transferred over the network. All results can be found in Table 1 below.

# Transactions	Total # Vertices	Total # Edges	# Nodes in Cluster	Time to ingest (ms)	Time to ingest (s)
1000	2702	4331	1	18152 ms	18.1 s
1000	2702	4331	2	27639 ms	27.6 s
1000	2702	4331	6	15141 ms	15.1s
10000	24102	42403	1	300305ms	300.3s
10000	24102	42403	2	178598 ms	178.6s
10000	24102	42403	6	74216 ms	74.2s
100000	211593	435515	1	2919918ms	2919.9s
100000	211593	435515	2	1824366 ms	1824.4s
100000	211593	435515	6	621816 ms	621.8s
1000000	1942614	4404349	1	28753458ms	28753.5s
1000000	1942614	4404349	2	17505452ms	17505.4s
1000000	1942614	4404349	6	6314416ms	6314.4s

Table 1: Results of Ingestion Engine Performance Testing

4.3.2 Query Service Performance Testing

The query testing was done on each of the previously ingested graphs. There are two major types of queries a user can run. There are queries that find a set of vertices, and there are queries that find a set of vertices and traverse a certain number of edges out from that set to find neighbors. We performed indexed and non-indexed tests for each of these, running the same 4 queries for all of the graphs. All of these results can be found in Table 2 below.

The results of this testing show the significant improvement that having properties indexed provides compared to non-indexed properties. For all except the indexed vertex set query, there is a trend of query times increasing as the cluster size increases. This was expected due to hosting the cluster on a SAN. We do however see some strange variation for the indexed vertex set queries. For the 2 node cluster the scaling advantage of the clustered indexing backend did not provide enough of a performance increase to overcome the disk sharing issue. This explains how the 1 node cluster can outperform the 2 node cluster due to its lack of disk sharing issues, yet the 6 node cluster can still provide the best performance.

# Transactions	Total # Vertices	Total # Edges	# Nodes in Cluster	Non-indexed Vertex Set Query Time (ms)	Indexed Vertex Set Query Time (ms)	Non Indexed Traversal Query Time (ms)	Indexed Traversal Time (ms)
1000	2702	4331	1	538	176	2355	603
1000	2702	4331	2	1307	183	7619	719
1000	2702	4331	6	3854	127	11795	736
10000	24102	42403	1	4421	197	54418	1995
10000	24102	42403	2	9798	216	81030	2554
10000	24102	42403	6	20759	143	128344	2864
100000	211593	435515	1	43372	483	114586	921
100000	211593	435515	2	118744	41	471346	127
100000	211593	435515	6	162925	39	480386	69
1000000	1942614	4404349	1	425277	803	848485	1642
1000000	1942614	4404349	2	1080917	1061	2072912	1976
1000000	1942614	4404349	6	1512667	59	2815516	266

Table 2 Results of Query Service Performance testing

4.3.3 Visualization Service Performance Testing

Visualization testing is concerned only with the time it takes to display the graph once the information has been received from the query service. The querying process from the UI is excluded because the queries have been evaluated separately, and the only additional factor is network latency, which has minimal effect. Not surprisingly, the time to build the graph scales significantly as the number of vertices and edges increases. Results can be found in Table 3.

Number of Vertices	Number of Edges	Time to Visualize (ms)
100	83	3623
500	488	16584
1000	1014	41932

Table 3 Results of Visualization Service Performance testing

4.4 Subject Expert Assessments

We spoke to ACI employees to get their inputs into how our system would be useful to them. Our evaluation sessions were conducted at ACI using either video conferencing software or at ACI in person. Each session was about an hour long and started off with a demonstration of GraphView. Once we explained and demonstrated our system, we asked for their overall opinions of the tool, how they thought it might be useful to them, and what features they would like to see. These discussions provided important feedback on the usefulness of GraphView and led to some novel ideas, some of which we were able to implement.

We first spoke with Cleber Martin who gave us the initial idea of creating synthetic nodes. He also had us show how we can query by only transactions that are known to be fraudulent and also for transactions that are less than a certain value. Cleber said that being able to immediately filter out non-fraudulent transactions would be helpful for identifying nodes related to fraud.

We next spoke to Joe Lividini and Sanjay Dodhia who gave us the idea of being able to ‘bucket’ transaction nodes that may not be useful for analysis. This is explained later in the Future Work section, but ‘bucketing’ would be consolidating similar nodes into a singular node in order to reduce the number of nodes on screen. They also asked whether the metadata we were using was able to be modified in order to allow analysts to select the data they wanted to see, not just the data that we had selected. We confirmed that the metadata can be changed to fit the needs of analysts and they said this would be helpful. Modifiable metadata would allow analysts to add and remove from the metadata that we have defined and also allow them to define metadata for any features they may add.

Lastly we spoke to Thea Ghiselli-Crippa, Linda Mesinger Nunez and Andrew Morse, who are all data scientists at ACI. They were interested in how we were computing our threat indices and also had a few suggestions for future work. They liked how we compute the threat index based on a node’s neighbors and thought that this would lead to an accurate calculation. A suggestion they had was to include a weight for some of the nodes so when fraud indices are being calculated, some nodes will affect the calculation more than others. The last suggestion they had was to modify the width of edge levels in order to express more data, such as a number of transactions from similar dimensions.

5. Conclusion

5.1 Project Summary

We accomplished the initial goals that we set for this project. Using big data analysis and distributed computing systems, we created a distributed graph database to store financial transactions. We also created the Ingestion Engine, Query Service, Metadata Manager, and Visualization Service to help the analysts complete their tasks. All of these components come together to form our solution, GraphView. With GraphView, analysts will have an easier time finding trends and patterns in the data. Our system will potentially allow analysts to find new patterns that they could not find using row-and-column based data. We introduced big data graph technology to ACI Worldwide, which opens the possibility that it will continue to be used in other projects in the future. While our system is a proof of concept, it acts as a stepping stone that could lead to further innovations that revolutionize the way fraud is found in financial transactions.

5.2 Teamwork

For the entirety of our project, we contacted our project advisor Eric Gieseke weekly. We would usually meet onsite at ACI Worldwide but occasionally spoke and met using video conferencing software. During these meetings, our team would present a powerpoint and demonstration to Mr. Gieseke, our WPI advisor Professor Rundensteiner, and any other ACI employees who were interested. We would discuss what we accomplished the previous week, any roadblocks that we hit, and questions we had about going forward. We frequently brainstormed when looking for solutions which is where concepts such as the Metadata Manager came from. As the project came to an end, we spoke to ACI employees in order for them to critique GraphView and these discussions would occur in small scale focus groups.

