CluSandra

A Framework for Data Stream Cluster Analysis

Design Specification
v0.5

Jose Fernandez
jrf15@students.uwf.edu

Department of Computer Science, University of West Florida
11000 University Parkway, Bldg. 4
Pensacola, FL 32514
(850) 474-2542

Table of Contents

Revision History

Name	Date	Reason For Changes	Version
Jose Fernandez	June 24, 2011	Initial Release	0.1
Jose Fernandez	June 29, 2011	Streamlined the Timeline Index	0.2
Jose Fernandez	July 5, 2011	Eliminated the need for MCA swarm to *share* microclusters from the data store. Each instance in swarm maintains its own microcluster set. This removes dependency for distributed locking package and also removes contention within swarm. Similar microclusters that temporally overlap can be aggregated by the ClusandraAggregator.	0.3
Joe Fernandez	July 15, 2011	Added captions to several figures and addressed incorrect years in a couple of references.	0.4
Joe Fernandez	August 7, 2011	Introduced both the formula used to calculate the standard deviation and its source code implementation. Updated how the timeline (cluster index) is managed. Updated the section that describes the maximum boundary threshold and how it is derived. Updated the section on the aggregator. Made minor revisions throughout the document.	0.5

1 Introduction

1.1 Scope

This document is the high level design for the CluSandra data stream clustering framework as defined in the CluSandra product definition (Fernandez, 2011). The goal of the CluSandra project is to design and develop a framework that facilitates the application of clustering analysis on one or more evolving high-speed data streams. The project also includes the design, development and deployment of a clustering algorithm onto the framework. This first algorithm, which this project provides, is simply referred to as the CluSandra algorithm.

The intended audience for this document includes anyone that will be involved in the design, development, testing, and extension of the CluSandra framework.

This project's initial work products may serve as the basis for an ongoing data stream management system (DSMS) project at the University of West Florida and/or whose work products can be contributed to an open source machine learning community such as Apache Mahout (http://mahout.apache.org). The project incorporates disciplines such as data mining, distributed systems, machine learning, statistics and various areas of mathematics (e.g. linear algebra). As such, the CluSandra framework can also serve as an excellent learning platform and whose extension can be realized by subsequent cross-disciplinary group projects.

The CluSandra framework will be written entirely in the Java programming language and its source code will be compatible with version 1.5 and higher of the language. Using Java as the programming language provides maximum portability and productivity. Java is available on all the major operating platforms and has an extremely rich set of utility libraries.

To mitigate development costs and take advantage of object reuse, the CluSandra framework leverages existing components that are offered by the following open source Java projects:

Organization	Projects	Description
New Zealand's University of Waikato	Massive Online Analysis (MOA) http://moa.cs.waikato.ac.nz/	MOA is a data stream mining library that includes machine learning algorithms, utilities, and tools used for testing and experimentation. MOA includes an implementation of the CluStream algorithm; this project will implement and deploy a derivative of CluStream, as well as leverage machine learning utilities offered by the framework.
New Zealand's University of Waikato	WEKA http://www.cs.waikato.ac.nz/ml/weka/	WEKA is a data mining framework that also includes machine learning algorithms and tools for testing and experimentation. Unlike MOA, WEKA is not designed for data streams; however, MOA does reuse some of the machine learning algorithms provided by WEKA.
Apache Software	Cassandra	Cassandra is an open source,

Foundation (ASF)	http://cassandra.apache.org/ ActiveMQ http://activemq.apache.org/	highly scalable, and distributed NoSQL or non-relational database management system (DMS). It will serve as this project's data store. ActiveMQ is a distributed, high-performance message queuing system. It will be used by the framework to reliably buffer data stream records.
Springsource.org	Spring http://www.springsource.org/	Spring is the leading platform for building Java applications. It provides a rich set of services for building Java applications and thus mitigates development costs and adds to overall productivity.

This paper is structured as follows. The rest of the introductory subsections provide the reader with a brief overview of data streams and cluster analysis. Section two describes the BIRCH and CluStream clustering algorithms. These two algorithms are covered in some detail, because the first clustering algorithm that will be developed and deployed on the CluSandra framework is heavily based on concepts and structures introduced by these two algorithms. Section three defines the framework's high-level data flow model (i.e., DFD) and provides an overview of the design. Section four defines the Cassandra data models for the CluSandra framework. Please note that Cassandra is a non-relational database system; therefore, traditional data modeling diagrams (e.g., Entity Relationship Diagrams) cannot be used. Section five provides the class diagrams and section six provides a quick overview on a SQL-like command line interface.

If the reader has not had any exposure to the Cassandra database system, she or he is strongly advised to spend a bit of time reviewing the following article, which helps explain the Cassandra data model: http://tinyurl.com/yawwgql . Section 4.1 also provides a brief overview of the Cassandra data model.

1.2 Data Stream

Unlike a traditional database or data set, a data stream is an ordered sequence of structured data records (a.k.a., data objects, tuples or n-dimensional vectors) that exhibit these characteristics: very fast arrival rate, temporally ordered, fast evolving, and unbounded. Its unbounded nature makes it unfeasible to store all the data that the stream produces in any form of secondary storage. The evolving data stream can be viewed as a fast and raging river of data whose arrival rate and composition can be assumed to be constantly evolving. One of the goals of this project is to develop a framework that can control this river of data so that one can harness knowledge from its evolving patterns. And so as a river of water is controlled by a dam to harness electricity from the river, the CluSandra framework serves as the data stream's dam such that knowledge can be harnessed from the data stream.

Examples of data streams include sensor networks, wireless networks, radio frequency identification (RFID), customer click streams, telephone records, multimedia data, scientific data, sets of retail chain transactions etc. What distinguishes current data from earlier one is automatic data feeds. We do not just have people who are entering information into a computer. Instead, we have computers entering data into each other (Gama & Mohamed, 2007). Nowadays there are applications in which the data are modeled best as transient data streams instead of as persistent tables. In these applications it is not feasible to load the arriving data into a traditional data base management system (DBMS) and traditional DBMSs are not designed to directly support the continuous queries required by these applications (Gama & Mohamed, 2007).

In this paper, a data stream S is viewed as an unbounded sequence of pairs $\langle s,t \rangle$, where *s* is a structured data record (set of attributes) and $t \in T$ 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 (Gama & Mohamed, 2007). The timestamp values of one data stream are independent from those of any other data stream that is being processed within the CluSandra framework.

This project assumes that data records entering the CluSandra framework are not necessarily time stamped; therefore, in those cases where data records are not time stamped, it is the responsibility of CluSandra, or one of its components, to immediately timestamp data records as soon as they are read from the data stream. These data records will also need to be wrapped by an object that implements a generic CluSandra data record interface. More on this later in the document.

Since data streams comprise structured records, streams (e.g., audio and video streams) comprising unstructured data are not considered data streams within the context of this project. Example data streams include IP network traffic, web click streams, mobile calls, ATM transactions, credit card transactions, and sensor network traffic. Therefore, a data stream produces a vast and endless amount of data that is assumed to continuously evolve over time, cannot be stored in a file system or data base of any kind, and at times can arrive at exceedingly fast rates.

1.3 Cluster Analysis

Clustering is the process by which one partitions objects into groups based on similarity. That is, all objects in a particular group are similar to one another, while objects in different groups are quite dissimilar. With respect to artificial intelligence and machine learning, cluster analysis can be regarded as "unsupervised pattern recognition". It automates the grouping process and allows the end-user to discover and explore distribution patterns within the data.

Cluster analysis lies at the intersection of many disciplines such as statistics, machine learning, data mining, and linear algebra (Han & Kamber, 2006). It is also used for many applications such as pattern recognition, fraud detection, market research, image processing, network analysis, psychology and biology.

1.3.1 Supervised vs. Unsupervised Learning

In most applications, clustering seeks to partition *unlabeled* data records from a large data set into clusters. The process of learning or gaining knowledge from an unlabeled data set is referred to as *unsupervised* learning.

Unlike the *supervised classification* algorithms, clustering algorithms do not rely on labeled training sets to partition data records into their respective clusters. It is said that clustering is a form of learning by observation instead of learning by example. Clustering takes a somewhat opposite approach from that of supervised learners, because it first processes unlabeled data records to group them into clusters; the clusters can then be assigned labels. Clustering presents an attractive advantage in that it is more easily adapted to changes in the data and can thus be used to identify features that distinguish different clusters (Han & Kamber, 2006). This is ideal for concept drifting data streams.

1.3.2 Data Types and Distance Measures

Depending on the application, clustering may need to take into account different data types and their preprocessing. The following are some examples of the different data types: numerical or interval scaled, binary, categorical, ordinal, and ratio-scaled. In some cases, clustering must preprocess a mixture of all these types. CluSandra's first clustering algorithm will only process numerical data; therefore, this project will only focus on that data type. In the future, additional algorithms, designed to handle the other data types, can be developed and deployed onto the CluSandra's framework.

This project will also assume that the values for all the data records' attributes or variables are *standardized*; therefore, there is no preprocessing of the data records. In other words, the records' attributes will be presented in the same units of measure. For example, all attributes that represent a height will be presented in meters and will not be a mixture of meters and yards or yards and feet. Again, future iterations of CluSandra can introduce preprocessing capabilities.

Numerical data types are continuous measurements of a roughly linear scale (Han et. al. 2006). Examples of continuous data or variables are crime rates, height, weight, and temperature. When working with data records comprising continuous attributes, the similarity or dissimilarity between individual records is typically quantified by some form of distance measure. There exists a variety of distance measures that are applied to continuous data. The following are some examples: Euclidean distance, city block or Manhattan distance, Minkowski distance, Caneberra distance, Pearson correlation, and Angular separation. Most if not all of these distance measures allow for some form of differential weighting of the variables. For example, a temporal weight may be assigned to the measures that place less importance on the measures as time progresses (e.g., time decay).

The most common distance measurement used for continuous numerical data is the Euclidean measure

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

where x_{ik} and x_{jk} are the k^{th} variable 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 would be the following:

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

If we view the two records as Euclidean vectors in Euclidean n-space (Figure 1.a), the distance between the two vectors is the length of the line connecting the two vectors' tips (points). The lower the resulting value, the closer (similar) are the two data records. Euclidean distance is an integral part of this project.

