

# Sporadic and Duplicate Cell Removal when Clustering Data Streams

*Nikolay Pavlovich Laptev*  
*nlaptev@ucla.edu*  
*University of California Los Angeles*

## *Abstract*

In recent years, data stream clustering became more popular due to the increased usage of interesting applications, such as credit card fraud detection, that rely on mining data streams. Current data stream clustering approaches are not scalable in terms of data dimensionality and produce poor results when high number of duplicate tuples are present. In this paper we describe the implementation of (1) Sporadic cell removal, and (2) Duplicate detection and removal over sliding windows which we have introduced in [6]. We use a grid-based clustering approach where we employ efficient bit operations to map tuples to cells. We have found that with both sporadic and duplicate cell removal features enabled our clustering algorithms is more efficient than previous state of the art approaches.

## I. INTRODUCTION

In this paper we address the problem of real time clustering of data streams. Real time clustering is a problem because of the size of data and response time requirements. The size of the data stream is potentially infinite, so inferences must be made on partially available data. The algorithm that analyzes data streams must give accurate results in real time because of precision and response time requirements imposed by data stream analysis applications, such as network traffic analysis. Above problems necessitate the development of a clustering algorithm that can deliver real time high accuracy clustering performance.

A growing number of applications, such as network traffic analyses and website click analysis, need to analyze data which flows on high capacity communication streams. Meaningful information is derived from these streams by the use of data mining algorithms. When dealing with data streams, efficient memory handling and real-time response time become major priorities. In the case of network traffic analysis, one needs to process a stream of data and take appropriate action in light of any suspicious behavior. Similarly in website click analyses a study of user's click patterns over time helps derive successful marketing strategies to maximize utility for a particular website. Both of these applications, and countless others, rely on clustering, or deriving patterns, from data streams in real-time.

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. Clustering analysis is part of everyday human activity. As children, we learn to distinguish between dogs and

cats, pens and pencils by subconsciously deriving similarities between these objects and assigning them to separate classes. Similarly, cluster analysis can be used in biology, to derive similarities between plants and animals and thus to categorize them based on their shared features. In business, cluster analysis can help find similarities between groups of shoppers to more precisely target marketing strategies.

Clustering algorithms need to be scalable, be able to deal with different types of attributes, require no previous knowledge of data, be able to deal with noisy data, be insensitive to order of data and be able to deal with high dimensional data. Scalability is necessary when dealing with large amounts of data. Clustering analysis may also be performed on numerical or binary data, which calls for type independent clustering algorithms. Network analysis applications have no knowledge of future attack, and similarly with other applications, therefore no previous knowledge must be assumed in cluster analysis. Because data streams can be *bursty* and unpredictable noise can occur, thus outliers and order of arriving data must not interfere with cluster accuracy. When clustering groups of shoppers, high number of variables can be used, therefore clustering algorithms must scale well with high dimensional data. These requirements make designing a clustering algorithm not a trivial task.

When dealing with evolving data streams, we cannot assume a specific number of clusters. Furthermore our algorithm must be able to deal with arbitrary shaped clusters and be able to handle outliers. Because access to entire data is unavailable, detection of arbitrary clusters can be problematic. Other algorithms, such as Clu-stream use distance metrics, thereby only producing spherical clusters.

Dealing with the evolution of clusters is equally important [4]. When clustering a data-stream, new clusters emerge and old ones fade out. In addition, given noise, the detection of clusters becomes trickier, because we need to be able to distinguish between junk and important data. Note that all of this analysis needs to occur in one pass. Figure 1 shows evolving clusters at time  $t$  and  $t+d$ .

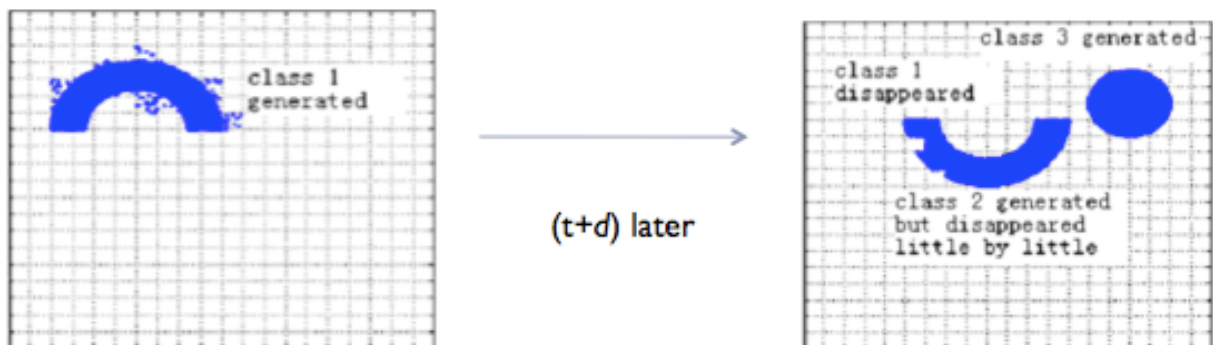


Figure 1: Evolving behavior of clusters, similar to what we implement in this paper.

