CluSandra: A Framework and Algorithm for Data Stream Cluster Analysis

Jose R. Fernandez
Department of Computer Science
University of West Florida
Pensacola, FL, USA

Eman M. El-Sheikh
Department of Computer Science
University of West Florida
Pensacola, FL, USA

***Abstract*—The clustering or partitioning of a dataset's records into groups of similar records is an important aspect of knowledge discovery from datasets. A considerable amount of research has been applied to the identification of clusters in very large multi-dimensional and static datasets. However, the traditional clustering and/or pattern recognition algorithms that have resulted from this research are inefficient for clustering data streams. A data stream is a dynamic dataset that is characterized by a sequence of data records that evolves over time, has extremely fast arrival rates and is unbounded. Today, the world abounds with processes that generate high-speed evolving data streams. Examples include click streams, credit card transactions and sensor networks. The data stream's inherent characteristics present an interesting set of time and space related challenges for clustering algorithms. In particular, processing time is severely constrained and clustering algorithms must be performed in a single pass over the incoming data. This paper presents both a clustering framework and algorithm that, combined, address these challenges and allows end-users to explore and gain knowledge from evolving data streams. Our approach includes the integration of open source products that are used to control the data stream and facilitate the harnessing of knowledge from the data stream. Experimental results of testing the framework with various data streams are also discussed.**

Keywords-data stream; data mining; cluster analysis; knowledge discovery; machine learning; Cassandra database; BIRCH; CluStream; distributed systems.

I. INTRODUCTION

According to the International Data Corporation (IDC), the size of the 2006 digital universe was 0.18 zettabytes[1] and the IDC has forecasted a tenfold growth by 2011 to 1.8 zettabytes [17]. One of the main sources of this vast amount of data are streams of high speed and evolving data. Clustering analysis is a form of data mining whose application has, relatively recently, started to be applied to data streams. The unbounded and evolving nature of the data that is produced by the data stream, coupled with its varying and high-speed arrival rate, require that the data stream clustering algorithm embrace these properties: efficiency, scalability, availability, and reliability. One of the objectives of this work is to produce a distributed framework that addresses these properties and, therefore, facilitates the development of data stream clustering algorithms for this extreme environment. Another objective is to implement a clustering algorithm that is specifically designed to leverage the distributed framework. This paper describes that clustering algorithm and the distributed framework, which is entirely composed of off-the-shelf open source components. The framework is referred to simply as *CluSandra*, while the algorithm, which is deployed onto the framework, is referred to as the *CluSandra algorithm*. CluSandra's primary pillars are a database system called Cassandra [9][15] and a message queuing system (MQS). Cassandra, which is maintained by the Apache Software Foundation (ASF), is a new breed of database system that is referred to as a NoSQL database. At its core, Cassandra is a distributed hash table (DHT) designed to tackle massive datasets, perform in near-time and provide linear scalability [9]. The MQS can be any number of either open source or commercial message queuing systems that implement the Java Message Service (JMS) API. All experimentation, related to this work, was performed using the Apache ActiveMQ [16] queuing system.

The combination of the CluSandra framework and algorithm provides a distributed, scalable and highly available clustering system that operates efficiently within the severe temporal and spatial constraints associated with real-time evolving data streams. Through the use of such a system, end-users can also gain a deeper understanding of the data stream and its evolving nature in both near-time and over different time horizons.

A. *Data Stream*

A data stream is an ordered sequence of structured data records with these inherent characteristics: fast arrival rate, temporally ordered, evolves over time, and is unbounded [8]. The data stream's arrival rate can be in the order of thousands of data records per second, the concepts that are derived from the data stream evolve at varying rates over time and, because the data stream is unbounded, it is unfeasible to store all of its records in any form of secondary storage (e.g., DBMS). The data stream's evolutionary characteristic is referred to as *concept drift* [3]. This type of change may come as a result of the changing environment of the problem; e.g., floating probability distributions, migrating clusters of data, loss of old and appearance of new classes and/or features, class label swaps, etc. [20] Examples of data streams include IP network traffic, sensor networks, wireless networks, radio frequency identification (RFID), customer click streams, telephone records, etc. Today, there are many applications whose data is best modeled as a data stream and not as a persistent set of

[1] One zettabyte equals 10^{21} bytes or one billion terabytes.

tables. The following are some examples of applications for data stream processing [11]:

- Real-time monitoring of information systems that generate vast amounts of data. For example, computer network management, telecommunications call analysis, internet applications (e.g., Google, eBay, recommendation systems, click stream analysis) and monitoring of power plants.
- Generic software for applications based on streaming data. For example, finance (fraud detection, stock market analysis), sensor networks (e.g., environment, road traffic, weather forecasting, electric power consumption).

In this paper, a data stream S is treated as an unbounded sequence of pairs $\langle s,t \rangle$, where *s* is a structured data record (set of attributes) and *t* is a system-generated timestamp attribute that specifies when the data record was created. Therefore, *t* may be viewed as the data stream's primary key and its values are monotonically increasing [7]. The timestamp values of one data stream are independent from those of any other data stream that is being processed within CluSandra. Since data streams comprise structured records, streams comprising unstructured data (e.g., audio and video streams) are not considered data streams within the context of this paper.

B. Cluster Analysis

Cluster analysis or clustering is a process by which similar objects are partitioned into groups. That is, all objects in a particular group are similar to one another, while objects in different groups are quite dissimilar. The clustering problem is formally defined as follows: *for a given set of data points, we wish to partition them into one or more groups of similar objects, where the notion of similarity is defined by a distance function [21]*. Clustering is a very broad topic that lies at the intersection of many disciplines such as statistics, machine learning, data mining, and linear algebra [12]. It is also used for many applications such as pattern recognition, fraud detection, market research, image processing, and network analysis.

The focus of this work is on *data clustering*, which is a type of data mining problem. Large multi-dimensional datasets are typically not uniformly distributed. By identifying the sparse and dense areas of the data space, data clustering uncovers the distribution patterns of the dataset [10]. In general, data clustering seeks to partition *unlabeled* data records from a large dataset into labeled clusters, which is a form of *classification*. Classification is an important problem that has been studied extensively within the context of data streams[3][4]. With respect to evolving data streams, clustering presents an attractive advantage, because it is easily adapted to changes in the data and can, therefore, be used to identify features that distinguish different clusters [12]. This is ideal for concept drifting data streams.

There are different data types (e.g., binary, numerical, discrete) that need to be taken into account by data clustering algorithms; the CluSandra algorithm only processes numerical data. Future work can deploy additional algorithms, designed to handle other data types, onto the CluSandra framework. This work also assumes that the values for all the data records' attributes are *standardized*; therefore, there is no preprocessing of the data records. Numerical data are continuous measurements of a roughly linear scale [12]. When working with data records whose attributes are of this data type, the records can be treated as n-dimensional vectors, where the similarity or dissimilarity between individual vectors is quantified by a distance measure. There are a variety of distance measures that can be applied to n-dimensional vectors; however, the most common distance measure used for continuous numerical data is the Euclidean measure:

$$d(i,j) = \sqrt{\sum_{k=1}^{n} (x_{ik} - x_{jk})^2} \quad (1)$$

where x_{ik} and x_{jk} are the kth variables for the n-dimensional data records i and j. For example, suppose you have two 2-dimenisonal data records as follows: (1,3) and (4,1). The Euclidean distance between these two records is the following:

$$\sqrt{(1-4)^2 + (3-1)^2} = 3.60 \quad (2)$$

If the data stream's records are viewed as Euclidean vectors in Euclidean n-space, the distance between any two vectors (records) is the length of the line connecting the two vectors' tips or points. The lower the resulting value, the closer (similar) the two vectors. The CluSandra algorithm utilizes Euclidean distance as a measure to determine how similar or close a new data record is to a cluster's centroid (mean). It is also used to find the distance between two clusters' centroids.

The next section discusses related work in this area. Section III describes the CluSandra framework and Section IV describes the cluster query language that was developed. Section V presents the experimental results and section VI discusses the conclusions and opportunities for future work.

