CLUES: A Unified Framework Supporting Interactive Exploration of Density-Based Clusters in Streams

Di Yang, Zhenyu Guo, Elke A. Rundensteiner, and Matthew O. Ward
Worcester Polytechnic Institute
100 Institute Road, Worcester, MA, USA
diyang, zyguo, rundenst, matt@cs.wpi.com

ABSTRACT

Although various mining algorithms have been proposed in the literature to efficiently compute clusters, few strides have been made to date in helping analysts to interactively explore such patterns in the stream context. We present a framework called CLUES to both computationally and visually support the process of real-time mining of density-based clusters. CLUES is composed of three major components. First, as foundation of CLUES, we develop an evolution model of density-based clusters in data streams that captures the complete spectrum of cluster evolution types across streaming windows. Second, to equip CLUES with the capability of efficiently tracking cluster evolution, we design a novel algorithm to piggy-back the evolution tracking process into the underlying cluster detection process. Third, CLUES organizes the detected clusters and their evolution interrelationships into a multi-dimensional pattern space – presenting clusters at different time horizons and across different abstraction levels. It provides a rich set of visualization and interaction techniques to allow the analyst to explore this multi-dimensional pattern space in real-time. Our experimental evaluation, including performance studies and a user study, using real streams from ground group movement monitoring and from stock transaction domains confirm both the efficiency and effectiveness of our proposed CLUES framework.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms, Management, Performance

Keywords

Density-Based Cluster, Evolution, Stream, Visualization

1. INTRODUCTION

The discovery of complex patterns such as clusters, outliers, and associations from huge volumes of streaming data has been recognized as critical for numerous domains, ranging from financial analysis to moving object monitoring. Such applications built upon high-speed data streams not only expect a stream processing system to computationally detect patterns in a highly efficient manner, but also require it to provide an exploratory platform that helps the analysts to interact with and thus comprehend both the detected patterns and their interrelationships. For example, in traffic monitoring applications, an analyst does not only need to know the major traffic congestions (clusters) just detected in the traffic streams, but also needs to keep track of how these clusters evolve over time. During routine monitoring, the analyst may monitor the clusters at a highly abstract level, such as whether new patterns formed recently or whether an earlier cluster now has split into several smaller ones. A closer exploration may be necessary in some circumstances. For example, a cluster that is growing quickly in size but not moving in position may indicate a traffic congestion or even an accident. The analyst then needs to zoom into that particular cluster to study the cluster structure (how vehicles are distributed in the cluster) and obtain information on the individual vehicles needing help. Obviously, using the traditional console or file output to simply report the clusters as tuples associated with cluster identifications would be of limited use, because the analysts can hardly comprehend such high volume data presented in plain text in real time.

To better support complex pattern mining in the streaming environments poses the following requirements: **1)** pattern evolution semantics as foundation are neccessary to characterize pattern changes over time in the streams; **2)** efficient pattern and evolution computation methods have to be designed to handle high input rate streams; **3)** appropriate visualization techniques must effectively convey the detected patterns and their interrelationships; and **4)** most importantly, all these critical components above need to be well integrated into a unified framework, which provides analysts an efficient and easy-to-use exploration platform.

Target Query Semantics. We support mining of density-based clusters [13, 14] in *sliding stream windows*. For this query semantics, arbitrarily shaped clusters are continuously detected within the most recent portion of the stream that fall into the query window. The traffic congestion monitoring task discussed above is an example that requires such query semantics. This is because the traffic congestions can be in any shape, and position information being valid for only a certain period of time necessitates cluster formation based on the most recent positions of the vehicles. Other applications that can be served by such query semantics include detecting intensive-transaction areas in most recent stock trades, and identifying malicious attacks in current network traffic.

1.1 Analysis of State-of-the-Art

While mining of streaming data has been studied for various pattern types [2, 9, 7], little attention has been paid to date for design of holistic frameworks that support the overall mining process for density-based clusters in sliding windows. First, an evolution model that effectively characterizes the changes of density-based cluster structures over time is needed. Reporting the changes of patterns in data streams [1, 2, 5, 6, 10, 17] has long been recognized as important. The previous efforts of clustering streaming data [2, 5, 6, 10, 17] assume that the streaming clusters are sphere-shaped and can be expressed using accumulative statistics. They use a "subtraction" function to calculate the difference between the statistics of clusters identified at different time points. However, these techniques usually do not deal with cluster changes caused by cluster membership transformations of individual objects, such as split and merge. Also they either cannot handle clusters with arbitrary shapes or with sliding window semantics, and thus do not meet the requirements of the applications mentioned earlier.

To our knowledge, no evolution semantics specific to density-based clusters in streaming environment have been presented so far in the literature. [17] is perhaps the closest research effort to us in terms of designing cluster evolution model. It presents a cluster change model, MONIC, to track cluster transactions. However, this model is neither designed for the streaming environment nor specifically for density-based clusters. In particular, in streaming window semantics, as object expirations and arrivals happen within a single window slide process, an evolution model needs to handle the composite effect caused by both of them in a single transaction (see section 3). While [17] provides evolution semantics for covering losing (expiration) or gaining (arrivals) cluster members separately, it does not offer semantics covering the situations in which both happen together. Second, to handle potentially high speed input streams, the cluster changes have to be detected and conveyed in a highly efficient manner. For density-based clusters, monitoring the changes of their complex cluster structures in streaming environment is particularly challenging. However, [17] provides neither efficient computational methods nor visualization and interaction techniques to allow analysts to quickly identify and understand the cluster changes in real time.

Second, to track cluster evolution across sliding windows, an efficient computational method needs to be designed. Existing algorithms for detecting density-based clusters in streams [8, 9] do not meet the requirements of our target applications, because they neither identify the individual members in the clusters nor enforce the sliding window semantics for the clustering process. [20], the only algorithm we are aware of that detects density-based clusters in sliding windows, suffers from dramatically increasing demands on system resources when the ratio between the window size and the slide size of the query window increases. We characterize this problem in depth in Section 4. More importantly, none of these clustering algorithms support evolution tracking for density-based clusters, which is one of the key focuses of this work.

Third, appropriate visualizations are needed to display the clusters as well as their evolution. Previous research focuses on visualizing time series data [15, 19], data associated with timestamps in static datasets. In such context, no issues of data arrival speed nor continuous updates of streams arise. Also, although visualization tools [11, 18] have been developed to present cluster structures in static environments, none of them tackle visualization and real-time visual interaction support for monitoring clusters and their evolution in the stream context.

1.2 Proposed Solution

In this work, we present a unified CLUster Evolution in Stream analysis framework, CLUES, to both computationally and visually support cluster mining in data streams.

First, we design the first *evolution semantics* for density-based clusters over sliding windows, deployed to model cluster evolutions in the CLUES framework. We address the following two key challenges. 1) Our proposed semantics not only cover statistical changes of individual clusters, such as the size or the centroid changes, but also classify structural cluster changes that involve multiple clusters, including splitting and merging of clusters. Such semantics are shown to effectively describe the changes of clusters over time in our user study. 2) Our proposed evolution semantics are efficiently computable. This is critical for real time applications, because a system conducting expensive clustering processes can hardly bear any additional computational burden.

Second, a novel algorithm is proposed to serve as the computational component of CLUES. It does not only efficiently extract clusters from sliding windows, but more importantly to also track cluster changes based on our proposed evolution semantics. This algorithm uses a compact hierarchical structure to incrementally store and update the progressive clusters and their evolution interrelationships. Our experimental study (Section 6) demonstrates that our proposed algorithm is not only on average 300% times faster than the state-of-the-art detection technique [20], but also consumes significantly (on average 60%) less memory space. In addition, this algorithm only consumes a very modest amount (less than 10%) of extra system resources to conduct evolution tracking along with the clustering process.