5.3 Future Work

After the completion of our project there were a few areas of work that we thought could be built upon. One area was adding more features to the visualization page. In our prototype the graph can be visualized, dragged, and queried but there are other features that may be useful. The ability to hide nodes and improving the formatting of query results are changes that could provide additional functionality. Hiding nodes would allow analysts to ignore some of the nodes and be able to focus on those that they most care about. Along with hiding nodes, the idea of 'bucketing' similar nodes would also allow the graph to be slimmed down and present the analyst with only the nodes they wanted to see. This would enable analysts to compress a group of uninteresting similar nodes into a single, visual node.

While we implemented a table to show query results, the display of the table could be modified so the results can be seen more clearly. This would utilize a sorting function to allow the user to sort by properties. While we are able to query some of the graph properties with any predicate operator, other properties, like date and time, currently only allow equality predicates. An analyst cannot find transactions within a desired range, they can only search for transactions on the exact time or date. Extending the functionality to enable analysts to specify a range of time would allow them to more easily focus on the most relevant transactions.

Next, display times could be designed to show how transactions occurred over time. This could be implemented by either making the transactions that occurred earliest smaller and then increase in size as they became closer to the future, or by using a different physics engine that allows for some arranging by transaction properties, or by using line thickness to denote relative time between transactions. In our current prototype, we similarly modified the nodes based on the fraud index, but modifying the nodes' visual properties based on time would require some way relating the date and time properties, as mentioned above.

Our graph currently only contains transaction data, but in the future it could be expanded to include data taken from social media or geographical location. Using social media data, such as Facebook friends, could allow more relationships to be formed. This would possibly allow fraud to be more easily detected in the graph. Geographic data from the transaction data could be used to overlay the data on a map. This would allow analysts to more easily visualize where different transactions occurred. This could allow for more insight into whether or not a transaction is fraudulent. For example, if a card was used in one part of the United States and then used on the other side of the country, this card would most likely be compromised.

Another concept that could be expanded upon would be the synthetic nodes. We showed that we could make synthetic nodes out of multiple properties, but, as of now, they can only be created by an administrator, someone with direct access to the Titan database. Future projects could focus on extending this functionality to the fraud analyst through the graphical interface. This way they would not have to know how to program or modify the system, they could easily create synthetic nodes based on the specific properties they want to better examine.

The last recommendation for future work on this project would be to update the technologies that are involved in the pipeline. One example of this would be our implementation uses Titan 0.5.4. During the lifespan of our project, Titan 1.0 was released, but we did not update our system to this latest version. In the future, all technologies could be updated to their latest versions to ensure that the system is secure and will run optimally.

6. Bibliography

Agarwal, Udit & Singh, Umesh Pal. 2009. *Graph theory*. http://common.books24x7.com.ezproxy.wpi.edu/toc.aspx?bookid=34046.

Apache. 2015. *Anti-Patterns in Cassandra*. https://docs.datastax.com/en/cassandra/2.1/cassandra/planning/architecturePlanningAntiPatterns_c.html

Apache Cassandra (version 2.0.8). 2015. Apache. http://cassandra.apache.org/

Apache Kafka. 2015. Apache. http://kafka.apache.org/

Apache Lucene. 2015. Apache. http://lucene.apache.org/solr/

Apache Storm. 2015. Apache. http://storm.apache.org/

Apache ZooKeeper. 2016. Apache. https://zookeeper.apache.org/

Aurelius. 2014. *Titan Documentation.* http://s3.thinkaurelius.com/docs/titan/0.5.4/index.html

Bolton, R. J., & Hand, D. J. 2002. *Statistical Fraud Detection: A Review. Statistical Science*, 235-249.

Brath, R., & Jonker, D. 2015. *In Graph analysis and visualization discovering business opportunity in linked data.* http://proquest.safaribooksonline.com.ezproxy.wpi.edu/book/databases/business-intelligence/9781118845875/chapter-1-why-graphs/h1_845844c01_0002_html?uicode=wpi

Elasticsearch (version 1.2.1). 2015. Elasticsearch. https://www.elastic.co/

Federal Trade Commission. 2015, February. *Consumer Sentinel Network Data Book for January - December 2014.* https://www.ftc.gov/reports/consumer-sentinel-network-data-book-january-december-2014

Li, T. & Pham, J. & Zhu, J. 2015. WPI. *WPI Major Qualifying Project: Fraud Detection Using Storm.*

Institute of Internal Auditors. 2012. *Standards.* https://na.theiia.org/standards-guidance/Public%20Documents/IPPF%202013%20English.pdf

Jouili & Vansteenberghe, 2013. *An empirical comparison of graph database.* http://www.odbms.org/wp-content/uploads/2014/05/an-empirical-comparison-of-graph-databases.pdf

Mallette, S. 2014, May 29. Aurelius. *Powers of Ten - Part 1*.
http://thinkaurelius.com/2014/05/29/powers-of-ten-part-i/

Mallette, S. 2014, June 2. Aurelius. *Powers of Ten - Part 2.*
http://thinkaurelius.com/2014/06/02/powers-of-ten-part-ii/

Rexster (version 2.6.0) 2014. Tinkerpop. https://github.com/tinkerpop/rexster/wiki/Downloads

Titan with Hadoop 2 (version 0.5.4). 2014. Aurelius. http://thinkaurelius.github.io/titan/

Vis.js (version 4.15.1). 2015. Vis.js. http://visjs.org/

Visa. 2015. *Visa Inc. Reports Fiscal Second Quarter 2015 Net Income of $1.6 billion or $0.63 per Diluted Share.*
http://investor.visa.com/news/news-details/2015/Visa-Inc-Reports-Fiscal-Second-Quarter-2015-Net-Income-of-16-billion-or-063-per-Diluted-Share/default.aspx