II. Related Work

A considerable amount of research has been applied to clustering very large multi-dimensional datasets. One of the key challenges, which has been the subject of much research, is the design of data clustering algorithms that efficiently operate within the time and space constraints presented by very large datasets. That is, the amount of available memory and required I/O time relative to the dataset's size. These constraints are greatly amplified by the data stream's extremely fast arrival time and unbounded nature. The research work done in [5], [6], and [10] introduced concepts, structures and algorithms that made great strides towards efficient clustering of data streams. The CluSandra algorithm is based on and expands on this work, as described below.

A. BIRCH

The CluSandra algorithm is based on the concepts and structures introduced by the Balanced Iterative Reducing and Clustering (BIRCH) [10] clustering algorithm. It is based on the K-means (center-based) clustering paradigm and, therefore, targets the *spherical Gaussian* cluster. K-means provides a well-defined objective function, which intuitively

coincides with the idea of clustering [19]. The simplest type of cluster is the spherical Gaussian [19]. Clusters that manifest non-spherical or arbitrary shapes, such as *correlation* and *non-linear correlation* clusters, are not addressed by the BIRCH and CluSandra algorithms. However, the CluSandra framework does not preclude the deployment of algorithms that address the non-spherical cluster types.

BIRCH mitigates the I/O costs associated with the clustering of very large multi-dimensional and persistent datasets. It is a batch algorithm that relies on multiple sequential phases of operation and is, therefore, not well-suited for data stream environments where time is severely constrained. However, BIRCH introduces concepts and a *synopsis* data structure that help address the severe time and space constraints associated with the clustering of data streams. BIRCH can typically find a good clustering with a single pass of the dataset [10], which is an absolute requirement when having to process data streams. It also introduces two structures: *cluster feature* (CF) and *cluster feature tree*. The CluSandra algorithm utilizes an extended version of the CF, which is a type of synopsis structure. The CF contains enough statistical summary information to allow for the exploration and discovery of clusters within the data stream. The information contained in the CF is used to derive these three spatial measures: *centroid*, *radius*, and *diameter*. All three are an integral part of the BIRCH and CluSandra algorithms. Given N n-dimensional data records (vectors or points) in a cluster where i = {1,2,3, …, N}, the centroid $\vec{x}_0$, radius R, and diameter D of the cluster are defined as

$$\vec{x}_0 = \frac{\sum_{i=1}^{N} \vec{x}_i}{N} \quad (3)$$

$$R = \sqrt{\frac{\sum_{i=1}^{N} \left(\vec{x}_i - \vec{x}_0\right)^2}{N}} \quad (4)$$

$$D = \sqrt{\frac{\sum_{i=1}^{N} \sum_{j=1}^{N} \left(\vec{x}_i - \vec{x}_j\right)^2}{N\left(N-1\right)}} \quad (5)$$

where $\vec{x}_i$ and $\vec{x}_j$ are the ith and jth data records in the cluster and N is the total number of data records in the cluster. The centroid is the cluster's mean, the radius is the average distance from the objects within the cluster to their centroid, and the diameter is the average pair-wise distance within the cluster. The radius and diameter measure the tightness or density of the objects around their cluster's centroid. The radius can also be referred to as the "root mean squared deviation" or RMSD with respect to the centroid. Typically, the first criterion for assigning a new object to a particular cluster is the new object's Euclidean distance to a cluster's centroid. That is, the new object is assigned to its closest cluster. Note, however, that equations 3, 4 and 5 require multiple passes of the data set. Like BIRCH, the CluSandra algorithm derives the radius and centroid while adhering to the single pass constraint. To accomplish this, the algorithm maintains statistical *summary* data in the CF. This data is in the form of the total number of data records (N), linear sum, and sum of the squares with respect to the elements of the n-dimensional data records. This allows the algorithm to calculate the radius, as follows:

$$\sqrt{\left(\sum x_i^2 - \left(\sum x_i\right)^2 / N\right) 1 / \left(N-1\right)} \quad (6)$$

For example, given these three data records: (0,1,1), (0,5,1), and (0,9,1), the linear sum is (0,15,3), the sum of the squares is (0,107,3) and N is 3. The CF, as described in BIRCH, is a 3-tuple or triplet structure that contains the aforementioned statistical summary data. The CF represents a cluster and records summary information for that cluster. It is formally defined as

$$CF = \langle N, LS, SS \rangle \quad (7)$$

where N is the total number of objects in the cluster (i.e., the number of data records absorbed by the cluster), LS is the linear sum of the cluster and SS is the cluster's sum of the squares:

$$LS = \sum_{i=1}^{N} \vec{x}_i \quad (8)$$

$$SS = \sum_{i=1}^{N} \vec{x}_i^2 \quad (9)$$

It is interesting to note that the CF has both the *additive* and *subtractive* properties. For example, if you have two clusters with their respective CFs, CF_1 and CF_2, the CF that is formed by merging the two clusters is simply $CF_1 + CF_2$.

B. CluStream

CluStream [5] is a clustering algorithm that is specifically designed for *evolving* data streams and is based on and extends the BIRCH algorithm. This section provides a brief overview of CluStream and describes the problems and/or constraints that it addresses. This section also describes how the CluSandra algorithm builds upon and, at the same time, deviates from CluStream.

One of CluStream's goals is to address the temporal aspects of the data stream's single-pass constraint. For example, the results of applying a single-pass clustering algorithm, like BIRCH, to a data stream whose lifespan is 1 or 2 years would be dominated by outdated data. CluStream allows end-users to explore the data stream over different time horizons, which provides a better understanding of how the data stream evolves over time. CluStream divides the clustering process into two phases of operation that are meant to operate simultaneously. The first, which is referred to as the *online* phase, efficiently computes and stores data stream summary statistics in a structure called the *microcluster*. A microcluster is an extension of the BIRCH CF structure

whereby the CF is given two temporal dimensions. The second phase, which is referred to as the *offline* phase, allows end-users to perform *macroclustering* operations on a set of microclusters. Macroclustering is the process by which end-users can explore the microclusters over different time horizons. To accomplish this, CluStream uses a *tilted time frame* model for maintaining the microclusters. The tilted time frame approach stores snapshots (sets) of microclusters at different levels of granularity based on elapsed time. In other words, as time passes, the microclusters are merged into coarser snapshots. The CluSandra algorithm is based upon and extends CluStream and the design of the CluSandra framework is based on the concepts of both microclustering and macroclustering. However, the general approach taken by CluSandra for these two operational phases is quite different than that taken by CluStream.

CluStream's microcluster extends the CF structure by adding two temporal scalars or dimensions to the CF. The first scalar is the sum of the timestamps of all the data records that have been absorbed by the cluster and the second is the sum of the squares of the timestamps. Thus the CF triplet, as defined by BIRCH, is extended as follows:

$$CF = \langle N, LS, SS, ST, SST \rangle \qquad (10)$$

Where ST is the sum of the timestamps and SST is the sum of the squares of the timestamps. Note that this extended CF retains its additive and subtractive properties. From henceforth, this extended version of the CF is simply referred to as a microcluster. The ST and SST can be applied to expression (6) to arrive at the temporal standard deviation of the microcluster as follows:

$$\sqrt{\left(SST - \left(ST\right)^2\right)1/\left(N-1\right)} \qquad (11)$$

CluStream's microclustering process collects and maintains the statistical information in such a manner that the offline macroclustering phase can make effective use of the information. For example, macroclustering over different time horizons and exploring the evolution of the data stream over these horizons.

CluStream defines a fixed set of microclusters that it creates and maintains in-memory. This set, $\mathcal{M} = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_q\}$ is the current online working set with q being the maximum number of microclusters in the set and each microcluster in $\mathcal{M}$ being given a unique *id*. The CluStream paper states that the algorithm is very sensitive to any further additions to the microcluster snapshot, as this negatively impacts q. This type of space constraint is one that CluSandra removes by relying on Cassandra to serve as a highly reliable and scalable real-time distributed cluster database. Like CluStream, the CluSandra algorithm also maintains a working set of microclusters in memory; however, the size of this working set is dictated by the size of a sliding time window in combination with a maximum boundary threshold (MBT).