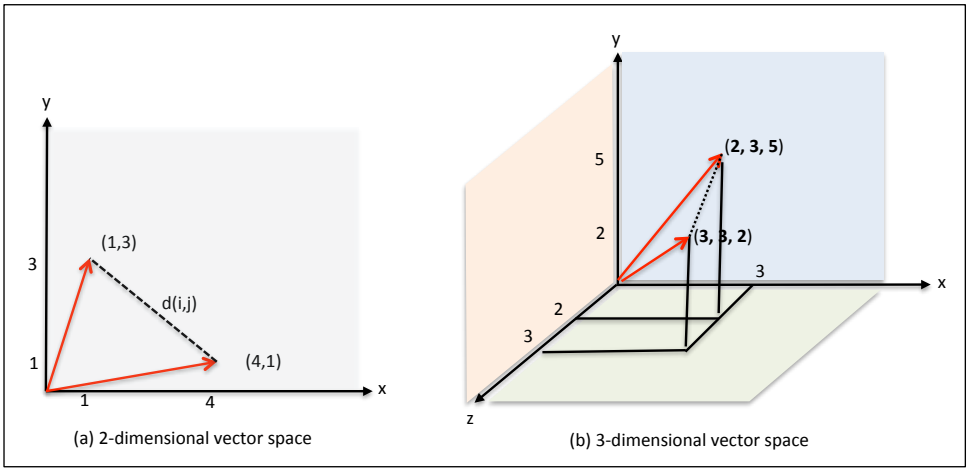


Figure 1. 2 and 3 dimensional Euclidean space

Figure 1 depicts two vectors in 2 and 3-dimensional Euclidean spaces. The Euclidean distance formula can be applied to any n-dimensional vector. A data type that satisfies the requirements of Euclidean space and triangular inequality is also referred to as a *metric* variable or attribute. Triangular inequality states that for any three points in n-dimensional space, i, j, and k, $d(i,j) + d(j,k) \geq d(i,k)$.

The clustering algorithm that will be developed by this project and deployed on to this initial release of CluSandra utilizes Euclidean distance as a measure to determine how similar or close a new data record is to an existing cluster's centroid (mean). It is also used to find the distance between two clusters' centroids.

2 BIRCH and CluStream

2.1 BIRCH

The CluSandra algorithm, which is the first clustering algorithm deployed onto the CluSandra framework, is heavily based on the concepts and structures introduced by the BIRCH (Zhang, Ramakrishnan & Livny, 1996) and CluStream (Aggarwal, Han, Wang, Yu, 2003) clustering algorithms. BIRCH is an acronym for "Balanced Iterative Reducing and Clustering". CluStream, which is based on BIRCH and extends it, is specifically designed to address evolving data streams.

There are many types of clusters that are represented by n-dimensional vector data. BIRCH, and thus CluStream, are based on the K-means (center-based) clustering paradigm; therefore, the CluSandra algorithm targets perhaps the simplest type of cluster, which is referred to as the *spherical Gaussian* cluster. Clusters that manifest non-spherical or arbitrary shapes, such as *correlation* and *non-linear correlation* clusters, are not addressed by the CluSandra algorithm. However, it may be feasible to deploy algorithms that address these cluster types onto the CluSandra framework.

The BIRCH incremental clustering algorithm was designed to mitigate the I/O costs associated with clustering very large multi-dimensional data sets. In general, BIRCH is a batch algorithm that includes and relies on multiple sequential phases of operation and is therefore not well-suited for data stream environments where time is severely constrained. However, it does introduce concepts and, in particular, a synopsis structure that can be applied to the clustering of data streams. For example, the algorithm can typically find a good clustering with a single scan of the data (Zhang, et. al., 1996), which is an absolute requirement when having to process data streams. BIRCH also introduces two structures: *cluster feature* (CF) and *cluster feature tree* (CF tree). This project is heavily dependent on the CF; however, it does not make use of the CF tree.

Before discussing the CF structure, we'll discuss these three spatial terms: *centroid*, *radius*, and *diameter*. All three are related to vectors and vector spaces and are an integral part of the BIRCH algorithm.

Given N n-dimensional data objects (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 follows:

$$\vec{x}_0 = \frac{\sum_{i=1}^{N} \vec{x}_i}{N}$$

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

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

The centroid is the cluster's mean or center of gravity, 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 of the objects around their cluster's centroid. The radius can also be viewed as the "root mean squared deviation" or RMSD with respect to the centroid. Note that the centroid itself is a data record or vector.

The clustering algorithm that this project is to develop uses the RMSD, but does not keep track of every data point; doing so is not feasible when processing data streams. So to arrive at the RMSD, the algorithm maintains statistical summary data in the form of the linear sum and linear sum of the squares with respect to the elements of the n-dimensional data records (more on this in the next section). This allows the algorithm to calculate the RMSD, as follows, and still only make one pass at all the data records.

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

For example, lets suppose we have 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. Here is the Java source code for deriving RMSD per the formula above.

```
// Get the root mean squared deviation (standard deviation)
private double getDeviation() {
  double[] variance = getVarianceVector();
  double sumOfDeviation = 0.0;
  for (int i = 0; i < variance.length; i++) {
        sumOfDeviation += Math.sqrt(variance[i]);
  }
  return sumOfDeviation;
}

// Get the variance used to calculate the standard deviation
private double[] getVarianceVector() {
  double[] res = new double[LS.length];
  for (int i = 0; i < LS.length; i++) {
        double lsDivNSquared = Math.pow(LS[i],2)/N;
        res[i] = ((SS[i]) - lsDivNSquared)/(N-1);
  }
  return res;
}
```

In the above source code snippet, LS is a double byte array that holds the linear sum and SS is another double byte array that holds the sum of the squares. This information is maintained in the synopsis structure referred to as a cluster feature (more on this in the next section) or cluster for short.

Typically, the first criteria for assigning a new object to a particular cluster is the new objects Euclidean distance (similarity, dissimilarity) to a cluster's centroid. That is, the new object is assigned to it closest cluster. To calculate the distance between two clusters, you can also apply the Euclidean distance formula to the two clusters' centroids. The one disadvantage to relying on these spherically related spatial concepts is that you cannot apply the algorithm to clusters that are not, by nature, spherically shaped; i.e., do not follow a Gaussian distribution.

2.1.1 The Cluster Feature (CF)

BIRCH introduces the CF, which is a 3-tuple or triplet synopsis structure whose elements summarize the statistical information required to calculate the centroid, radius, and diameter of a cluster. In other words,

the CF captures summary information for a particular pattern in the data set. The CF vector is formally defined as

$$CF = \langle N, LS, SS \rangle$$

where N is the total number of objects in the cluster, LS (vector) is the linear sum of the objects in the cluster and SS (vector) is the squared sum of the objects in the cluster.

$$LS = \sum_{i=1}^{N} \vec{x}_i$$

$$SS = \sum_{i=1}^{N} \vec{x}_i^2$$

For example, suppose you have three 2-dimenisonal data records (2,5), (3,2) and (4,3) in a cluster C_1. The CF of C_1 is $\langle 3,(2+3+4,5+2+3),(4+9+16,25+4+9) \rangle = \langle 3,(9,10),(29,38) \rangle$. From the summary information held by the CF, you can calculate the centroid, radius, and diameter of the cluster and thus there is no need to store all the cluster's objects. What is also interesting about CFs is that they are *additive* and *subtractive*. For example, if you have two clusters, C_1 and C_2 with their respective cluster features, CF_1 and CF_2, the CF that is formed by merging the two clusters is simply $CF_1 + CF_2$.

Since it is not feasible to store each and every data record in an unbounded data stream, we instead store synopsis data structures like the CF. These structures are designed to contain enough statistical information to allow for the exploration and discovery of patterns in the data stream.

2.2 CluStream

As previously mentioned, CluStream is a clustering algorithm that is specifically designed for evolving data streams and borrows from the BIRCH algorithm. From an analysis perspective, the general intent of CluStream is to address the one-pass constraint imposed by data streams on the design of existing data stream clustering algorithms. For example, the results of applying a one-pass clustering algorithm, like BIRCH, to a data stream lasting 1 or 2 years would be dominated by outdated data. CluStream allows end-users to explore the data stream over different time windows or horizons and thus get an understanding of how and when the data stream has evolved over time.

CluStream tackles the one-pass constraint by dividing the clustering process into two phases of operation that are meant to operate simultaneously, if desired. The first, which is referred to as the *online* phase, efficiently computes and stores summary statistics about the data stream in *microclusters*. A microcluster is an extension of the CF whereby the CF is given a temporal dimension. The second phase, which is referred to as the *offline* phase, allows end-users to perform *macroclustering* operations on the 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 sets of microclusters (snapshots) 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 extends CluStream and the CluSandra framework's design is based on the concept of there being a microclustering and macroclustering phase of operation.

2.2.1 Microcluster

As previously mentioned, a data stream S is viewed as an unbounded sequence of pairs ⟨s,t⟩, where *s* is a structured data record and t ∈ T is the timestamp that specifies the arrival time of data record *s* on stream S. The CluStream microcluster extends the BIRCH CF structure by adding two temporal scalars to the CF. The first scalar is the sum of the timestamps of all the data records whose summary information is contained in the CF and the second is the sum of the squares of the timestamps. Thus the CF triplet is extended as follows.

$$CF = \langle N, LS, SS, ST, SST \rangle$$

ST is the sum of the timestamps and SST is the squared sum of the timestamps. Note that this new extended CF retains its additive and subtractive properties. From henceforth, this new structure is referred to as a microcluster. The ST and SST can be applied to the following formula, which was introduced earlier,

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

to arrive at the temporal standard deviation of the cluster as follows:

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

2.2.2 Online Phase

CluStream's online microclustering phase 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.

The algorithm defines a fixed set of *q* microclusters that it creates and maintains in-memory. This set, $\mathcal{M} = \{\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_q\}$ is referred to as the current online snapshot with *q* being the maximum number of microclusters in the snapshot, with each microcluster in $\mathcal{M}$ being given a unique *id*. The value for *q* is based on the available amount of memory; however, there is no mention of how *q* is calculated. The CluStream papers do state that the algorithm is very sensitive to any further additions to the microcluster data structure (i.e., the CFT), as this negatively impacts *q*. This type of space constraint is one that the CluSandra framework lifts by relying on Cassandra to serve as a highly reliable and scalable real-time distributed data store.

The updating of the microclusters is similar to that of the BIRCH method. When a new data record is presented by the data stream, it is assigned to the closest microcluster $\mathcal{M}_i$ in $\mathcal{M}$. Most of the algorithm's time is spent finding the closest microcluster in $\mathcal{M}$. After finding the closest microcluster $\mathcal{M}_i$, it is then determined if $\mathcal{M}_i$ can absorb the new record based on a maximum boundary threshold. CluStream defines this threshold as, "a factor of *t* of the RMSD of the data points in $\mathcal{M}_i$ from the centroid", where RMSD is the root mean squared deviation. This is just another way of referring to the standard deviation of the cluster times sum factor *t* to arrive at the final radius. For example, if t = 1.6 and the standard deviation is 2, then the final radius or maximum boundary threshold is 3.2. The microcluster $\mathcal{M}_i$ is updated whenever it absorbs a new data record. If $\mathcal{M}_i$ has only one entry, the RMSD cannot be calculated; therefore, for those clusters having only one entry, the maximum boundary is derived as the distance from $\mathcal{M}_i$ to its closest

neighbor times a factor *r*. The CluSandra algorithm will use a similar approach in both locating the closest cluster and determining whether the targeted cluster can absorb the data record.

If $\mathcal{M}_i$ cannot absorb the new data record, a new microcluster must be created to host the data record. To conserve memory, this 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.

CluStream uses the temporal scalars (ST, SST) of the microcluster, in combination with a user-specified threshold δ, 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 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 it refers 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.