Previous work treated stream data as static data using finite space. Therefore, outdated and recent data tuples were treated the same. Moving windows were introduced to partially alleviate this problem. Furthermore, previous approaches required additional knowledge about the data, such as in the case of CluStream  $k$  was required, which indicated the number of clusters in the stream.

Newer algorithms rely on density information to mine clusters. In order to more efficiently mine density information, the space is divided into grids or cells, and *dense* cells are termed to be part of a cluster.

In this paper we build upon our previous algorithm developed in [6] and show the implementation of (1) Sporadic cell removal and (2) Duplicate deletion in more detail. We also briefly show the implementation of bit operations used for mapping tuples into cells.

We demonstrate extensive experimental data which shows the clear benefits of the above mentioned features. Furthermore we show that with addition of these features, our grid based algorithm in [6] becomes more efficient than previous state of the art algorithms.

The next section of this paper presents previous work. In section III we present the implementation of our work. After, we conclude and present ideas for future work.

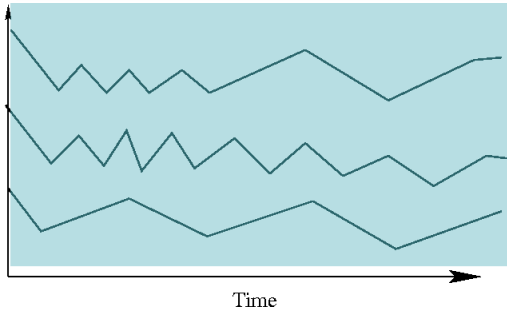
## II. BACKGROUND

Cluster analysis came from the necessity to analyze data streams. As mentioned before, data streams unlike static data, cannot be stored and analyzed locally due to space requirements. As Figure 2 shows, in data streams we have to make inferences when only partial data available.

Below are methods adopted in literature to solve the problem of clustering. Several approaches to the problem combine two methods together, which makes categorizing them difficult. In general, however, the following are the methods, along with examples of each, for solving clustering problem.

## Time series Analysis

Inference using the complete information



## Data Streams

Inference using only partial information

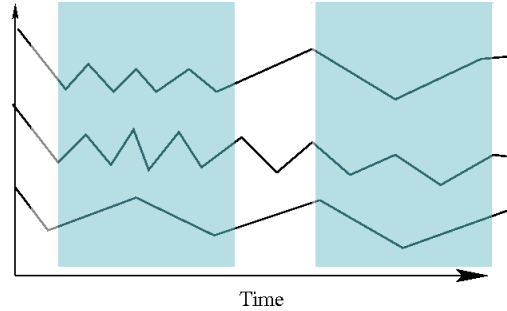


Figure 2: When clustering data streams, only access to partial information is available.

### Clustering Methods - Partitioning Methods

Given  $N$  objects and input  $k$ , partitioning methods organize these  $N$  objects into  $k$  groups based on a similarity metric. Two constraints must be maintained when dividing the data-space into  $k$  partitions, (1) each group must contain at least one object, and (2) each object must belong to exactly 1 group. Initially  $k$  objects are picked at random, creating  $k$  random partitions. Iteratively, objects are re-partitioned to improve an objective partition criterion, such as dissimilarity function based on distance. This helps objects in the same cluster to be similar in terms of the data attributes. Partitioning methods find spherical clusters easily, however finding arbitrary shaped clusters proves to be a problem. Algorithms belonging to this method type are easy to implement, however the iterative step requires multiple scans of data which is impractical for data streams.

A classic partitioning method algorithm is called  $k$ -means [\cite{clustering}](#).  $k$ -means algorithm first selects  $k$  objects at random, which become centers of clusters. The remaining objects in data space are assigned to one of the closest  $k$  initial clusters. The algorithm then iteratively re-computes the mean of clusters and adjusts clusters by assigning objects to an appropriate group. The algorithm terminates when an objective function converges. Square error criterion is typically used as an objective function. It makes  $k$  clusters as compact as possible and as far away from other clusters as possible. Pseudo code for  $k$ -means is given below.

1. *Select  $k$  points as initial centroids*
2. *Form clusters by assigning all points to the closest centroid*
3. *Recompute the centroid of each cluster until the centroids do not change.*

k-means guarantees only local optimum solutions. The recomputing step also requires several passes over data. For these reasons, k-means is practical for static data, however for data streams, several passes over the data is not feasible. Other disadvantage of k-means include the inability to find clusters of arbitrary shape. k-means also requires previous knowledge about the data, when specifying k, which contradicts the notion of unsupervised learning required for clustering. k-means advantages are the ease of implementation and its speed.