CluStream's updating of the microclusters is similar to that of BIRCH. When a new data record is presented by the data stream, it is assigned to the closest microcluster $\mathcal{M}_i$ in $\mathcal{M}$. After finding the closest microcluster $\mathcal{M}_i$, it is then determined if $\mathcal{M}_i$ can absorb the new record without exceeding its MBT. CluStream defines the MBT as, "*a factor of t of the RMSD of the data points in $\mathcal{M}_i$ from the centroid*". This is another way of referring to the standard deviation of the cluster times a factor t to arrive at the final maximum radius. The microcluster $\mathcal{M}_i$ is updated whenever it absorbs a new data record. If $\mathcal{M}_i$ has absorbed only one data record, the RMSD cannot be calculated; therefore, for those clusters having absorbed only one data record, the MBT is derived as the distance from $\mathcal{M}_i$ to its closest neighbor times a factor r. The CluSandra algorithm uses a similar approach in locating the closest microcluster. However, it does not use the nearest neighbor approach for those instances where the closest microcluster has absorbed only one entry. It instead determines if the closest microcluster can absorb the data record without exceeding a configurable and fixed maximum radius, which is CluSandra's MBT.

If $\mathcal{M}_i$ cannot absorb the new data record, CluStream creates a new microcluster to host the data record. To conserve memory, CluStream requires that either one of the existing microclusters in $\mathcal{M}$ be deleted or two microclusters be merged. Only those microclusters that are determined to be *outliers* are removed from $\mathcal{M}$. If an outlier cannot be found in $\mathcal{M}$, two microclusters are merged. The CluSandra algorithm deviates from this approach since it simply adds a new microcluster to its current working set. Again, the size of CluSandra's in-memory working set is managed according to a temporal sliding window and the specified MBT. Any microclusters that are no longer active within the current sliding window are removed from the working set, but not before being persisted to the Cassandra cluster database.

The temporal scalars (ST, SST) of the microcluster, in combination with a user-specified threshold δ, are used to look for an outlier microcluster in $\mathcal{M}$. The ST and SST scalars allows CluStream to calculate the mean and standard deviation of the arrival times of the data records in $\mathcal{M}$'s microclusters. CluStream assumes that the arrival times adhere to a normal distribution. With the mean and standard deviation of the arrival times calculated, CluStream calculates a "*relevance stamp*" for each of the microclusters. A microcluster whose relevance stamp is less than the threshold δ is considered an outlier and subject to removal from $\mathcal{M}$. If all the relevance stamps are recent enough, then it is most likely that there will be no microclusters in $\mathcal{M}$ whose relevance stamp is less than δ. If and when this occurs, CluStream merges the two closest microclusters in $\mathcal{M}$ and assigns the resulting merged microcluster a *listid* that is used to identify the clusters that were merged to create this new merged microcluster. So as time progresses, one microcluster may end up comprising many individual microclusters.

Unlike BIRCH, CluStream does not utilize a tree structure to maintain its microclusters. At certain time intervals, and while $\mathcal{M}$ is being maintained as described above, $\mathcal{M}$ is persisted to secondary storage. Each instance of $\mathcal{M}$ that is persisted is referred to as a *snapshot*. CluStream employs a logarithmic based time interval scheme, which is referred to as

a *pyramidal time frame***,** to store the snapshots. This technique guarantees that all individual microclusters in $\mathcal{M}$ are persisted prior to removal from $\mathcal{M}$ or being merged with another microcluster in $\mathcal{M}$. This allows a persisted and merged microcluster (i.e., those having a listid) in a snapshot to be broken down (via the microclusters subtractive property) into its constituent (individual), finer-grained microclusters during the macroclustering portion of the process. The opposite is also available, whereby the additive property allows finer-grained/individual and merged microclusters to be merged into more course grained microclusters that cover specified time horizons. Snapshots are classified into orders, which can vary from 1 to $\log_2(T)$, where T is the amount of clock time elapsed since the beginning of the stream[5]. The number of snapshots stored over a period of T time units is

$$(a+1) * \log_2(T) \quad (12)$$

For example, if $\alpha = 2$ and the time unit or granularity is 1 second, then the number of snapshots maintained over 100 years is as follows:

$$(2+1) * \log_2(100 * 365 * 24 * 60 * 60) \gg 95 \quad (13)$$

The CluSandra algorithm does not operate within the same memory constraints as CluStream. During the CluSandra algorithm's microclustering process, microclusters are not merged to accommodate a new microcluster and there is no need to search for possible outliers that can be targeted for removal from $\mathcal{M}$. When a new microcluster is created, it is simply added to the current in-memory working set and persisted to the Cassandra cluster database. Also, when a microcluster absorbs a data record, the microcluster is immediately persisted to the cluster database; there is no dependence on a periodic time interval scheme for persisting $\mathcal{M}$ to secondary storage.

CluStream does not include a database system of any kind for persisting its snapshots and does not fully address the transactional integrity associated with snapshot persistence. After the in-memory snapshot $\mathcal{M}$ has been updated, there may exist a relatively lengthy time period before the snapshot is persisted to the local file system. This raises the risk that the snapshot's state may be lost due to process, system or hardware failure. Also, CluStream does not address $\mathcal{M}$'s recoverability. If the machine fails and is restarted, CluStream cannot reliably recover $\mathcal{M}$'s state prior to the failure. In other words, there is no transactional integrity associated with $\mathcal{M}$'s persistence. One of the goals of the CluSandra framework is to introduce the necessary components that guarantee the transactional integrity of microcluster persistence.

III. CLUSANDRA FRAMEWORK

Figure 1 is a level 1 context-level data flow diagram[2] (DFD) that represents, at a high-level, the CluSandra framework. The framework's core components are written entirely in version 1.5 of the Java programming language; however, the framework supports client components that are written in a variety of programming languages. The CluSandra framework and algorithm are based on temporal and spatial aspects of clustering. The algorithm, which is implemented in a MicroClustering Agent (MCA) framework component and described in more detail in section D, uses temporal and spatial measures (radius, distance) to group the data stream's records into microclusters. The microclusters are stored in the Cassandra database and later accessed by offline processes (e.g., aggregator) and/or end-users. A Cluster Query Language (CQL) is provided by the framework to facilitate the querying and analysis of the microclusters in the database. As previously noted, Cassandra is an implementation of a DHT that comprises two or more distributed machines configured in a peer-to-peer ring network topology. The ring of machines or nodes is called a Cassandra cluster. To meet the most demanding environments, Cassandra can *elastically* scale up from a one or two node cluster to a cluster comprising 10s, if not 100s, of nodes and then back down to one or two nodes. Each node in the cluster may also optionally host the CluSandra framework's other executable components.

A. *StreamReader*

The StreamReader component is responsible for reading the data stream's structured data records, wrapping those data records in a CluSandra-specific DataRecord object, and then sending the DataRecords to the CluSandra message queuing system (MQS), where they are temporarily stored or buffered for subsequent processing. The CluSandra framework automatically time stamps the DataRecords when they are created, but the StreamReader can also override the framework's timestamp. This may be required if, for example, the raw data stream records are already time-stamped. The time stamps are critically important, because they are used to record the data stream's timeline. The raw data stream record that is read by the StreamReader is treated as a multidimensional vector containing one or more continuous numerical values. This vector is encapsulated by the DataRecord object. It is critical for the StreamReader to keep up with the data stream's arrival rate, which is assumed to be in the 1000s of records per second[2]. However, this should not be an issue given the simple and straightforward nature of the StreamReader's main purpose.

[2] It is also quite possible that the StreamReader is the stream generator.

