A Framework for Clustering Evolving Data Streams

Charu C. Aggarwal
T. J. Watson Resch. Ctr.

Jiawei Han, Jianyong Wang
UIUC

Philip S. Yu
T. J. Watson Resch. Ctr.

Abstract

The clustering problem is a difficult problem for the data stream domain. This is because the large volumes of data arriving in a stream renders most traditional algorithms too inefficient. In recent years, a few one-pass clustering algorithms have been developed for the data stream problem. Although such methods address the scalability issues of the clustering problem, they are generally blind to the evolution of the data and do not address the following issues: (1) The quality of the clusters is poor when the data evolves considerably over time. (2) A data stream clustering algorithm requires much greater functionality in discovering and exploring clusters over different portions of the stream.

The widely used practice of viewing data stream clustering algorithms as a class of one-pass clustering algorithms is not very useful from an application point of view. For example, a simple one-pass clustering algorithm over an entire data stream of a few years is dominated by the outdated history of the stream. The exploration of the stream over different time windows can provide the users with a much deeper understanding of the evolving behavior of the clusters. At the same time, it is not possible to simultaneously perform dynamic clustering over all possible time horizons for a data stream of even moderately large volume.

This paper discusses a fundamentally different philosophy for data stream clustering which is guided by application-centered requirements. The idea is divide the clustering process into an online component which periodically stores detailed summary statistics and an offline component which uses only this summary statistics. The offline component is utilized by the analyst who can use a wide variety of inputs (such as time horizon or number of clusters) in order to provide a quick understanding of the broad clusters in the data stream. The problems of efficient choice, storage, and use of this statistical data for a fast data stream turns out to be quite tricky. For this purpose, we use the concepts of a *pyramidal time frame* in conjunction with a *microclustering* approach. Our performance experiments over a number of real and synthetic data sets illustrate the effectiveness, efficiency, and insights provided by our approach.

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

1 Introduction

In recent years, advances in hardware technology have allowed us to automatically record transactions of everyday life at a rapid rate. Such processes lead to large amounts of data which grow at an unlimited rate. These data processes are referred to as *data streams*. The data stream problem has been extensively researched in recent years because of the large number of relevant applications [1, 3, 6, 8, 13].

In this paper, we will study the clustering problem for data stream applications. The clustering problem is defined as follows: for a given set of data points, we wish to partition them into one or more groups of similar objects. The similarity of the objects with one another is typically defined with the use of some distance measure or objective function. The clustering problem has been widely researched in the database, data mining and statistics communities [4, 9, 12, 10, 11, 14] because of its use in a wide range of applications. Recently, the clustering problem has also been studied in the context of the data stream environment [8, 13].

Previous algorithms on clustering data streams such as those discussed in [13] assume that the clusters are to be computed over the entire data stream. Such methods simply view the data stream clustering problem as a variant of one-pass clustering algorithms. While such a task may be useful in many applications, a clustering problem needs to be defined carefully in the context of a data stream. This is because a data stream should be viewed as an infinite process consisting of data which continuously evolves with time. As a result, the underlying clusters may also change con-

siderably with time. The nature of the clusters may vary with both the moment at which they are computed as well as the *time horizon* over which they are measured. For example, a user may wish to examine clusters occurring in the last month, last year, or last decade. Such clusters may be considerably different. Therefore, a data stream clustering algorithm must provide the flexibility to compute clusters over user-defined time periods in an interactive fashion.

We note that since stream data naturally imposes a one-pass constraint on the design of the algorithms, it becomes more difficult to provide such a flexibility in computing clusters over different kinds of time horizons using conventional algorithms. For example, a direct extension of the stream-based k-means algorithm in [13] to such a case would require a simultaneous maintenance of the intermediate results of clustering algorithms over all possible time horizons. Such a computational burden increases with progression of the data stream and can rapidly become a bottleneck for online implementation. Furthermore, in many cases, an analyst may wish to determine the clusters at a previous moment in time, and compare them to the current clusters. This requires even greater book-keeping and can rapidly become unwieldy for fast data streams.

Since a data stream cannot be revisited over the course of the computation, the clustering algorithm needs to maintain a substantial amount of information so that important details are not lost. For example, the algorithm in [13] is implemented as a continuous version of k-means algorithm which continues to maintain a number of cluster centers which change or merge as necessary throughout the execution of the algorithm. Such an approach is especially risky when the characteristics of the stream evolve over time. This is because the k-means approach is highly sensitive to the order of arrival of the data points. For example, once two cluster centers are merged, there is no way to informatively split the clusters when required by the evolution of the stream at a later stage.

Therefore a natural design to stream clustering would separate out the process into an online micro-clustering component and an offline macro-clustering component. The online micro-clustering component requires a very efficient process for storage of appropriate summary statistics in a fast data stream. The offline component uses these summary statistics in conjunction with other user input in order to provide the user with a quick understanding of the clusters whenever required. Since the offline component requires only the summary statistics as input, it turns out to be very efficient in practice. This two-phased approach also provides the user with the flexibility to explore the *nature of the* evolution of the clusters over different time periods. This provides considerable insights to users in real applications.

This paper is organized as follows. In section 2, we will discuss the basic concepts underlying the stream clustering framework. In section 3, we will discuss how the micro-clusters are maintained throughout the stream generation process. In section 4, we discuss how the micro-clusters may be used by an offline macro-clustering component to create clusters of different spatial and temporal granularity. Since the algorithm is used for clustering of evolving data streams, it can also be used to determine the nature of cluster evolution. This process is described in section 5. Section 6 reports our performance study on real and synthetic data sets. Section 7 discusses the implication of the method and concludes our study.

2 The Stream Clustering Framework

In this section, we will discuss the framework of our stream clustering approach. We will refer to it as the CluStream framework. The separation of the stream clustering approach into online and offline components raises several important questions:

• What is the nature of the summary information which can be stored efficiently in a continuous data stream? The summary statistics should provide sufficient temporal and spatial information for a horizon-specific offline clustering process, while being prone to an efficient (online) update process.

• At what moments in time should the summary information be stored away on disk? How can an effective trade-off be achieved between the storage requirements of such a periodic process and the ability to cluster for a specific time horizon to within a desired level of approximation?

• How can the periodic summary statistics be used to provide clustering and evolution insights over user-specified time horizons?

In order to address these issues, we utilize two concepts which are useful for efficient data collection in a fast stream:

• **Micro-clusters:** We maintain statistical information about the data locality in terms of micro-clusters. These micro-clusters are defined as a temporal extension of the *cluster feature vector* [14]. The additivity property of the micro-clusters makes them a natural choice for the data stream problem.

• **Pyramidal Time Frame:** The micro-clusters are stored at snapshots in time which follow a pyramidal pattern. This pattern provides an effective trade-off between the storage requirements and the ability to recall summary statistics from different time horizons.

This summary information in the micro-clusters is used by an offline component which is dependent upon a wide variety of user inputs such as the time horizon or the granularity of clustering. We will now discuss a number of notations and definitions in order to introduce the above concepts.

It is assumed that the data stream consists of a set of multi-dimensional records $\overline{X}_1 \ldots \overline{X}_k \ldots$ arriving at time stamps $T_1 \ldots T_k \ldots$. Each $\overline{X}_i$ is a multi-dimensional record containing d dimensions which are denoted by $\overline{X}_i = (x_i^1 \ldots x_i^d)$.

We will first begin by defining the concept of micro-clusters and pyramidal time frame more precisely.

