CDS-Tree: An Effective Index for Clustering Arbitrary Shapes in Data Streams*

Huanliang Sun[1,2], Ge Yu[1],Yubin Bao[1], Faxin Zhao[1], Daling Wang[1]
[1]School of Information Science and Engineering, Northeastern University, Shenyang 110004, China
yuge@mail.neu.edu.cn
[2] Shenyang Jianzhu University, Shenyang 110168, China
sunhl@sjzu.edu.cn

Abstract

Finding clusters of arbitrary shapes in data streams is a challenging work for advanced applications. An effective approach to clustering arbitrary shapes is the clustering algorithm based on space partition. However, it cannot be applied directly into data stream clustering since it costs large memory spaces while data stream processing has strict memory space limitation. In addition, it has low efficiency for high dimensional data and fine granularity. Moreover, its fixed granularity partition isn't suitable for the changes on data distribution of data streams. Therefore, we propose a novel index structure CDS-Tree and design an improved space partition based clustering algorithm, which aims to cluster arbitrary shapes on high dimension streams data with high accuracy. CDS-Tree stores only non-empty cells and keeps the position relationship among cells, so its compact structure costs small memory spaces and gets high efficiency. Moreover, we propose a novel measure for data skew—DSF (Data Skew Factor) to be used to adjust automatically the partition granularity according to the change of data streams, thus the algorithm can gain high analysis accuracy within limited memory. The experimental results on real datasets and synthetic datasets show that this algorithm has higher clustering accuracy, and better scalability with the size of windows and data dimensionality than other typical algorithms applied in trivial style.

1. Introduction

A data stream can be considered as a massive unbounded sequence of data elements continuously generated at a rapid rate [1-2]. Clustering has been widely studied in data mining fields [3-8] because of its popular applications. Clustering is also studied in the data stream field to describe the characterization of data streams [9-12].

Guha et al proposed an early cluster algorithm for data stream—STREAM [9-10], which is the one-pass data-stream clustering algorithm. Babcock et al expended STREAM algorithm to sliding window [11], and solved the *k*–median problem for the most recent *N* points in a stream. Aggarwal et al [12] presented a framework for clustering evolving data streams, named as CluStream. They divided the clustering procedure into an online component and an offline component. The online component maintains detailed summary information by the pyramidal time window, and the offline component analyzes the data streams over different horizons by using summary information that is maintained in online component.

All of these methods are based on *k*–Medians or *k*-means algorithms, which don't emphasize particularly on finding clusters of arbitrary shape in data streams. Our work aims at finding clusters of arbitrary shapes in data streams. The density-based algorithms, such as DBSCAN [7], can find arbitrary shape clusters. However, these algorithms need scan dataset more than one time and have higher complexity, thus it is different to apply these algorithms into data streams.

The major approaches to clustering arbitrary shapes are the clustering algorithms based on space partition, such as CLIQUE [8]. We call them the cell-based algorithms. The basic idea of the cell-based algorithms

* Supported by the National Natural Science Foundation of China under Grant No. 60473073, 60173051, the Teaching and Research Award Program for Outstanding Young Teachers in Higher Education Institutions of the Ministry of Education, China.

is to divide data space into uniform cells (or grids), to count the number of points in them, and to get clusters by connecting neighbor *dense* cells. The traditional cell-based algorithms cannot be applied into data stream clustering in trivial way since it has three major problems as follows.

Firstly, the cell-based algorithms are not suitable for data streams with high dimensionality. CLIQUE finds subspaces clusters for high dimensional datasets by using apriori property to prune non-dense portions. However, it needs a lot of join operations among subspaces, which is not suitable to data streams. For the whole space clustering, if the cell-based algorithm stores all of cells, clustering procedure is easy to be executed since the position relationships among cells are kept, but the number of cells will increase exponentially with the dimensionality. For example, the dimensionality of dataset is 4, and the number of partition intervals in each dimension is 100, then the total number of cells will be 100^4, which results in large memory cost. If only storing non-empty cells, the position relationships among cells are eliminated, thus a neighbor search will traverse the whole list of cells, and clustering algorithm will suffer high complexity.

Secondly, the finer the partition is, the higher clustering accuracy is. When we try our best to partition data space finely for the higher the accuracy of clustering, it also costs high memory.