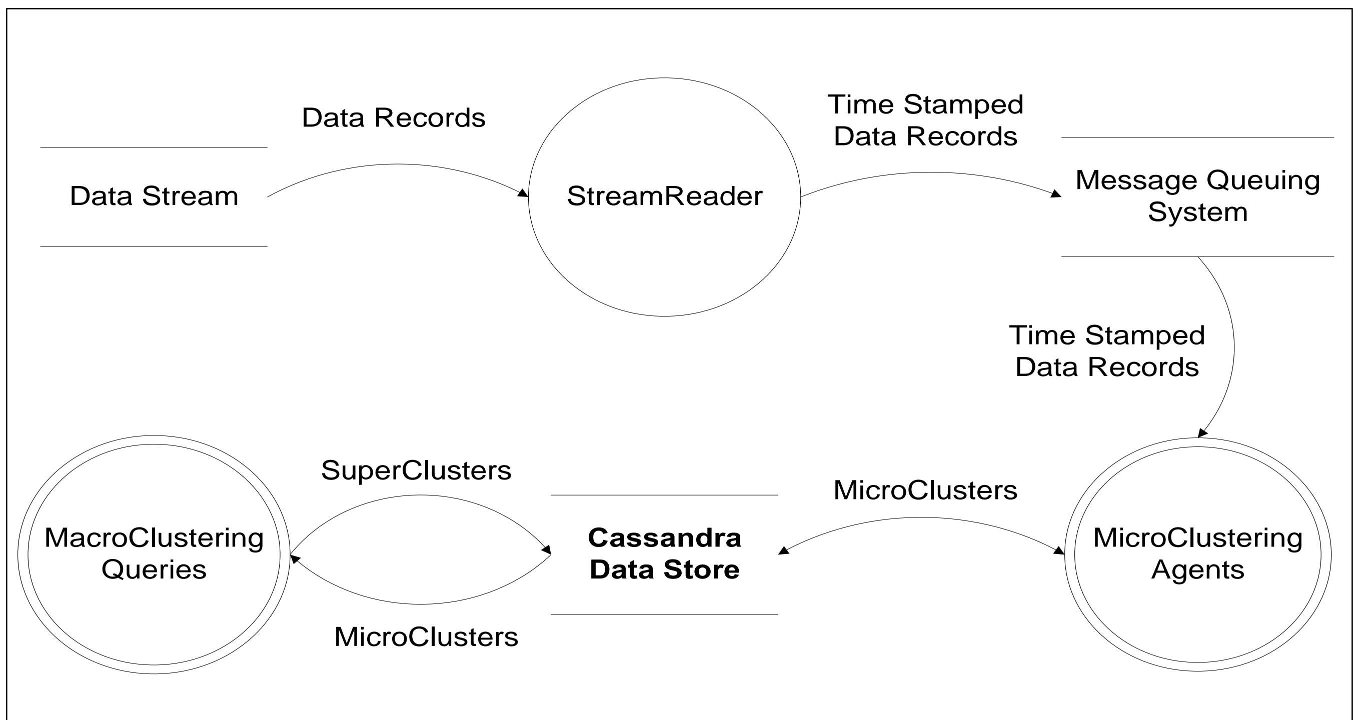

Figure 1. CluSandra data flow diagram

CluSandra's MQS provider is remotely accessed via a TCP/IP network; therefore, the StreamReader component does not have to reside on the same node as the MQS provider nor does it have to reside in a Cassandra cluster node. The StreamReader can even be embedded within the component or device (e.g., sensor or router) that produces the data stream. StreamReaders can be distributed across a population of such devices, all writing to the same or different MQS queue, with each queue dedicated to a particular data stream. Most MQS providers, like ActiveMQ, support clients written in a variety of languages. If a StreamReader is not written in the Java programming language and would like to take advantage of the component that implements the CluSandra algorithm, then it requires a transformational component; in other words, a Java proxy that can create a Java DataRecord object and place it in the MQS for the non-Java StreamReader.

B. Timeline Index

The timeline index (TI), which is not depicted in the DFD, represents the data stream's timeline. It is an important temporal-based index that is maintained in the Cassandra database and used for maintaining the clusters in the database. In the Cassandra vernacular, the TI is a type of *secondary index*. There is one TI defined for each data stream that is being processed within the CluSandra framework. The TI is implemented as a Cassandra *SuperColumnFamily*. At an abstract level, each entry in the TI represents a second in the data stream's lifespan. Each entry's contents or value is a Cassandra *SuperColumn* whose entries (Columns) form a collection or set of row keys to clusters that were or are still active during that second in time. A data stream's timeline can run for any amount of time. During development and testing, the stream's timeline may extend over a handful of seconds, while in production environments, the timeline may extend over months, if not years.

The TI's implementation comprises a Cluster Index Table (CIT). The CIT includes a row for each day of the year and each row comprises 86400 SuperColumns; one SuperColumn for each second of that particular day. Each SuperColumn contains one or more Columns whose values are keys to clusters in the Cluster Table (CT). Also, a SuperColumn can be assigned different names from row-to-row, and in this case the name of a SuperColumn is a timestamp that corresponds to a second for that day. Each row key of the CIT is given a value that corresponds to the zeroth second of a particular day. The motivation behind the TI's implementation is that Cassandra is not currently set up to perform well with sorted rows and returning ranges within those sorted rows. However, it is set up to sort columns and return ranges or slices of columns, based on their names.

C. Message Queuing System

The message queuing system (MQS) is a critical piece of the CluSandra framework. It serves as a reliable asynchronous message store (buffer) that guarantees delivery of its queued messages (DataRecords). So as a dam is used to control a wild raging river and harness it to produce electricity, so is the MQS used to control the evolving high-speed data stream so that it can be harnessed to produce knowledge. In CluSandra's case, a MQS queue serves as the stream's dam and its contents of time-stamped DataRecords form the reservoir. The CluSandra framework supports the simultaneous processing of many data streams; therefore, there may very well be many queues defined within the MQS; one queue for each data stream.

The primary motivation for having the CluSandra framework incorporate a MQS is to *control the data stream*. More precisely, the MQS provides a reliable DataRecord store that temporarily buffers and automatically distributes DataRecords across one or more instances of the microclustering agent (MCA) component. A group of identical MCAs that consume DataRecords from the same queue is called a *swarm*. Many swarms can be defined and distributed across the framework, with each swarm reading from its unique queue. The queue is capable of retaining thousands of DataRecords and guarantees their delivery to the swarm, which is responsible for ensuring that the number of DataRecords in the MQS queue is maintained at an acceptable level. The swarm size is, therefore, a function of the data stream rate. Data streams with extremely fast arrival rates produce very large volumes of DataRecords and require a correspondingly large swarm.

The CluSandra framework is designed to utilize any MQS that implements the Java Message Service (JMS) API. The vast majority of MQS providers, both open source and commercial, implement the JMS. The JMS is an industry standard Java messaging interface that decouples applications, like the StreamReader, from the different JMS implementations. JMS thus facilitates the seamless porting of JMS-based applications from one JMS-based queuing system to another. Within the context of the JMS, the process that places messages in the MQS's queues is called the *producer*, the process that reads messages from the queue is called the *consumer*, and both are generically referred to as *clients*. So the StreamReader is a producer, the MCA is a consumer and they are both clients.

The MQS *guarantees* delivery of DataRecords to the swarm. This guarantee means that DataRecords are delivered even if the MQS or machine hosting the MQS were to fail and be restarted. The MQS achieves this guarantee via a combination of message persistence and a broker-to-client acknowledgment protocol. These MQSs can be configured to persist their messages to either file systems (distributed or local) or database management systems (DBMS). For the CluSandra framework, the MQS is configured to use a distributed or shared file system and not a DBMS. The file system provides much better throughput performance than does a DBMS.

These MQS systems are also architected to provide fault-tolerance via redundancy. For example, you can run multiple "*message broker*" processes across multiple machines, where certain message brokers can act as hot or passive standbys for failover purposes. The message broker is the core component of the MQS and is the component responsible for message delivery. One overarching requirement is to ensure that this level of reliability and/or fault tolerance be an inherent quality of the CluSandra framework. These MQS systems include many features, but it is beyond the scope of this paper to list all the features.

D. Microclustering Agent

The microclustering agent (MCA) is the framework component that consumes DataRecords from a particular MQS queue and implements a clustering algorithm that produces microclusters. This section describes the MCA that implements the CluSandra algorithm and is delivered with the CluSandra framework.