Definition 1 *A micro-cluster for a set of d-dimensional points $X_{i_1} \ldots X_{i_n}$ with time stamps*

$T_{i_1} \ldots T_{i_n}$ is defined as the $(2 \cdot d + 3)$ tuple $(\overline{CF2^x}, \overline{CF1^x}, CF2^t, CF1^t, n)$, wherein $\overline{CF2^x}$ and $\overline{CF1^x}$ each correspond to a vector of d entries. The definition of each of these entries is as follows:

- *For each dimension, the sum of the squares of the data values is maintained in $\overline{CF2^x}$. Thus, $\overline{CF2^x}$ contains d values. The p-th entry of $\overline{CF2^x}$ is equal to $\sum_{j=1}^{n} (x_{i_j}^p)^2$.*
- *For each dimension, the sum of the data values is maintained in $\overline{CF1^x}$. Thus, $\overline{CF1^x}$ contains d values. The p-th entry of $\overline{CF1^x}$ is equal to $\sum_{j=1}^{n} x_{i_j}^p$.*
- *The sum of the squares of the time stamps $T_{i_1} \ldots T_{i_n}$ is maintained in $CF2^t$.*
- *The sum of the time stamps $T_{i_1} \ldots T_{i_n}$ is maintained in $CF1^t$.*
- *The number of data points is maintained in n.*

We note that the above definition of micro-clusters is a temporal extension of the cluster feature vector in [14]. We will refer to the micro-cluster for a set of points $\mathcal{C}$ by $\overline{CFT}(\mathcal{C})$. As in [14], this summary information can be expressed in an additive way over the different data points. This makes it a natural choice for use in data stream algorithms. At a given moment in time, the statistical information about the dominant micro-clusters in the data stream is maintained by the algorithm. As we shall see at a later stage, the nature of the maintenance process ensures that a very large number of micro-clusters can be efficiently maintained as compared to the method discussed in [13]. The high granularity of the online updating process ensures that it is able to provide clusters of much better quality in an evolving data stream.

The micro-clusters are also stored at particular moments in the stream which are referred to as *snapshots*. The offline macro-clustering algorithm discussed at a later stage in this paper will use these finer level micro-clusters in order to create higher level clusters over specific time horizons. Consider the case when the current clock time is t_c and the user wishes to find clusters in the stream based on a history of length h. The macro-clustering algorithm discussed in this paper will use some of the *subtractive* properties[1] of the micro-clusters stored at snapshots t_c and $(t_c - h)$ in order to find the higher level clusters in a history or *time horizon* of length h. The subtractive property is a very important characteristic of the micro-clustering representation which makes it feasible to generate higher level clusters over different time horizons. Of course, since it is not possible to store the snapshots at each and every moment in time, it is important to choose particular instants of time at which the micro-clusters are stored. The aim of choosing these particular instants is to ensure that clusters in any user-specified time horizon $(t_c - h, t_c)$ can be approximated.

In order to achieve this, we will introduce the concept of a *pyramidal time frame*. In this technique, the snapshots are stored at differing levels of granularity depending upon the recency. Snapshots are classified into different *orders* which can vary from 1 to $\log(T)$, where T is the clock time elapsed since the beginning of the stream. The order of a particular class of snapshots defines the level of granularity in time at which the snapshots are maintained. The snapshots of different ordering are maintained as follows:

- Snapshots of the i-th order occur at time intervals of α^i, where α is an integer and $\alpha \geq 1$. Specifically, each snapshot of the i-th order is taken at a moment in time when the clock value[2] from the beginning of the stream is exactly divisible by α^i.
- At any given moment in time, only the last $\alpha + 1$ snapshots of order i are stored.

We note that the above definition allows for considerable redundancy in storage of snapshots. For example, the clock time of 8 is divisible by 2^0, 2^1, 2^2, and 2^3 (where $\alpha = 2$). Therefore, the state of the micro-clusters at a clock time of 8 simultaneously corresponds to order 0, order 1, order 2 and order 3 snapshots. From an implementation point of view, a snapshot needs to be maintained only once. We make the following observations:

- For a data stream, the maximum order of any snapshot stored at T time units since the beginning of the stream mining process is $\log_\alpha(T)$.
- For a data stream the maximum number of snapshots maintained at T time units since the beginning of the stream mining process is $(\alpha + 1) \cdot \log_\alpha(T)$.
- For any user-specified time window of h, at least one stored snapshot can be found within $2 \cdot h$ units of the current time.

While the first two results are quite easy to verify, the last one needs to be proven formally.

Lemma 1 *Let h be a user-specified time window, t_c be the current time, and t_s be the time of the last stored snapshot of any order just before the time $t_c - h$. Then $t_c - t_s \leq 2 \cdot h$.*

Proof: Let r be the smallest integer such that $\alpha^r \geq h$. Therefore, we know that $\alpha^{r-1} < h$. Since we know that there are $\alpha + 1$ snapshots of order $(r - 1)$, at least one snapshot of order $r - 1$ must *always* exist before $t_c - h$. Let t_s be the snapshot of order $r - 1$ which occurs just before $t_c - h$. Then $(t_c - h) - t_s \leq \alpha^{r-1}$. Therefore, we have $t_c - t_s \leq h + \alpha^{r-1} < 2 \cdot h$. ■

Thus, in this case, it is possible to find a snapshot within a factor of 2 of any user-specified time window. Furthermore, the total number of snapshots which need to be maintained is relatively modest. For example, for a data stream running[3] for 100 years with a clock time granularity of 1 second, the total number of snapshots which need to be maintained is given by

[1] This property will be discussed in greater detail in a later section.

[2] Without loss of generality, we can assume that one unit of clock time is the smallest level of granularity. Thus, the 0-th order snapshots measure the time intervals at the smallest level of granularity.

[3] The purpose of this rather extreme example is only to illustrate the efficiency of the pyramidal storage process in the most demanding case. In most real applications, the data stream is likely to be much shorter.

Order of Snapshots	Clock Times (Last 5 Snapshots)
0	55 ~~54~~ 53 ~~52~~ 51
1	54 ~~52~~ 50 ~~48~~ 46
2	52 ~~48~~ 44 ~~40~~ 36
3	~~48~~ 40 ~~32~~ 24 ~~16~~
4	48 ~~32~~ 16
5	32

Table 1: An example of snapshots stored for $\alpha = 2$ and $l = 2$

$(2+1) \cdot \log_2(100 * 365 * 24 * 60 * 60) \approx 95$. This is quite a modest storage requirement.

It is possible to improve the accuracy of time horizon approximation at a modest additional cost. In order to achieve this, we save the $\alpha^l + 1$ snapshots of order r for $l > 1$. In this case, the storage requirement of the technique corresponds to $(\alpha^l + 1) \cdot \log_\alpha(T)$ snapshots. On the other hand, the accuracy of time horizon approximation also increases substantially. In this case, any time horizon can be approximated to a factor of $(1 + 1/\alpha^{l-1})$. We summarize this result as follows:

Lemma 2 *Let h be a user-specified time horizon, t_c be the current time, and t_s be the time of the last stored snapshot of any order just before the time $t_c - h$. Then $t_c - t_s \leq (1 + 1/\alpha^{l-1}) \cdot h$.*

Proof: Similar to previous case. ∎

For larger values of l, the time horizon can be approximated as closely as desired. Consider the example (discussed above) of a data stream running for 100 years. By choosing $l = 10, \alpha = 2$, it is possible to approximate any time horizon within 0.2%, while a total of only $(2^{10} + 1) \cdot \log_2(100 * 365 * 24 * 60 * 60) \approx 32343$ snapshots are required for 100 years. Since historical snapshots can be stored on disk and only the current snapshot needs to be maintained in main memory, this requirement is quite feasible from a practical point of view. It is also possible to specify the pyramidal time window in accordance with user preferences corresponding to particular moments in time such as beginning of calendar years, months, and days. While the storage requirements and horizon estimation possibilities of such a scheme are different, all the algorithmic descriptions of this paper are directly applicable.