Finally, CLUES organizes the detected clusters and their evolution information into a multi-dimensional pattern space, with one dimension representing the cluster changes over time and the other representing clusters in different abstraction levels. We have designed a rich set of visualization and interaction techniques to enable analysts to easily navigate through the proposed pattern space by reviewing, monitoring, and even predicting cluster changes over time at different levels of abstraction. Our user study in Section 6 shows that our system can effectively improve both the efficiency and accuracy of human analysts for tracking cluster evolution compared to the alternative strategies.

2. PRELIMINARIES

We now introduce the concept of density-based clusters [13, 14]. We use the term "data point" to refer to a multi-dimensional tuple in the data stream. Density-based cluster detection uses a range threshold $\theta^{range} \geq 0$ to define the *neighborship* between any two data points. For two data points p_i and p_j, if the distance between them is no larger than θ^{range}, p_i and p_j are said to be neighbors. We use the function $NumNeigh(p_i, \theta^{range})$ to denote the number of neighbors a data point p_i has, given the θ^{range} threshold.

Definition 2.1. ***Density-Based Cluster:*** *Given θ^{range} and a count threshold θ^{count}, a data point p_i with $NumNeigh(p_i, \theta^{range}) \geq \theta^{count}$ is defined as a core point. Otherwise, if p_i is a neighbor of any core point, p_i is an edge point. p_i is a noise point if it is neither a core point nor an edge point. Two core points c_0 and c_n are connected, if they are neighbors of each other, or there exists a sequence of core points $p_0, p_1, ...p_{n-1}, c_n$, where for any i with $0 \leq i \leq n-1$, each pair of core points p_i and p_{i+1} are neighbors of each other. Finally, a density-based cluster is defined as a maximum group of "connected core points" and the edge points attached to them. Any pair of core points within a cluster are "connected". Figure 2 shows an example of a density-based cluster composed of 11 core points (black) and 2 edge points (grey) in window W_0.*

We focus on periodic sliding window semantics as proposed in *Countinuous Query Language* [3] and widely used in the literature [4, 20]. These proposed semantics can be either time-based or count-based. In both cases, each query has a query window W with with a fixed window size win and a fixed slide size $slide$ (either a time interval or a tuple count). The query window covers the most recent portion of the stream quantified by the window size win, say the latest 10K tuples (count-based) or the tuples coming into the system within the last 10 minutes. When a certain amount of time has elapsed (time-based) or a certain number of tuples have arrived (count-based), the query window slides (moves forward) by the slide size $slide$. During each window slide, old tuples expire from the query window, while new tuples arrive and fall into the query window. Such query semantics periodically extract patterns based on all tuples falling into the query window.

3. EVOLUTION MODEL OF CLUSTERS

In this section, we propose the first evolution model that models the semantics of density-based clusters changes over sliding windows. The proposed evolution semantics is defined between any two adjacent sliding windows upon a data stream, say W_n and W_{n+1}. The process of sliding a query window can be divided into two phases, namely purging the expired and then adding the new data points. Our evolution semantics covers not only the cluster changes caused by each of these two phases in isolation, but also by both of them together. We call the cluster changes caused by either of these two phases, *single-step evolution*, and those caused by both of them, *multiple-step evolution*.

Model for Single-Step Evolution. Our single-step evolution semantics covers seven change types. Given two cluster sets Clu_set1 and Clu_set2 identified in W_n and W_{n+1} respectively, the change types of single-step evolution are:

***birth**:* If a cluster C_i in Clu_set2 does not include any core point members from any cluster C_j in Clu_set1, we define this as the *birth* of a new cluster C_i in Clu_set2.

***termination**:* If none of the core point members of a cluster C_i in Clu_set1 appear in any cluster C_j in Clu_set2, we define this as the *termination* of C_i.

***split**:* A cluster C_i in Clu_set1 *splits*, if its non-expired core point members now belong to at least two different clusters in Clu_set2.

***merge**:* Clusters C_i, C_{i+1}... C_{i+n} in Clu_set1 *merge*, if all their core point members remaining in W_{n+1} now belong to a single cluster C_j in Clu_set2.

***preserve/shrink/expand**:* A cluster C_i in Clu_set1 *preserves/ shrinks/expands*, if no *termination*, *split* or *merge* has happened to it, and its size doesn't change/decrease/increase significantly in Clu_set2 respectively. Here the definition of "significance" can be controlled by an adjustable threshold θ^{sig}, which denotes a percentage of the original cluster size of C_i in W_n.

These seven change types describe all possible cluster changes from one window to the next if considering either tuple expirations or tuple insertions only.

Lemma 3.1. *For any density-based cluster set Clu_set identified in a window W_n, the seven change types above cover all potential cluster changes that can be caused by either inserting or deleting data points from W_n.*

Here we give an intuitive explanation for Lemma 3.1. When inserting new tuples into W_n, these tuples can only either join the existing clusters or fall into non-cluster areas in W_n. **1)** When new tuples join existing clusters, they can only cause either ***preservation*** or ***expansion*** of individual clusters, depending on whether the number of new members each cluster gains is significant or not, or ***merge*** of several clusters, when new tuples build connections between existing clusters. **2)** When new tuples fall into non-cluster areas, they can only cause either the **birth** of new clusters (as density reaches the threshold) or no cluster changes (as they become new noise). Thus, insertion of new data points into W_n can only cause ***birth***, ***merge***, ***expansion*** or ***preservation*** of the existing clusters in Clu_set.

Second, when data points are deleted from W_n, they can either be members of existing clusters in Clu_set or belong to non-cluster areas (noises). **1)** Deleting cluster members from existing clusters can only cause ***shrinkage*** or ***preservation*** of clusters, depending on whether the number of members each cluster loses is significant or not, ***termination*** of clusters, when the whole cluster structure is deleted or collapsed after losing members, or ***split*** of the clusters, when the connections between subparts of an existing cluster are broken but at least two subparts still constitute smaller clusters. **2)** Deleting noises causes no cluster changes. Thus, deleting data points from W_n can only cause ***shrinkage*** or ***preservation***, ***termination*** or ***split*** of clusters

This shows that the seven single-step change types cover all potential cluster changes that can be caused by either inserting or deleting tuples from W_n.

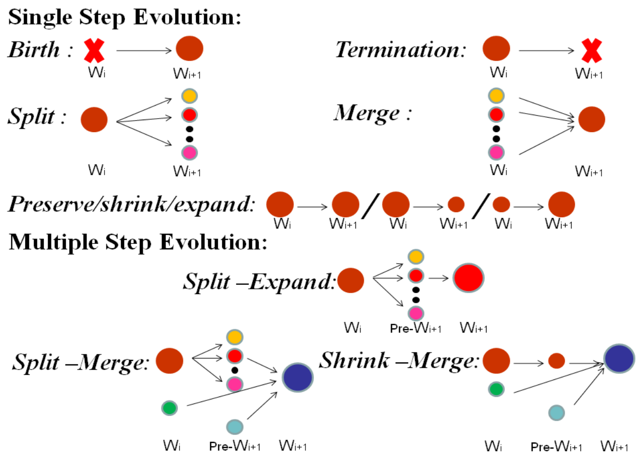


Figure 1: Depiction of Evolution Semantics

Model for Multi-Step Evolution. We now consider change types caused by the combination of both inserting and deleting data points from W_n. For example, given two clusters C_i and C_j both in W_n, two subparts of C_i, say C_{i_sub1} and C_{i_sub2}, and two subparts of cluster C_j, say C_{j_sub1} and C_{j_sub2}, C_{i_sub1} and C_{j_sub1} may belong to a new cluster C_k in W_{n+1}, while C_{i_sub2} and C_{j_sub2} may belong to another new cluster C_l in W_{n+1}.