Like CluStream, the CluSandra algorithm tackles the one-pass data stream constraint by dividing the data stream clustering process into two operating phases: online and offline. Microclustering takes place in real-time, computing and storing summary statistics about the data stream in microclusters. There is also an optional offline aggregation phase of microclustering that merges temporally and spatially similar microclusters. Macroclustering is another offline process by which end-users create and submit queries against the stored microclusters to discover, explore and learn from the evolving data stream. As CluStream extended the BIRCH CF data structure, so does the CluSandra algorithm extend the CluStream's CF structure as follows:

$$CFT = \langle N, LS, SS, ST, SST, CT, LAT, IDLIST \rangle \quad (14)$$

The CFT is used to represent either a microcluster or supercluster in the CluSandra data store. The term *cluster* applies to both micro and superclusters. The CT and LAT parameters are two timestamp scalars that specify the creation time and last absorption time of the cluster, respectively. More precisely, CT records the time the cluster was created and the LAT records the timestamp associated with the last DataRecord that the cluster absorbed. When a microcluster is first created, to absorb a DataRecord that no other existing microcluster can absorb, both the CT and LAT parameters are assigned the value of the DataRecord's timestamp. All timestamps in CluSandra are measured in the number of milliseconds that have elapsed since Unix or Posix epoch time, which is January 1,1970 00:00:00 GMT. The IDLIST is also new and is used by superclusters.

Over time, a particular pattern in the data stream may appear, disappear and then reappear. This is reflected or captured by two microclusters with identical or very similar spatial values, but different temporal values. The time horizon over which a cluster was active can be calculated by the CT and LAT parameters and more detailed statistical analysis can be performed based on the other temporal, as well as spatial parameters. For example, the temporal density of the DataRecords and their spatial density with respect to one another and/or their centroid. An *inactive* microcluster is no longer capable of absorbing data records; however, during macroclustering, it can be merged with other inactive and active microclusters to form a supercluster. The algorithm's microclustering phase, therefore, works within a specified temporal sliding window and updates only those microclusters that are within that time window. Such a temporal sliding window is required, because of the unbounded nature of the data stream.

As previously mentioned, the MCA consumes sets of DataRecords from its assigned MQS queue. The framework provides a *read template* for the MCA that includes this functionality; therefore, the one implementing the MCA's clustering algorithm does not need to concern herself with this functionality. The members of a set $\mathcal{D}$ of DataRecords are consumed, by the read template, in the same order that they

were produced by the StreamReader and the temporal order of the DataRecords is maintained by the MQS. The set $\mathcal{D}$ can, therefore, be viewed as a temporal window of the data stream.

$$\mathcal{D} = \{d_1, d_2, \ldots, d_k\} \quad (15)$$

The maximum number of DataRecords in $\mathcal{D}$ is configurable. Also, the amount of time the read template blocks on a queue, waiting to read the maximum number of DataRecords in $\mathcal{D}$, is configurable. The read template processes the set $\mathcal{D}$ whenever the maximum number has been reached or the read time expires and $\mathcal{D} \neq \varnothing$. The read time should be kept at a relatively high value. So, $0 < k <= m$, where k is the total number of DataRecords in $\mathcal{D}$ and m is the maximum. If the read time expires and $\mathcal{D} = \varnothing$, the MCA simply goes back to blocking on the queue for the specified read time.

After the read template has read a set $\mathcal{D}$ from the queue, it gives it to the clustering algorithm via a specified interface. The following describes the CluSandra clustering algorithm and from henceforth it is simply referred to as the CA. When the CA receives a set $\mathcal{D}$, it begins the process of partitioning the DataRecords in $\mathcal{D}$ into a set $\mathcal{M}$ of currently active microclusters. On startup, $\mathcal{M}$ is an empty set, but as time progresses, $\mathcal{M}$ is populated with microclusters. The CA then determines the time horizon h associated with $\mathcal{D}$. The range of h is R_h and it is defined by the newest and oldest DataRecords in $\mathcal{D}$. Therefore, $R_h = \{t_o\text{-}t_e, t_y\}$, where t_e is the configurable microcluster expiry time and t_o and t_y represent the oldest and newest DataRecords in $\mathcal{D}$, respectively. All microclusters in $\mathcal{M}$, whose LAT does not fall within R_h, are considered inactive microclusters and removed from $\mathcal{M}$. Thus the CA always works within a sliding temporal window that is defined by R_h. When the CA completes the processing of $\mathcal{D}$, it writes all new and updated microclusters in $\mathcal{M}$ to the Cassandra data store, gives control back to read template, and starts the partitioning process over again when it is given a new set $\mathcal{D}$ of DataRecords.

To partition the DataRecords in $\mathcal{D}$, the CA iterates through each DataRecord in $\mathcal{D}$ and selects a subset $\mathcal{S}$ of microclusters from $\mathcal{M}$, where all microclusters in $\mathcal{S}$ are active based on the current DataRecord's (d_c) timestamp. For example, if the timestamp for d_c is t_d, then only those microclusters in $\mathcal{M}$ whose LAT value falls within the range, $\{t_d\text{-}t_e, t_d\}$ are added to $\mathcal{S}$. The value t_e is the configurable microcluster expiry time. Depending on the rate of the data stream and the microcluster expiry time, it is possible that $\mathcal{S} = \mathcal{M}$ and so, $\mathcal{S} \subseteq \mathcal{M}$. The CA uses the Euclidean distance measure to find the microcluster in $\mathcal{S}$ that is closest to d_c. The MBT is then used to determine if the closest microcluster $\mathcal{M}_i$ in $\mathcal{S}$ can absorb d_c. The MBT is a configurable numeric value that specifies the maximum radius of a microcluster. Again, the radius or RMSD is the root mean squared deviation of the cluster and is derived according to (6). If $\mathcal{M}_i$ can absorb d_c without breaching the MBT, $\mathcal{M}_i$ is allowed to absorb d_c and is placed back in $\mathcal{M}$, else a new microcluster is created to absorb d_c and that new microcluster is then added to $\mathcal{M}$. If $\mathcal{S} = \varnothing$, the CA simply creates a new microcluster to absorb d_c and adds the new microcluster to $\mathcal{M}$.

One method for deriving a MBT value for the target data stream is by sampling the data stream to derive an average distance (measure of density) between DataRecords and then using some fraction of that average density. If the CA has not been assigned a MBT, it will derive the MBT based on the first set of data records that it receives from the read template. If $\mathcal{M}_i$ has previously absorbed only one DataRecord (i.e., N=1), the RMSD cannot be calculated. In such a case, the MBT is used to determine if the microcluster can absorb the DataRecord.

Depending on the distribution of the data stream, as it evolves over time, microclusters of all sizes appear, disappear, and may reappear. The number and sizes of the microclusters are a factor of not only the data stream's evolutionary pattern, but also of the MBT and microcluster expiry-time. The smaller the MBT, the more microclusters will be produced and vice versa. However, the optional offline aggregation and/or macroclustering phases can be used to merge those microclusters that are deemed similar. The next section describes the CluSandra Aggregator component, which is responsible for the aggregation and merges those microclusters whose radii overlap. Please note that the Aggregator does not produce superclusters; it simply *merges* overlapping microclusters.

E. Aggregating Microclusters

If there is an MCA swarm distributed across the CluSandra framework's nodes, then it is very likely for the swarm to create microclusters that are very similar, if not equal, both temporally and spatially (see figure 2). This occurs if two or more MCAs in the swarm process a set of DataRecords with equal or overlapping time horizons (h_e). This may also occur as a natural side effect of the clustering algorithm.