Lastly, in the cell-based algorithm, only the number of points is stored in a cell, each cell needs the same memory regardless of the number of points in this cell. Thus, memory cost doesn't depend on the number of points in the window but the number of non-empty cells created in the range of sliding window. With fixed partition granularity, the cell-based algorithm cannot get good clustering accuracy with the limited memory for changing data streams. If the partition granularity is too coarse, the accuracy will become low. On the other hand, if the partition granularity is too fine, the memory cost may exceed the limit, which isn't permitted in data stream analysis.

In fact, we find there are many empty cells in space partition that are meaningless for analyzing data, so an index structure named CDS-Tree (Cell Dimension Tree for Streams) is proposed for indexing non-empty cells for the first two problem. CDS-Tree not only stores non-empty cells, but also keeps the position relationship among cells, which makes it very convenient to cluster data streams on CDS-Tree. The clustering algorithm based on CDS-Tree divides the sliding window into some buckets, and keeps the information of buckets on CDS-Tree.

For the last problem, we dynamically adjust the partition granularity based on the current cost of memory space for maximizing the accuracy of clustering. When the granularity changing, we must estimate the memory cost under the new granularity. We find the number of non-empty cells is related to data skew, that is, the skewer datasets are, the larger the number of created cells is. Existing measures for data skew (e.g. variance) cannot be used to estimate the number of non-empty cells. A new measure called DSF is proposed to estimate the number of non-empty cells, and adjust partition granularity automatically for higher clustering accuracy under limited memory.

The experimental results on real datasets and synthetic datasets show that, without memory increment with the arrival of data, this algorithm provides higher clustering accuracy and has better scalability with data dimensionality and size of windows.

2. CDS-Tree

In this section, we first give the definition of clustering based on space partition for data streams. In sub-section 2.2 we discuss CDS-Tree, and sub-section 2.3 gives related algorithms of CDS-Tree.

2.1 Problem Definitions

Suppose $A=\{A_1,A_2,...,A_k\}$ be a set of bounded, totally ordered domains and $S= A_1 \times A_2 \times ... \times A_k$ be a k-dimensional numerical space. We refer to A_1, A_2,..., A_k as the dimensions(attributes) of S. A k-dimension data stream $X=\{x_1, x_2, ..., x_n\}$ is a set of ordered objects at t time point, where $x_i=<x_{i1}, x_{i2},..., x_{ik}>$, and x_{ij}, the *jth* component of x_i, is drawn from domain A_j.

Definition 1. Sliding window model on data stream *X*. Let $[t\text{-}h, t]$ be a time interval in a data stream where t is a time point (In many practical scenarios, t is the current clock time), and h be a time span on X. In a sliding window model, points in $[t\text{-}h, t]$ are divided into u buckets(hops), which are numbered B_1, B_2, …, B_u, starting from the most recent one (B_1) to the oldest one (B_u), and in which there are n points. The window slides by creating a new bucket and discarding the oldest bucket.

Definition 2. Partition *P* of data stream *X*. Let P be a set of non-overlapping rectangular cells, which is obtained by partitioning every dimension of X into equal length. Each cell C is the intersection of one interval from each dimension. It is represented as the form $\{c_1,c_2,...,c_k\}$ where $c_i=[l_i, h_i)$ is a right-open interval in the partition of A_j. A cell can also be denoted as $(cNO_1, cNO_2, ..., cNO_k)$ named the coordinate of the cell, where cNO_i is the interval number of the cell on *i-th* dimension, and it is coded from 1 to the total number of interval. We call P a partition of data stream X.

Definition 3. Sub-partition. Let P'and P be two partitions of dataset X, if $\forall C' \in P'$, $\exists C \in P$, such that $C' \subset C$, i.e. For the interval $[l_{1i}, h_{1i})$ of the cell C', then $\exists [l_{2i}, h_{2i})$ of C, such that $[l_{1i}, h_{1i}) \subset [l_{2i}, h_{2i}), i = 1,...,k$. We call P' a sub-partition of P, and P a super-partition of P'denoted as $P' \subset P$.