In order to clarify the way in which snapshots are stored, let us consider the case when the stream has been running starting at a clock-time of 1, and a use of $\alpha = 2$ and $l = 2$. Therefore $2^2 + 1 = 5$ snapshots of each order are stored. Then, at a clock time of 55, snapshots at the clock times illustrated in Table 1 are stored.

We note that a large number of snapshots are common among different orders. From an implementation point of view, the states of the micro-clusters at times of 16, 24, 32, 36, 40, 44, 46, 48, 50, 51, 52, 53, 54, and 55 are stored. It is easy to see that for more recent clock times, there is less distance between successive snapshots (better granularity). We also note that the storage requirements estimated in this section do not take this redundancy into account. Therefore, the requirements which have been presented so far are actually worst-case requirements.

An important question is to find a systematic rule which will eliminate the redundancy in the snapshots at different times. We note that in the example illustrated in Table 1, all the snapshots of order 0 occurring at odd moments (nondivisible by 2) need to be retained, since these are non-redundant. Once these snapshots have been retained and others discarded, all the snapshots of order 1 which occur at times that are not divisible by 4 are non-redundant. In general, all the snapshots of order l which are not divisible by 2^{l+1} are non-redundant. A redundant (hence not be generated) snapshot is marked by a crossbar on the number, such as ~~54~~, in Table 1. This snapshot generation rule also applies to the general case, when α is different from 2. We also note that whenever a new snapshot of a particular order is stored, the oldest snapshot of that order needs to be deleted.

3 Online Micro-cluster Maintenance

The micro-clustering phase is the online statistical data collection portion of the algorithm. This process is not dependent on any user input such as the time horizon or the required granularity of the clustering process. The aim is to maintain statistics at a sufficiently high level of (temporal and spatial) granularity so that it can be effectively used by the offline components such as horizon-specific macro-clustering as well as evolution analysis.

It is assumed that a total of q micro-clusters are maintained at any moment by the algorithm. We will denote these micro-clusters by $\mathcal{M}_1 \ldots \mathcal{M}_q$. Associated with each micro-cluster i, we create a unique *id* whenever it is first created. If two micro-clusters are merged (as will become evident from the details of our maintenance algorithm), a *list* of *id*s is created in order to identify the constituent micro-clusters. The value of q is determined by the amount of main memory available in order to store the micro-clusters. Therefore, typical values of q are significantly larger than the natural number of clusters in the data but are also significantly smaller than the number of data points arriving in a long period of time for a massive data stream. These micro-clusters represent the current snapshot of clusters which change over the course of the stream as new points arrive. Their status is stored away on disk whenever the clock time is divisible by α^i for any integer i. At the same time any micro-clusters of order r which were stored at a time in the past more remote than α^{l+r} units are deleted by the algorithm.

We first need to create the initial q micro-clusters. This is done using an offline process at the very beginning of the data stream computation process. At the very beginning of the data stream, we store the first $InitNumber$ points on disk and use a standard

k-means clustering algorithm in order to create the q initial micro-clusters. The value of $InitNumber$ is chosen to be as large as permitted by the computational complexity of a k-means algorithm creating q clusters.

Once these initial micro-clusters have been established, the online process of updating the micro-clusters is initiated. Whenever a new data point $\overline{X_{i_k}}$ arrives, the micro-clusters are updated in order to reflect the changes. Each data point either needs to be absorbed by a micro-cluster, or it needs to be put in a cluster of its own. The first preference is to absorb the data point into a currently existing micro-cluster. We first find the distance of each data point to the micro-cluster centroids $\mathcal{M}_1 \ldots \mathcal{M}_q$. Let us denote this distance value of the data point $\overline{X_{i_k}}$ to the centroid of the micro-cluster $\mathcal{M}_j$ by $dist(\mathcal{M}_j, \overline{X_{i_k}})$. Since the centroid of the micro-cluster is available in the cluster feature vector, this value can be computed relatively easily.

We find the closest cluster $\mathcal{M}_p$ to the data point $\overline{X_{i_k}}$. We note that in many cases, the point $\overline{X_{i_k}}$ does not naturally belong to the cluster $\mathcal{M}_p$. These cases are as follows:

- The data point $\overline{X_{i_k}}$ corresponds to an outlier.
- The data point $\overline{X_{i_k}}$ corresponds to the beginning of a new cluster because of evolution of the data stream.

While the two cases above cannot be distinguished until more data points arrive, the data point $\overline{X_{i_k}}$ needs to be assigned a (new) micro-cluster of its own with a unique id. How do we decide whether a completely new cluster should be created? In order to make this decision, we use the cluster feature vector of $\mathcal{M}_p$ to decide if this data point falls within the *maximum boundary* of the micro-cluster $\mathcal{M}_p$. If so, then the data point $\overline{X_{i_k}}$ is added to the micro-cluster $\mathcal{M}_p$ using the CF additivity property. The maximum boundary of the micro-cluster $\mathcal{M}_p$ is defined as a factor of t of the RMS deviation of the data points in $\mathcal{M}_p$ from the centroid. We define this as the *maximal boundary factor*. We note that the RMS deviation can only be defined for a cluster with more than 1 point. For a cluster with only 1 previous point, the maximum boundary is defined in a heuristic way. Specifically, we choose it to be the distance to the closest cluster.

If the data point does not lie within the maximum boundary of the nearest micro-cluster, then a new micro-cluster must be created containing the data point $\mathcal{X}_{i_k}$. This newly created micro-cluster is assigned a new id which can identify it uniquely at any future stage of the data steam process. However, in order to create this new micro-cluster, the number of other clusters must be reduced by one in order to create memory space. This can be achieved by either deleting an old cluster or joining two of the old clusters. Our maintenance algorithm first determines if it is safe to delete any of the current micro-clusters as outliers. If not, then a merge of two micro-clusters is initiated.

The first step is to identify if any of the old micro-clusters are possibly outliers which can be safely deleted by the algorithm. While it might be tempting to simply pick the micro-cluster with the fewest number of points as the micro-cluster to be deleted, this may often lead to misleading results. In many cases, a given micro-cluster might correspond to a point of considerable cluster presence in the past history of the stream, but may no longer be an active cluster in the recent stream activity. Such a micro-cluster can be considered an outlier from the current point of view. An ideal goal would be to estimate the average time-stamp of the last m arrivals in each micro-cluster [4], and delete the micro-cluster with the least recent time-stamp. While the above estimation can be achieved by simply storing the last m points in each micro-cluster, this increases the memory requirements of a micro-cluster by a factor of m. Such a requirement reduces the number of micro-clusters that can be stored by the available memory and therefore reduces the effectiveness of the algorithm.