To cover such more complex cases, we define the notion of multi-step evolution as a composition of two single-step evolutions to express cluster changes caused by first deleting all expired tuples and then inserting new tuples.

***split-expand**:* In W_{n+1}, new cluster C_i formed by splitting of cluster C_j in W_n has, in addition, gained more members.

***split-merge**:* In W_{n+1}, new cluster C_i formed by splitting of a cluster C_j in W_n merges with other clusters from W_n.

***shrink-merge**:* In W_{n+1}, cluster C_i in W_n shrinks in size and then merges with other clusters from W_n.

Theorem 3.1. *For any density-based cluster set Clu_set identified in a query window W_i, the seven single-step evolution types and the three multiple-step evolution types cover all potential cluster changes caused by window sliding.*

We give an intuitive explanation of Theorem 3.1 below. Each phase of the window sliding process, namely deleting expired and inserting new tuples, can only cause four types of single-step evolution (Lemma 3.1). Among these single-step evolution types, ***termination*** and ***birth*** are non-composable single step evolutions. This is because a cluster already terminated cannot participate in any further evolution, and a cluster just born cannot have any evolution before its birth. Thus nine (3×3) possible combinations remain. Among them, six of these combinations composed by two single-step evolution types regarding cluster size changes only, namely ***preservation***, ***shrinkage*** and ***expansion***, can be collapsed into single-step evolution again. For example, ***shrinkage*** and then ***expansion*** will be classified as ***preservation***, ***shrinkage*** or ***expansion***, depending on the specific size change of the cluster. So, the only three real multi-step evolution types remaining are the ones we listed above. In conclusion, our proposed evolution semantics (demonstrated in Figure 1) cover all possible cluster changes caused by window slides.

4. A NOVEL CLUSTERING STRATEGY SUPPORTING EVOLUTION TRACKING

We now design an algorithm that not only significantly improve the performance of the state-of-the-art techniques in terms of clustering, but more importantly supports tracking of cluster evolution based on our proposed evolution semantics (Section 3).

4.1 Proposed Stream Clustering Algorithm

As both analytically and experimentally shown in [20], detecting clusters from scratch at each window is prohibitively expensive. [20] also demonstrates that Incremental DBSCAN [13], which was designed to handle a single insertion or deletion to density-based clusters in data warehouse, does not scale well to handle large numbers of updates in each stream window. Thus, designing an efficient incremental computation method is critical for cluster detection.

For incremental clustering, one key challenge is to efficiently maintain cluster memberships of cluster member tuples over time. The expiration of data points may cause complex cluster changes, such as splitting, which may require cluster membership relabeling of all the remaining cluster members from scratch. Thus, discounting the effect of expired points from the detected clusters can be computationally very expensive.

To address this problem, we exploit the property of sliding window semantics and apply the general concept of "predicted view" [4, 20] as described below. Since sliding windows tend to partially overlap ($slide < win$) [3], some data points falling into the window W_i will also participate in some of the windows right after W_i. Based on the knowledge about data points in the current window and the slide size, we can pre-determine which subsets of the current data points will participate in certain future windows. Figure 2 shows "predicted views" for four future windows. In particular, 16 data points, namely p_1 to p_{16} (each depicted as circles, and the number in the circle representing its time stamp), fall into the current window W_0 and 14 of them (p_1 to p_{13}) form a single cluster. By analyzing the slide size (equal to 4) and the time stamps on each point, we know that data points p_{13} to p_{16} will also participate in three future windows, namely W_1 to W_3. While data points p_9 to p_{12} and data points p_5 to p_8 can only participate in two and one future windows respectively.

By using this concept, we can avoid the computational effort needed for discounting the effect of expiring data points. The idea is to pre-generate "partial clusters" for future windows based on the data points in the current window that are known to participate in the respective future windows. Using the same example shown in

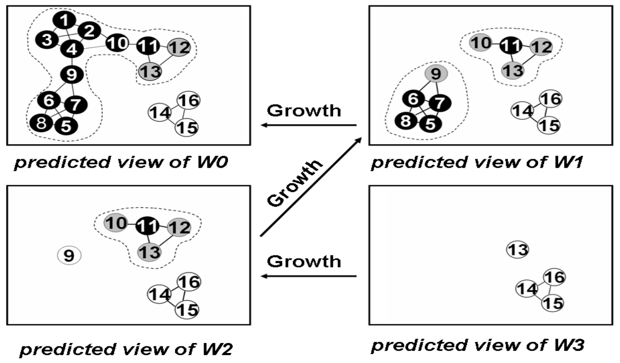


Figure 2: Predicted views of four consecutive windows at time of W_0 (slide size = 4 tuples)

Figure 2, the 12 data points in W_0 that are known to participate in W_1, namely p_5 to p_{16}, will form two clusters in W_1, without considering the new objects that later may come into W_1.

When the window slides, we simply update the pre-generated clusters in the "predicted views" by inserting the new data points into them. No effort for handling expiration of data points is needed, because these views were generated without considering these to-be-expired data points. Note all the cluster changes caused by insertion of additional data points, namely birth of new clusters or expansion or merge of existing clusters, are incrementally maintainable. They are computationally much cheaper than handling expiration-related changes, which may require recomputation from scratch. Figure 3 demonstrates the updated clusters at the time of W_1 after the window slides from W_0 to W_1. The two clusters in W_1 (pre-generated at time of W_0) actually merge into one after the new data points p_{17} to p_{20} arrive.

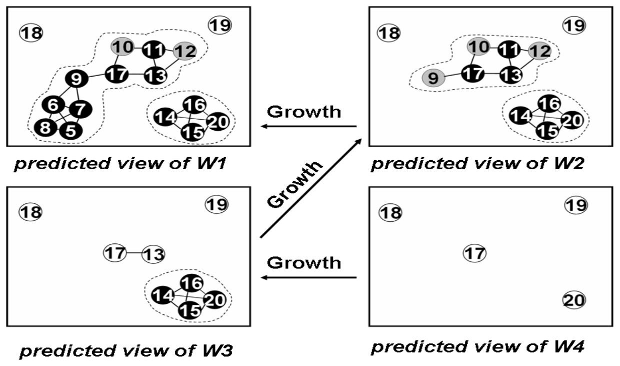


Figure 3: Updated predicted views of four consecutive windows at time of W_1 (slide size = 4 tuples)

Independent View Maintenance. As a straightforward application of the predicted view concept, the "pre-generated" clusters in each future window can be maintained independently. In particular, one can represent and thus update the "pre-generated" clusters in these views independently when the window slides. However, the demands of this approach on both CPU and memory resources increase linearly as the number of predicted views to be maintained increases. More specifically, for a clustering query Q_i, the number of predicted views to be maintained equals $\lceil \frac{Q_i.win}{Q_i.slide} \rceil$. For example, given a query Q_i with $Q_i.win$=10000(s) and $Q_i.slide$=10(s), this approach must maintain 1000 independent "predicted views". Clearly, this represents a heavy burden in terms of both memory and CPU resources. Figure 8 (left part) shows an example of the independent storage for the predicted views depicted in Figure 2. Here, information for representing cluster structures is repeatedly stored even if unchanged across predicted views. More importantly, as the clusters in different windows are treated independently, it does not facilitate cluster evolution tracking, which is based on interrelationships between cluster sets identified in adjacent windows. Extra-N [20], the only algorithm in the literature conducting density-based clustering in sliding windows, suffers from both of these shortcomings.