Definition 4. Selectivity p_c of cell C. Let $x=<x_1, x_2, \dots, x_k>$ be a point in data stream X, $C=\{c_1,c_2,...,c_k\}$ be a cell in partition P. If $l_i \leqslant x_i < h_i$ for all c_i ,then we call x belongs to C, denoted by $x \in C$. The number of points that belong to C defines the selectivity p_c of cell C.

Definition 5. Clustering based on cells data stream X in a sliding window. Map data points of a data stream X in a sliding window into a partition P. If the selectivity of a cell is larger than a threshold τ, we call the cell *dense*. A cluster is the largest set of cells that are adjacent and *dense*. The adjacency is transitive. That is to say, two cells C_1 and C_2 are connective when they are neighboring, or there exists a cell C_3, C_1 and C_3 are neighboring, C_2 and C_3 are neighboring.

2.2 CDS-Tree Definition

Space partition will produce many empty cells over a data stream, which makes the clustering algorithm inefficient on data stream. We seek an index structure to store the statistical information of cells in data streams. Such data structure should retain the essential information of all non-empty cells, and enable to be merged efficiently with the new continuously arriving points in data streams. In addition, based on this structure, clustering algorithm is easy to be implemented. We propose a new index structure named as CDS-Tree that can meet these requirements.

Definition 6. CDS-Tree. The CDS-Tree of a dataset X under a partition is a balanced tree <*Root-Node*, *Mid-Nodes*, *Leaf-Nodes*, *Edges*>. Suppose dataset X has the dimensions $A_1, A_2, \dots, A_k$, where

1. *Root-Node* is the root-node of the tree and corresponding to the first dimension A_1.
2. *Mid-Nodes* are the set of intermediate nodes that are between the root nodes and the leaf nodes. The *Mid-Nodes* at the *i*-th level corresponds to the dimension A_i, uniquely, and contain all the distinct interval numbers of the dimension A_i corresponding to the non-empty cells based on intervals of the anterior the *i*-th dimension.
3. *Leaf-Nodes* is the set of the leaf-nodes *Leaf-Node*, which is at the k+1 level. A leaf-node has the form <*Coor*, *Total-Num, Num-Point-List*>, where *Coor* is the coordinate of a cell, and its' form is (cNO_1, cNO_2, ..., cNO_k). *Total-Num* is the total number of points falling in this cell, and *Num-Point-List* is a list of the number of points, whose length is equal to the number of buckets in the sliding window, and a value of list represents the number of points in the corresponding bucket that fall in the cell.
4. The *Edges* is a set of the pointers in non-leaf nodes (*Root-Node* and *Mid-Nodes*). The non-leaf nodes have the form <*cNO*, *pointers*>, where *cNO* is a keyword. The keyword of the *i*-th level is a distinct interval number of the *i*-th dimension corresponding to a cell; and *pointers* is a set of pointers, each of which points to the next level node. The *pointer* of the *k*-th level internal-nodes points a leaf node.
5. A path from the root node to a leaf node corresponds to a cell.

Figure 1(a) illustrates a sliding window model of the data stream. There are *u* buckets in window, which are numbered with B_1, B_2, …, B_u. Figure 1(b) shows the cell structure of a 2-dimension (A_1 and A_2) dataset under a partition, and each dimension is divided into 6 intervals, so the interval number of each dimension is from 1 to 6. Suppose the *gray* cells represent the non-empty cells, and they show the distribution of data stream in a window. Note that in a cell only the number of points is recorded, rather than real points. The cell (2,3) denotes the one whose interval number is 2, 3 on the dimension A_1 and A_2 respectively, where '2' represents the 2nd interval of A_1-dimension, and '3' represents the 3rd interval of A_2-dimension. Figure 1(c) is a CDS-Tree structure corresponding to the cell structure, which only stores non-empty cells. The CDS-Tree has 3 levels. The first two levels correspond to A_1-dimension and A_2-dimension of the dataset respectively, and the bottom level is the cell level. The first dimension A_1 has 6 intervals, but only the interval 2, 3, 5 and 6 contain data objects, so the root node has 4 internal-nodes. The first internal-node '2' of the root node points to the second level of the CDS-Tree, there are two intervals, '2' and '3'. A path from the root to a leaf such as (2, 3) corresponds to a cell. To maintain a sliding window on CDS-Tree, a list is stored in each cell, whose length is equal to the number of buckets in the sliding window, and a value of list represents the number of points in the corresponding bucket that fall in the cell. For example, the list 5, 12, … , 0, 8 in cell (2,2) shows 5 points in B_1 fall in the cell(2,2), 12 points in B_2 fall in this cell, and so on.