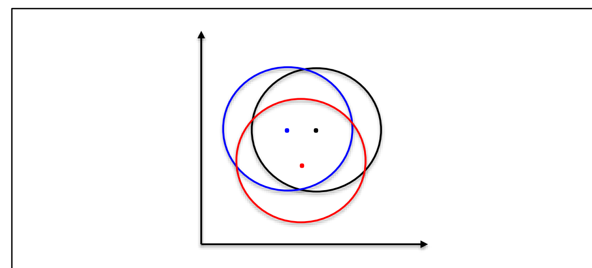

Figure 2. Overlapping clusters

If any microclusters temporally and spatially overlap, then those microclusters may be viewed as one microcluster. Given a two-dimensional vector-space, the figure above illustrates an example where three MCAs have created three microclusters (dots represent the microclusters' centroids) that are so close to one another, both spatially and temporally, that they should be merged into one microcluster. The merging of these microclusters is performed by an offline aggregator component that sweeps through the data stream timeline (CIT) performing such merges. Microcluster aggregation is a type of *agglomerative* clustering procedure whereby individuals or

groups of individuals are merged based upon their temporal and spatial proximity to one another [13]. Agglomerative procedures are probably the most widely used of the hierarchical clustering methods [13]. The result of a merger, performed by the aggregator, is immutable (see figure 3). This aggregator, which is provided as part of the CluSandra algorithm package, should not be confused with superclustering and macroclustering, which is discussed in the following section.

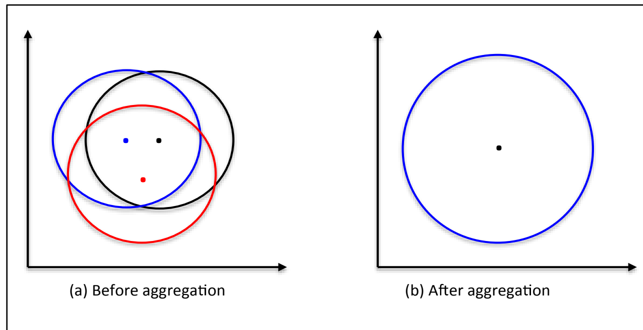

Figure 3. Before and after aggregation

The microcluster expiry time is used to determine if two microclusters temporally overlap. To determine if two temporally overlapping microclusters also spatially overlap, the aggregator compares the distance between their centroids with the sum of their radii. If the distance is less than the sum of the radii, then the two microclusters spatially overlap. There may even be instances where one microcluster is entirely within the other. The aggregator is given a configurable property called the *overlapFactor*. This property is used to specify the amount of overlap that is required to deem two microclusters similar enough to merge. The radii of the microcluster that results from the merge may be greater than the radii of the two merged microclusters; therefore, it will occupy more space, as well as time and be capable of absorbing more surrounding microclusters that temporally overlap. So the less the overlapFactor, the greater the probability of creating very large clusters that may mask out interesting patterns in the data stream.

To minimize the occurrence of overlapping microclusters, all members of a MCA swarm can work from the same or shared set of microclusters in the data store. However, this would have required coordination between the distributed members of the swarm. Unfortunately, this level of coordination requires a distributed locking mechanism that introduces severe contention between the members of the swarm. It is for this reason, that this approach of sharing microclusters was not followed. Also, overlapping microclusters is a natural side effect of the CluSandra clustering algorithm.

F. Superclusters and Macroclustering

The CluSandra framework introduces the supercluster, which is created when two or more clusters, either micro or super, are merged via their additive properties. A supercluster can also be reduced or even eliminated via its subtractive property. The IDLIST in the CluSandra CFT is a collection or vector of microcluster ids (Cassandra row keys) that identify a supercluster's constituent parts (i.e., microclusters). If the IDLIST is empty, then it identifies the cluster as a microcluster, else a supercluster.

Superclusters are created by the end-user during the macroclustering process. Superclusters are created based on a specified distance measure (similarity) and time horizon. Like aggregation, superclustering is another type of *agglomerative* clustering procedure; however, unlike traditional agglomertive procedures where the result of a merger is immutable, CluSandra's superclusters can be undone. What makes this possible is the subtractive property of the CFT. When a supercluster is created, the earliest CT and LAT of its constituent parts are used as the supercluster's CT and LAT. The CT and LAT thus identify the supercluster's lifespan.

G. The Spring Framework

The open source Spring Framework [1] is relied upon by the CluSandra framework for configuration and to reuse Spring's components. Except for Cassandra, all CluSandra framework components are comprised of one or more Spring POJOs (Plain Old Java Objects) that are configured via Spring XML configuration files. The Spring Framework, which is henceforth referred to simply as *Spring*, was created for the specific purpose of minimizing the development costs associated with Java application development. Spring provides a number of Java packages whose components are meant to be reused and that, in general, hide the complexities of Java application development. However, at its core, Spring is a modular dependency injection and aspect-oriented container and framework. By using Spring, Java developers benefit in terms of simplicity, testability and loose coupling.

The StreamReader and MCA components make heavy use of Spring's support for the JMS to send and receive DataRecords to and from the MQS, respectively. Apache ActiveMQ [16] has been selected as the CluSandra framework's default JMS provider (i.e., MQS). There are many open source JMS providers or implementations. ActiveMQ was chosen, because of its rich functionality, beyond that specified by the JMS specification, and its robust support for and integration with Spring.

As a JMS producer, the StreamReader indirectly uses Spring's JmsTemplate class to produce DataRecords destined for the ActiveMQ message broker. The JmsTemplate is based on the template design pattern and it is a convenience class that hides much of the complexity of sending messages to a JMS message broker. The JmsTemplate is integrated into a *send template* (convenience class) provided by CluSandra; therefore, the person implementing the StreamReader does not have to concern herself with the implementation of this functionality. This send template is similar to the previously described *read template* for the MCA. In general, the use of the JmsTemplate by both the send and read templates, in combination with the corresponding Spring XML file, helps ensure a decoupling between the CluSandra JMS clients and whatever JMS provider is being used as the CluSandra MQS.

IV. Cluster Query Language

The CluSandra framework includes a query language that is used for querying the CluSandra algorithm's cluster

database. The query language, which is referred to as the *cluster query language* or *CQL* for short, includes the following statements:

- *Connect*: This statement is used to connect to a particular node in the Cassandra cluster.
- *Use*: This statement, which must be invoked immediately after the Connect, is used to specify the keyspace to use within the Cassandra cluster. A Cassandra keyspace is analogous to a schema that is found in a relational database.
- *Select*: This statement, which is the one most often invoked, is used for projecting clusters from the cluster store.
- *Aggregate*: This statement is used for invoking the aggregator on all or a portion of the cluster database.
- *Merge*: This statement is used, as part of the offline macroclustering process, to form superclusters.
- *Sum*: This is a relatively simple statement that is used to return the total number of DataRecords that have been absorbed by all the microclusters in the cluster database.
- *Distance*: This statement is used for acquiring the distance between pairs of clusters.
- *Overlap*: This statement is used for acquiring the amount of overlap between pairs of clusters or in other words, the overlap percentage of the clusters' radii.

CluSandra's CQL is not to be confused with the Cassandra Query Language, which is also referred to as CQL. CluSandra's CQL operates in either batch or interactive mode. When invoked in batch mode, the user specifies a file that contains CQL statements.

V. Empirical Results

Several experiments were conducted to evaluate the accuracy of the CluSandra algorithm, as well as the scalability and reliability provided by the CluSandra framework. The experiments are considered small-scale where the framework comprised a cluster of 2-3 compute nodes. More large-scale testing that comprises larger clusters of compute nodes is planned for future work.

A. Test Environment and Datasets

Experiments designed to test the accuracy of the CluSandra algorithm were conducted on an Intel Core i5 with 8 GB of memory running OS/X version 10.6.8. The experiments included a real and synthetic dataset.