APPENDIX A: Evaluation Results

# Transactions	Total # Vertices	Total # Edges	# Nodes in Cluster	Time to ingest (ms)	Time to ingest (s)
1000	2702	4331	1	18152 ms	18.1 s
1000	2702	4331	2	27639 ms	27.6 s
1000	2702	4331	6	15141 ms	15.1s
10000	24102	42403	1	300305ms	300.3s
10000	24102	42403	2	178598 ms	178.6s
10000	24102	42403	6	74216 ms	74.2s
100000	211593	435515	1	2919918ms	2919.9s
100000	211593	435515	2	1824366 ms	1824.4s
100000	211593	435515	6	621816 ms	621.8s
1000000	1942614	4404349	1	28753458ms	28753.5s
1000000	1942614	4404349	2	17505452ms	17505.4s
1000000	1942614	4404349	6	6314416ms	6314.4s

Table 1: Results of Ingestion Engine Performance Testing

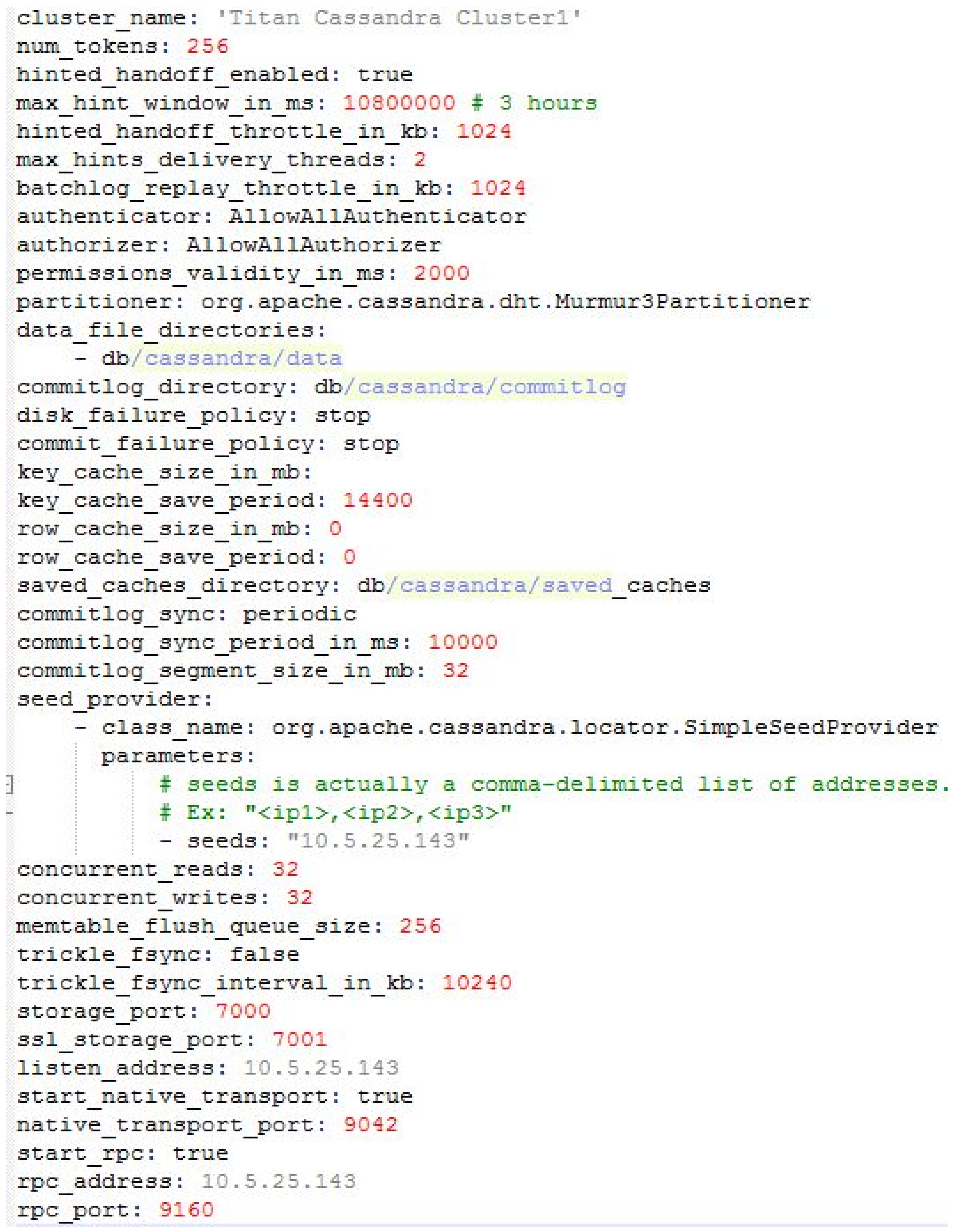
# Transactions	Total # Vertices	Total # Edges	# Nodes in Cluster	Non-indexed Vertex Set Query Time (ms)	Indexed Vertex Set Query Time (ms)	Non Indexed Traversal Query Time (ms)	Indexed Traversal Time (ms)
1000	2702	4331	1	538	176	2355	603
1000	2702	4331	2	1307	183	7619	719
1000	2702	4331	6	3854	127	11795	736
10000	24102	42403	1	4421	197	54418	1995
10000	24102	42403	2	9798	216	81030	2554
10000	24102	42403	6	20759	143	128344	2864
100000	211593	435515	1	43372	483	114586	921
100000	211593	435515	2	118744	41	471346	127
100000	211593	435515	6	162925	39	480386	69
1000000	1942614	4404349	1	425277	803	848485	1642
1000000	1942614	4404349	2	1080917	1061	2072912	1976
1000000	1942614	4404349	6	1512667	59	2815516	266

Table 2 Results of Query Service Performance testing

Number of Vertices	Number of Edges	Time to Visualize (ms)
100	83	3623
500	488	16584
1000	1014	41932