CluStream classifies snapshots 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 (Aggarwal et. al., 2003). The number of snapshots stored over a period of T time units is

$$(\alpha + 1) * \log_2(T)$$

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:

$$(2 + 1) * \log_2(100 * 365 * 24 * 60 * 60) \approx 95$$

CluSandra will not have to operate within the same space (memory) constraints as CluStream; therefore, for CluSandra'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 $\mathcal{M}$. If a new microcluster is required, it will be created by a CluSandra microclustering agent (MCA) and simply added to the Cassandra data store. Also, the MCAs will write directly to the Cassandra data store; therefore, there is no need for a periodic time interval scheme that persists $\mathcal{M}$ to secondary storage.

CluStream cannot guarantee that microclusters will not be lost. After a microcluster is added to $\mathcal{M}$, there exists the possibility that the machine may fail prior to the microcluster being persisted, in which case it will be lost. Also, CluStream does not address $\mathcal{M}$'s recoverability. If the machine fails and is restarted,

CluStream cannot recover $\mathcal{M}$'s state prior to the failure. In other words, there is no transactional integrity applied to $\mathcal{M}$. One of the goals of the CluSandra framework is to introduce the necessary components that can guarantees the persistence of microcluster. More on this later in the document.

2.2.3 Curse of Dimensionality

Conventional clustering methods based on K-means (e.g., BIRCH and CluStream) that have been designed to identify clusters from the whole data space may suffer from what is called, *curse of dimensionality* (Plant & Bohm, 2008). This curse is directly related to the sparsity of the data inherent in high-dimensional vectors/data records. With increasing dimensional space, all pairs of data records tend to be almost equidistant from one another. Another problem with high-dimensional data is that there exists clusters embedded within subspaces of the high dimensional data; at times, different clusters may exists in different subspaces. As a result, it may be unrealistic to apply distance-based measures to high-dimensional data records for discovering clusters (Aggarwal, Han, Wang, Yu, 2004). Relatively recent work uses a technique called, *projected clustering* to address this curse. Projected clustering discovers clusters based on specific subsets of the dimensions, which in turn alleviates the sparsity issue. Even though sparsity of the data may preclude discovering meaningful clusters from on all the dimensions, some subset of the dimensions can always be found that yield meaningful clusters (Aggarwal et. al., 2004). The people behind CluStream designed another CF-based algorithm called, HPStream whose design attempts to address this curse of dimensionality. The CluSandra project does not attempt to address this curse and works with a conventional clustering method and full-dimensional data records. The project assumes that the data record's number of dimensions are relatively low and thus the initial clustering algorithm employed by CluSandra does not fall victim to this curse. However, it should be noted that CluSandra is a clustering framework, and not just a specific clustering algorithm, that can accommodate many CF-based algorithms. Future work may apply HPStream, or derivative of, to the CluSandra framework.

3 CluSandra Context Level Data Flow

Figure 2 is a context-level (level 0) data flow diagram (DFD) that represents, at a high-level, the CluSandra framework as a whole.

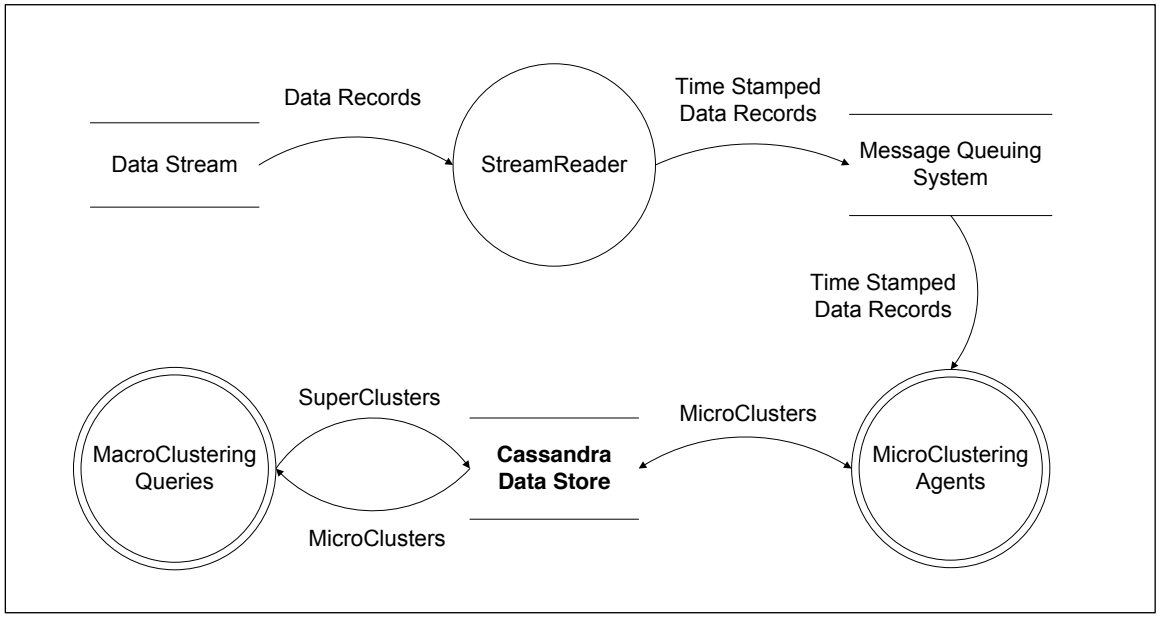


Figure 2. Context Data Flow Diagram for CluSandra

As the reader may have surmised, the CluSandra framework is heavily based on temporal, as well as spatial aspects of clustering. It is used to recognize patterns in the data stream according to spatial characteristics

and records those patterns according to their temporal characteristics. For example, it answers the following temporally related questions: when do patterns occur, how long do they remain active, when do they reappear.

The Cassandra data store comprises two or more distributed machines configured in a peer-to-peer ring network topology. The ring of nodes is referred to as a Cassandra cluster. Each node in the cluster may also optionally host CluSandra's executable components. Cassandra is a distributed hash table (DHT) and will serve as CluSandra's distributed database. CluSandra stores micro and superclusters in its distributed database or data store. More information on Cassandra can be found in the CluSandra product definition (Fernandez, 2011) and the Cassandra web site: http://cassandra.apache.org/

The following subsections provide a high-level overview of all the components, data flows and data stores depicted in the DFD.

3.1 StreamReader

The StreamReader component is primarily responsible for reading structured data records directly from the data stream, wrapping the data records in objects that implement a CluSandra-specific data record interface (DataRecord), and then placing these objects in the CluSandra message queuing system (MQS), where they are temporarily stored for subsequent processing. If not already time stamped, the StreamReader time stamps the DataRecord objects prior to being given to the MQS. The time stamps are critically important, as they record the instant of time that the DataRecord[1] was produced by the stream. More information on the time stamps and their usage will be provided later in this document.

The raw data record that is read by the StreamReader, from the stream, is essentially a vector containing one or more continuous numerical values; therefore, that vector can be represented simply as a Java vector whose elements are of type double. This vector is encapsulated by the DataRecord and represents the DataRecord's payload. Again, this first release of CluSandra does not support other data types (e.g., categorical, discrete, binary, etc.); only continuous numerical data types.

The StreamReader must keep up with the data stream's arrival rate, which is assumed to be in the 1000s of records per second. Therefore, the StreamReader's design and implementation must take into account this extreme time constraint. To ensure utmost performance and throughput, it is important that the StreamReader send or deliver DataRecords to the MQS in an *asynchronous*, and not synchronous, manner. Another approach used for ensuring good throughput to the MQS is to send a batch of DataRecords within the context of a local transaction. More on this later in the document.

It is entirely possible to have many StreamReaders concurrently active within the CluSandra framework, with each reader reading from its assigned and distinct data stream. As this document will later describe in more detail, the CluSandra framework is capable of concurrently processing many data streams. A StreamReader needs to be developed for each and every data stream type, because each StreamReader must understand the format or structure of its corresponding data stream's records. However, the CluSandra framework provides a StreamReader superclass that facilitates the development of the different readers.

CluSandra's MQS provider can be remotely accessed via a TCP/IP network; therefore, the StreamReader component does not have to reside on the same node as the MQS provider. The StreamReader can even be embedded within the component or device that produces the data stream. For example, the StreamReader can be embedded within a sensor device or network router. You can have many StreamReaders distributed across a population of such devices all writing to the same MQS queue.

[1] The MOA framework has an interface called, *Instance* that is similar to the DataRecord interface.

3.2 Timeline Index (TI)

Before reading this section, you may want to jump-ahead and read the beginning of section 4 for a quick overview of Cassandra Columns, SuperColumns, ColumnFamilies and SuperColumnFamilies.

The TI is a very important temporal-based index structure that is maintained in the CluSandra data store and it is used to maintain the micro and supercluster objects that reside in the CluSandra data store. Superclusters will be described later in this document. The TI is not reflected in the high-level DFD that is depicted in Figure 4, but it is important that it be discussed prior to the remaining subsections of section 3.

Maintaining micro and supercluster objects implies creating the objects, as well as accessing and deleting them from the CluSandra data store. 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, which represents the data stream's active timeline or lifespan, 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 micro and superclusters that were or are still active during that second in time. In Cassandra, a row key is essentially a primary key. Temporal-based queries are used to select clusters from the CluSandra data store. These queries are temporal, because they must specify a time horizon over the stream's lifespan. The query returns all the clusters that were active during that time horizon and does so by referencing the TI.

A data stream's lifespan can run for any amount of time. During development and test, the stream's lifespan may extend over just a couple of hours; however, in production environments, the lifespan may extend over months, if not years.

Figure 3 provides an abstract view of the TI's implementation, which is comprised of the Cluster Index Table (CIT). The Cluster Table is not part of the TI. The CIT, which is a Cassandra SuperColumnFamily, includes a row for each day and the row comprises 86400 SuperColumns; one for each second of that particular day. Each SuperColumn contains one or more Columns whose values are keys to micro and superclusters in the Cluster Table. Please note that Cassandra is a column-oriented database and thus 86400 SuperColumns in a row is not really an issue. Also, a SuperColumn can be assigned different names from row-to-row, and in this case the name of a SuperColumn is a timestamp (milliseconds) that corresponds to a second for that day. Each row key of the CIT is given a value that corresponds to the zero'th second of a particular day. For example, lets suppose the date is, "Wed Jun 29, 2011". The row key for that particular date would have a value of "1309320000000". This same value would also be assigned to the name of the first supercolumn of that row since it is the first second of the day. The next supercolumn (i.e., next second) in that row would be assigned "1309320001000", followed by "1309320002000", and so on. The second row (i.e., next day) would be assigned, "1309406400000" and the sequence of supercolumn names for that row would be, {"1309406400000", "1309406401000", "1309406401000",…, "1309406483000"}.

Each node in a Cassandra cluster manages one or more rows in a Column or SuperColumnFamily; therefore, if you intend to have many, many Columns (e.g., in the 100,000s) or SuperColumns in a particular row, it is best to partition them across multiple rows. For example, the TI's data model could have been designed such that the CIT has only one row containing all possible seconds in the data stream's lifespan. However, with that sort of design, the one row in the CIT would have millions of SuperColumns and the burden of maintaining this one row would fall on one node in the cluster. This would have placed a heavy burden on one node in the cluster and thus created a bottleneck of sorts.