### *Partitioning Methods - k-Medoids*

An improvement is made to k-means by introducing another partitioning method called k-Medoids. k-means is sensitive to outliers. When an outlier is present, the mean value used in k-means is distorted producing inaccurate clustering results. k-medoids's idea is that instead of taking the mean of objects in a cluster as the main reference point, a representative object is picked from a cluster of objects. Other objects are assigned to a cluster with the most similar reference point. Iteratively, the partitioning step minimizes the dissimilarity between objects in a cluster and eventually each representative object becomes the most centrally located object of its cluster.

### *Clustering Methods - Hierarchical methods*

Hierarchical methods decompose a data-set into a tree-like structure. A top-down hierarchical approach, initially considers whole data-set as one cluster. This initial cluster is then split into smaller clusters, until k clusters remain. Similarly, bottom-up approach begins with n clusters. A merge step merges similar clusters until k clusters remain, where  $k \leq n$ . This method is fast and memory efficient. A major draw-back of this algorithm is that a merge step can never be un-done, and therefore wrongly merged (or split) clusters cannot be corrected by undoing the previous step. Before merging or splitting of clusters a thorough analysis can be performed. This alleviates the need for undoing a previous step at the cost of some computation overhead [4].

### *Hierarchical methods - BIRCH*

A popular hierarchical algorithm is called BIRCH [7]. Recall that hierarchical methods suffer from the inability to undo a merge or split of clusters and high overhead due to numerous examinations and evaluations when merging or splitting clusters. BIRCH tries to solve these problems. It begins by partitioning clusters hierarchically, using tree structures. The leaf nodes of a tree represent micro-clusters which BIRCH then merges using other clustering algorithms. Micro-clusters which are dense areas of radius r that can be merged together will be discussed in more detail later. BIRCH overcomes the difficulties of previous hierarchical methods by introducing two concepts, clustering feature and clustering feature tree (CF tree) which are used to summarize cluster representation. These structures help BIRCH achieve dynamic clustering, where

we can undo previous merges/splits which were performed by relying on the summaries stored in these data structures.

### *Clustering Methods - Density methods*

Clustering algorithms which rely on distance metrics results in only finding spherical clusters. More recent approaches employ density based clustering algorithms. The main idea of this approach involves growing clusters, as long as the density in some specified neighborhood of each point belonging to a particular cluster contains at least a minimum number of points. This approach, while able to detect clusters of arbitrary shape, is also useful for identifying the noise in data.

### *Density methods - DBSCAN*

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [8] is a popular density based algorithm for clustering both static data and data streams. In DBSCAN a notion of micro-clusters is used. A micro-cluster framework uses concepts of  $\epsilon$ -neighborhoods of point  $p$ , core object and direct density reachability.  $\epsilon$ -neighborhood of point  $p$  is a neighborhood within a radius  $\epsilon$  of object  $p$ . A core object is an object which contains at least  $MinPts$  number of objects within its  $\epsilon$ -neighborhood. Object  $p$  is said to be directly density reachable from object  $q$ , if  $p$  is within  $\epsilon$ -neighborhood of  $q$  and  $q$  is a core object. DBSCAN works by searching for core points in a database  $D$ . The algorithm collects all directly density reachable objects from those core objects. This may involve merging density reachable clusters (which are clusters that contain a chain of objects which are directly density reachable from each other). The process terminates when no new points are added to any cluster.

DBSCAN can be extended to work on data-streams. To do so, two lists are maintained, PL and OL. PL is a list of potential micro-clusters and OL is a list of outlier-micro-clusters. Because of always changing data, micro-clusters must be created rather than discovered as in the static case. Every tuple read from the stream is mapped into PL or OL. When this mapping fails, the new tuple becomes its own micro-cluster. DBSCAN expects all clusters to have similar densities, which can be a disadvantage, nevertheless DBSCAN is an accurate and fast algorithm for finding clusters of arbitrary shape.

### *Clustering Methods - Grid methods*

Grid based methods divide the space into a fixed number of cells (grids). Because the grid can be multidimensional grid-based methods experience a curse of dimensionality, where as more dimensions are added, the number of cells grows exponentially. Density based approaches are used to find clusters by identifying dense grids which thereby contain objects belonging to a cluster. Grids can be grouped based on a distance metric to form larger clusters.