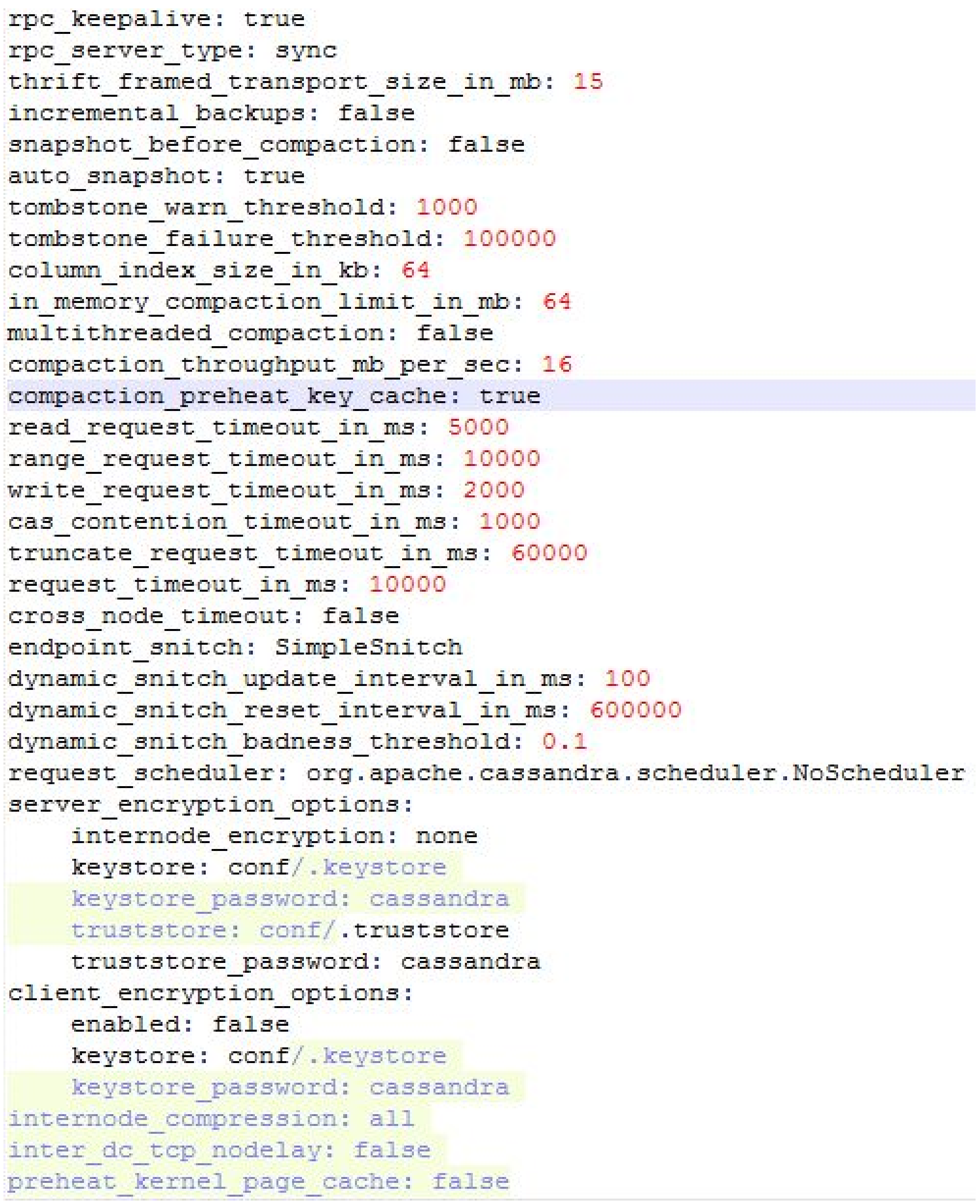
Table 3 Results of Visualization Service Performance testing

APPENDIX B: Tool Configurations



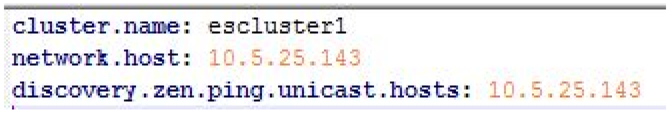
```
cluster_name: 'Titan Cassandra Cluster1'
num_tokens: 256
hinted_handoff_enabled: true
max_hint_window_in_ms: 10800000 # 3 hours
hinted_handoff_throttle_in_kb: 1024
max_hints_delivery_threads: 2
batchlog_replay_throttle_in_kb: 1024
authenticator: AllowAllAuthenticator
authorizer: AllowAllAuthorizer
permissions_validity_in_ms: 2000
partitioner: org.apache.cassandra.dht.Murmur3Partitioner
data_file_directories:
    - db/cassandra/data
commitlog_directory: db/cassandra/commitlog
disk_failure_policy: stop
commit_failure_policy: stop
key_cache_size_in_mb:
key_cache_save_period: 14400
row_cache_size_in_mb: 0
row_cache_save_period: 0
saved_caches_directory: db/cassandra/saved_caches
commitlog_sync: periodic
commitlog_sync_period_in_ms: 10000
commitlog_segment_size_in_mb: 32
seed_provider:
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
          # seeds is actually a comma-delimited list of addresses.
          # Ex: "<ip1>,<ip2>,<ip3>"
          - seeds: "10.5.25.143"
concurrent_reads: 32
concurrent_writes: 32
memtable_flush_queue_size: 256
trickle_fsync: false
trickle_fsync_interval_in_kb: 10240
storage_port: 7000
ssl_storage_port: 7001
listen_address: 10.5.25.143
start_native_transport: true
native_transport_port: 9042
start_rpc: true
rpc_address: 10.5.25.143
rpc_port: 9160
```

Figure 1. Cassandra Configuration Part 1. (continued in next figure)