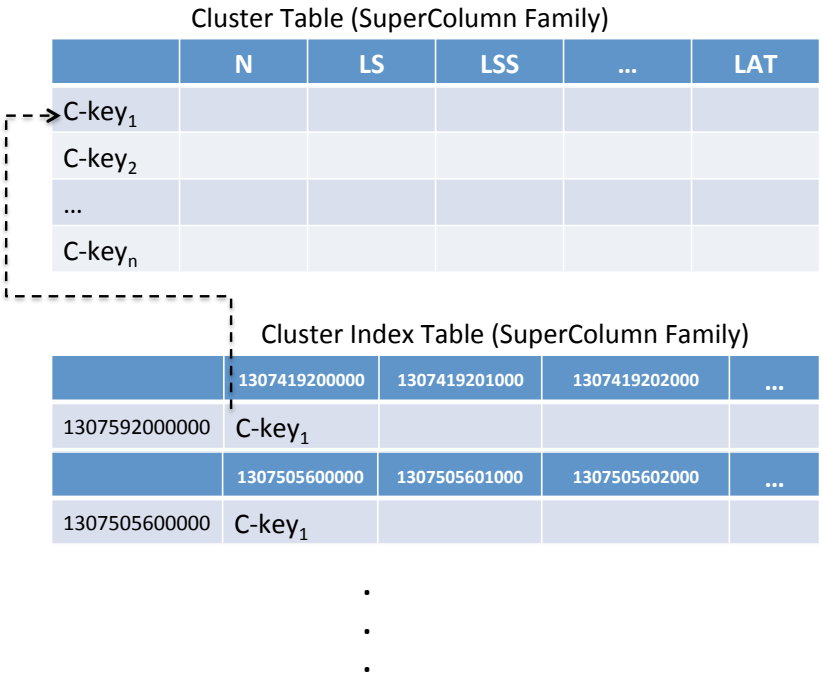


Figure 3. Timeline Index

The motivating factor behind the TI is that Cassandra is not currently set up to perform all that 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. Relatively recently, Cassandra has been upgraded so that you can define a column as an index key (native key), which does help with returning ranges of rows, but the column that is chosen as the index must have a small set of discrete values.

3.3 CluSandra Message Queuing System (MQS)

The 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, which in this case are DataRecords. So as a dam is used to control a wild raging river and thus 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. As you'll read in the next section, the Microclustering Agent (MCA) is analogous to a turbine within the dam, but instead of producing electricity, it produces a series of microclusters from which one can gain knowledge about the data stream. It is also the responsibility of the MCA, or group of identical MCAs, to ensure that the queue's level of DataRecords be maintained at an acceptable level.

As previously mentioned, CluSandra can support the simultaneous processing of many data streams; therefore, there may very will be many queues defined within the MQS; one queue for each data stream.

There are many open source MQSs that can be leveraged for this project. Some examples are ActiveMQ, RabbitMQ, HornetQ, and openJMS. The CluSandra framework is designed to utilize any MQS that implements the Java Message Service (JMS) API. The vast majority of MQSs, both open source and commercial, implement the JMS API. 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 API or specification, the process that places messages in the MQS's queues is referred to as a "*producer*", those processes that read messages from the queues is referred to as a "*consumer*", while both are generically referred to as *clients*.

As previously mentioned, 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, in batches or sets, across one or more instances of the MicroCluserting Agent (MCA) component (consumer); the MCA is described in the next subsection. A group of identical MCA processes that consume DataRecords from the same queue is referred to as a MCA swarm. Many MCA swarms can be defined, with each swarm reading from its unique queue. It is important that each individual member of a swarm process sets of DataRecords from the queue, and not one DataRecord at a time; this will be further explained in the next subsection.

The MQS *guarantees* delivery of DataRecords to the MCA 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. For example, the MQS's message broker does not acknowledge receipt of a message to its producer until the message has been safely persisted to secondary storage. On the receiving or consuming end, the message broker does not remove a message from secondary storage until its consumer has acknowledged receipt of the message. These MQSs can be configured to persist their messages to either file systems (distributed or local) or database management systems (DBMS). There is even an open source project that has positioned Cassandra as a MQS message store; however, that project is still in its infancy. 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.

All of these MQA 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. It is the goal of this project that this form 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. However, as the paper progresses, it will point out those MQS features that will be most important to the CluSandra framework.

The use of a MQS also facilitates the passing of messages (DataRecords) through different stages of preprocessing prior to being given to a final clustering analysis process (e.g., MCA). The design approach or pattern of decomposing an application into a set of stages is referred to as "*staged event-driven architecture*" (SEDA). One of the advantages of this design pattern is that it allows for modularity and code reuse. This first release of CluSandra utilizes a limited version of the SEDA design pattern; however, future releases can expand upon this pattern to provide stages for the transformation, preprocessing and/or standardization of the data.

3.4 Microclustering Agent (MCA)

The MCA consumes DataRecords from the MQS's queues and implements a clustering algorithm that produces microclusters. The MCA that is delivered with this release of CluSandra implements the first clustering algorithm that will be deployed onto the CluSandra framework. It is referred to as the CluSandra clustering algorithm and is described later in this section.

Many identical instances of a MCA process can be simultaneously executed, across the Cassandra/CluSandra cluster, to produce a MCA swarm. To meet the most demanding environments, Cassandra can elastically scale from a one or two node cluster to a cluster comprising 10s if not 100s of nodes. It is also possible to deploy swarms on non-Cassandra nodes and to define/configure many different swarms, with each swarm dedicated to a different queue and data stream. To take advantage of multiprocessor/multicore based nodes, the MCA can be configured to spawn multiple independent threads of execution.

The CluSandra clustering algorithm incorporates concepts and structures from both BIRCH and CluStream. Like CluStream, the CluSandra algorithm tackles the one-pass data stream constraint by dividing the data stream clustering process into two general operating phases: online and offline. Microclustering takes place in in real-time, computing and storing summary statistics about the data stream in microclusters; this functionality is provided by the MCA component. There is also an offline aggregation phase of microclustering that merges similar microclusters into one microcluster. 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. Both the offline aggregation and macroclustering are described in subsequent subsections.

As CluStream extended the BIRCH CF data structure to produce the CFT structure or microcluster, so does the CluSandra algorithm by extending CluStream's CFT structure as follows:

$$CFT = \langle N, LS, SS, ST, SST, CT, LAT, IDLIST \rangle$$

The CFT is also referred to as the *cluster structure* and it 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; more on the IDLIST later in this section.

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. CluSandra's online 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 reads or consumes sets of DataRecords from its assigned MQS queue. The members of a set of DataRecords $\mathcal{D}$ are consumed in the same order that they were produced by the data stream 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.

$$\mathcal{D} = \{d_1, d_2, \ldots, d_k\}$$

The maximum number of DataRecords in $\mathcal{D}$ is configurable. Also, the amount of time that a MCA blocks on a queue, waiting to read the maximum number of DataRecords in $\mathcal{D}$, is configurable. The MCA processes the set $\mathcal{D}$ whenever the maximum number has been reached or the read time expires and $\mathcal{D} \neq \emptyset$. 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 $\mathcal{D} = \emptyset$, the MCA simply goes back to blocking on the queue for the specified read time. Figure 4 depicts the logic flow for the MCA queue reading.

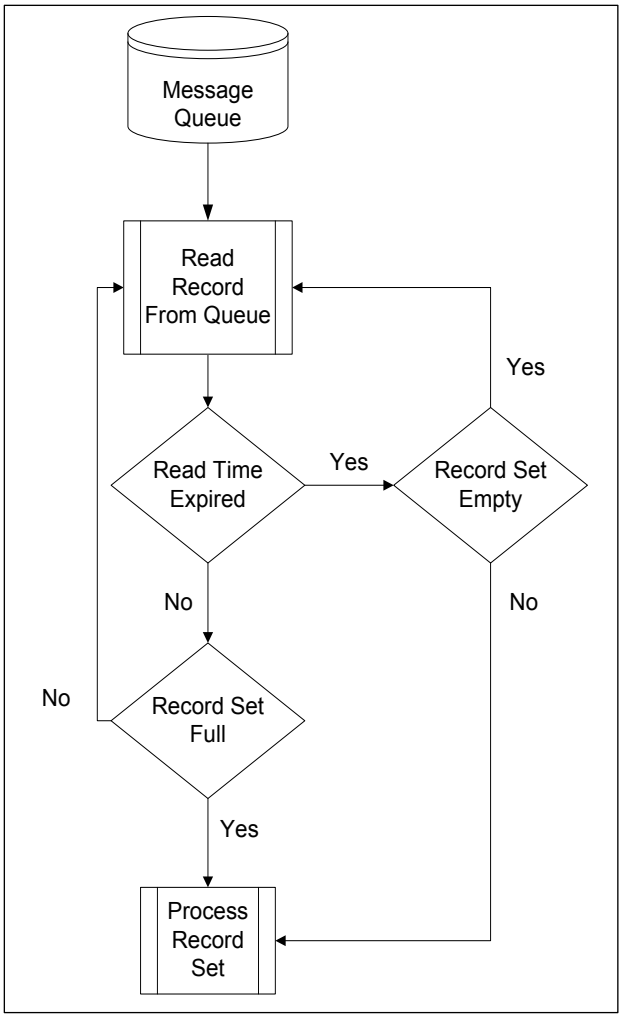


Figure 4 Flow Diagram for Queue Reads

After the MCA has read a set $\mathcal{D}$ from the queue, it starts the process of partitioning the DataRecords in $\mathcal{D}$ into a set of currently active microclusters $\mathcal{M}$. On startup, $\mathcal{M}$ is an empty set of microclusters, but as time progresses, $\mathcal{M}$ is populated with microclusters. The MCA then determines the time horizon h associated with $\mathcal{D}$. The range of h is R_h and it is defined by the youngest and oldest DataRecords in $\mathcal{D}$. Therefore, $R_h = \{t_o, t_y\}$, where t_o and t_y represent the oldest and youngest 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 MCA always works within a sliding temporal window that is defined by R_h. When the MCA completes the processing of $\mathcal{D}$, which is explained in the next paragraph, it writes all microclusters in $\mathcal{M}$ to the data store, goes back to reading from the queue, and starts the partitioning process over again with a new set of DataRecords.

To partition the DataRecords in $\mathcal{D}$, the MCA 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_e) timestamp. For example, if the timestamp for d_e is t_d, then only those microclusters in $\mathcal{M}$ whose LAT value falls within the range, $\{t_d - t_e, t_d\}$ 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 MCA then uses the Euclidean distance measure to find the microcluster in $\mathcal{S}$ that is closest to d_e.

A maximum boundary threshold (MBT) is then used to determine if the closest microcluster $\mathcal{M}_i$ in $\mathcal{S}$ can absorb d_e. If $\mathcal{M}_i$ can absorb d_e without breaching the MBT, then $\mathcal{M}_i$ is allowed to absorb d_e and is placed back in $\mathcal{M}$, else a new microcluster is created to absorb d_e and that new microcluster is then added to $\mathcal{M}$. If $\mathcal{S} = \varnothing$, the MCA simply creates a new microcluster to absorb d_e and adds the new microcluster to $\mathcal{M}$. 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 as follows:

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

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 $\mathcal{M}_i$ 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 expiry-time specified for the microclusters. Obviously, the smaller the MBT, the more microclusters will be produced and vice versa. However, the offline macroclustering phase can be used to merge those microclusters that are deemed similar. The next section describes the CluSandra Aggregator component, which is used to merge those microclusters whose radii overlap. Please note that the Aggregator does not produce superclusters; it simply *merges* overlapping microclusters.