### *Grid methods - D-Stream*

A recently proposed grid-based method is called D-Stream [1]. D-stream partitions its space into grids. Every 'gap' amount of time it adjusts the densities of the grids and explores new clusters. The 'gap' amount of time is determined by considering the minimum time necessary for a dense cell to degenerate and become a sparse cell and the minimum time necessary for a sparse cell to receive sufficient amount of tuples in order to be considered a dense cell. The gap value is set to the minimum of these two values. D-Stream considers a group of grids to be a cluster when these grids are more dense than the grids outside of that group. As with every grid-based method, high dimensionality results in an exponential increase in the number of cells. D-Stream algorithm is able to find and remove sporadic grids, thereby significantly reducing the number of cells. In addition, D-Stream is able to detect clusters of arbitrary shape. Furthermore it does not require (k), or the expected number of clusters as input. In D-Stream the number of grids is fixed; this can be a problem when dealing with data sets having low variance, which would result in a grid becoming overly dense. When a grid is too dense, or contains a large number of points, clustering accuracy is degraded and a finer granularity is necessary to produce more detailed clusters.

### *Clustering Methods - Model methods and EM*

Model based methods rely on a model of the data to find clusters. A model based clustering method may use a density distribution model of the data to detect not only the noise but also the number of clusters present in the data-set by relying on statistical approaches.

EM (Expectation-Maximization) algorithm is one of the model-based clustering methods. Model-based methods try to fit data to some underlying mathematical model. In practice, it is also true that every cluster can be represented mathematically by a parametric probability distribution. Entire data can be viewed as a mixture of these distributions. Therefore, a finite mixture density model of k distributions (each distribution represents a cluster) can be used to cluster our data. A model based clustering algorithm must compute the parameters of the probability distributions so as to best fit the data. EM uses an iterative refinement algorithm to find these parameter estimates. First it starts with an initial guess for the parameters of the model. Then it rescores the objects, based on the mixture density produced by the parameter vector. The rescored objects are then used to update parameter estimates until the algorithm converges.

### *Choice of method*

The choice of the clustering method depends on the type of data to be clustered as well as on the purpose of clustering application. Previous work described above gives strong reasons to use grid-based density clustering technique for clustering data streams. This method provides the ability to find clusters of arbitrary shape and can be implemented efficiently to scale well with high dimensional data. We use this approach in [6] to

implement GStream. GStream eliminates drawbacks seen in previous methods. Unlike k-means and k-medians GStream is able to find clusters of arbitrary shape. By implementing auto-partitioning, GStream is able to adopt to varying density in data streams which is a problem seen in both D-Stream and DBScan. By operating on cells, GStream can efficiently merge and split any cluster of cells at any time eliminating problems experienced by hierarchical methods. GStream also implements sporadic cell removal, thus improving high dimension clustering scalability.

### III. IMPLEMENTATION

As stated before, in this paper we show our implementation of sporadic and duplicate cell removal mechanisms. The implementation of sporadic cell removal consists of mainly an efficient linked list hash table implementation. The duplicate removal implementation is simple and only consists of the binary search of tuples in a cell of interest. We also show code for efficiently inserting a tuple into an appropriate cell using bit operations.

#### *Sporadic Cell Implementation - Hash Table*

Sporadic cell removal is a method for removing cells, which contain none or very few tuples, from processing. Below is a simple pseudo code for managing the list of active cells:

```
while merging a slide
    check if any new cell became dense
        add it to the active grid list
end while

while expiring a slide
    check if any existing dense cell became sparse
        remove it from the active grid list
end while
```

As the above pseudo code shows, we update the list of active grids when merging and expiring slides. Below we show the implementation of the hash table structure which holds active grids.

```

structure
  struct item_el {
    int val;
    struct item_el* next;
  };

typedef struct hash {
  struct item_el* curr;
  struct item_el* head;
}hash;

typedef struct item_el item;

```

*Figure 3: A simple Hash Table of linked lists structure*

Figure 3 presents a simple hash table structure. Each entry in the hash table contains a pointer to a linked list which contains a list of active grids. Our clustering algorithm only processes these active grids, and therefore no extra resources are wasted for empty or sporadic grids.

Figure 4 presents a detailed implementation of both the hash table and its methods. (a) is a simple declaration of a hash table. (b) is a method for inserting a cell number into a linked list inside of a hash table. Provided a cell number, (c) deletes that grid integer from our active list. Finally the *search* method returns true if a grid is in the active list and a *print* method, outputs all of the cells which are dense.

```

hash* activeGrids;      (a)
void insert(int c);     (b)
int delete(int c);     (c)
void print(int c);     (d)
int search(int c);     (e)

void insert(int c) {
  activeGrids[c%ACTIVE_SIZE].curr = (item*)malloc(sizeof(item));
  activeGrids[c%ACTIVE_SIZE].curr->val = c;
  activeGrids[c%ACTIVE_SIZE].curr->next = activeGrids[c%ACTIVE_SIZE].head;
  activeGrids[c%ACTIVE_SIZE].head = activeGrids[c%ACTIVE_SIZE].curr;
}

int delete(int c) {
  if (activeGrids[c%ACTIVE_SIZE].head->val == c) {
    activeGrids[c%ACTIVE_SIZE].head = activeGrids[c%ACTIVE_SIZE].head->next;
    return 1;
  }
  activeGrids[c%ACTIVE_SIZE].curr = activeGrids[c%ACTIVE_SIZE].head;
  while (activeGrids[c%ACTIVE_SIZE].curr->next){
    if (activeGrids[c%ACTIVE_SIZE].curr->next->val == c) {
      activeGrids[c%ACTIVE_SIZE].curr->next = activeGrids[c%
ACTIVE_SIZE].curr->next->next;
      return 1;
    }
  }
  // prev = cur;