We will find a way to approximate the average time-stamp of the last m data points of the cluster $\mathcal{M}$. This will be achieved by using the data about the time-stamps stored in the micro-cluster $\mathcal{M}$. We note that the time-stamp data allows us to calculate the mean and standard deviation[5] of the arrival times of points in a given micro-cluster $\mathcal{M}$. Let these values be denoted by $\mu\mathcal{M}$ and $\sigma\mathcal{M}$ respectively. Then, we find the time of arrival of the $m/(2 \cdot n)$-th percentile of the points in $\mathcal{M}$ assuming that the time-stamps are normally distributed. This time-stamp is used as the approximate value of the recency. We shall call this value as the *relevance stamp* of cluster $\mathcal{M}$. When the least relevance stamp of any micro-cluster is below a user-defined threshold δ, it can be eliminated and a new micro-cluster can be created with a unique id corresponding to the newly arrived data point $\overline{X_{i_k}}$.

In some cases, none of the micro-clusters can be readily eliminated. This happens when all relevance stamps are sufficiently recent and lie above the user-defined threshold δ. In such a case, two of the micro-clusters need to be merged. We merge the two micro-clusters which are closest to one another. The new micro-cluster no longer corresponds to one id. Instead, an $idlist$ is created which is a union of the ids in the individual micro-clusters. Thus, any micro-cluster which is the result of one or more merging operations can be identified in terms of the individual micro-clusters merged into it.

While the above process of updating is executed at the arrival of each data point, an additional process is executed at each clock time which is divisible by α^i for any integer i. At each such time, we store away the current set of micro-clusters (possibly on disk) together with their id list, and indexed by their time of storage. We also delete the least recent snapshot of order i, if α^l+1 snapshots of such order had already been

[4] If the micro-cluster contains fewer than $2 \cdot m$ points, then we simply find the average time-stamp of all points in the cluster.

[5] The mean is equal to $CF1^t/n$. The standard deviation is equal to $\sqrt{CF2^t/n - (CF1^t/n)^2}$.

stored on disk, and if the clock time for this snapshot is not divisible by α^{i+1}.

4 Macro-Cluster Creation

This section discusses one of the offline components, in which a user has the flexibility to explore stream clusters over different horizons. The micro-clusters generated by the algorithm serve as an intermediate statistical representation which can be maintained in an efficient way even for a data stream of large volume. On the other hand, the macro-clustering process does not use the (voluminous) data stream, but the compactly stored summary statistics of the micro-clusters. Therefore, it is not constrained by one-pass requirements.

It is assumed, that as input to the algorithm, the user supplies the time-horizon h, and the number of higher level clusters k which he wishes to determine. We note that the choice of the time horizon h determines the amount of history which is used in order to create higher level clusters. The choice of the number of clusters k determines whether more detailed clusters are found, or whether more rough clusters are mined.

We note that the set of micro-clusters at each stage of the algorithm is based on the entire history of stream processing since the very beginning of the stream generation process. When the user specifies a particular time horizon of length h over which he would like to find the clusters, then we need to find micro-clusters which are specific to that time-horizon. How do we achieve this goal? For this purpose, we find the additive property of the cluster feature vector very useful. This additive property is as follows:

Property 1 *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two sets of points. Then the cluster feature vector $\overline{CFT(\mathcal{C}_1 \cup \mathcal{C}_2)}$ is given by the sum of $\overline{CFT(\mathcal{C}_1)}$ and $\overline{CFT(\mathcal{C}_2)}$*

Note that this property for the temporal version of the cluster feature vector directly extends from that discussed in [14]. The following subtractive property is also true for exactly the same reason.

Property 2 *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be two sets of points such that $\mathcal{C}_1 \supseteq \mathcal{C}_2$. Then, the cluster feature vector $\overline{CFT(\mathcal{C}_1 - \mathcal{C}_2)}$ is given by $\overline{CFT(\mathcal{C}_1)} - \overline{CFT(\mathcal{C}_2)}$*

The subtractive property helps considerably in determination of the micro-clusters over a pre-specified time horizon. This is because by using two snapshots at pre-defined intervals, it is possible to determine the approximate micro-clusters for a pre-specified time horizon. Note that the micro-cluster maintenance algorithm always creates a unique *id* whenever a new micro-cluster is created. When two micro-clusters are merged, then the micro-clustering algorithm creates an *idlist* which is a list of all the original *ids* in that micro-cluster.

Consider the situation at a clock time of t_c, when the user wishes to find clusters over a past time horizon of h. In this case, we find the stored snapshot which occurs just before the time $t_c - h$. (The use of a pyramidal time frame ensures that it is always possible to find a snapshot at $t_c - h'$ where h' is within a pre-specified tolerance of the user-specified time horizon h.) Let us denote the set of micro-clusters at time $t_c - h$ by $\mathcal{S}(t_c - h')$ and the set of micro-clusters at time t_c by $\mathcal{S}(t_c)$. For each micro-cluster in the current set $\mathcal{S}(t_c)$, we find the list of *ids* in each micro-cluster. For each of the list of *ids*, we find the corresponding micro-clusters in $\mathcal{S}(t_c - h')$, and subtract the CF vectors for the corresponding micro-clusters in $\mathcal{S}(t_c - h')$. This ensures that the micro-clusters created before the user-specified time horizon do not dominate the results of the clustering process. We will denote this final set of micro-clusters created from the subtraction process by $\mathcal{N}(t_c, h')$. These micro-clusters are then subjected to the higher level clustering process to create a smaller number of micro-clusters which can be more easily understood by the user.

The clusters are determined by using a modification of a k-means algorithm. In this technique, the micro-clusters in $\mathcal{N}(t_c, h')$ are treated as *pseudo-points* which are re-clustered in order to determine higher level clusters. The k-means algorithm [10] picks k points as random seeds and then iteratively assigns database points to each of these seeds in order to create the new partitioning of clusters. In each iteration, the old set of seeds are replaced by the centroid of each partition. When the micro-clusters are used as pseudo-points, the k-means algorithm needs to be modified in a few ways:

- At the initialization stage, the seeds are no longer picked randomly, but are sampled with probability proportional to the number of points in a given micro-cluster. The corresponding seed is the centroid of that micro-cluster.
- At the partitioning stage, the distance of a seed from a given pseudo-point (or micro-cluster) is equal to the distance of the seed from the centroid of the corresponding micro-cluster.
- At the seed adjustment stage, the new seed for a given partition is defined as the weighted centroid of the micro-clusters in that partition.

It is important to note that a given execution of the macro-clustering process only needs to use two (carefully chosen) snapshots from the pyramidal time window of the micro-clustering process. The compactness of this input thus allows the user considerable flexibilities for querying the stored micro-clusters with different levels of granularity and time horizons.

5 Evolution Analysis of Clusters

Many interesting changes can be recorded by an analyst in an evolving data stream for effective use in a number of business applications [1]. In the context of the clustering problem, such evolution analysis also has significant importance. For example, an analyst may wish to know how the clusters have changed over the last quarter, the last year, the last decade and so on. For this purpose, the user needs to input a few

parameters to the algorithm:

• The two clock times t_1 and t_2 over which the clusters need to be compared. It is assumed that $t_2 > t_1$. In many practical scenarios, t_2 is the current clock time.

• The time horizon h over which the clusters are computed. This means that the clusters created by the data arriving between $(t_2 - h, t_2)$ are compared to those created by the data arriving between $(t_1 - h, t_1)$.

Another important issue is that of deciding how to present the changes in the clusters to a user, so as to make the results appealing from an intuitive point of view. We present the changes occurring in the clusters in terms of the following broad objectives:

• Are there new clusters in the data at time t_2 which were not present at time t_1?