Integrated View Maintenance. To overcome these shortcomings, we now propose our new clustering algorithm based on an Integrated predicted WINdow maintenance strategy, called IWIN. IWIN exploits the interrelationships among the "pre-generated" cluster structures in a sequence of future windows to compress them into one single compact structure. This integrated storage mechanism enables efficient maintenance of a sequence of predicted views.

We first exploit the notion of **"growth"** between any two density-based cluster sets.

Definition 4.1. *If Clu_Set2 is a "growth" of Clu_Set1, any two data points that belong to the same cluster in Clu_Set1 will also belong to a single cluster in Clu_Set2.*

Figures 4 and 5 demonstrate an example of two cluster sets between which the "growth property" holds. The white circles represent the data points belonging to both cluster sets, while the black ones represent those belonging to Clu_Set2 only. As demonstrated in Figures 4 and 5 also, ***birth*** of "new" clusters, ***expansion*** and ***merge*** are the only three possible cluster changes between any two cluster sets between which the "growth property" holds (see Section 3 for definitions for those change types).

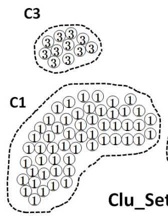


Figure 4: Cluster Set 1 containing 3 clusters

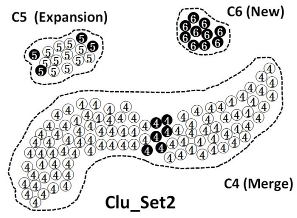


Figure 5: Cluster Set 2 containing 3 clusters

If the growth property transitively holds among a sequence of cluster sets, a **Hierarchical Cluster Structure** can be built to incrementally store the clusters in these cluster sets. The key idea of such a hierarchical cluster membership structure is to incrementally store the cluster "growth information" from one cluster set to another. Figures 6 and 7 respectively give examples of independent and hierarchical cluster membership structures built for the two cluster sets shown in Figures 4 and 5.

As shown in Figure 6, if we store the clusters in these two cluster sets independently, each cluster member (white squares) belonging to both clusters has to store two cluster memberships, one for each cluster set. However, if we store them in the hierarchical cluster structure (Figure 7), we no longer need to repeatedly store the cluster memberships for these "shared" cluster members. Instead, we simply store cluster memberships for each cluster member belonging to Clu_Set1, and then store the cluster growth informa-

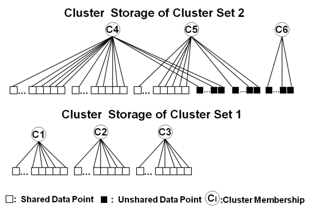

Cluster Storage of Cluster Set 2
C4
C5
C6
Cluster Storage of Cluster Set 1
C1
C2
C3
□: Shared Data Point ■ : Unshared Data Point Ci :Cluster Membership

Figure 6: Independent Cluster Storage for Cluster Sets 1 and 2

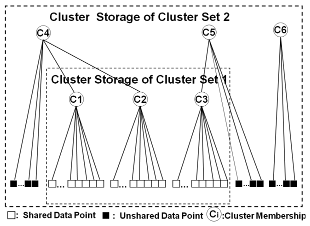

Cluster Storage of Cluster Set 2
C4
C5
C6
Cluster Storage of Cluster Set 1
C1
C2
C3
□: Shared Data Point ■ : Unshared Data Point Ci :Cluster Membership

Figure 7: Hierarchical Cluster Storage for Cluster Sets 1 and 2

tion from Clu_Set1 to Clu_Set2. Such growth information is based on the **granularity of clusters** rather than the granularity of individual tuples. In particular, we correlate each cluster C_i in Clu_Set1 with the "grown clusters" in Clu_Set2 that contain C_i.

Such hierarchical cluster structure does not only saves the memory space, but will also allow us to incrementally tracking cluster evolution. Now we explore the insights from this hierarchical cluster structure to design the IWIN algorithm. We first make the observation that, at any given time T with W_n being the current window, the pre-generated cluster set in a future W_{n+i} is always a **growth** of that identified in the predicted view of W_{n+i+1}. The intuitive explanation for this observation is that a cluster set in an earlier window W_{n+i} can be considered as formed by adding data points to the cluster set in the later window W_{n+i+1} (data points expire in the opposite direction). Put differently, adding data points to a cluster set can only cause "growth" of it.

Given the growth property holds among pre-generated cluster sets in any pair of adjacent future windows, we now design an integrated structure to maintain the pre-generated cluster structures in adjacent windows. We call it the ***Integrated Representation of Predicted Views (IntView)***. For any window W_n, $IntView\_W_n$ is an integrated data structure that represents the predicted views of W_n, W_{n+1} ... W_{n+i}, where W_{n+i} is the last window that any current data point in W_n will participate in before its expiration. The right part of Figure 8 gives an example of $IntView\_W_0$ representing the "predicted views" of W_0, W_1,W_2 and W_3 at the time of W_0 (Figure 2), with the left part depicting independent representation. By using the *IntView* structure, IWIN realizes incremental storage of pre-generated clusters for future windows, reducing the memory utilization.

Since the clusters in different "predicted views" are now correlated in $IntView$ ("grow" from one to the next) , we design an update method that is able to incrementally update the pre-generated clusters in all future windows in one single pass. More specifically, for each new data point, the update process to $IntView\_W_n$ starts from the predicted window with the largest window number (the lowest level in the $IntView$ hierarchy). It then incrementally updates other windows in decreasing order of window numbers (in the higher levels of $IntView$). When updating the clusters in a window W_{n+i}, a new data point p_{new} communicates with its neighbors in this window only. If p_{new} causes a merge of two clusters C_i and C_j in W_{n+i}, and C_i and C_j are subparts of two different clusters C_n and C_m in the earlier window(s), it propagates the "growth information", namely merge C_n and C_m, as well. This assures that all the cluster "growth" in W_{n+i} will be "known" by the upper level windows. Thus, p_{new} never needs to re-communicate with the data points stored in W_{n+i} when updating the upper level windows. The pseudo-code of the IWIN algorithm is listed in Figure 9.