```

```

    // cur = cur->next;
    activeGrids[c%ACTIVE_SIZE].curr = activeGrids[c%ACTIVE_SIZE].curr->next;
}
}

int search(int c) {
item* head = activeGrids[c%ACTIVE_SIZE].head;
while (head){
    if (head->val == c){
        return 1;
    } head = head->next;
}
return 0;
}

void print(int index) {
item* head = activeGrids[index].head;

while (head){
    printf("%i ", head->val);
    head = head->next;
}
printf("\n");
}

void print_all() {
int i;

for (i=0; i<ACTIVE_SIZE; i++) {

item* head = activeGrids[i].head;

while (head){
    printf("%i ", head->val);
    head = head->next;
}

printf("\n");
}
}

```

*Figure 4: A detailed implementation of hash table functions*

If our data stream contains many duplicate points, our clustering results will be poor because majority of cells which are sparse, will be considered to be dense. Pseudo code below presents the duplicate cell removal procedure.

#### *Duplicate cell removal implementation*

```

while merging a slide
    check if a new point to be merged is already in a cell
    if so ignore the point
end while

```

Figure 5 below presents pseudo code for checking if a particular point already exists in a cell. Note that only a tuple *struct* is passed into the method. This is sufficient because it contains all of the necessary information such as the tuple's cell number, its dimensions etc.

```

void add_t_to_plus_summary(tuple t) {
    int i;
    int j;
    //plus_summary. we insert the tuple into the grid when we process
the plus summary window
    // if this if statement catches fire, then we must be waiting for a
SLIDE_CHAR or SLIDE_TIME.
    if (DISABLE_DUPLICATES)
        duplicate=point_exists(t);
    if (!duplicate) {

// returns true if the point which we are trying to insert is a duplicate
int point_exists(tuple t) {
    int i,j,cell_count,correct;
    cell_count = grid.cells[t.cell_id].tuple_count;
    for (i=0;i<cell_count;i++) {
        correct = 0;
        for (j=0;j< d;j++) {
            if(grid.cells[t.cell_id].tuple_list[i].coords[j] ==
t.coords[j])
                correct++;
            if (correct == d) {
                return 1;
            }
        }
    }
    return 0;
}
}

```

*Figure 5: Duplicate removal code*

### *Tuple mapping implementation*

Lastly, we show the bit operation procedure for finding where a particular tuple must be inserted in the grid. The code below first computes where the point belongs in each dimension. Afterwards, the cell number of dimension  $i$  is concatenated with cell number of dimension  $i+1$  using binary shift and *or* operators. The resulting binary string, when interpreted as an unsigned integer gives us the integer of a cell where a tuple must map to.

```

int which_cell(tuple t) {
    int i;
    int which_cell_num = 0;
    int nocilt_d = 1; // This will hold number of cells in previous
    // dimensions.
    // this is to find out where the point maps in each dimension
    for (i=0; i<d;i++){
        cell_num[i] = (t.coords[i]/dimensions[i].partition);
        cell_num[i] %= number_of_cells_in_d[i];
    }
    // and now we find out the cell number
    which_cell_num = cell_num[0];
    // compute final cell number
    for (i=1; i<d;i++){
        which_cell_num = (which_cell_num | (cell_num[i] << (int)(log
(number_of_cells_in_d[i-1])/log(2))));
    }
    return which_cell_num;
}

```

## IV. RESULTS

In this section we measure how well our algorithm performs in terms of quality and speed. In the figures below, *GStream* denotes the name of our clustering algorithm which we implemented in [6]. In order to test the benefits of duplicate removal we use a data-set with arbitrary shaped clusters. This data-set contains duplicate tuples and as Figure 7 shows, clustering this data-set without duplicate removal produces poor results. Figure 8 on the other hand shows clear improvement when duplicate tuple removal is enabled. To test how sporadic cell removal improves performance we use a data-set which contains two clusters with a lot of outliers. Figure 9 shows clustering performance versus number of tuples in the data-set (again, the number of clusters is kept at 2). Figure 10 demonstrates the overall performance impact when none, one or both duplicate and sporadic cell removal features are enabled. We conclude with Figure 11 where scalability in the number of dimensions is demonstrated. All of the testing is done on a Fedora Linux with P4 2GHz, 1GB of RAM.