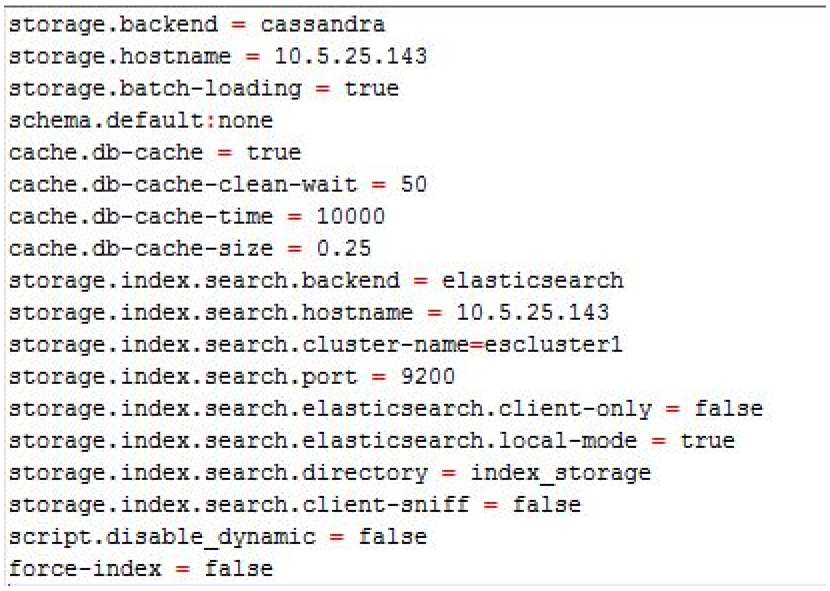
```
rpc_keepalive: true
rpc_server_type: sync
thrift_framed_transport_size_in_mb: 15
incremental_backups: false
snapshot_before_compaction: false
auto_snapshot: true
tombstone_warn_threshold: 1000
tombstone_failure_threshold: 100000
column_index_size_in_kb: 64
in_memory_compaction_limit_in_mb: 64
multithreaded_compaction: false
compaction_throughput_mb_per_sec: 16
compaction_preheat_key_cache: true
read_request_timeout_in_ms: 5000
range_request_timeout_in_ms: 10000
write_request_timeout_in_ms: 2000
cas_contention_timeout_in_ms: 1000
truncate_request_timeout_in_ms: 60000
request_timeout_in_ms: 10000
cross_node_timeout: false
endpoint_snitch: SimpleSnitch
dynamic_snitch_update_interval_in_ms: 100
dynamic_snitch_reset_interval_in_ms: 600000
dynamic_snitch_badness_threshold: 0.1
request_scheduler: org.apache.cassandra.scheduler.NoScheduler
server_encryption_options:
    internode_encryption: none
    keystore: conf/.keystore
    keystore_password: cassandra
    truststore: conf/.truststore
    truststore_password: cassandra
client_encryption_options:
    enabled: false
    keystore: conf/.keystore
    keystore_password: cassandra
internode_compression: all
inter_dc_tcp_nodelay: false
preheat_kernel_page_cache: false
```

Figure 2 Cassandra Configuration Part 2



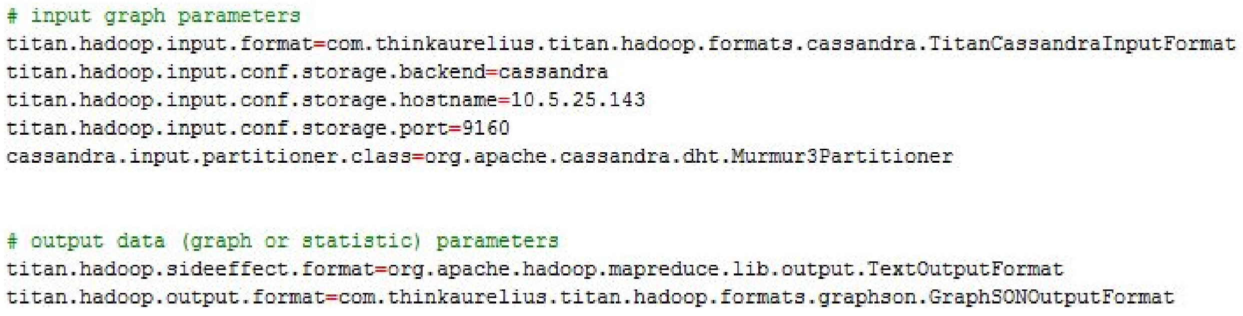
```
cluster.name: escluster1
network.host: 10.5.25.143
discovery.zen.ping.unicast.hosts: 10.5.25.143
```

Figure 3 Elasticsearch Node Configuration.



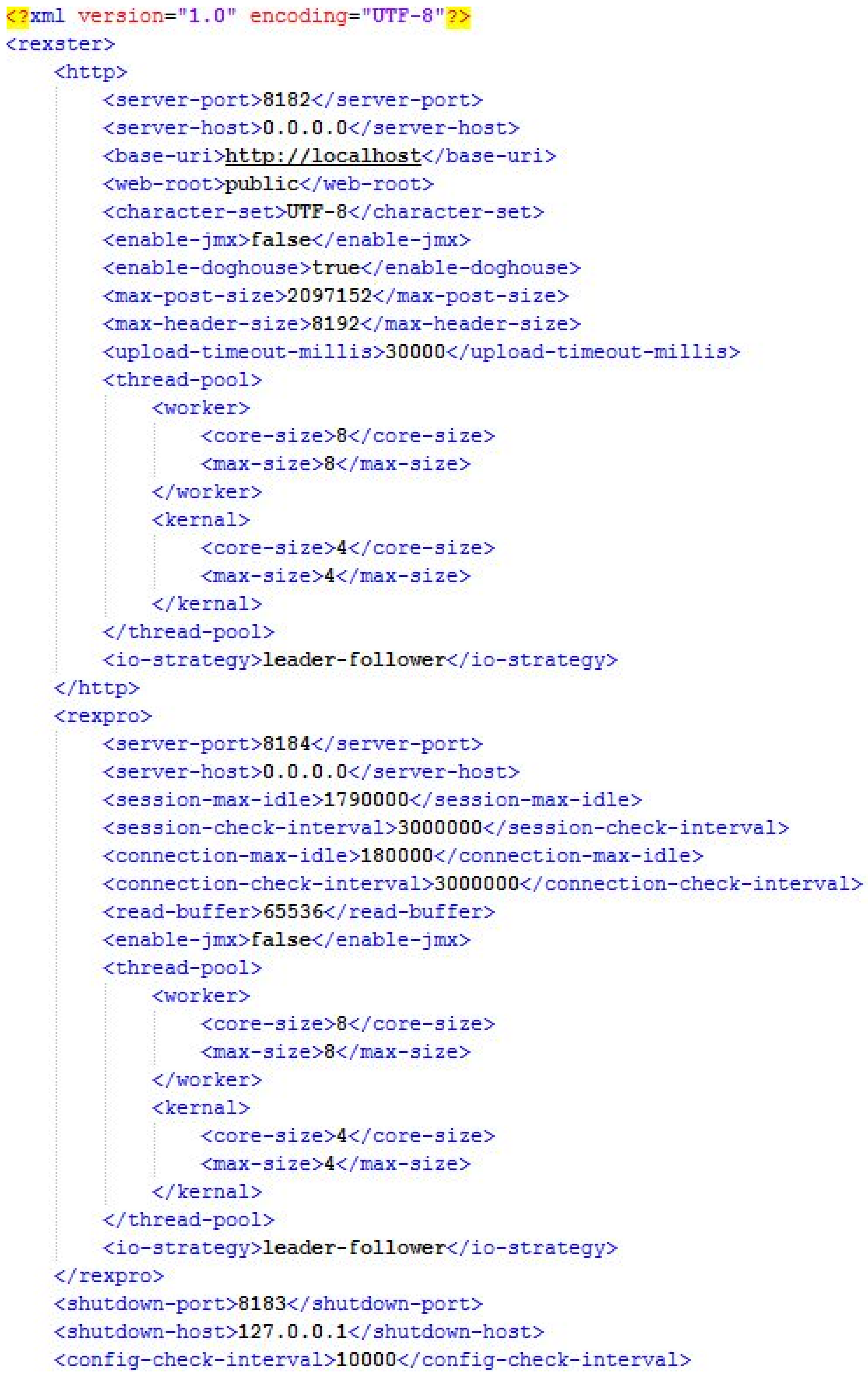
```
storage.backend = cassandra
storage.hostname = 10.5.25.143
storage.batch-loading = true
schema.default:none
cache.db-cache = true
cache.db-cache-clean-wait = 50
cache.db-cache-time = 10000
cache.db-cache-size = 0.25
storage.index.search.backend = elasticsearch
storage.index.search.hostname = 10.5.25.143
storage.index.search.cluster-name=escluster1
storage.index.search.port = 9200
storage.index.search.elasticsearch.client-only = false
storage.index.search.elasticsearch.local-mode = true
storage.index.search.directory = index_storage
storage.index.search.client-sniff = false
script.disable_dynamic = false
force-index = false
```