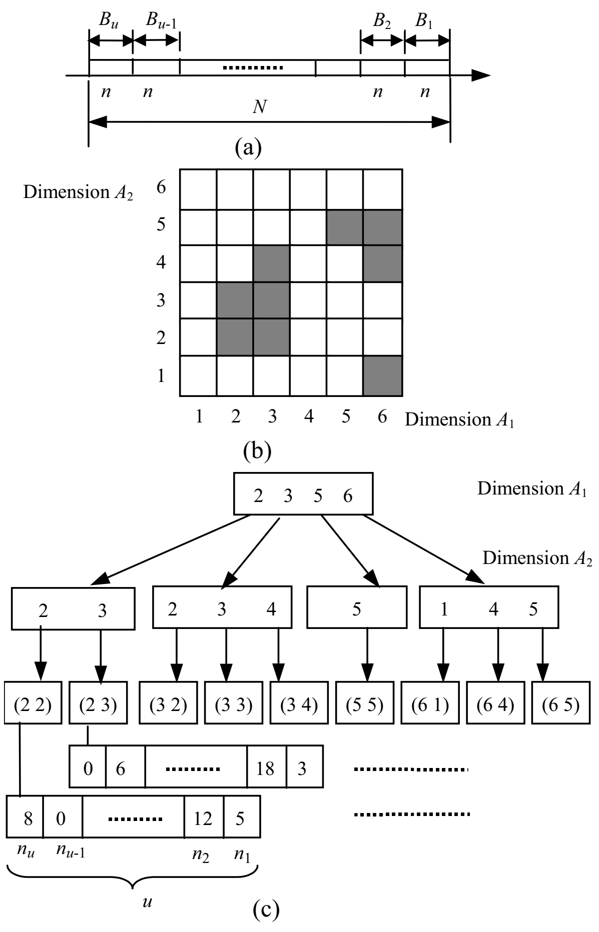

Figure 1. An example of a CDS-Tree

From the above example, we know that CDS-Tree keeps the relationships among cells, which is very important for clustering procedure because it facilities neighbors' searches. For example, to search the neighbors of cell (2,2), we can computer the coordinate of the neighbor cells by position relationships, they are (1,2), (2,1), (3,2), (2,3). On CDS-Tree, we can search the value of coordinate corresponding to dimension level, so that it needs search fewer keys. Existing multi-dimension indexes such as R*-Tree [13] cannot be used to index cells, because they cannot keep position relationships among cells.

2.3 Related Algorithms of CDS-Tree

In this section we give the primary operation algorithms on CDS-Tree, which are used in data streams clustering. They are CDS-Tree building, CDS-Tree merging, and clustering algorithm based on CDS-Tree.

2.3.1. CDS-Tree building algorithm. The algorithm receives the current data object and computes coordinates (i.e. the interval number of each dimension) of them. If the cell corresponding to this object exists, add the number of points in this cell. Otherwise, create a new cell. The algorithm doesn't deal with the discard of buckets, which will be introduced in Algorithm 4. The details are shown in Algorithm 1.

```
Algorithm 1. CreateCDSTree
Input: Stream X
Output: CDS-Tree
1.  Create the root node of CDS-Tree;
2.  For the current object $x^t$ in stream X then
3.     Compute the coordinate ($cNO_1$, $cNO_2$, ..., $cNO_k$) of the cell C corresponding to $x^t$;
4.     For each value of C coordinates $cNO_i$
5.        Search in corresponding level;
6.        If it exists then go 4;
7.        Else insert $cNO_i$ with ascending into corresponding level of CDS-Tree;
8.     If the cell C exists then insert $x^t$ into it, and update Num-Point-List of the cell C;
9.     Else create a new cell as leaf node and insert $x^t$ into it, and create Num-Point-List ;
10. Output the CDS-Tree;
```