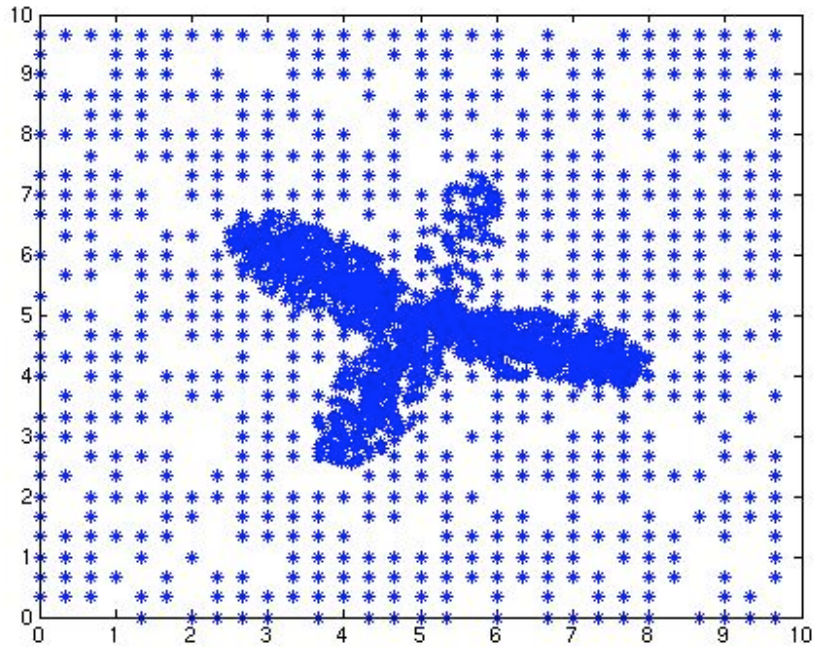


Figure 6: Initial data set with duplicate tuples

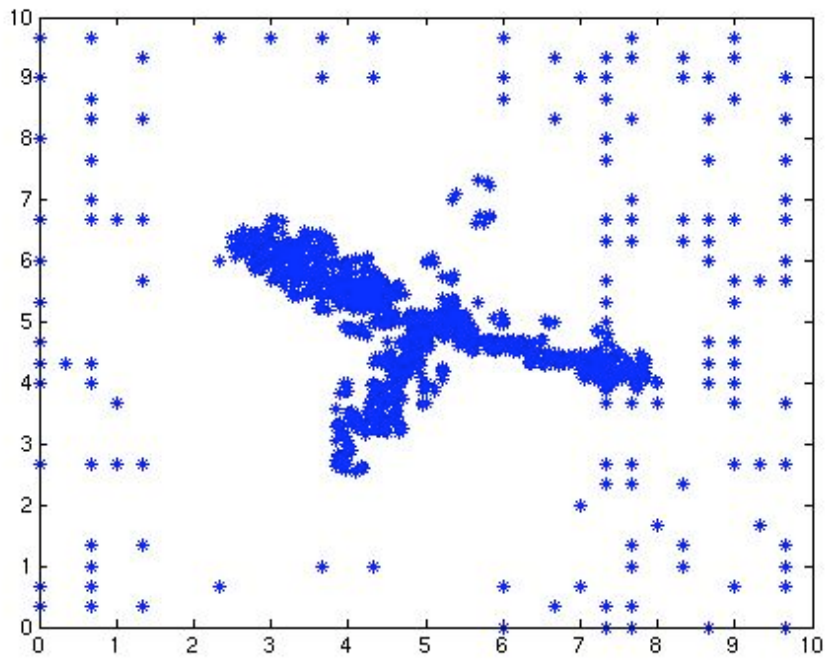


Figure 7: Clustering without duplicate cell removal

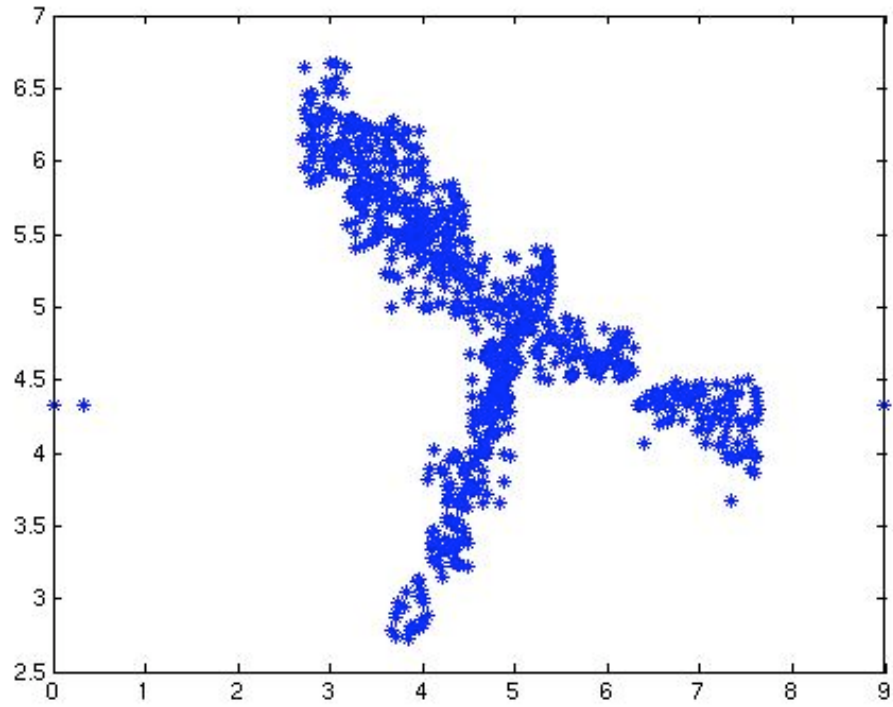


Figure 8: Clusters with duplicate cell removal

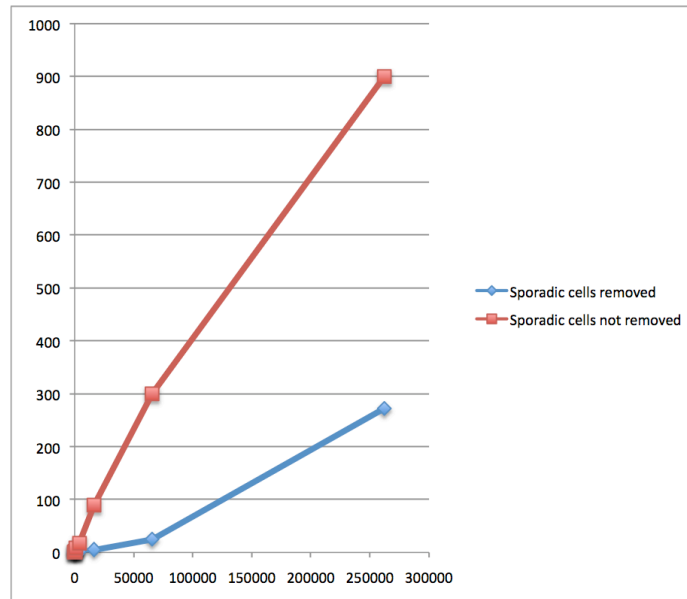


Figure 9: Sporadic cell removal benefits

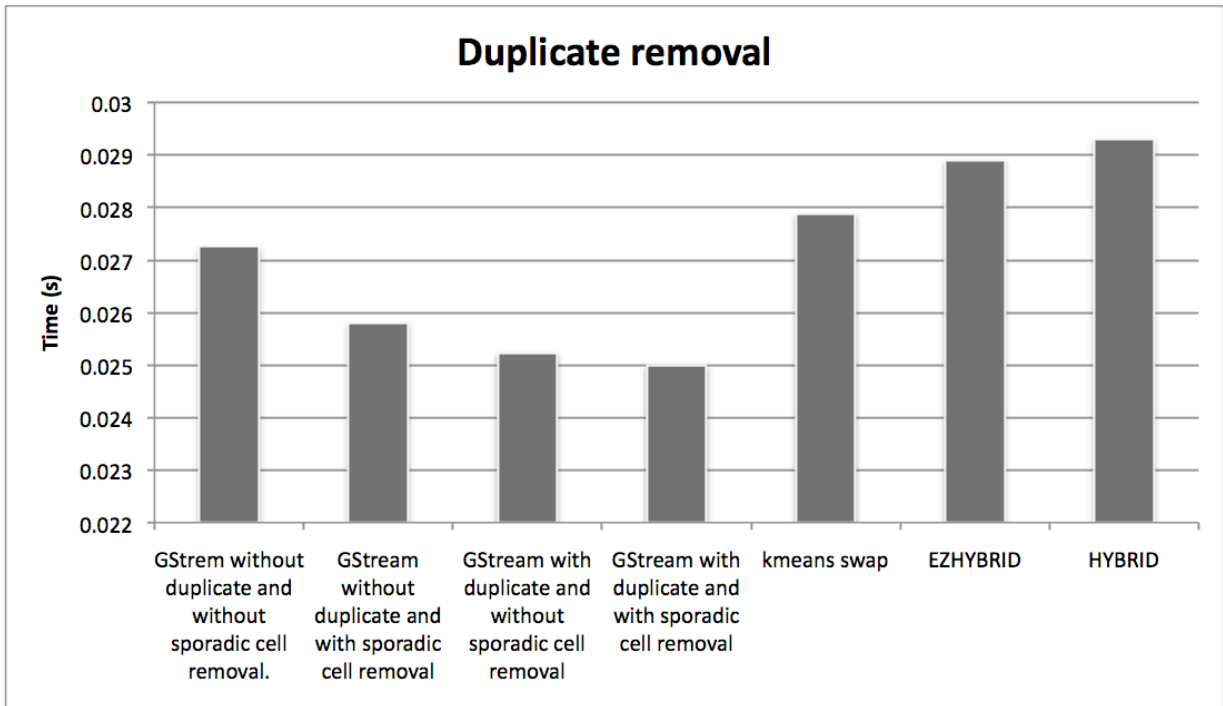


Figure 10: Duplicate and sporadic cell removal benefits

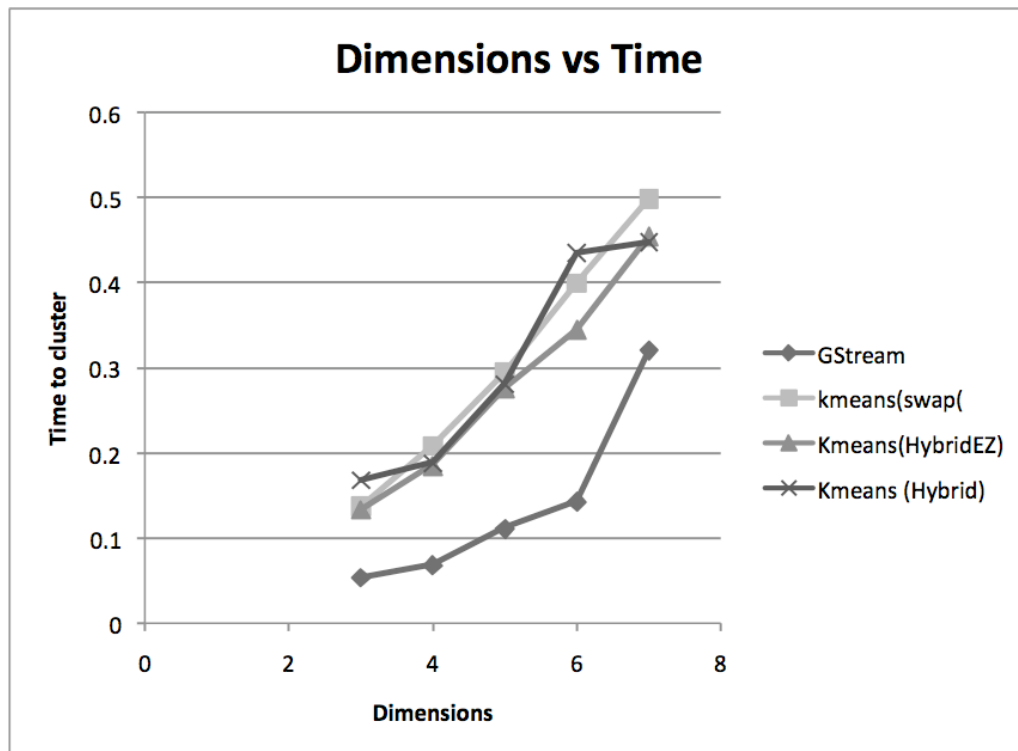


Figure 11: Dimension scalability graph

## V. CONCLUSION

In this paper we have described in detail the implementation of sporadic and duplicate cell removal when clustering data streams. The implementation of these features was also done in [6]. We have used an efficient hash table