• Have some of the original clusters been lost because of changes in the behavior of the stream?

• Have some of the original clusters at time t_1 shifted in position and nature because of changes in the data?

We note that the micro-cluster maintenance algorithm maintains the idlists which are useful for tracking cluster information. The first step is to compute $\mathcal{N}(t_1, h)$ and $\mathcal{N}(t_2, h)$ as discussed in the previous section. Therefore, we divide the micro-clusters in $\mathcal{N}(t_1, h) \cup \mathcal{N}(t_2, h)$ into three categories:

• Micro-clusters in $\mathcal{N}(t_2, h)$ for which none of the ids on the corresponding *idlist* are present in $\mathcal{N}(t_1, h)$. These are new micro-clusters which were created at some time in the interval (t_1, t_2). We will denote this set of micro-clusters by $\mathcal{M}^{added}(t_1, t_2)$.

• Micro-clusters in $\mathcal{N}(t_1, h)$ for which none of the corresponding ids are present in $\mathcal{N}(t_2, h)$. Thus, these micro-clusters were deleted in the interval (t_1, t_2). We will denote this set of micro-clusters by $\mathcal{M}^{deleted}(t_1, t_2)$.

• Micro-clusters in $\mathcal{N}(t_2, h)$ for which some or all of the ids on the corresponding *idlist* are present in the idlists corresponding to the micro-clusters in $\mathcal{N}(t_1, h)$. Such micro-clusters were at least partially created before time t_1, but have been modified since then. We will denote this set of micro-clusters by $\mathcal{M}^{retained}(t_1, t_2)$.

The macro-cluster creation algorithm is then separately applied to each of this set of micro-clusters to create a new set of higher level clusters. The macro-clusters created from $\mathcal{M}^{added}(t_1, t_2)$ and $\mathcal{M}^{deleted}(t_1, t_2)$ have clear significance in terms of clusters added to or removed from the data stream. The micro-clusters in $\mathcal{M}^{retained}(t_1, t_2)$ correspond to those portions of the stream which have not changed very significantly in this period. When a very large fraction of the data belongs to $\mathcal{M}^{retained}(t_1, t_2)$, this is a sign that the stream is quite stable over that time period.

6 Empirical Results

A thorough experimental study has been conducted for the evaluation of the CluStream algorithm on its accuracy, reliability, efficiency, scalability, and applicability. The performance results are presented in this section. The study validates the following claims: (1) CluStream derives higher quality clusters than traditional stream clustering algorithms, especially when the cluster distribution contains dramatic changes. It can answer many kinds of queries through its micro-cluster maintenance, macro-cluster creation, and change analysis over evolved data streams; (2) The pyramidal time frame and micro-clustering concepts adopted here assures that CluStream has much better clustering accuracy while maintaining high efficiency; and (3) CluStream has very good scalability in terms of stream size, dimensionality, and the number of clusters.

6.1 Test Environment and Data Sets

All of our experiments are conducted on a PC with Intel Pentium III processor and 512 MB memory, which runs Windows XP professional operating system. For testing the accuracy and efficiency of the CluStream algorithm, we compare CluStream with the STREAM algorithm [8, 13], the best algorithm reported so far for clustering data streams. CluStream is implemented according to the description in this paper, and the STREAM K-means is done strictly according to [13], which shows better accuracy than BIRCH [14]. To make the comparison fair, both CluStream and STREAM K-means use the same amount of memory. Specifically, they use the same stream incoming speed, the same amount of memory to store intermediate clusters (called Micro-clusters in CluStream), and the same amount of memory to store the final clusters (called Macro-clusters in CluStream).

Because the synthetic datasets can be generated by controlling the number of data points, the dimensionality, and the number of clusters, with different distribution or evolution characteristics, they are used to evaluate the scalability in our experiments. However, since synthetic datasets are usually rather different from real ones, we will mainly use real datasets to test accuracy, cluster evolution, and outlier detection.

Real datasets. First, we need to find some real datasets that evolve significantly over time in order to test the effectiveness of CluStream. A good candidate for such testing is the KDD-CUP'99 Network Intrusion Detection stream data set which has been used earlier [13] to evaluate STREAM accuracy with respect to BIRCH. This data set corresponds to the important problem of automatic and real-time detection of cyber attacks. This is also a challenging problem for dynamic stream clustering in its own right. The offline clustering algorithms cannot detect such intrusions in real time. Even the recently proposed stream clustering algorithms such as BIRCH and STREAM cannot be very effective because the clusters reported by these algorithms are all generated from the entire history of data stream, whereas the current cases may have evolved significantly.

The Network Intrusion Detection dataset consists

of a series of TCP connection records from two weeks of LAN network traffic managed by MIT Lincoln Labs. Each n record can either correspond to a normal connection, or an intrusion or attack. The attacks fall into four main categories: DOS (i.e., denial-of-service), R2L (i.e., unauthorized access from a remote machine), U2R (i.e., unauthorized access to local superuser privileges), and PROBING (i.e., surveillance and other probing). As a result, the data contains a total of five clusters including the class for "*normal connections*". The attack-types are further classified into one of 24 types, such as buffer-overflow, guess-passwd, neptune, portsweep, rootkit, smurf, warezclient, spy, and so on. It is evident that each specific attack type can be treated as a sub-cluster. Most of the connections in this dataset are *normal*, but occasionally there could be a burst of attacks at certain times. Also, each connection record in this dataset contains 42 attributes, such as duration of the connection, the number of data bytes transmitted from source to destination (and vice versa), percentile of connections that have "SYN" errors, the number of "root" accesses, etc. As in [13], all 34 continuous attributes will be used for clustering and one outlier point has been removed.

Second, besides testing on the rapidly evolving network intrusion data stream, we also test our method over relatively stable streams. Since previously reported stream clustering algorithms work on the entire history of stream data, we believe that they should perform effectively for some datasets with a relatively stable distribution over time. An example of such a data set is the KDD-CUP'98 Charitable Donation data set. We will show that even for such datasets, the CluStream can consistently outperform the STREAM algorithm.

The KDD-CUP'98 Charitable Donation data set has also been used in evaluating several one-scan clustering algorithms, such as [7]. This dataset contains 95412 records of information about people who have made charitable donations in response to direct mailing requests, and clustering can be used to group donors showing similar donation behavior. As in [7], we will only use 56 fields which can be extracted from the total 481 fields of each record. This data set is converted into a data stream by taking the data input order as the order of streaming and assuming that they flow-in with a uniform speed.

Synthetic datasets. To test the scalability of CluStream, we generate some synthetic datasets by varying base size from 100K to 1000K points, the number of clusters from 4 to 64, and the dimensionality in the range of 10 to 100. Because we know the true cluster distribution a priori, we can compare the clusters found with the true clusters. The data points of each synthetic dataset will follow a series of Gaussian distributions. In order to reflect the evolution of the stream data over time, we change the mean and variance of the current Gaussian distribution every 10K points in the synthetic data generation.

The quality of clustering on the real data sets was measured using the sum of square distance (SSQ), defined as follows. Assume that there are a total of nh points in the past horizon at current time T_c. For each point p_i in this horizon, we find the centroid C_{p_i} of its closest macro-cluster, and compute $d(p_i, C_{p_i})$, the distance between p_i and C_{p_i}. Then the SSQ at time T_c with horizon H (denoted as $SSQ(T_c, H)$) is equal to the sum of $d^2(p_i, C_{p_i})$ for all the nh points within the previous horizon H. Unless otherwise mentioned, the algorithm parameters were set at $\alpha = 2$, $l = 10$, $InitNumber = 2000$, $\delta = 512$, and $t = 2$.