Figure 4 Titan Graph Configuration



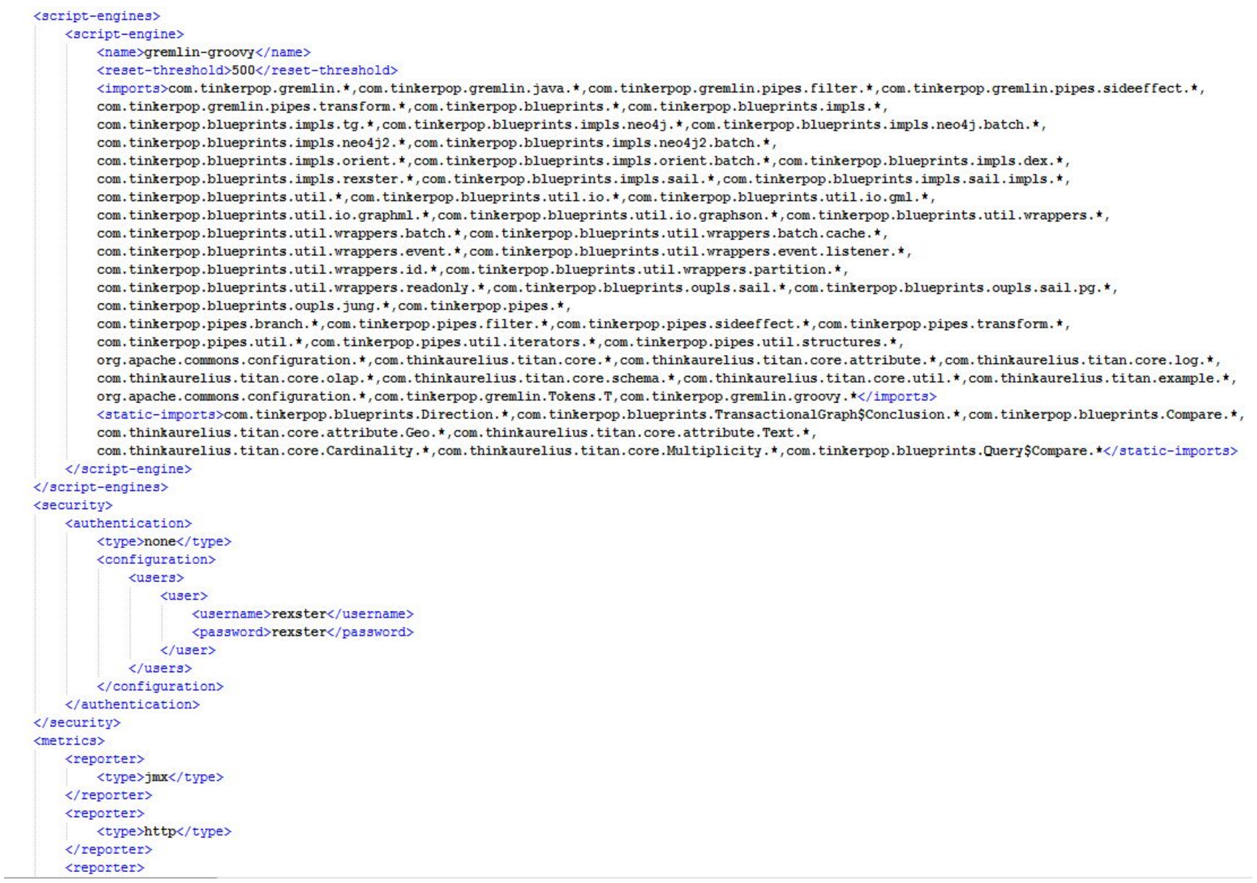
```
# input graph parameters
titan.hadoop.input.format=com.thinkaurelius.titan.hadoop.formats.cassandra.TitanCassandraInputFormat
titan.hadoop.input.conf.storage.backend=cassandra
titan.hadoop.input.conf.storage.hostname=10.5.25.143
titan.hadoop.input.conf.storage.port=9160
cassandra.input.partitioner.class=org.apache.cassandra.dht.Murmur3Partitioner


# output data (graph or statistic) parameters
titan.hadoop.sideeffect.format=org.apache.hadoop.mapreduce.lib.output.TextOutputFormat
titan.hadoop.output.format=com.thinkaurelius.titan.hadoop.formats.graphson.GraphSONOutputFormat
```