3.5 Aggregating Microclusters

If there is a MCA swarm distributed across the CluSandra nodes, then it is likely for the swarm to create microclusters that are very similar, if not equal, both temporally and spatially (see Figure 5). This occurs if two or more MCAs 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.

If any microclusters temporally and spatially overlap, then those microclusters should in fact be viewed as one microcluster. Given a two-dimensional vector-space, the figure to the right 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 addressed by an offline aggregator component (see section 5.4.1) that sweeps through the timeline performing such merges. This aggregator, which is provided as part of the Clusandra algorithm package, should not be confused with superclustering and macroclustering.

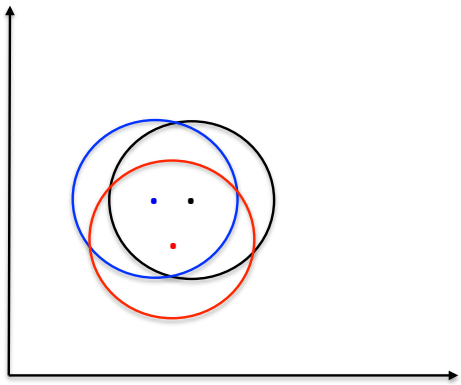


Figure 5 Temporally and Spatially Similar Microclusters

To determine if two microclusters spatially overlap, the aggregator determines if their radii overlap. To determine if two microclusters overlap, the aggregator compares the distance between their centroids and the sum of their radii. If the distance is less than or equal to the sum of the radii, then the two microclusters overlap. There may even be instances where one microcluster is entirely within the other. Consideration may be given to making the amount of overlap configurable. For example, if the overlap is 50% or more of the distance, then merge the two microclusters.

Design note: To avoid having similar microclusters and thus eliminate aggregation, all the members of a swarm can work off the same 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 and would also introduce severe contention between the members of the swarm. It is for this reason, that this approach of sharing microclusters was not followed.

3.6 Superclusters and Macroclustering

CluSandra introduces the supercluster, which is created when two or more clusters are merged via their additive properties. A supercluster can also be reduced or even eliminated via its subtractive property. The IDLIST in the cluster feature structure (CFT) is a collection or vector of microcluster ids (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 CluSandra's macroclustering process or phase of operation. Superclusters are created based on a specified distance measure (similarity) and time horizon. CluSandra superclustering is a type of *agglomerative* clustering procedure whereby individuals or groups of individuals are merged based upon their proximity to one another (Everitt et. al. 2001). However, unlike traditional agglomertive procedures where the results of a merger cannot be undone or separated, CluSandra's superclusters can be undone. What makes this possible is the subtractive property of the CF. Agglomerative procedures are probably the most widely used of the hierarchical clustering methods (Everitt et. al. 2001). When a supercluster is created, the earliest CT and LAT of its constituent parts is used as the supercluster's CT and LAT. The CT and LAT thus identify the supercluster's lifespan.

This first release of CluSandra will deliver the core data collection portion of the system and a set of canned queries that can be executed against CluSandra's cluster data store. This set of queries can be used to test the data store and demonstrate aspects of macroclustering. However, this first release will not go as far as delivering any visual or graphical front-end to the data store nor any sort of query engine. That functionality can be assigned to one or a series of follow-up projects.

3.7 The Spring Framework

Although the open source Spring Framework (http://www.springsource.org/) is not depicted in Figure 2 (CluSandra DFD), it is nonetheless relied upon by both the StreamReader and MCA components. Except for Cassandra, all CluSandra 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 as just "Spring", was created by its founders for the specific purpose of making Java application development a lot easier and thus mitigating the development effort. Spring provides a number of Java packages whose functionality is 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.

3.7.1 JMS

The StreamReader and MCA components make heavy use of Spring's support for the Java Message Service (JMS) API to send and receive DataRecords to and from the MQS, respectively. Apache ActiveMQ (http://activemq.apache.org/) has been selected as CluSandra's default JMS provider (i.e., MQS); there are many such open source JMS providers or implementations. ActiveMQ was chosen, because of its rich set of feature-functionality, beyond that specified by the JMS specification, and its robust support for and integration with Spring. Even the ActiveMQ message broker (core) is implemented as a Spring POJO.

As a JMS producer, the StreamReader uses Spring's JmsTemplate class to produce DataRecords destined for the ActiveMQ message broker. The JmsTemplate is based on the template design pattern (http://en.wikipedia.org/wiki/Template_method_pattern) and it is a convenience class that hides much of the complexity of sending messages to a JMS message broker. The JmsTemplate, in combination with the corresponding Spring XML file, also helps ensure a decoupling between the StreamReader and whatever JMS provider is being used as the CluSandra MQS. The StreamReader is therefore JMS provider agnostic.

The DataRecord is the object that is passed from StreamReader to MCAs, via the ActiveMQ message broker. The JMS supports the sending and receiving of objects, and not just text messages; however, the objects must be *serializeable*. Therefore, the CluSandra DataRecord object must implement the standard Java Serializeable interface. More detailed information on the StreamReader and DataRecord is provided in the section that describes their class diagrams.

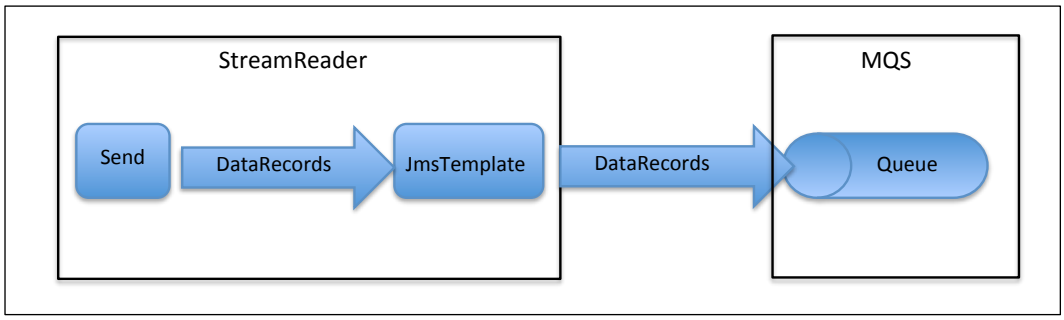


Figure 6 StreamReader's us of the Spring JmsTemplate

The MQS queue is capable of holding 1000s of DataRecords and guarantees their delivery to the MCA swarm, which is responsible for ensuring that the number of DataRecords in the MQS queue is maintained at an acceptable level. The MCA 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 size. The MCA also uses Spring's JmsTemplate to facilitate the synchronous receiving of messages from the MQS queue; therefore, like the StreamReader, the MCA is also JMS provider agnostic. Figure 4 is a flow diagram for the MCA queue read logic and Figure 7 depicts a MCA swarm. As depicted in Figure 4, each MCA processes a set of DataRecords at a time. Again, this keeps down the number of reads it will have to perform to process the set of DataRecords, which is processed within the context of a local JMS transaction. When all the DataRecords in the set have been processed, the MCA commits the transaction, at which point the MCA can discard the DataRecords from secondary storage. By default, the MCA receives the set within the context of a transaction; however, the MCA can be configured to not use a transaction.

To achieve a high level of message throughput, the ActiveMQ message broker dispatches or pushes messages to a consumer (e.g., MCA) as fast as possible so that the consumer always has messages in its local memory (message buffer) to process. This is opposed to the consumer having to pull each individual message from the broker, which adds a significant amount of overhead or latency per message. However, this method of pushing as many messages as possible on to the consumer must incorporate a throttling mechanism. This is because it typically takes much less time for the broker to deliver a message than it does for the consumer to process it; therefore, the consumer could very easily be overwhelmed with messages. The throttling mechanism that ActiveMQ incorporates is referred to as the "*prefetch limit*". This limit specifies how many messages the broker can push or stream to the consumer without first getting an acknowledgement or commit back from the consumer for the messages it has processed. When the prefetch limit has been reached, the broker will stop delivering messages to the consumer until the consumer starts to acknowledge or commit the processed messages, which in this case are DataRecords. In general, the higher the prefetch limit that you assign to a consumer, the faster that consumer is expected to process messages. So you want to give slow and fast consumers lower and higher prefetch limits, respectively. The default value assigned to the prefetch limit is 1000 and so by default, the ActiveMQ message broker will send chunks of DataRecords to the MCA.

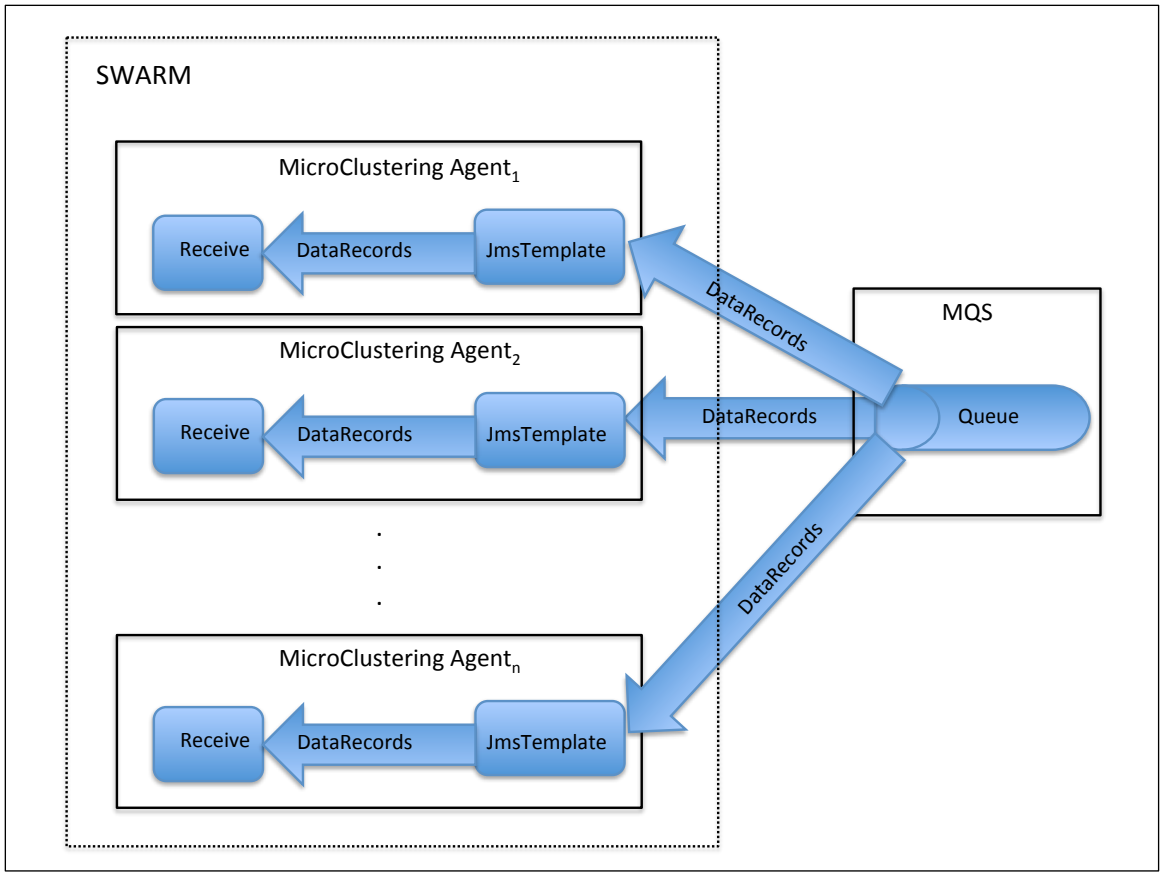


Figure 7 A MCA Swarm Receiving From a Queue

4 Data Models

This section of the design specification describes the Cassandra data models for the CluSandra timeline index (3.2) and cluster structure. Cassandra is neither a relational nor SQL database; therefore, entity relationship diagrams (ERDs) are not used as the modeling diagrams. The document instead uses JavaScript Object Notation (JSON)-like syntax to describe the data models.