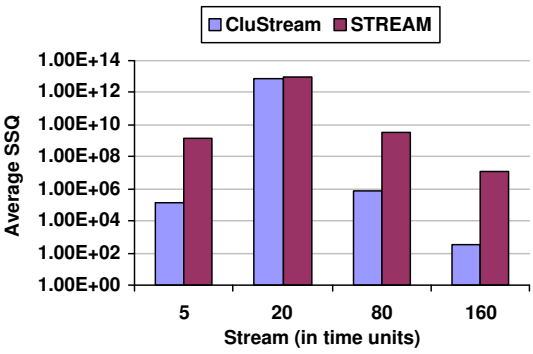

Figure 1: Quality comparison (Network Intrusion dataset, horizon=1, stream_speed=2000)

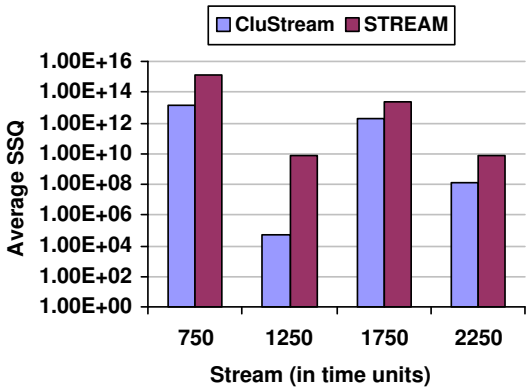

Figure 2: Quality comparison (Network Intrusion dataset, horizon=256, stream_speed=200)

6.2 Clustering Evaluation

One novel feature of CluStream is that it can create a set of macro-clusters for any user-specified horizon at any time upon demand. Furthermore, we expect CluStream to be more effective than current algorithms at clustering rapidly evolving data streams. We will first show the effectiveness and high quality of CluStream in detecting network intrusions.

We compare the clustering quality of CluStream with that of STREAM for different horizons at different times using the Network Intrusion dataset. For

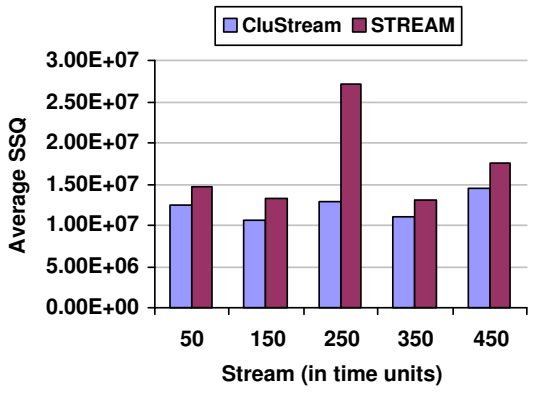

Figure 3: Quality comparison (Charitable Donation dataset, horizon=4, stream_speed=200)

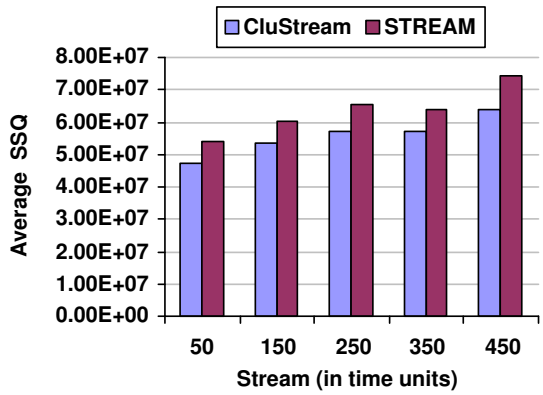

Figure 4: Quality comparison (Charitable Donation dataset, horizon=16, stream_speed=200)

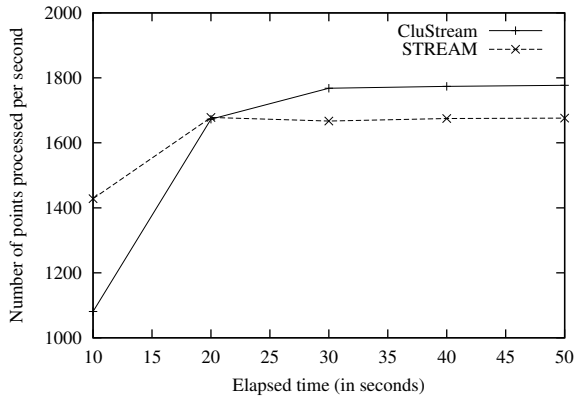

Figure 5: Stream Processing Rate (Charitable Donation dataset, stream_speed=2000)

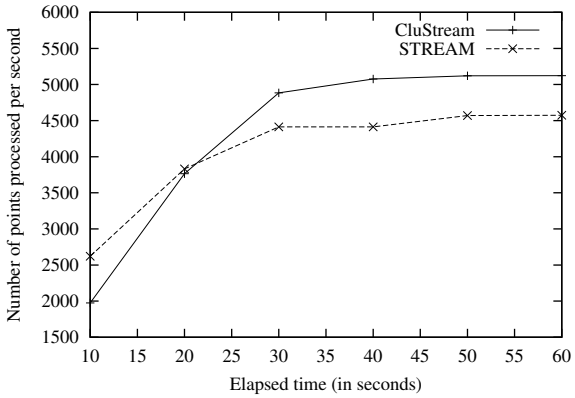

Figure 6: Stream Processing Rate (Network Intrusion dataset, stream_speed=2000)

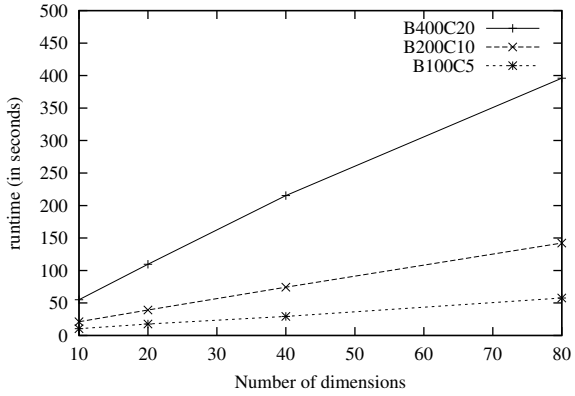

Figure 7: Scalability with Data Dimensionality (stream_speed=2000)

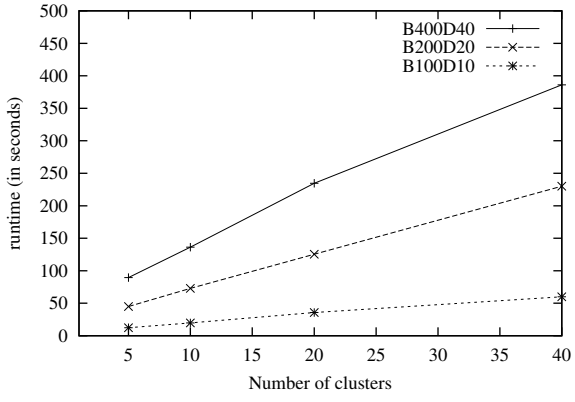

Figure 8: Scalability with Number of Clusters (stream_speed=2000)

each algorithm, we determine 5 clusters. All experiments for this dataset have shown that CluStream has substantially higher quality than STREAM. Figures 1 and 2 show some of our results, where stream_speed = 2000 means that the stream in-flow speed is 2000 points per time unit. We note that the Y-axis is drawn on a logarithmic scale, and therefore the improvements correspond to orders of magnitude. We run each algorithm 5 times and compute their average SSQs. From Figure 1 we know that CluStream is almost always better than STREAM by several orders of magnitude. For example, at time 160, the average SSQ of CluStream is almost 5 orders of magnitude smaller than that of STREAM. At a larger horizon like 256, Figure 2 shows that CluStream can also get much higher clustering quality than STREAM. The average SSQ values at different times consistently continue to be order(s) of magnitude smaller than those of STREAM. For example, at time 1250, CluStream's average SSQ is more than 5 orders of magnitude smaller than that of STREAM.