In conclusion, the upper bound of the memory consumption for

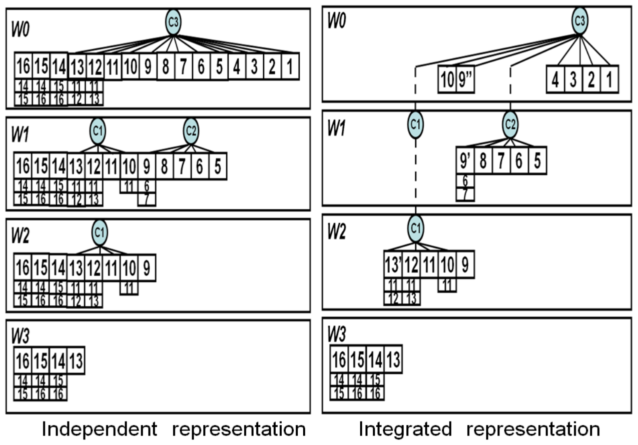


Figure 8: Independent vs. Integrated Representation of Predicted Views

```
$p_i$: a data point.   $p_{new}$: a new data point.
$p_i.T$:$p_i$'s time stamp.
$clu\_mem$:cluster membership.   $W_i$ : predicted window.
$W.T_{end}/_{start}$ : ending/starting time of W.
$PV$ :predicted view. $W_i.PV$ :$PV$ built for $W_i$.
$PV.neighbors$: neighbors distributed to $PV$.
evo optional: optional, for evolution maintenance only
  IWIN ($\theta^{range}, \theta^{cnt}, win, slide$)
  1 For each new data point $p_{new}$
  2    if $p_{new}.T > W_{oldest}.T_{end}$
 (2.1) PregenerateEvolution(Evo,sig); //(evo optional)
  3       Purge the oldest $W_i$
  4    load $p_{new}$ into index
  5    $neighbors$:=RangeQuerySearch($p_{new}, \theta^{range}$)
  6    UpdateIntView $(p_{new}, neighbors)$
  7    if $p_{new}.T == T_{output}$
  8       Output();
  9       add new window $W_{newest}$ to $IntView$
  10      $T_{output} = T_{output} + slide$

  UpdateIntView $(p, neighbors)$
  1 For each predicted window $W_i$ (from $W_{newest}$)
  2    UpdatePredictedView($p, W_i.PV$);
 (2.1)If $W_i = W_{oldest}$ (evo optional)
 (2.2)    UpdateEvolution(p,Evo) (evo optional)

  UpdatePredictView $(p, PV)$
  1 $p.neighborcount = PV.neighbors.size()$;
  2 For i:=1 to $PV.neighbors.size()$
  3       $PV.neighbors[i].neighborcount++$;
  4       if $PV.neighbors[i]$ becomes a new core
  5           HandleNewCore($PV.neighbors[i]$);
  6 if $p.neighborcount \geq Q_i.\theta^{cnt}$
  7       HandleNewCore($p, PV$);

  HandleNewCore$(p, PV)$
  1 $p.type = core$;
  2 $p.clu\_mem$=new $clu\_mem$ ;
  3 For i:=1 to $PV.neighbors.size()$
  4       if $PV.neighbors[i].type == core$
  5           Merge $PV.neighbors[i]$. and $p.\ clu\_mem$;
  6       if $PV.neighbors[i].type == noise$
  7           $PV.neighbors[i].type := edge$;
  8           $PV.neighbors[i].clu\_mem := p.clu\_mem$;
```

Figure 9: The *IWIN* Algorithm

IWIN is now independent of the number of predicted views to be maintained, while that for the state-of-the-art technique [20] increases linearly with the number of predicted views. Computationally, at each window, IWIN only requires a single pass through the new data points, each communicating with its neighbors once, while Extra-N requires each new data point to communicate with each of its neighbors in all windows in which they both will participate.

4.2 Evolution Tracking Procedures

Next we discuss how IWIN incrementally determines the evolution of clusters at each window slide. Our key insight here is that this evolution semantics computation can be piggy-backed with the process of IWIN's cluster structure maintenance. Generally, the computation for evolution semantics is composed of two steps. First, before each window slide, when the view of the current window is still available, information about "predicted evolution" will be extracted based on the relationship between the current view and the predicted view of the next window. Second, after each window slide, when new data points arrive, an evolution update process will be carried out along with the cluster structure update to compute the impact of each new data point on the cluster evolution. Now we further explain the proposed techniques for these two steps.

Step 1: Extract Predicted Evolution Relationships. The key idea here is that using $IntView$ techniques the single-step evolutions caused by purging are already known before the window slides. In particular, since our proposed IWIN algorithm uses the hierarchical structure $IntView$ to incrementally store the clusters identified in the current window and those pre-generated for later windows, the cluster changes caused by the expiration of data points are already captured in our structure.

Using the earlier example in Figure 2, at W_0, the only cluster identified in this window is "predicted" to be split after purging the to-be-expired data points in the next window W_1. This information is already captured in the $IntView$ structure depicted on the right of Figure 8. Namely, this cluster is associated with two pre-generated clusters in W_1. So, before each window slide (purge), we extract the predicted evolution based on the view of the current window and the predicted view for the next window.

Such predicted evolution reflects future cluster changes after purging the to-be-expired data points. Although subsequent arrivals of new data points may *expand* or *merge* these pre-generated clusters or cause *birth* of new clusters, it is guaranteed that the pre-generated clusters will never *terminate*, *shrink* or *split* with the arrival of new data points. In other words, any two data points predicted to be in the same cluster are guaranteed to remain in the same cluster. This is important for applications in which the life span and stability of clusters needs to be analyzed.

Step 2: Update Evolution Relationships. We incrementally maintain the evolution relationships between clusters in adjacent windows at the arrival of each new data point. Since the new data points may only cause growth related changes, this maintenance process of the evolution semantics is fairly straightforward. We simply keep track of which type of cluster change is caused by the insertion of each new data point. Since such effort is anyway needed for cluster computation itself, the only extra effort here is to record the accumulative cluster changes caused by incoming tuples.

In general, this evolution tracking is an incorporated but optional part of our IWIN algorithm. It can be activated during the IWIN clustering process by simply calling the two functions, namely $PregenerateEvolution()$ and $UpdateEvolution(p, Evo)$, shown in Figure 11. Where these two functions are called in IWIN algorithm are shown in Figure 9.

5. MULTI-DIMENSIONAL PATTERN SPACE AND VISUALIZATION

We now describe our proposed pattern space of CLUES framework, which organizes the detected patterns and their evolution in an intuitively accessible and understandable fashion. Our current system supports a two dimensional pattern space with dimensions representing the change over time and across different abstraction levels (Figure 10).

5.1 Past, Present and Future Views

Along the time dimension, our pattern space provides a sequence of views representing the clusters identified in the different portions of the data stream over time. More specifically, each view in the pattern space is a snapshot of the clusters identified in a single window. Those views are organized in the order of their recentness, and thus systematically reflect the *past*, the *present* and the predicted *future* of clusters identified in the stream.

Evo: Evolution Semantics Maintained.
W_{cur}: current window (W_{oldest} on IntView).
W_{next}: the window after W_{cur}.
W_{cur-p}: current window before inserting p.
W_{last}: the (expired) window before W_{cur}.
C_i: a cluster. $W_i.C_a$ a cluster in W_i.
$DistClu(W_i, D)$: distinct clusters in W_i to which data points in dataset D belong to .

```
PregenerateEvolution()
1  For each $W_{cur}.C_i$
2    If $|DistClu(W_{next}, C_i)| == 0$
3      add [$W_{cur}.C_i$ terminates] to Evo;
4    If $|DistClu(W_{next}, C_i)| == 1$
5      $C_a.clu\_mem = C_i.clu\_men$;
                        $(C_a \in DistClu(W_{next}, C_i))$
6      If $|W_{cur}.C_i| - |W_{next}.C_a| > sig$
7        add [$W_{cur}.C_i$ shrinks] to Evo;
8      Else add [$W_{cur}.C_i$ remains] to Evo;
9    If $DistClu(W_{next}, C_i)| > 1$
10     For each $C_a \in DistClu(W_{next}, C_i)$
11       add [$W_{cur}.C_i$ splits into $W_{next}.C_a$] to Evo;

UpdateEvolution($p$,Evo)
1  If $p$ is a new core
2    If $|DistClu(W_{cur-p}, p.neighbors)| == 0$
3      add [$W_{cur}.C_{new}$ birth] to Evo;
4    If $|DistClu(W_{cur-p}, p.neighbors)| == 1$
5      If $|W_{last}.C_i| - |W_{cur}.C_i| \leq sig$ $(p \in C_i)$
6        remove [$W_{cur}.C_i$ shrinks] from Evo if any;
7      If $|W_{cur}.C_i| - |W_{last}.C_i| > sig$
8        add [$W_{cur}.C_i$ expands] to Evo;
9    If $|DistClu(W_{cur-p}, p.neighbors)| > 1$
10     If $DistClu(W_{cur-p}, p.neighbors)$ covers
       all clusters split from $W_{last}.Clu_i$
11       remove all [$W_{last}.Clu_i$ split into ...] from Evo;
12     add [$DistClu(W_{cur-p}, p.neighbors)$ merge
       $Clu_{new}$] into Evo;
13 Else If $p$ belongs to any $C_i$
15   If $|W_{last}.C_i| - |W_{cur}.C_i| \leq sig$
16     remove [$W_{cur}.C_i$ shinks] from Evo if any;
17   If $|W_{cur}.C_i| - |W_{last}.C_i| > sig$
18     add [$W_{cur}.C_i$ expands] to Evo;
```