The CluSandra framework can process many distinct data streams. All the data structures (i.e., TI and clusters) that are used for a particular data stream must be assigned to a unique Cassandra *keyspace*. The framework includes scripts and components that are used to create a keyspace and the data structures for a particular data stream. In a RDBMS setting, the keyspace is analogous to a schema and the data stream's structures are the tables within that schema.

4.1 The Cassandra Column-based Data Model

Before describing the CluSandra data models, lets quickly go over the Cassandra column-based data models as presented in this article: http://tinyurl.com/yawwgql

Column

The column is the smallest or atomic unit of data and its best represented as a tuple (triplet) comprising a name, a value and a timestamp. Both the name and value pair are binary. More precisely, they are Java byte arrays (i.e., byte[]) with no size limit.

This is the column's representation in JSON-like notation:

```
// this is a column
{
   name: "emailAddress",
   value: "arin@example.com",
   timestamp: 123456789
}
```

For now, we'll ignore the timestamp and thus we can simply treat the Column as a name-value pair.

SuperColumn

The SuperColumn is also a tuple with a binary name (i.e., byte[]); however, the SuperColumn's value is a map containing an unbounded number of Columns, which are keyed by their names. The SuperColumn is analogous to a row in a relational database. The following is a JSON-like notation for a SuperColumn with a name of "homeAddress" and a value comprising three Columns (street, city, and zip).

```
// this is a SuperColumn
{
   // the SuperColumn's name
   name: "homeAddress",
   // the SuperColumn's value, which is a collection of Columns
   value: {
      // note that the key for each Column in this map must match the Column's name
      street: {name: "street", value: "1234 x street",  timestamp: 123456789},
      city:   {name: "city",   value: "san francisco", timestamp: 123456789},
      zip:    {name: "zip",    value: "94107",         timestamp: 123456789},
   }
}
```

So Columns and SuperColumns are both tuples with name and value elements. The main difference between the two is that a Column's value is a byte array, while the SuperColumn's value is a map of Columns. One minor difference is that the SuperColumn tuple does not have a timestamp element.

For the sake of this overview, we'll make the following two simplifications: 1) ignore the Column's timestamp and 2) ignore the Column and SuperColumn's name elements. So the above homeAddress SuperColumn can now be represented as follows:

```
homeAddress: {
   street: "1234 x street",
   city: "san francisco",
   zip: "94107",
}
```

ColumnFamily

A ColumnFamily is a table-like structure that contains one or more rows, where each row is identified by an application-supplied row key and contains a collection or map of Columns. The row key is analogous to a unique id index column found in the relational table. Also, there is no schema enforced on the ColumnFamily. If you'd like, you can define each row to have its own unique set of Columns, where each set has a different number of Columns and the Columns can all have different names. One row can have 1000 Columns , while next has only two Columns. For example, note in the example below how the second row (i.e., the one with row-key 'john') has more Columns than the first row.

```
UserProfile = { // start ColumnFamily
```

```
mary: {   // this is the key for this first row
          //these are the Columns for this first row
          username: "mary",
          email:      "mary@example.com",
          phone:      "(800) 987-1234"
}, // end row

john: {   // this is the key to a second row
          //these are the Columns for the second row
          username:   "john",
          email:        "ieure@example.com",
          phone:       "(800) 543-2100"
          age:          "66",
          gender:      "male"
}, // end row

} // end ColumnFamily
```

Remember that for simplicity, we're only showing the Column value and ignoring the Column name and timestamp.

SuperColumn Family

Like the ColumnFamily, the SuperColumnFamily is also a table-like structure, where each row is identified by a row-key. However, instead of each row containing a collection or map of Columns, as is the case for a ColumnFamily, each row in a SuperColumnFamily contains a collection or map of SuperColumns. Here's an example of a SuperColumnFamily:

```
AddressBook = {
        // This is a SuperColumnFamily (SCF) called 'AddressBook' that is a collection of
        // address books for different individuals

   john: {
        // This is a row in the SCF with a row key of 'john'.
        // This row is a map of SuperColumns that represents the address
        // book for John. This row contains one SuperColumn for each of John's
        // friends.  The keys inside the row are the names for the SuperColumns.

        // Maria is the first row in John's address book and it is a SuperColumn
        // containing Maria's contact information
        Maria: {street: "Howard street", zip: "94404", city: "Bakesfield", state: "CA"},
        Kim: {street: "X street", zip: "87876", city: "Baylor", state: "VA"},
        Todd:{street: "Jerry street", zip: "54556", city: "Carbon", state: "CO"},
        Bob: {street: "Q Blvd", zip: "24252", city: "Bloomfield", state: "MN"},
       ...
       // we can have an infinite # of ScuperColumns (aka address book entries)
   }, // end row for 'john'

   linda: {     // this is the key to another row in the Super CF
        // This is a row in the SCF with a row key of 'linda'.
        // This row is a map of SuperColumns that represents the address
        // book for Linda. This row contains one SuperColumn for each of Linda's
        // friends.  The keys inside the row are the names for the SuperColumns.

       // all the address book entries for ieure
       Ana: {street: "A ave", zip: "55485", city: "Sin City", state: "NV"},
       Sandy: {street: "Armslength Dr", zip: "93301", city: "Fresno", state: "CA"},
```

```
        …
    }, // end row for 'linda'
  …
} // end SCF
```

4.2 The Timeline Index (TI) Data Model

Section 3.2 introduced the TI and described its usage. In this section, its data model is presented in the same JSON-like notation used in the previous section. As you may recall from section 3.2, the TI comprises a Cluster Index Table (CIT). This section describes the CIT's data model and please refer to Figure 3 for a pictorial representation of the TI.

The CIT is a SuperColumnFamily, where each row-key is assigned the value of the zero 'th hour, minute, and second of a particular day/date. Each row in the CIT contains 84000 SuperColumns, where each SuperColumn represents a second in the day represented by its corresponding row-key. Each SuperColumn may contain zero or more Columns, where each Column's name and value is a row key of the Cluster Table. Each Column in the SuperColumn, therefore, serves as an index that identifies a microcluster that absorbed a DataRecord at this particular second.

```
ClusterIndexOne = {

  // This is a row in the CIT SCF. Each row in the CIT represents a particular
  // day (date) and contains 84000 SuperColumns. The value for a row-key in
  // the CIT is a timestamp value for the zero'th hour of a particular day. This
  // particular row represents June 7, 2011.
  1307419200000: {

        // This is the first SuperColumn for this row. The name assigned to the first
        // SuperColumn always matches its corresponding row's key value, because
        // it is the first second of the day.  This particular SuperColumn has two
        // Columns. Each Column is an index to the ClusterTable and identifies a
        // cluster that absorbed a DataRecord at this particular second in time.
        1307419200000: {
            // This is first of two Columns in this SuperColumn and the value
            // assigned to this Column's name and value is a row-key (index)
            // of the ClusterTable, which contains all the micro and superclusters.
            c6743991: c6743991,
            // The second Column in the SuperColumn
            c6743972: c67439972
         }, // end SuperColumn

        …

        // This is the 84000th SuperColumn for this row.
        1307419283000: {
            c6743001: c6743001,
            c6743011: c6743011,
            c6753972: c67539972
         } // end SuperColumn

   }, // end row for 1307419200000
```

```
   // This is a  second row in the CIT SCF. It represents the second day.
  1307505600000: {
      1307505600000: {
           // This is first of two Columns in this SuperColumn and the value
           // assigned to this Column's name and value is a row-key (index)
           // of the ClusterTable, which contains all the micro and superclusters.
           c6743981: c6743981,
           // The second Column in the SuperColumn
           c6743973: c67439973
       }, // end SuperColumn

        …

      // This is the 84000th SuperColumn for this row.
      1307505683000: {
           c6743001: c6743101,
           c6743011: c6743211,
           c6753972: c67539872
       } // end SuperColumn
  }, // end row for 1307505600000

} // end SCF
```

4.3 The Cluster Table (CT)

The CT is implemented as a SuperColumnFamily (SCF), where each row in the SCF represents either a micro or supercluster. Note in the table below that SuperColumns for the CT are assigned two types: *Scalar* and *Vector*. A Scalar SuperColumn has at most one Column, while a Vector SuperColumn has one or more Columns. The Scalar and Vector type has been introduced within this context, because you cannot mix Columns and SuperColumns in a SCF.

The table below describes the SuperColumns that comprise a row in the CT SCF. The vector representation of a cluster is represented as follows, where the 'T' in CFT defines the cluster feature as being a temporal extension of the BIRCH cluster feature (CF):

$$CFT = \langle N, LS, SS, ST, SST, CT, LAT, ID, LISTID \rangle$$

SuperColumn Name	Type	Description
N	Scalar	Total number of DataRecords absorbed by the cluster.
LS	Vector	Each DataRecord encapsulates a vector of continuous numerical values of type double. LS is the linear sum of all the absorbed DataRecords' vectors. $LS = \sum_{i=1}^{N} \vec{x}_i$

SS	Vector	SS is the squared sum of all the absorbed DataRecords' vectors. $$SS = \sum_{i=1}^{N} \vec{x}_i^2$$ The SS and LS vectors must have the same number of elements across all SuperColumns in this SuperColumnFamily.
LST	Scalar	The linear sum of the timestamps associated with all the DataRecords absorbed by this cluster.
SST	Scalar	The squared sum of the timestamps associated with all the DataRecords absorbed by this cluster.
CT	Scalar	The creation time, in the form of a timestamp, for this cluster.
LAT	Scalar	The time, in the form of a timestamp, that this cluster last absorbed a DataRecord. This scalar, along with the CT, is used to determine how long this cluster remained active during the data steam's lifespan.
LISTID	Vector	The list of ids (row keys) that represent the microclusters that comprise this supercluster. In other words, if the LISTID is not empty, this cluster represents a supercluster that was created by aggregating or adding together all the microclusters listed in the LISTID. Note that superclusters are subtractive, as well as additive.