implementation for sporadic cell removal. A fast search technique for duplicate cell removal was used and an efficient set of bit operations was used for mapping tuples into cells which makes for an overall efficient algorithm.

A tremendous improvement is achieved in both clustering quality and performance when duplicate and sporadic cell removal features are enabled. Outliers and duplicate tuples were removed, thereby improving quality, when enabling duplicate removal. When increasing dimensionality and number of points in our dataset, enabling sporadic cell removal, showed dramatic increase in performance. Overall, when employing sporadic and duplicate cell removal our algorithms is more efficient than previous state of the art algorithms.

In future work we hope to implement visualization mechanism for visualizing the evolution of data streams by ordering micro-clusters in data-streams.

## VI. REFERENCES

- [1] Density Based Clustering for Real-Time Stream Data, Yixin Chen, Li Tu, KDD '07
- [2] Visualizing the Cluster Structure of Data Streams, Dimitris K. Tasoulis et al, IDA 2007
- [3] Streaming Algorithms for High-Quality clustering, Liadan O'Callaghan
- [4] A framework for Clustering Evolving Data Streams, Charu C. Aggrawal
- [5] Dynamic Clustering of Evolving Streams with a Single Pass, Jiong Yang, ICDE '03
- [6] Implementing data-stream clustering with ANNCAD, Nikolay Pavlovich Laptev
- [7] BIRCH: An Efficient Data Clustering Method for Very Large Databases, Tian Zhang, SIGMOD 96.
- [8] A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise, Martin Ester, KDD 96