Figure 11: IWIN Functions for Tracking Evolution

Two important characteristics distinguish our pattern space from simply lining up the clusters identified in previous windows. First, we depict the cluster evolution interrelationships between the clusters from one window to the next. For example, our system can tell the analyst that a cluster identified in the current window has been formed by a "merge" of two clusters in the previous window, or reversely, two clusters identified in the current window are the results of the "splitting" of a previous cluster. Second, besides the views presenting the clusters identified in the previous and current windows, our system depicts a sequence of views for the near future, each presenting the clusters that are predicted to appear in a subsequent future window. As in the past and the present views, these future views are also "linked" by lineage information, indicating that users can track the potential evolution of clusters over multiple views. For example, a future view generated by our system may indicate that a cluster identified in the current window may split into several smaller clusters or totally disappear in the next window. Although future views are predictions and thus may differ from actual clusters identified later, it is guaranteed that the "predicted" clusters will never shrink, split, or disappear (see Section 3).

Generally, by navigating along the time dimension, analysts can 1) monitor the cluster in the current window, 2) review the cluster evolution in the recent past, and 3) study the prediction of cluster evolution in the near future.

5.2 Tuple vs. Pattern Level Cluster Views

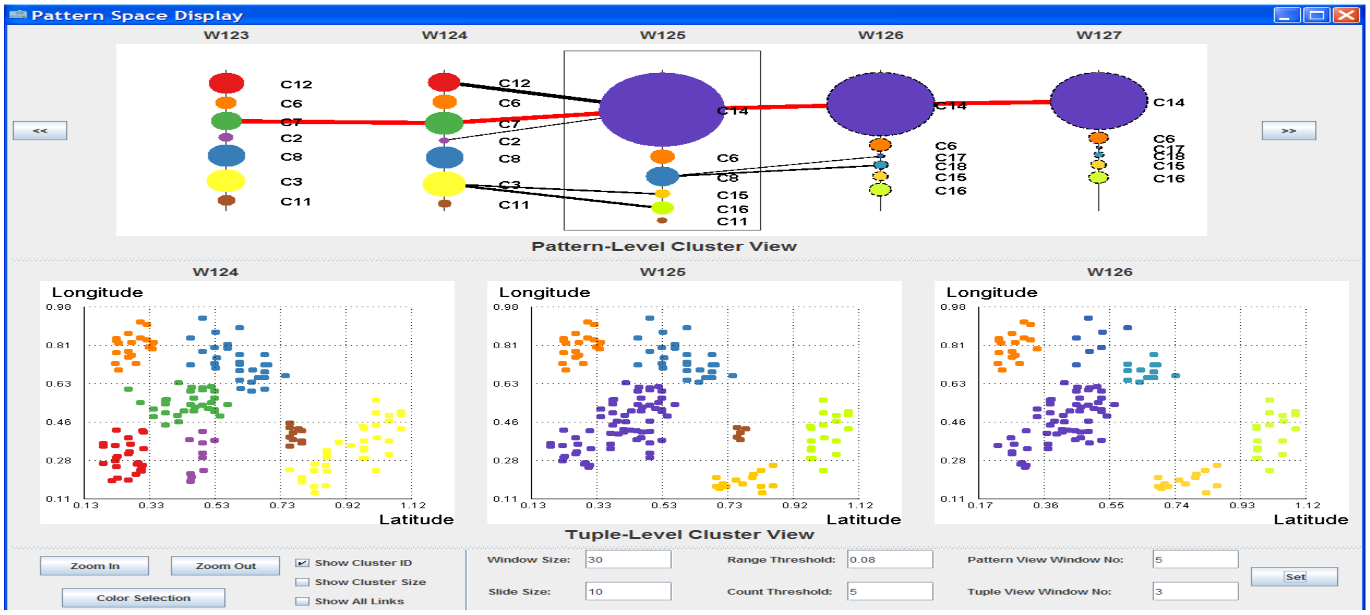
The second dimension in the pattern space represents changes of the abstraction levels, namely tuple vs. pattern level.

In the *pattern level views*, each cluster in a window is abstracted as a single object. Visually, our system presents each cluster using a colored circle. Two important characteristics of a cluster, namely its size (population) and its position, are conveyed by its radius and relative position in the view. More specifically, the radius of each circle is increased as the size of the corresponding cluster increases. We also layout the clusters by mapping their centroids into a single dimension (vertical axis in Figure 10) based on their positions in the data space. Other statistical properties about each cluster, such as density, can be retrieved for any cluster by moving the mouse over it.

The pattern level cluster views also explicitly depict the cluster evolution across windows. First, in each window, a color mapping mechanism is used to preserve the lineage of the identified clusters. For example, if a cluster identified in W_{n+1} is defined to be the "same" cluster as one identified in W_n, we use the same color to present both of them in these two adjacent windows. The specific color mapping is driven by our cluster evolution semantics discussed in Section 3.

Second, we propose to customize the "river metaphor" technique [15], initially designed to visualize frequency changes, to express the evolution of clusters. In particular, the derivation of each cluster from the last window is presented by the "river" (links) between them. For a given cluster C_i in W_{n+1}, if it is considered to be (partially) derived from a cluster C_j in W_n, our system draws a "river" between C_i and C_j. In addition, the volume of each river (width of each link) represents the number of cluster members the cluster in the later window inherits from the previous window. The larger the volume of a river, the more cluster members the clusters in the two windows share.

Tuple level cluster views present specific information about the members of identified clusters. Visually, cluster members are mapped to a two dimensional display. The members of a cluster C_i are all presented in the same color associated with C_i, which is consistent with the color used in the cluster level view. Our system allows analysts to study the characteristics of any individual cluster in depth by zooming into it. For the zoomed-in clusters, we provide a general color mapping mechanism, including hue, saturation, and lightness variations, to express useful information about the cluster



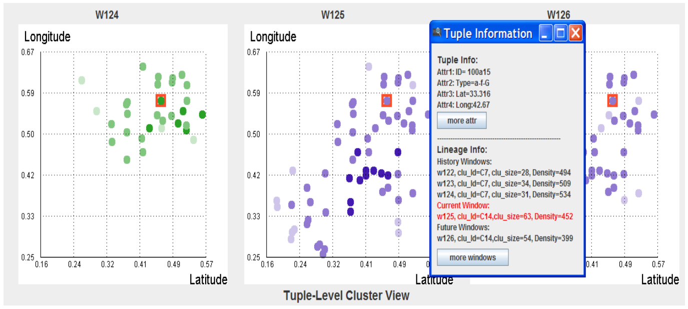
[✓ Show Cluster ID]
Show Cluster Size
Show All Links

Window Size: 30
Slide Size: 10
Range Threshold: 0.08
Count Threshold: 5
Pattern View Window No: 5
Tuple View Window No: 3

Figure 10: A screen shot of visualized pattern space.

members, such as their "freshness" or "object type (core object or not)". Analysts can retrieve detailed information about a tuple, including its lineage across the windows, by a mouse click on it. A screen shot of our system about a zoomed-in view for a cluster at tuple level view is shown in Figure 12.