The following is an example of a CT in JSON-like notation. In this particular example the CT has only one cluster and that cluster's LISTID is not empty; therefore, the cluster is a supercluster.

```
ClusterTable = {

  c6743001: {

        // This particular cluster has absorbed 2 DataRecords
       N: {
            2: 2
        }, // end SuperColumn
       LS: {
            9: 9,
            10: 10
        }, // end SuperColumn
       SS: {
            29: 29,
            38: 38
        }, // end SuperColumn
       ST: {
```

```
            1307419283000: 1307419283000
        }, // end SuperColumn
        SST: {
            3347439285010: 3347439285010
        }, // end SuperColumn
        CT: {
            1007419284200: 1007419284200
        }, // end SuperColumn
        LAT: {
            1007419284200: 1007419284200
        }, // end SuperColumn
        // This is a supercluster comprised of the clusters in this list.
        LISTID: {
            c6743004: c6743004,
            c6743011: c6743011,
            c6753972: c67539972
        }, // end SuperColumn

    }, // end row for cluster c6743001
    ...

} // end SCF
```

5 Primary CluSandra Classes

CluSandra's Java package name has the "clusandra." prefix. For example, "clusandra.core", "clusandra.clusterers", "clusandra.util", etc. The following subsections describe the classes that comprise CluSandra's primary components.

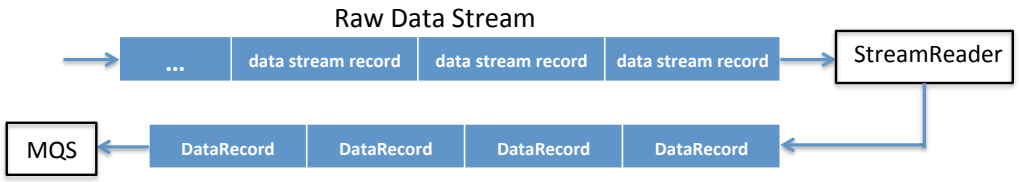
5.1 CluRunner

The clusandra.core.CluRunner class is used to run or start the CluSandra-provided runnable components, such as the StreamReader and Microclustering Agent (MCA). The CluRunner is a very simple component that includes a main() method, is started from the command line, and is responsible for reading a Spring XML configuration file that is specified via a command line option. If the XML file is not specified, it looks for a default XML file in its class path. A Spring application context (Bean or POJO factory) is created from this XML file. The CluRunner searches the application context for all POJOs that implement the clusandra.util.CluRunnable interface and invokes their cluRun() methods, which are expected to spawn one or more threads of execution.

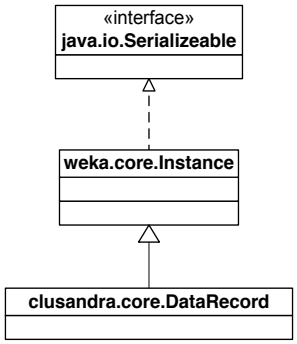
The CluRunner also includes a Java Management Extension (JMX) bean that is used to manage the CluRunner component from any JMX-compliant management platform (e.g., JConsole and Hyperic). For example, you can send the CluRunner a shutdown command or ask it for certain statistics. The first release of the CluRunner will come with a very simple JMX bean that allows you to shutdown the component or perhaps send it a command to reload the Spring XML configuration file.

5.2 DataRecord

The DataRecord is the object that encapsulates the relevant attributes of a raw data stream record, and it is also the object that is passed from StreamReader to MCA, via the MQS. It is the StreamReader's responsibility to transform raw data stream records into core CluSandra DataRecord objects.



The DataRecord's constructor method assigns the newly created DataRecord a timestamp; however, that timestamp can be overridden by the application or object that creates the DataRecord. For example, if the raw data stream record includes a timestamp, the StreamReader can override the DataRecord constructor's timestamp with the one included in the data stream record. The DataRecord must implement the Java Serializeable interface so that it can be passed through the MQS's message queues. The open source WEKA data mining package includes a class called, "Instance" that is very suitable for this purpose. It implements the Serializeable interface, encapsulates different data types within a vector and has a rich set of supporting operations; therefore, the DataRecord extends the WEKA's Instance superclass.

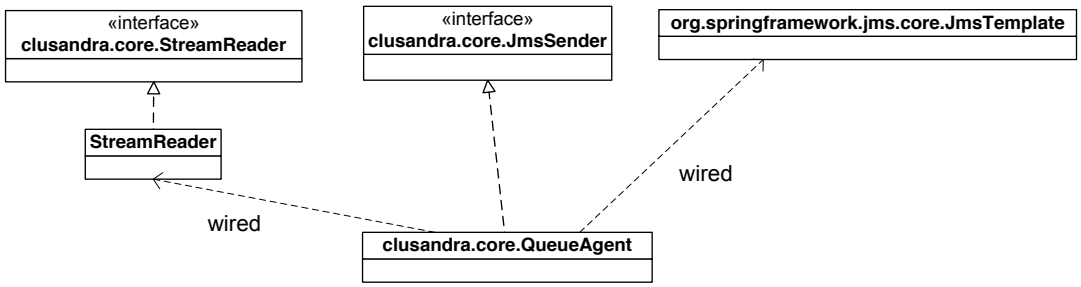


For this first release of CluSandra, the only data type being supported is the numerical type; however, future releases of CluSandra may support other data types (e.g., binary and nominal), for which the Instance superclass is well-suited to handle.

5.3 StreamReader

The StreamReader is comprised of three main classes: StreamReader, QueueAgent and JmsTemplate. All three components are collectively referred to as the StreamReader. Each distinct data stream that is processed by the CluSandra framework must be assigned a StreamReader.

The main driver component is the QueueAgent, which is wired to both a Spring JmsTemplate and StreamReader object; all three objects are POJOs (http://en.wikipedia.org/wiki/Plain_Old_Java_Object) and are wired together via a Spring XML file. This is the file that the CluRunner component reads in as a Spring application context.

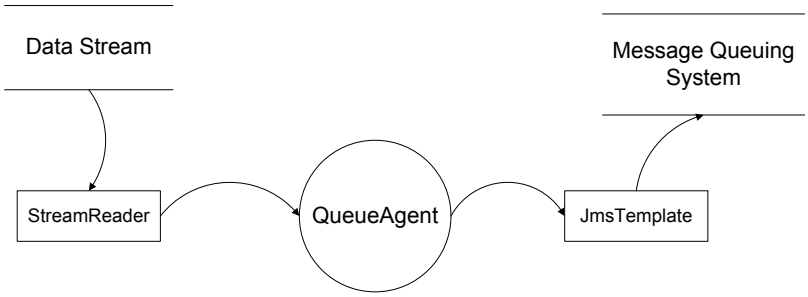


The end user must provide a class that implements the clusandra.core.StreamReader interface. The following lists the more important methods defined by the StreamReader interface.

- setQueueAgent() - This method is used by Spring to wire the StreamReader to its QueueAgent. The wiring is done through the corresponding Spring XML configuration file. If the end user's StreamReader includes a constructor with a QueueAgent parameter, then the StreamReader can be auto-wired, by Spring, to its QueueAgent.

- getQueueAgent () – Returns the QueueAgent object that is wired to the StreamReader. The StreamReader uses this method to invoke the QueueAgent's helper methods.
- startReader() - This method is invoked by the QueueAgent to start and give control to the StreamReader.

The QueueAgent serves as a coupling that is used to join the end user's StreamReader and Spring JmsTemplate classes. It also implements the clusandra.core.JmsSender interface, which defines the helper methods that the StreamReader invokes to send DataRecords to the MQS.



The following lists and describes the more important methods in the QueueAgent class:

- cluRun() – This method is invoked by the CluRunner to run the QueueAgent, which runs within its own thread of execution.
- setJmsTemplate() – This method is used for wiring the QueueAgent to its corresponding Spring JmsTemplate.
- setStreamReader() – This method is used for wiring the QueueAgent to its corresponding StreamReader.
- sendDataRecord() – This method is invoked by the StreamReader to send a DataRecord to the MQS queue. The QueueAgent is configured to send a collection of DataRecords to the MQS within the context of a local JMS transaction. When the collection's maximum size has been reached, the QueueAgent automatically sends or flushes the collection to the MQS within the context of a local JMS transaction. The collection's maximum size is configured via the Spring XML configuration file. The use of transactions can be disabled and if disabled, all messages are sent asynchronously to the MQS. If the collection's maximum size is set to 1, each invocation of sendDataRecord() will result in a send to the MQS system.
- flushDataRecords() – This method is invoked by the StreamReader to force the QueueAgent to ignore the collection's maximum size and immediately send (flush) the collection of DataRecords to the MQS.

The classes also include mutator (setter) methods used to set their corresponding configuration parameters, which are specified via the Spring XML file. For example, the QueueAgent includes a setSendSize() method to set the maximum size of the DataRecord send collection.

The wiring of the three objects provides flexibility, as well as decouples the QueueAgent from the different implementations of the StreamReaders and JMS providers. It also decouples the end-user's StreamReader from JMS implementations.

As previously mentioned, all the wiring and configuration of the three objects (i.e., QueueAgent, StreamReader and JmsTemplate) is accomplished via a Spring XML configuration file. It is also possible to have the XML file define any number of sets of these three objects. For example, suppose you have two data streams, you can define two sets of these three objects, where each set is assigned to one of the two streams. However, all resulting components will execute within the same JVM and under their own threads of execution. See the following web page for more information on Spring and how POJOs are wired via Spring's Inversion of Control (IOC) facility:

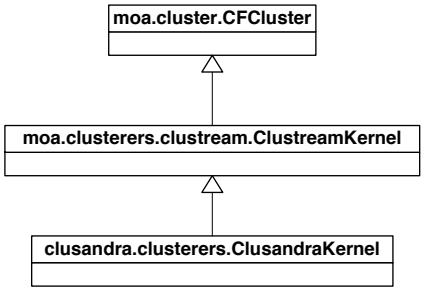
http://static.springsource.org/spring/docs/3.0.5.RELEASE/reference/

These three objects do not have to reside in a Cassandra cluster node nor a node that hosts the MQS message broker. The only requirement is that the node's operating platform support version 1.5 or higher of the Java Virtual Machine (JVM). As such, these three objects may reside in the devices that produce the actual data stream (e.g., routers, sensor, servers, etc.).

5.4 Clusandra

The "clusandra.clusterers" package contains the classes that implement the distinct clustering algorithms within the CluSandra framework. The first such class or algorithm, which is described in section 3.4, is called the Clusandra class. This class is one of a number of classes that collectively comprise an instance of a MicroclusteringAgent or MCA. As you'll read in this and the following subsections, these classes are wired together to form a MCA instance.

Given a set of DataRecords, the Clusandra class generates and/or updates one or more microclusters that absorb the given set of DataRecords. These Clusandra microclusters extend the Clustream microcluster class, which is called, ClustreamKernel. Therefore, the Clusandra-generated microcluster is called, ClusandraKernel and it extends ClustreamKernel. All the Clustream-related classes are made available by the open source MOA framework.



Note in the class diagram that ClustreamKernel extends CFCluster. The latter is a representation of the cluster feature structure or CF that is introduced by the BIRCH algorithm.