The surprisingly high clustering quality of CluStream benefits from its good design. On the one hand, the pyramidal time frame enables CluStream to approximate any time horizon as closely as desired. On the other hand, the STREAM clustering algorithm can only be based on the entire history of the data stream. Furthermore, the large number of micro-clusters maintain a sufficient amount of summary information in order to contribute to the high accuracy. In addition, our experiments demonstrated CluStream is more reliable than STREAM algorithm. In most cases, no matter how many times we run CluStream, it always returns the same (or very similar) results. More interestingly, the fine granularity of the micro-cluster maintenance algorithm helps CluStream in detecting the real attacks. For example, at time 320, all the connections belong to the neptune attack type for any horizon less than 16. The micro-cluster maintenance algorithm always absorbs all data points in the same micro-cluster. As a result, CluStream will successfully cluster all these points into one macro-cluster. This means that it can detect a distinct cluster corresponding to the network attack correctly. On the other hand, the STREAM algorithm always mixes up these neptune attack connections with the normal connections or some other attacks. Similarly, CluStream can find one cluster (neptune attack type in underlying data set) at time 640, two clusters (neptune and smurf) at time 650, and one cluster (smurf attack type) at time 1280. These clusters correspond to true occurrences of important changes in the stream behavior, and are therefore intuitively appealing from the point of view of a user.

Now we examine the performance of stream clustering with the Charitable Donation dataset. Since the Charitable Donation dataset does not evolve much over time, STREAM should be able to cluster this data set fairly well. Figures 3 and 4 show the comparison results between CluStream and STREAM. The results show that CluStream outperforms STREAM even in this case, which indicates that CluStream is effective for both evolving and stable streams.

6.3 Scalability Results

The key to the success of the clustering framework is high scalability of the micro-clustering algorithm. This is because this process is exposed to a potentially large volume of incoming data and needs to be implemented in an efficient and online fashion. On the other hand, the (offline) macro-clustering part of the process required only a (relatively) negligible amount of time. This is because of its use of the compact micro-cluster representation as input.

The most time-consuming and frequent operation during micro-cluster maintenance is that of finding the closest micro-cluster for each newly arrived data point. It is clear that the complexity of this operation increases linearly with the number of micro-clusters. It is also evident that the number of micro-clusters maintained should be sufficiently larger than the number of input clusters in the data in order to obtain a high quality clustering. While the number of input clusters cannot be known a priori, it is instructive to examine the scalability behavior when the number of micro-clusters was fixed at a constant large factor of the number of input clusters. Therefore, for all the experiments in this section, we will fix the number of micro-clusters to 10 times the number of input clusters. We tested the efficiency of CluStream micro-cluster maintenance algorithm with respect to STREAM on the real data sets.

Figures 5 and 6 show the stream processing rate (the number of points processed per second) with progression of the data stream. Since CluStream requires some time to compute the initial set of micro-clusters, its precessing rate is lower than STREAM at the very beginning. However, once steady state is reached, CluStream becomes faster than STREAM in spite of the fact that it needs to store the snapshots to disk periodically. This is because STREAM takes a few iterations to make k-means clustering converge, whereas CluStream just needs to judge whether a set of points will be absorbed by the existing micro-clusters and insert into them appropriately. We make the observation that while CluStream maintains 10 times higher granularity of the clustering information compared to STREAM, the processing rate is also much higher.

We will present the scalability behavior of the CluStream algorithm with data dimensionality, and the number of natural clusters. The scalability results report the total processing time of the micro-clustering process over the entire data stream. The first series of data sets were generated by varying the dimensionality from 10 to 80, while fixing the number of points and input clusters. The first data set series B100C5 indicates that it contains 100K points and 5 clusters. The same notational convention is used for the second data set series B200C10 and the third one B400C20. Figure 7 shows the experimental results, from which one can see that CluStream has linear scalability with

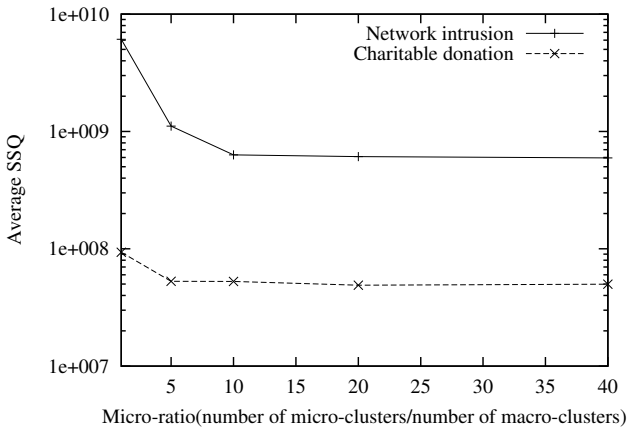

Figure 9: Accuracy Impact of Micro-clusters

data dimensionality. For example, for dataset series B400C20, when the dimensionality increases from 10 to 80, the running time increases less than 8 times from 55 seconds to 396 seconds.

Another three series of datasets were generated to test the scalability against the number of clusters by varying the number of input clusters from 5 to 40, while fixing the stream size and dimensionality. For example, the first data set series B100D10 indicates it contains 100K points and 10 dimensions. The same convention is used for the other data sets. Figure 8 demonstrates that CluStream has linear scalability with the number of input clusters.

6.4 Sensitivity Analysis

In section 3, we indicated that the number of micro-clusters should be larger than the number of natural clusters in order to obtain a clustering of good quality. However, a very large number of micro-clusters is inefficient in terms of running time and storage. We define *micro-ratio* as the number of micro-clusters divided by the number of natural clusters. It is desirable that a high quality clustering can be reached by a reasonably small micro-ratio. We will determine the typical micro-ratios used by the CluStream algorithm in this section.

We fix the stream_speed at 200 points (per time unit), and horizon at 16 time units. We use the two real datasets to test the clustering quality by varying the number of micro-clusters. For each dataset, we determine the macro-clusters over the corresponding time horizon, and measure the clustering quality using the sum of square distance (SSQ).

Figure 9 shows our experimental results related to the accuracy impact of micro-ratio, where we fix T_c at 200 for Charitable Donation dataset and at 1000 for Network Intrusion dataset. We can see that if we use the same number of micro-clusters as the natural clusters, the clustering quality is quite poor. This is because the use of a very small number of micro-clusters defeats the purpose of a micro-cluster approach. When the micro-ratio increases, the average SSQ reduces. The average SSQ for each real dataset becomes stable when the micro-ratio is about 10. This indicates that to achieve high-quality clustering, the micro-ratio does not need to be too large as compared to the natural clusters in the data. Since the number of micro-clusters is limited by the available memory, this result brings good news: for most real applications, the use of a very modest amount of memory is sufficient for the micro-clustering process.