Generally, by navigating along the abstraction dimension, analysts can 1) examine the summary information for the clusters in each window; and 2) zoom into a specific cluster to retrieve information on individual members.



Tuple Info:
Attr1: ID= 100a15
Attr2: Type=a-f-G
Attr3: Lat=33.316
Attr4: Long:42.67

Lineage Info:
History Windows:
w122, clu_Id=C7, clu_size=28, Density=494
w123, clu_Id=C7, clu_size=34, Density=509
w124, clu_Id=C7, clu_size=31, Density=534
Current Window:
w125, clu_Id=C14,clu_size=63, Density=452
Future Windows:
w126, clu_Id=C14,clu_size=54, Density=399

Figure 12: Zoomed-In View for A Single Cluster.

6. EXPERIMENTAL EVALUATION

We conducted our experiments on a Dell desktop with an Intel Core2 2.2GHz processor and 3GB memory, which runs Windows 7 professional operating system. We implemented the algorithms in VC++ 7.0.

Real Datasets. We used two streaming datasets in our experiments. The first dataset, GMTI (Ground Moving Target Indicator) [12], records the real-time information on moving objects gathered by 24 different ground stations or aircraft over 6 hours. It has around 100,000 records regarding the information on vehicles and helicopters (speed ranging from 0-200 mph) moving in a certain geographic region. The second real dataset we use is the Stock Trading Traces data (STT) from [16], which has one million transaction records throughout the trading hours of a day. In our experiment, we detect clusters based on targets' latitude and longitude in GMTI and transactions' price, volume and time in STT.

6.1 Performance Evaluation

Evaluation of IWIN Algorithm. We experimentally verified the correctness of IWIN by comparing the cluster sets identified by it with those identified by two alternative algorithms from the literature, namely Extra-N [20] and Incremental DBSCAN [13]. In all our test cases the cluster sets identified by IWIN were identical with those identified by Extra-N and Incremental DBSCAN.

We then compared the performance of IWIN (without evolution tracking) with the two alternatives mentioned above. Based on our analysis in Section 4, Extra-N is expected to suffer from scalability issues when the number of predicted views maintained increases. This arises when the ratio between the window size and slide size increases. Also, as reported in [20], Incremental DBSCAN has scalability problems when the number of data points expiring at each window slide is large. Thus, one goal of our experiment was to evaluate the performance of the three competitor algorithms when handling queries with different query windows. In particular, we first used all three algorithms to cluster the GMTI data using a fixed window size 5K, while varying the slide size from 0.25K to 2.5K, implying that 5 to 50 percent of data points will be expired at each window slide in different testing cases. We set θ^{range} equal to 0.02 and θ^{cnt} equal to 5, respectively. To evaluate the performance for each algorithm in term of both CPU and memory utilization, we measured two major performance metrics: 1) the average processing time for each data point, and 2) the memory footprint to store the progressive clusters.

As shown in C1 of Figure 13, the CPU time consumed by IWIN was significantly lower than in both alternatives in all the test cases. The CPU time of all three algorithms increased as the ratio between the slide size and the window size increases. For Extra-N and IWIN, this is because the number of predicted views they need to maintain increases. However, such increase is quite modest for IWIN, while dramatic (almost linear) for Extra-N. This matches our analysis in Section 4, as IWIN used an integrated maintenance strategy for multiple predicted views, while Extra-N maintains them independently. For Incremental DBSCAN, as it always maintains each individual update (a single addition or removal of a data point) independently, such increase in CPU time is mainly caused by more frequent output. However, since such single-update-oriented optimization does not scale to handle large numbers of data points expiring at each window, its performance is significantly worse than that of the other two competitors.

For memory consumption, as shown in C2 of Figure 13, IWIN uses much less memory compared with Extra-N, especially for smaller slide sizes. Incremental DBSCAN, which uses a simple cluster maintenance strategy, consumes even less memory in many test cases. However, because of its significantly worse performance in CPU time, its overall performance was deemed worse than IWIN.

We also have conducted a series of experiments showing the performance of the competitor algorithms when handling data with different characteristics. As discussed in [20], one of the most important factors that affects the performance of density-based clustering algorithms is the Average Number of Neighbors (ANN) each data point has in each window. ANN is indicated as a percentage of the window size. So, we utilize a query with the same parameter settings as the previous experiment but a fixed slide size equal to 5K. We use a synthetic cluster generator as described in [20] to control ANN to vary from 1% to 20%. Experimental results in C3 and C4 of Figure 13 show that the superiority of IWIN is not affected by changes to ANN of the input data.

In conclusion, IWIN is on average 3 times faster in all our test cases than Extra-N, the best state-of-the-art technique, while using 60% less memory space. Put differently, we have a win-win selection in terms of gains in both CPU and memory resources.

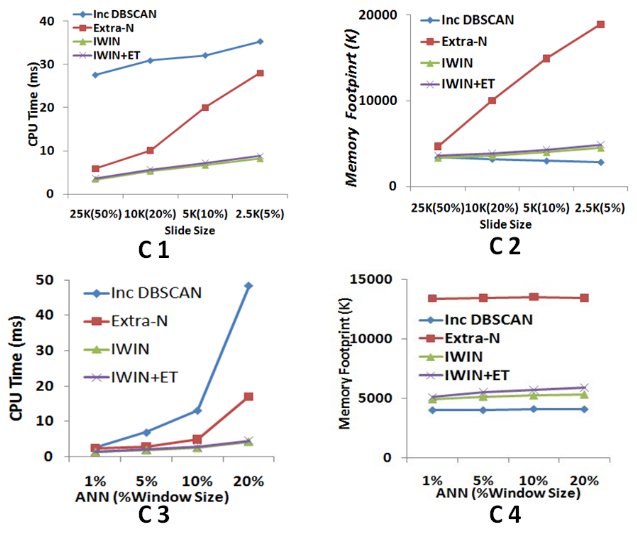


Figure 13: CPU (C1) and memory (C2) comparison when handling queries with different slide sizes. CPU (C3) and memory (C4) comparison when handling streams with different ANN

Evaluation for Evolution Semantics Computation. In the previous test cases, we also evaluate the performance of our IWIN algorithm when evolution tracking is activated ("IWIN+ET"). As shown in all charts in Figure 13, the overhead caused by the computation of the evolution semantics was modest, if not negligible, compared to the maintenance of clusters. In particular, both the CPU and memory utilization caused by semantics computation were always less than 10% of those used for cluster maintenance in all our test cases. On average, only 6% overhead on CPU and 4% overhead on memory are caused by computing evolution in all our test cases. This is expected, because in our proposed method, this evolution computation process is naturally embedded and thus elegantly piggy-bagged by the clustering process.

6.2 User Study for Evolution Tracking

To show that our system effectively helps analysts to track cluster evolution in streams, we conducted a user study based on the STT data. We invited fifteen users (Users 1 - 15), all students from a college, to monitor the transaction pattern changes in stock trades using our proposed system and state-of-the-art techniques. In particular, the users were asked to identify the evolution of intensive-transaction areas (clusters formed based on transaction price, volume and time) with the support of three different exploration systems. **1)** The traditional console view (state-of-the-art technique). **2)** The tuple-level view of our system. **3)** Our full-fledged system with both tuple- and pattern-level views. All three systems used the same cluster detection and evolution tracking logic based on our semantics. The console view showed the statistics of clusters in each window and individual tuples by text. It conveyed cluster evolution using a consistent cluster identification mechanism, namely the clusters detected in different windows were given the same cluster identification, if they were considered to be the same cluster by our evolution semantics. The other two systems displayed the clusters and their evolution as we described in Section 5.