5.4.1 ClusandraAggregator

The Clusandra algorithm also includes a class called, clusandra.clusterers.ClusandraAggregator. This class implements the CluRunner interface and is responsible for aggregating or consolidating microclusters as described in section 3.5. It is not part of a MCA and is instead designed to run as a standalone process. The aggregator can be configured to sweep through any portion of the data stream timeline index (TI). As it sweeps through the TI, it aggregates similar microclusters that temporally overlap one another. Microclusters are aggregated via their additive property and the results of aggregation are new microclusters; the microclusters that were aggregated are removed from the data store. The figure on the right illustrates the before and after results of aggregation. In this case, there were three very similar 2-dimensional microclusters, which are represented by their centroids and maximum boundary thresholds, that all temporally overlap. The aggregator then consolidates all three into one microcluster and the original three are removed from the data store.

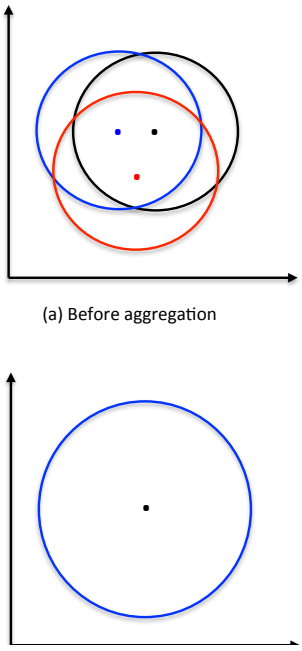


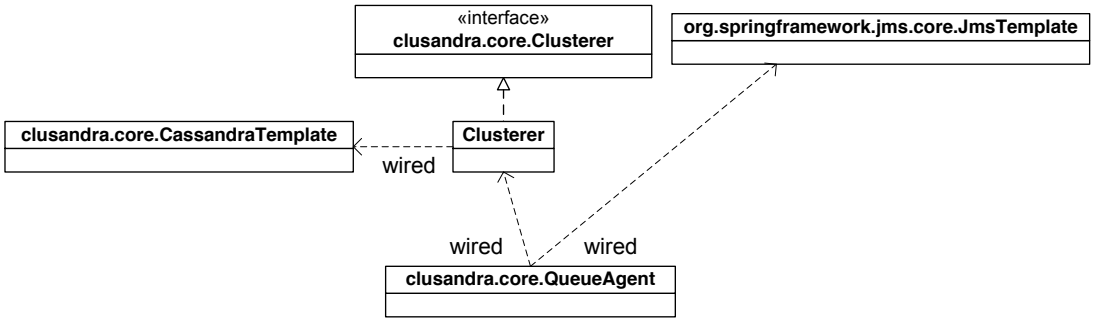
Figure 8 Aggregation Process

The aggregation of microclusters is not to be confused with superclustering.

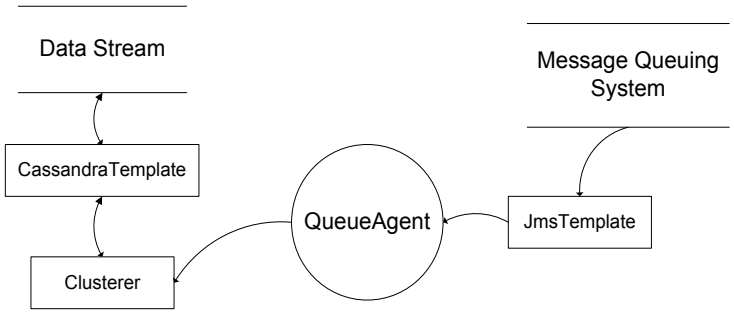
The aggregator must not aggregate that portion of the TI that represents the current online window; doing so would result in a conflict with the current online phase. As is the case for all CluRunners, the aggregator has a corresponding Spring XML configuration file through which it is configured. The aggregator is wired with a CassandraTemplate, which is described in the following section.

5.5 MicroClustering Agent (MCA)

The MicroclusteringAgent, or MCA, is comprised of four main components: QueueAgent, JmsTemplate, CassandraTemplate, and Clusterer. All components are wired together via a Spring XML file that is given to a CluRunner.



Each distinct data stream that is processed by the CluSandra framework must be assigned at least one MCA. Multiple instances of a MCA is referred to as a MCA swarm and the swarm can be distributed across multiple nodes. The QueueAgent is the same class that is used to couple the JmsTemplate with a StreamReader.



The Clusterer is actually an interface that must be implemented by all classes that implement a clustering algorithm. Even though it is not depicted in the previous subsection's class diagram, the Clusandra class implements this interface. These are some of the more important methods defined by the Clusterer interface:

- setConfig(Map cfg): This method is invoked by Spring and is used to set a Clusterer-specific configuration Map, as defined in the Spring XML file. The Map comprises a set of name-value pairs that are specific to the Clusterer. For example, the Clusandra clusterer, requires a factor that is used for calculating the maximum boundary threshold for the microclusters and it also requires a temporal scalar that specifies how long the microclusters can remain active without absorbing a DataRecord.
- initialize() – This method is invoked by the QueueAgent to allow the Clusterer to perform its initialization tasks.
- processDataRecords() – This method is invoked by the QueueAgent to have the Clusterer process a set of DataRecords that have been read from the MQS (see section 3.4).
- setCassandraTemplate() – This method is invoked by Spring to wire the Clusterer to a CassandraTemplate.

The CassandraTemplate class is similar to the JmsTemplate in that it hides or abstracts all the complexities of dealing with Cassandra. It exposes a handful of methods that are used for persisting and fetching DataRecords to and from the Cassandra data store, respectively.

The QueueAgent serves as a coupling that is used to join the Clusterer and Spring JmsTemplate classes. The following lists and describes the more important methods in the QueueAgent class, relative to the Clusterer. Some of the other important methods are listed and described in the StreamReader section:

- setClusterer() – This method is used for wiring the QueueAgent to its corresponding Clusterer. Note that the QueueAgent can only be wired to a Clusterer or StreamReader. The QueueAgent throws an exception and exits if it is wired to both a StreamReader and Clusterer.
- cluRun() – This method is invoked by the CluRunner (described in the following subsection) to start the QueueAgent, which runs within its own thread of execution.

As previously mentioned, the classes also include mutator (setter) methods used to set their corresponding configuration parameters, which can be specified via the Spring XML file. For example, the QueueAgent includes a setReadSize() and setReadTime() method to set the maximum number of DataRecords to read and the read expiry time, respectively.

The wiring of the objects provides flexibility, as well as decouples the QueueAgent from the different implementations of Clusterers and JMS providers. It also decouples the end-user's Clusterer from JMS implementations.

6 Command Line Interface

Starting with version 0.8, Cassandra includes a SQL-like language that is referred to as the Cassandra Query Language or CQL for short. Within the context of CluSandra, the CQL is used to maintain the data structures for a particular data stream keyspace. For example, using CQL, you can truncate or drop a Timeline Index (TI).

To fetch microclusters and superclusters from the CluSandra data store, you use a simple CluSandra-specific query language, which we'll refer to simply as the MicroCluster Query Language or MQL for short. CQL cannot be used for this purpose, because it cannot properly index the TI to fetch the clusters. The following provides an overview of the MQL.

Select

The select statement allows you to project one or more clusters from the data store. When using the select statement, you must always specify a time horizon over the data stream's lifespan.

select * from <*schema name*> where date between <*date-time*> and <*date-time*>

Here's an example query that selects all clusters that were active between June 7, 2011 and June 8, 2011:

select * from datstream1 where date between 2011:06:07 and 2011:06:08

The following select statement projects all clusters that were active on June 7, 2011.

select * from datastream1where date is 2011:06:07

The time can be specified down to the second. In the following example query, the select projects all clusters that were active between 13:10:00 and 13:10:30 on June 7, 2011.

select * from <*schema name*> where date between 2011:06:07:13:10:00 and 2011:06:07:13:10:30

You can specify whether to select micro or superclusters.

select * from <*schema name*> where date between 2011:06:07:13:10:00 and 2011:06:07:13:10:30 and cluster=micro

select * from <*schema name*> where date between 2011:06:07:13:10:00 and 2011:06:07:13:10:30 and cluster=super

You can select a particular cluster.

select <*cluster id*> from <*schema name*>

For example: select c3567989 from datastream1

Merge

The merge statement allows you to merge clusters, whether micro or super, to create new superclusters.

merge <*cluster id*>, <*cluster id*>, …, <*cluster id*> from <*schema name*>

For example: merge c3567989, c3567923 from datastream5

7 References

Fernandez, J. (2011). Clustering evolving data streams with Cassandra (Product Definition). University of West Florida.

Walls, C. (2008). Spring in action (2nd Ed.) Greenwick, CT: Manning Publications Co.

Pressman, R. (2009). Software engineering A Practitioners Approach (7th Ed.) New York: McGraw-Hill.

Domingos, P., Hulten, G. (2000). Mining high-speed data streams. *Knowledge Discovery and Data Mining*, pages 71-80.

Aggarwal, C.C., Han, J., Wang, J., Yu, P.S. (2003). A framework for clustering evolving data streams. Proceedings of the 29th VLDB Conference, Berlin, Germany.

Aggarwal, C.C., Han, J., Wang, J., Yu, P.S. (2004). A framework for projected clustering of high dimensional data streams. Proceedings of the 30th VLDB Conference, Toronto, Canada.

Gama, J. (2010). Knowledge discovery from data streams. Boca Raton, FL: Chapman & Hall/CRC

Gama, J., Mohamed, M.G. (2007). Learning from data streams: Processing techniques in sensor data. Springer-Verlag

Bifet, B. and Kirkby, R. (2009). Data stream mining. University of Waikato, New Zealand: Centre for Open Software Innovation

Hewitt, E. (2010). Cassandra, the definitive guide. Sebastopol, CA: O'Reilly Media, Inc.

Zhang, T., Ramakrishnan, R., Livny, M. (1996). BIRCH: An efficient data clustering method for very large databases. ACM SIGMOD, pages 103-114

Golab, L., Ozsu, M.T. (2003). Issues in data stream management. SIGMOD Record, Vol. 32, No. 2

Hebrail, G. (2008). Introduction to data stream querying and processing. *International Workshop on Data Stream Processing and Management*. Beijing.

Han, J., Kamber, M. (2006). Data mining, concepts and techniques (2nd edition). San Francisco, CA: Morgan Kaufmann Publishers.

Everitt, B.S., Landau, S., Leese M. (2001). Cluster analysis. (4th edition). London, England: Arnold Publishers

Zhang, P., Zhu, X., Guo, L. (2009). Mining data streams with labeled and unlabeled training examples. Ninth IEEE International Conference on Data Mining, pages 627-635.

Masud, M., Gao, J., Latifur, K., Han, J. (2008). A practical approach to classify data streams: training with limited amount of labeled data. Dept. of Computer Science, University of Dallas @ Texas, Dept. of Computer Science, University of Illinois @ Urbana Champaign

Giuseppe, D., Deniz H., Madan J., Gunavardhan K., Avinash L., Alex P., Swaminathan S., Peter V., Werner V. (2007). Dynamo: Amazon's Highly Available Key-value Store. Stevenson, WA: ACM

Plant, C., Bohm, C. (2008). Novel trends in clustering. Technishe Universitat Munchen, Munchen Germany, Ludwig Maximialns Universitat Munchen Munich Germany