Figure 5 Titan-Hadoop Graph Configuration



```xml
<?xml version="1.0" encoding="UTF-8"?>
<rexster>
    <http>
        <server-port>8182</server-port>
        <server-host>0.0.0.0</server-host>
        <base-uri>http://localhost</base-uri>
        <web-root>public</web-root>
        <character-set>UTF-8</character-set>
        <enable-jmx>false</enable-jmx>
        <enable-doghouse>true</enable-doghouse>
        <max-post-size>2097152</max-post-size>
        <max-header-size>8192</max-header-size>
        <upload-timeout-millis>30000</upload-timeout-millis>
        <thread-pool>
            <worker>
                <core-size>8</core-size>
                <max-size>8</max-size>
            </worker>
            <kernal>
                <core-size>4</core-size>
                <max-size>4</max-size>
            </kernal>
        </thread-pool>
        <io-strategy>leader-follower</io-strategy>
    </http>
    <rexpro>
        <server-port>8184</server-port>
        <server-host>0.0.0.0</server-host>
        <session-max-idle>1790000</session-max-idle>
        <session-check-interval>3000000</session-check-interval>
        <connection-max-idle>180000</connection-max-idle>
        <connection-check-interval>3000000</connection-check-interval>
        <read-buffer>65536</read-buffer>
        <enable-jmx>false</enable-jmx>
        <thread-pool>
            <worker>
                <core-size>8</core-size>
                <max-size>8</max-size>
            </worker>
            <kernal>
                <core-size>4</core-size>
                <max-size>4</max-size>
            </kernal>
        </thread-pool>
        <io-strategy>leader-follower</io-strategy>
    </rexpro>
    <shutdown-port>8183</shutdown-port>
    <shutdown-host>127.0.0.1</shutdown-host>
    <config-check-interval>10000</config-check-interval>
```

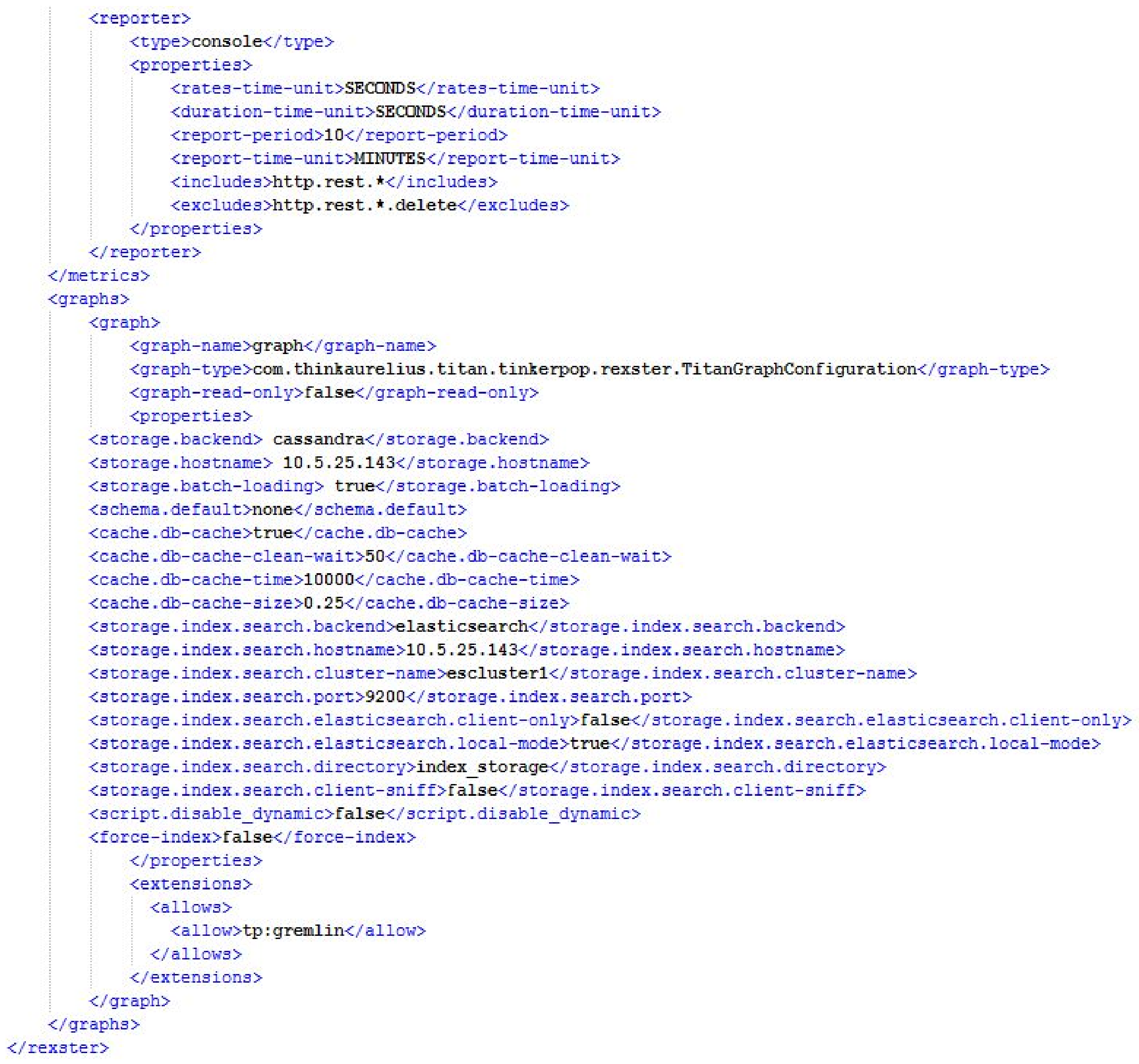
Figure 6 Rexster Configuration Part 1



```
<script-engines>
    <script-engine>
        <name>gremlin-groovy</name>
        <reset-threshold>500</reset-threshold>
        <imports>com.tinkerpop.gremlin.*,com.tinkerpop.gremlin.java.*,com.tinkerpop.gremlin.pipes.filter.*,com.tinkerpop.gremlin.pipes.sideeffect.*,
        com.tinkerpop.gremlin.pipes.transform.*,com.tinkerpop.blueprints.*,com.tinkerpop.blueprints.impls.*,
        com.tinkerpop.blueprints.impls.tg.*,com.tinkerpop.blueprints.impls.neo4j.*,com.tinkerpop.blueprints.impls.neo4j.batch.*,
        com.tinkerpop.blueprints.impls.neo4j2.*,com.tinkerpop.blueprints.impls.neo4j2.batch.*,
        com.tinkerpop.blueprints.impls.orient.*,com.tinkerpop.blueprints.impls.orient.batch.*,com.tinkerpop.blueprints.impls.dex.*,
        com.tinkerpop.blueprints.impls.rexster.*,com.tinkerpop.blueprints.impls.sail.*,com.tinkerpop.blueprints.impls.sail.impls.*,
        com.tinkerpop.blueprints.util.*,com.tinkerpop.blueprints.util.io.*,com.tinkerpop.blueprints.util.io.gml.*,
        com.tinkerpop.blueprints.util.io.graphml.*,com.tinkerpop.blueprints.util.io.graphson.*,com.tinkerpop.blueprints.util.wrappers.*,
        com.tinkerpop.blueprints.util.wrappers.batch.*,com.tinkerpop.blueprints.util.wrappers.batch.cache.*,
        com.tinkerpop.blueprints.util.wrappers.event.*,com.tinkerpop.blueprints.util.wrappers.event.listener.*,
        com.tinkerpop.blueprints.util.wrappers.id.*,com.tinkerpop.blueprints.util.wrappers.partition.*,
        com.tinkerpop.blueprints.util.wrappers.readonly.*,com.tinkerpop.blueprints.oupls.sail.*,com.tinkerpop.blueprints.oupls.sail.pg.*,
        com.tinkerpop.blueprints.oupls.jung.*,com.tinkerpop.pipes.*,
        com.tinkerpop.pipes.branch.*,com.tinkerpop.pipes.filter.*,com.tinkerpop.pipes.sideeffect.*,com.tinkerpop.pipes.transform.*,
        com.tinkerpop.pipes.util.*,com.tinkerpop.pipes.util.iterators.*,com.tinkerpop.pipes.util.structures.*,
        org.apache.commons.configuration.*,com.thinkaurelius.titan.core.*,com.thinkaurelius.titan.core.attribute.*,com.thinkaurelius.titan.core.log.*,
        com.thinkaurelius.titan.core.olap.*,com.thinkaurelius.titan.core.schema.*,com.thinkaurelius.titan.core.util.*,com.thinkaurelius.titan.example.*,
        org.apache.commons.configuration.*,com.tinkerpop.gremlin.Tokens.T,com.tinkerpop.gremlin.groovy.*</imports>
        <static-imports>com.tinkerpop.blueprints.Direction.*,com.tinkerpop.blueprints.TransactionalGraph$Conclusion.*,com.tinkerpop.blueprints.Compare.*,
        com.thinkaurelius.titan.core.attribute.Geo.*,com.thinkaurelius.titan.core.attribute.Text.*,
        com.thinkaurelius.titan.core.Cardinality.*,com.thinkaurelius.titan.core.Multiplicity.*,com.tinkerpop.blueprints.Query$Compare.*</static-imports>
    </script-engine>
</script-engines>
<security>
    <authentication>
        <type>none</type>
        <configuration>
            <users>
                <user>
                    <username>rexster</username>
                    <password>rexster</password>
                </user>
            </users>
        </configuration>
    </authentication>
</security>
<metrics>
    <reporter>
        <type>jmx</type>
    </reporter>
    <reporter>
        <type>http</type>
    </reporter>
    <reporter>
```