We randomly picked twelve time periods in the STT stream, each covering four successive windows. The monitoring tasks for each user were to identify cluster changes from one window to the next in each of these twelve time periods. We call them Tasks 1 to 12, and put them into three task groups (A, B and C), each covering four tasks. For Task Group A, Users 1 - 5 used the console view, while Users 6 - 10 and Users 11 -15 used the tuple-level view and the full-fledged system respectively. For Task Group B, Users 1 - 5 used the tuple-level view, while Users 6 - 10 and Users 11 - 15 used the full-fledged system and the console view respectively. For Task Group C, Users 1 - 5 used our full-fledged system, while Users 6 - 10 and Users 11 - 15 used the console view and the tuple-level view respectively. The specific task assignment is also shown in Table 1. In this task assignment, no "pre-knowledge" problem will rise, as each of the users was only asked to finish particular task once.

	Console View	Tuple-Level View	Full-Fledged System
Users 1- 5	Task Group A (task 1-4)	Task Group B (task 5-8)	Task Group C (task 9-12)
Users 6-10	Task Group C (task 9-12)	Task Group A (task 1-4)	Task Group B (task 5-8)
Users 11-15	Task Group B (task 5-8)	Task Group C (task 9-12)	Task Group A (task 1-4)

Table 1: Task Assignment for User Study.

We measured both the efficiency and accuracy of the users finishing the monitoring tasks. We asked each user to report all pattern changes they observed in each task. For efficiency, we measured the time consumed by each user for each task. For accuracy, we measured the "correctness rate" of each user's report, namely the number of correct pattern changes reported by the user divided by the total number of pattern changes within each task (acquired by pre-analysis of the stream). Figures 14 and 15 depicted the average time consumed and the average accuracy of users for finishing each task within three task groups when supported by different exploration techniques.

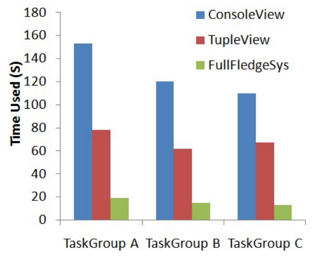


Figure 14: Comparison of Users' Efficiency

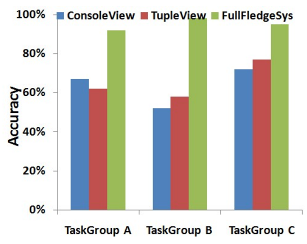


Figure 15: Compassion of Users' Accuracy

As shown in Figure 14, for the same tasks, the users used much less time when supported by our full-fledged system (10-20 sec for each task), while they needed a significantly longer time using the tuple-level view (60-80 sec) and the console view (110-160 sec). This is because both the console and tuple view do not provide any intuitive mechanism for users to identify the pattern changes. Users needed to learn about the relationships among clusters by analyzing the specific cluster members shown in either the text or the tuple level visualization. This was clearly time-consuming. While our full-fledged system provides an intuitive pattern-level view that allows analysts to quickly observe the cluster evolution.

Accuracy-wise, the performance of users was similar when using the console and tuple views ($52 - 70\%$). By analyzing the specific evolution types identified by the users, we observed that although the users could easily identify single-cluster evolution (preservation, expansion and shrinkage) using these two methods, they had difficulties in identifying the evolutions involving multiple clusters, such as splits and merges. In particular, by using the console or tuple view, only 25 and 32 $\%$ of such evolution types respectively were correctly identified by the users. This is because although these two methods are able to convey the size changes of single clusters, they lack effective mechanisms to express complex relationships among multiple clusters. In contrast, our full-fledged system intuitively conveys all evolution types. Thus, users were able to correctly identify almost all the cluster evolutions (92% on average) using our full-fledged system. In conclusion, our proposed solution is shown to help human analysts to track cluster evolution in the streams in an accurate and efficient manner.

7. CONCLUSION AND FUTURE WORK

In this work, we present a unified framework CLUES supporting interactive exploration of density-based clusters in streaming windows. As a foundation, we designed the first comprehensive model for classifying the evolution of density-based clusters in sliding windows. At the back-end, a new density-based clustering algorithm over sliding windows is introduced to efficiently compute not only clusters in isolated windows but also their evolution across windows. At the front-end, new visualization and interaction techniques were developed to effectively convey the detected clusters to the analysts. Our performance studies and user studies confirm the efficiency and effectiveness of our system for tracking clusters in data streams, respectively.

As future work, based on the platform built in this work, we will extend our study to more evolution models for clusters and also other pattern types, including allowing users to customize the models based on their own applications. At the visualization end, domain experts will be invited to help us design more customized visualization and interaction technics, which will provide functionalities specifically needed by some critical application domains.

8. REFERENCES

[1] C. C. Aggarwal. A framework for change diagnosis of data streams. In *SIGMOD*, pages 575–586, 2003.

[2] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *VLDB*, pages 81–92, 2003.

[3] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[4] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

[5] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining variance and k-medians over data stream windows. In *PODS*, pages 234–243, 2003.

[6] J. Beringer and E. Hüllermeier. Online clustering of parallel data streams. *Data Knowl. Eng.*, 58(2):180–204, 2006.

[7] A. Bifet and R. Gavaldà. Mining adaptively frequent closed unlabeled rooted trees in data streams. In *KDD*, pages 34–42, 2008.

[8] F. Cao, M. Ester, W. Qian, and A. Zhou. Density-based clustering over an evolving data stream with noise. In *SDM*, 2006.

[9] Y. Chen and L. Tu. Density-based clustering for real-time stream data. In *KDD*, pages 133–142, 2007.

[10] B.-R. Dai, J.-W. Huang, M.-Y. Yeh, and M.-S. Chen. Adaptive clustering for multiple evolving streams. *IEEE Trans. Knowl. Data Eng.*, 2006.

[11] Denny, G. J. Williams, and P. Christen. Redsom: Relative density visualization of temporal changes in cluster structures using self-organizing maps. In *ICDM*, pages 173–182, 2008.

[12] J. N. Entzminger, C. A. Fowler, and W. J. Kenneally. Jointstars and gmti: Past, present and future. *IEEE Trans on Aero and Elec Sys*, 35(2):748–762, april 1999.

[13] M. Ester, H. Kriegel, J. Sander, M. Wimmer, and X. Xu. Incremental clustering for mining in a data warehousing environment. In *VLDB*, 1998.

[14] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[15] S. Havre, E. Hetzler, P. Whitney, and L. Nowell. ThemeRiver: Visualizing thematic changes in large document collections. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):9–20, January 2002.

[16] I. INETATS. Stock trade traces. http://www.inetats.com/.

[17] M. Spiliopoulou, I. Ntoutsi, Y. Theodoridis, and R. Schult. Monic: modeling and monitoring cluster transitions. In *KDD*, pages 706–711, 2006.

[18] A. K. H. Tung, X. Xu, and B. C. Ooi. Curler: Finding and visualizing nonlinear correlated clusters. In *SIGMOD*, pages 467–478, 2005.

[19] M. Wattenberg. Baby names, visualization, and social data analysis. *Proc. IEEE Symp. Information Visualization*, pages 1–7, 2005.

[20] D. Yang, E. A. Rundensteiner, and M. O. Ward. Neighbor-based pattern detection for windows over streaming data. In *EDBT*, pages 529–540, 2009.