The real dataset was acquired from the 1998 DARPA Intrusion Detection Evaluation Program, which was prepared and managed by MIT Lincoln Labs. This is the same dataset used for The Third International Knowledge Discovery and Data Mining Tools Competition (KDD-99 Cup). According to [5], this dataset was also used to run experiments on the CluStream algorithm. The dataset is contained in a comma-separated values (CSV) file where each line comprises a TCP connection record; there are 4,898,431 records in the file. The dataset records two week's worth of normal network traffic, along with bursts of different types of intrusion attacks, that was simulated for a fictitious US military base. The attacks fall into four main categories: DOS (denial-of-service), R2L (unauthorized access from a remote machine), U2R (unauthorized access to local super user privileges), and PROBING (surveillance and other probing). Each connection record contains 42 attributes, which provide information regarding the individual TCP connection between hosts inside and outside the fictitious military base. Some example attributes are protocol (e.g., Telnet, Finger, HTTP, FTP, SMTP, etc), duration of the connection, the number of root accesses, number of bytes transmitted to and from source and destination. The connection records are not time stamped. Of the 43 attributes, 34 are of type continuous numerical. Every record in the dataset is labeled as either a *normal* connection or a connection associated with a particular attack. The following is a list of all possible labels, with the number in parenthesis being the total number of records in the dataset having that particular label: *back*(2203), *buffer_overflow*(30), *ftp_write*(8), *guess_passwd*(53), *imap*(1069), *ipsweep*(12481), *land*(21), *loadmodule*(9), *multihop*(7), *neptune*(1072017), *nmap*(2316), *normal*(972781), *perl*(3), *phf*(4), *pod*(264), *portsweep*(10413), *rootkit*(10), *satan*(15892), *smurf*(2807886), *spy*(2), *teardrop*(979), *warezclient*(1020), *warezmaster*(20). Together, the normal, smurf and neptune records comprise 99% of all records. The StreamReader (KddStreamReader) for this experiment reads each connection record and creates a DataRecord that encapsulates the 34 numerical attributes for that record. The KddStreamReader then sends the DataRecord the framework's MQS for processing by the MCA, which in this case is an implementation of the CluSandra algorithm. The KddStreamReader includes a filter that is configured to read all or any combination of records based on the label type. For example, the end-user can configure the KddStreamReader to process only the neptune records, only the smurf and neptune records, or all the records.

The first series of experiments focused on processing the more ubiquitous record types in the dataset. The first experiment in the series had the KddStreamReader process only the smurf records and assigned the MCA a time window that captured the entire stream and an MBT (i.e., maximum radius) of 1000; this same time window and MBT were maintained throughout this series of experiments. The result was one microcluster that had absorbed all 2,807,886 smurf records and had a relatively dense radius of 242.34. This experiment was executed five times with identical results. The second experiment was identical to the first, except that the KddStreamReader processed only the neptune records, which is the second most ubiquitous record type. The result was, once again, one microcluster that had absorbed all 1,072,017 neptune records, but with an even smaller radius of 103.15. The next experiment targeted the normal record, which is the third most ubiquitous record. One might expect that the result would, once again, be only one microcluster. However, the result was a set of 1,048 microclusters with very little to no overlap between the microclusters and a high degree of variance with respect to their radii and number of absorbed DataRecords. This relatively large set of microclusters is to be expected, because there exists a high degree of variance within a set of normal connections. In other words, in a TCP network, the usage across a set of normal connections is typically not the same; the connections are being used for a variety of different reasons. For example, some connections are being used for email (SMTP), file transfer (FTP), and

terminal interfaces (Telnet, HTTP). It was also noted that it took appreciably longer for this experiment to complete. This is due to the specified time window encompassing the data stream's entire lifespan, which results in a much larger number of microclusters in the in-memory working set. The StreamReader's next target was the ipsweep records. The result was one microcluster absorbing all but one ipsweep record; the radius of this one microcluster was 145.66. The distance between this microcluster and the one that had absorbed the remaining ipsweep record was 53,340. It was, therefore, assumed that this one neptune record was an outlier. Finally, the KddStreamReader was focused on the portsweep records. Unlike the previous attack-related experiments, where only one or two microclusters were produced, this one resulted in 24 microclusters; however, one microcluster absorbed 90% of all the portsweep records and it had a radius of 569. The other 23 microclusters had radii ranging in the 900-1000 range with very little overlap. Overall, the results of this first series of experiments indicated a high level of accuracy for the CluSandra algorithm. Also, in this series of experiments, the average throughput rate for the KddStreamReader was approximately 28,000 records per second.

In the next series of experiments, the KddStreamReader processed combinations of two or more connection record types. To start, the KddStreamReader processed only the neptune and smurf records, along with the same time window and MBT as the previous series of experiments. The result was two microclusters; the first having absorbed 2,807,639 records with a radius of 242.17, while the second absorbed 1,072,264 records with a radius of 101.52. It was clear that the first and second microcluster accurately grouped the smurf and neptune records, respectively. There was also no overlap between the two microclusters, but it was interesting to see that a relatively small number of smurf records were grouped into the neptune microcluster. The KddStreamReader was then configured to process all smurf, neptune and normal records. The result was a set of 1,053 microclusters. The most populated microcluster had absorbed 2,804,465 DataRecords and had a radius of 232.62. This is clearly the smurf microcluster, but note how it had lost approximately 3000 DataRecords. The second most populated microcluster had absorbed 1,438,988 DataRecords and had a radius of 497.34. This is clearly the neptune microcluster, but it had gained over 300,000 DataRecords and its radius had, as would be expected, appreciably increased from 101.52. It was concluded that a considerable number of normal DataRecords had been absorbed by the neptune microcluster. This can be addressed by reducing the MBT; however, this will also result in an appreciably larger number of denser normal microclusters. The third most populated microcluster had absorbed 140,885 DataRecords with a radius of 544.89. Population-wise, this is approximately one-tenth the size of the neptune microcluster.

To test the accuracy of the sliding time window, the window's size was reduced to capture the different bursts of attacks. It was noted from the analysis of the dataset file that the neptune and smurf attacks occur across a handful of different bursts. So the size of the sliding time window was reduced to 3 seconds, the MBT was kept at 1000, and the StreamReader processed only the neptune records. The result was 4 very dense microclusters (attacks) that had a radius of 100 or less, with 18 being the smallest radius. The first microcluster absorbed 15 DataRecords and its lifespan (duration) was only one second. The second microcluster was created 4 seconds later, absorbed approximately 411,000 DataRecords and its lifespan was 12 seconds. Thus there was a gap of 4 seconds, between the first and second microcluster, where there were no neptune attacks. The third microcluster was created 15 seconds after the second expired, absorbed approximately 450,000 DataRecords, and its lifespan was 9 seconds. Finally, the fourth microcluster was created 4 seconds after the third expired, absorbed approximately 211,000 DataRecords and its lifespan was 6 seconds. The test was repeated for the smurf connection records and the result was one microcluster that had absorbed all the DataRecords and had a lifespan of over one minute. The experiment was run again, but with a window of 2 seconds. This time, the result was 5 very dense microclusters; three had a lifespan of 1 or 2 seconds, one a lifespan of 12 seconds and the last a lifespan of 42 seconds. These experiments proved the accuracy of the sliding window.

The synthetic dataset was generated by a stream generator that is loosely based on the Radial Basis Function (RBF) stream generator that is found in the University of Waikato's Massive Online Analysis (MOA) open source Java package [8]. This RBF type of generator was used, because it produces data streams whose data distribution adhere to a spherical Gaussian distribution, which is the distribution that the CluSandra algorithm is designed to process. During its initialization, the RBF generator creates a set of randomly generated centroids. The number of centroids in the set is specified by one of the generator's configurable parameters. Each centroid, which represents a distinct class, is given a random standard deviation and a multivariate center that is a proper distance from all the other centroids' centers. Not ensuring a proper distance between centroids leads to ambiguous results, because the resulting radial fields associated with two or more centroids may overlap. In some cases, the amount of overlap is considerable. The number of variables or attributes assigned to the centroids' centers is also specified via a configurable parameter. A new data record is generated by first randomly selecting one of the centroids. Then a random offset with direction is created from the chosen centroid's center. The magnitude of the offset is randomly drawn from a Gaussian distribution in combination with the centroid's standard deviation. This effectively creates a normally distributed hypersphere of data records, with distinct density, around the corresponding centroid [14].