Figure 6 Rexster Configuration Part 2

]



```
        <reporter>
            <type>console</type>
            <properties>
                <rates-time-unit>SECONDS</rates-time-unit>
                <duration-time-unit>SECONDS</duration-time-unit>
                <report-period>10</report-period>
                <report-time-unit>MINUTES</report-time-unit>
                <includes>http.rest.*</includes>
                <excludes>http.rest.*.delete</excludes>
            </properties>
        </reporter>
    </metrics>
    <graphs>
        <graph>
            <graph-name>graph</graph-name>
            <graph-type>com.thinkaurelius.titan.tinkerpop.rexster.TitanGraphConfiguration</graph-type>
            <graph-read-only>false</graph-read-only>
            <properties>
        <storage.backend> cassandra</storage.backend>
        <storage.hostname> 10.5.25.143</storage.hostname>
        <storage.batch-loading> true</storage.batch-loading>
        <schema.default>none</schema.default>
        <cache.db-cache>true</cache.db-cache>
        <cache.db-cache-clean-wait>50</cache.db-cache-clean-wait>
        <cache.db-cache-time>10000</cache.db-cache-time>
        <cache.db-cache-size>0.25</cache.db-cache-size>
        <storage.index.search.backend>elasticsearch</storage.index.search.backend>
        <storage.index.search.hostname>10.5.25.143</storage.index.search.hostname>
        <storage.index.search.cluster-name>escluster1</storage.index.search.cluster-name>
        <storage.index.search.port>9200</storage.index.search.port>
        <storage.index.search.elasticsearch.client-only>false</storage.index.search.elasticsearch.client-only>
        <storage.index.search.elasticsearch.local-mode>true</storage.index.search.elasticsearch.local-mode>
        <storage.index.search.directory>index_storage</storage.index.search.directory>
        <storage.index.search.client-sniff>false</storage.index.search.client-sniff>
        <script.disable_dynamic>false</script.disable_dynamic>
        <force-index>false</force-index>
            </properties>
            <extensions>
              <allows>
                <allow>tp:gremlin</allow>
              </allows>
            </extensions>
        </graph>
    </graphs>
</rexster>
```

Figure 7 Rexster Configuration Part 3

APPENDIX C: Metadata Information

Field Name	**Dimension Type**	**Description**
CARDBIN	Account	The Bank Identification Number of the card used
CARDTYPE	Account	The card type
CARDEXPDT	Account	The card’s expiration date
BILLZIPCD	Account	Billing zip code of the account holder
BILLSTATE	Account	Billing state of the account holder
BILLCITY	Account	Billing city of the account holder
CUSTFIRSTNAME	Account	Customer’s first name
CUSTLASTNAME	Account	Customer’s last name
WEBSITE	Website	Website address where the transaction was made
MERCHANTID	Merchant	ID of the merchant
MCC_CODE	Merchant	Merchant category code
MERCHANT_ZIP	Merchant	Zip code of the merchant
TRX_ID	Transaction	ID of the transaction

TRANSACTION_AMOUNT	Transaction	Cost of the transaction
PURCHASE_DATE	Transaction	Date the transaction was made
PRCHSTM	Transaction	Time the transaction was made
FRAUD_IDX	Transaction	Character specifying if the transaction was fraudulent or not (G for good, B for bad)
CUSTIP	Transaction	IP Address used to make the transaction

Table 1. Metadata fields from the labeled shield data used during evaluation