Factor t	1	2	4	6	8
Net. Int.	14.85	1.62	0.176	0.0144	0.0085
Cha. Don.	11.18	0.12	0.0074	0.0021	0.0021

Table 2: Exception percent vs. Max. Boundary Factor t

Another important parameter which may significantly impact the clustering quality is the maximal boundary of a micro-cluster. As discussed earlier, this was defined as a factor t of the RMS deviation of the data points from the corresponding cluster centroid. The value of t should be chosen small enough, so that it can successfully detect most of the points representing new clusters or outliers. At the same time, it should not generate too many unpromising new micro-clusters or outliers. By varying the factor t from 1 to 8, we ran the CluStream algorithm for both the real datasets and recorded all the exception points which fall outside of the maximal boundary of its closest micro-cluster. Table 2 shows the percentage of the total number of data points in each real dataset that are judged belonging to exception points at different values of the factor t. Table 2 shows that if factor t is less than 1, there will be too many exception points. Typically, a choice of $t = 2$ resulted in an exception percentile which did not reduce very much on increasing t further. We also note that if the distances of the data points to the centroid had followed a Gaussian distribution, the value $t = 2$ results in more than 95% of the data points within the corresponding cluster boundary. Therefore, the value of the factor t was set at 2 for all experiments in this paper.

6.5 Evolution Analysis

Our experiments also show that CluStream facilitates cluster evolution analysis. Taking the Network Intrusion dataset as an example, we show how such an analysis is performed. In our experiments, we assume that the network connection speed is 200 connections per time unit.

First, by comparing the data distribution for $t_1 = 29, t_2 = 30, h = 1$ CluStream found 3 micro-clusters (8 points) in $\mathcal{M}^{added}(t_1, t_2)$, 1 micro-cluster (1 point) in $\mathcal{M}^{deleted}(t_1, t_2)$, and 22 micro-clusters (192 points) in $\mathcal{M}^{retained}(t_1, t_2)$. This shows that only 0.5% of all the connections in $(28, 29)$ disappeared and only 4% were added in $(29, 30)$. By checking the original dataset, we find that all points in $\mathcal{M}^{added}(t_1, t_2)$

and $\mathcal{M}^{deleted}(t_1, t_2)$ are normal connections, but are outliers because of some particular feature such as the number of bytes of data transmitted. The fact that almost all the points in this case belong to $\mathcal{M}^{retained}(t_1, t_2)$ indicates that the data distributions in these two windows are very similar. This happens because there are no attacks in this time period.

More interestingly, the data points falling into $\mathcal{M}^{added}(t_1, t_2)$ or $\mathcal{M}^{deleted}(t_1, t_2)$ are those which have evolved significantly. These usually correspond to newly arrived or faded attacks respectively. Here are two examples: (1) During the period $(34, 35)$, all data points correspond to normal connections, whereas during $(39, 40)$ all data points belong to smurf attacks. By applying our change analysis procedure for $t_1 = 35, t_2 = 40, h = 1$, it shows that 99% of the smurf connections (i.e., 198 connections) fall into two $\mathcal{M}^{added}(t_1, t_2)$ micro-clusters, and 99% of the normal connections fall into 21 $M^{deleted}(t_1, t_2)$ micro-clusters. This means these normal connections are non-existent during $(39, 40)$; (2) By applying the change analysis procedure for $t_1 = 640, t_2 = 1280, h = 16$, we found that all the data points during $(1264, 1280)$ belong to one $\mathcal{M}^{added}(t_1, t_2)$ micro-cluster, and all the data points in $(624, 640)$ belong to one $\mathcal{M}^{deleted}(t_1, t_2)$ micro-cluster. By checking the original labeled data set, we found that all the connections during $(1264, 1280)$ are smurf attacks and all the connections during $(624, 640)$ are neptune attacks.

7 Discussion and Conclusions

In this paper, we have developed an effective and efficient method, called CluStream, for clustering large evolving data streams. The method has clear advantages over recent techniques which try to cluster the whole stream at one time rather than viewing the stream as a changing process over time. The CluStream model provides a wide variety of functionality in characterizing data stream clusters over different time horizons in an evolving environment. This is achieved through a careful division of labor between the online statistical data collection component and an offline analytical component. Thus, the process provides considerable flexibility to an analyst in a real-time and changing environment. These goals were achieved by a careful design of the statistical storage process. The use of a *pyramidal time window* assures that the essential statistics of *evolving* data streams can be captured without sacrificing the underlying *space- and time-efficiency* of the stream clustering process. Further, the exploitation of *microclustering* ensures that CluStream can achieve *higher accuracy* than STREAM due to its registering of more detailed information than the k points used by the k-means approach. The use of micro-clustering ensures scalable data collection, while retaining the sufficiency of data required for effective clustering.

A wide spectrum of clustering methods have been developed in data mining, statistics, machine learning with many applications. Although very few have been examined in the context of stream data clustering, we believe that the framework developed in this study for separating out periodic statistical data collection through a pyramidal time window provides a unique environment for re-examining these techniques. As future work, we are going to examine the application of the CluStream methodology developed here to other clustering paradigms for data streams.

References

[1] C. C. Aggarwal. A Framework for Diagnosing Changes in Evolving Data Streams. *ACM SIGMOD Conference*, 2003.

[2] M. Ankerst et al. OPTICS: Ordering Points To Identify the Clustering Structure. *ACM SIGMOD Conference*, 1999.

[3] B. Babcock et al. Models and Issues in Data Stream Systems, *ACM PODS Conference*, 2002.

[4] P. Bradley, U. Fayyad, C. Reina. Scaling Clustering Algorithms to Large Databases. *SIGKDD Conference*, 1998.

[5] C. Cortes et al. Hancock: A Language for Extracting Signatures from Data Streams. *ACM SIGKDD Conference*, 2000.

[6] P. Domingos, G. Hulten. Mining High-Speed Data Streams. *ACM SIGKDD Conference*, 2000.

[7] F. Farnstrom, J. Lewis, C. Elkan. Scalability for Clustering Algorithms Revisited. *SIGKDD Explorations*, 2(1):51-57, 2000.

[8] S. Guha, N. Mishra, R. Motwani, L. O'Callaghan. Clustering Data Streams. *IEEE FOCS Conference*, 2000.

[9] S. Guha, R. Rastogi, K. Shim. CURE: An Efficient Clustering Algorithm for Large Databases. *ACM SIGMOD Conference*, 1998.

[10] A. Jain, R. Dubes. Algorithms for Clustering Data, *Prentice Hall,* New Jersey, 1998.

[11] L. Kaufman, P. Rousseuw. Finding Groups in Data- An Introduction to Cluster Analysis. *Wiley Series in Probability and Math. Sciences,* 1990.

[12] R. Ng, J. Han. Efficient and Effective Clustering Methods for Spatial Data Mining. *Very Large Data Bases Conference*, 1994.

[13] L. O'Callaghan et al. Streaming-Data Algorithms For High-Quality Clustering. *ICDE Conference*, 2002.

[14] T. Zhang, R. Ramakrishnan, M. Livny. BIRCH: An Efficient Data Clustering Method for Very Large Databases. *ACM SIGMOD Conference,* 1996.