The first series of experiments, with the synthetic RBF generator, configured the generator to produce a data stream comprising 200,000 data records with 5 classes and whose records had five attributes. The MBT was set to 3.0 and the sliding time window captured the entire data stream. On some occasions, the result was as expected; i.e., five very dense microclusters with radii ranging from 0.18 to 1.4 and whose population of absorbed data records was rather evenly distributed. On other occasions, the result was more than five

microclusters. However, on these occasions, there were always five dense microclusters that had no overlap and absorbed the vast majority of the data stream's records. Of those five, there was always one that had absorbed substantially less data records than the other four. Using the CQL, it was noted that there was a considerable amount of overlap between this one microcluster and the other sparsely populated 'extra' microclusters. When the CluSandra framework's aggregator was run with an overlap factor of 1.0, that one microcluster absorbed all the extra sparsely populated microclusters. There was also a rather even distribution of data records across all five microclusters. A second series of experiments was executed that was identical to the first; the one exception being that the generator was configured to produce 7 classes, instead of 5. The results were consistent with those of the previous series of experiments.

VI. CONCLUSIONS AND FUTURE WORK

This work presents a distributed framework and algorithm for clustering evolving data streams. The Java-based framework, which is named the CluSandra framework, exhibits the following characteristics:

- It is entirely composed of proven open source components that can be deployed on a variety of commodity hardware and operating systems; therefore, it is very economical to implement.
- It is leveraged by clustering processes to address the severe time and space constraints presented by the data stream. In other words, the functionality required to address these constraints is offloaded from clustering process to the framework, and it is the framework that controls the data stream.
- It provides a distributed platform through which ensembles of clustering processes can be seamlessly distributed across many processors. This results in high levels of reliability, scalability, and availability for the clustering processes.
- It allows its hosted ensembles of clustering processes to elastically scale up and down to meet the most demanding dynamic data stream environments.
- It provides convenience classes or objects whose purpose is to facilitate the implementation of clustering algorithms and their subsequent deployment onto the framework.
- It provides an effective mechanism through which the hosted clustering processes can reliably and efficiently store their byproduct of real-time statistical summary information (i.e., microclusters) in a cluster database.
- It provides a Cluster Query Language (CQL) that is used to perform near-time or offline analytics against the cluster database. The CQL offloads the offline analytics from the clustering algorithm and provides a mechanism for the implementation of a variety of offline analytical processes. The CQL can also be used by offline processes to monitor, in near-time, clusters in the database and raise alerts whenever clusters of a particular nature appear and/or disappear.
- It lays down the foundation for a data management system whose focus is on clustering high speed and evolving data streams.

The algorithm developed is named the CluSandra algorithm and it is based upon concepts, structures, and algorithms introduced in [10] and [5]. The algorithm's implementation serves as an example of a clustering process that is designed to leverage the services and functionality provided by the CluSandra framework. The algorithm is more closely related to CluStream with its concept of viewing the data stream as a changing process over time and its functionality being divided between two operational phases: real-time statistical data collection and offline analytical processing. However, it deviates from CluStream primarily in how it addresses these operational phases. Unlike CluStream, it only performs the real-time statistical data collection, in the form of microclustering, leaving the offline analytical processing to end-users and/or processes that leverage the CQL. This results in a simpler and higher performing algorithm; primarily, because the agglomeration of microclusters is not performed by the algorithm. It also provides added flexibility to the end-user and/or offline analytical processes, because it affords the opportunity to analyze the collected data prior to agglomeration. Also, due to its reliance on the framework, the CluSandra algorithm can reliably and efficiently persist its microclusters to a cluster database; this is addressed in neither [10] nor [5]. The end result is the development of a clustering algorithm that exhibits these characteristics: configurable, distributable, elastically scalable, highly available and reliable, and simpler to implement.

The following are topics for future work:

- The implementation of clustering algorithms designed to address other data types and distributions, besides numerical and Gaussian, and their deployment onto the framework.
- The introduction of an integration framework, such as [18], that allows for the quick implementation of messaging design patterns meant to further control the data stream and facilitate the implementation of multi-staged data stream processing. For example, patterns to address data processing steps such as cleansing, standardization and transformation, and patterns used for routing data records based on their content.
- The implementation of a graphical user interface that leverages the CQL and provides a visual representation of the data in the cluster database.
- Additional research and development on algorithms that automate the calculation of an optimal MBT for the target data stream.

REFERENCES

[1] C. Walls, Spring In Action, 2nd ed., Greenwick, CT: Manning Publications Co., 2008.

[2] R. Pressman, Software Engineering A Practitioners Approach, 7th ed., New York: McGraw-Hill, 2009.

[3] P. Domingos, G. Hulten, "Mining high-speed data streams", in Knowledge Discovery and Data Mining, 2000, pp.71-80.

[4] P. Domingos, G. Hulten, L. Spencer "Mining time-changing data streams", in ACM KDD Conference, 2001, pp.97-106.

[5] C. Aggarwal, J. Han, J. Wang, P.S. Yu, "A framework for clustering evolving data streams", in Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003.

[6] J. Gama, Knowledge Discovery From Data Streams. Boca Raton, FL: Chapman & Hall/CRC, 2010.

[7] J. Gama, M. Gaber, Learning From Data Streams, Processing Techniques in Sensor Data, Berlin-Hiedelberg: Springer-Verlag, 2007.

[8] B. Bifet, R. Kirkby, R., Data Stream Mining, University of Waikato, New Zealand: Centre for Open Software Innovation, 2009 .

[9] E. Hewitt, Cassandra, The Definitive Guide. Sebastopol, CA: O'Reilly Media, Inc., 2010.

[10] T. Zhang, R. Ramakrishnan, M. Livny, "BIRCH: an efficient data clustering method for very large databases", in ACM SIGMOD, 1996, pp. 103-114.

[11] G. Hebrail, Introduction to Data Stream Querying and Processing, International Workshop on Data Stream Processing and Management, Beijing: 2008.

[12] J. Han, M. Kamber, Data Mining Concepts and Techniques, 2nd ed., San Francisco, CA: Morgan Kaufmann Publishers, 2006.

[13] B. Everitt, S. Landau, M. Leese, Cluster Analysis, 4th ed., London, England: Arnold Publishers, 2001.

[14] B. Bifet, R. Kirkby, Massive Online Analysis Manual. University of Waikato, New Zealand: Centre for Open Software Innovation, 2009.

[15] "The Apache Cassandra Project." Available: http://cassandra.apache.org, [Accessed Sept. 2011].

[16] "Apache ActiveMQ." Available: http://activemq.apache.org, [Accessed Sept. 2011].

[17] T. White, Hadoop: The Definitive Guide, Sebastopol, CA: O'Reilly Media, Inc., 2010.

[18] "Apache Camel." Available: http://camel.apache.org, [Accessed Sept. 2011].

[19] C. Plant, C. Bohm, Novel Trends In Clustering, Technische Universitat, Munchen Germany, Ludwig Maximialns Universitat Munchen, Germany, 2008.

[20] L. Kuncheva, "Classifier ensembles for detecting concept change in streaming data: overview and perspectives", in Proceedings of the Second Workshop SUEMA, ECAI, Partas, Greece, July 2008, pp. 5-9.

[21] C. Aggarwal, J. Han, J. Wang, P.S. Yu, "A framework for projected clustering of high dimensional data streams", in Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004, pp. 852-863.

AUTHORS' PROFILES

Jose R. Fernandez received his MSc (2011) in Computer Science at the University of West Florida and his BSc (Eng., 1983) in Computer and Information Sciences at the University of Florida. He has over 25 years of commercial software engineering experience and has held senior architecture and management positions at NCR, AT&T, and BEA Systems. He has also been instrumental in starting several companies whose focus was on the development of Java Enterprise Edition (JEE) software products. He is currently a senior consultant with special research interests in machine learning and data mining.

Eman M. El-Sheikh is the Associate Dean for the College of Arts and Sciences and an Associate Professor of Computer Science at the University of West Florida. She received her PhD (2002) and MSc (1995) in Computer Science from Michigan State University and BSc (1992) in Computer Science from the American University in Cairo. Her research interests include artificial intelligence-based techniques and tools for education, including the development of intelligent tutoring systems and adaptive learning tools, agent-based architectures, knowledge-based systems, machine learning, and computer science education.