2.3.2. Clustering algorithm based on CDS-Tree. Algorithm 2 gives the clustering procedure on data streams, which marks adjacent *dense* cells with the same cluster number. The algorithm executes a width-first search on CDS-Tree. The result of clustering is a CDS-Tree in which all cells are marked by the number of clusters.

```
Algorithm 2. ClusteringDataStream
Input: CDS-Tree, density threshold $\tau$
Output: ClusteringResult
1. For each cell C of CDS-Tree
2.    If C is unmarked and C.Total-Num>=$\tau$ then
3.       Mark C with a new cluster number ClusterNo;
4.    For each neighbor C' of C in CDS-Tree
5.       If C' meets: (1) adjacent with C, (2) are unmarked, and (3) C'. Total-Num>=$\tau$ then
6.       Mark C' with ClusterNo of C and push C' into stack S;
7.    If stack S is not empty then;
8.       Pop stack S as C, go step 4;
```

3 Granularity Adjustment

In this paper, sliding window model is applied to discard stale data. Because only the number of points is stored in a cell, each cell needs the same memory regardless of the number of points in this cell. Thus, memory cost doesn't depend on the number of points in the window but the number of cells in the window. In addition, in the sliding window we cannot get the distribution of data, so it is impossible to estimate the memory cost of N points.

As we know, the finer the partition is, the higher the accuracy is, but the more number of the cells is created. So, in a sliding window, users aim to get the analysis accuracy as high as possible.

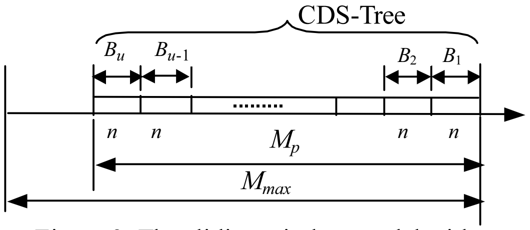

Figure 2. The sliding window model with granularity adjustment

Our basic idea is as follows: Given fixed N points in sliding window and the limited memory M_{max}, if the current cost memory M_p is far less than M_{max}, we can execute finer granularity partition for higher accuracy. On the contrary, if the current memory cost M_p is close to M_{max}, we should use coarser partition to avoid memory overflow. The algorithm adjusts the partition granularity periodically according to the current memory cost M_P to get the largest accuracy. Figure 2 shows the sliding window model with granularity adjustment. Before creating a new bucket, the algorithm will examine if the granularity adjustment is required. If an adjustment occurs, a new CDS-Tree is created with the new granularity. After the new bucket is created, an old bucket B_i becomes B_{i+1}.

Algorithm 3 gives the procedure of granularity adjustment. The algorithm sets two safety factor λ and η in case of exhausting memory. The factor λ is used to avoid the memory required exceeding the limited memory M_{max} when the granularity turns finer, here we set it larger than 1. It is not safe to adjust the granularity coarser when M_p is close to M_{max} in case that the adjustment cannot keep up the speed of data stream and leads to the overflow of memory, therefore, adjustment of granularity should be executed ahead. So we set a factor η to decide the time point to adjust the granularity, where η is less than 1. For example, η is set 0.1, which represents when left memory is less than 10% of M_{max}, the algorithm will turn granularity coarse to save more memory.

Algorithm 3. *PartitionAdjustment*

```
Input: Stream $X$, $M_{max}$, $P$  // $M_{max}$ is the limit of memory; $P$ is the current partition
1.  If create a new bucket then
        Compute the current memory $M$;
2.      If ($M_{max}$ - $M_p$)>= $\lambda$ ($M_p$'- $M_p$) then // $M_p$ is the current memory cost; $M_p$'is memory cost under the sub-partition $P$'.
3.          Return sub-partition $P$' of $P$
4.      If ($M_{max}$- $M_p$)<=$\eta M_{max}$ then
5.          Return super-partition $P_0$ of $P$;
6.      Return $P$. // No adjustment
```

Because the number of cells under coarse granularity is less than that of under fine granularity, it needn't estimate the number of cells when the granularity is adjusted from fine to coarse. But when the granularity is adjusted from coarse to fine, we must estimate the number of cells in order to judge if there is enough memory for fine granularity in the current window. Without the new coming data, we can only estimate it from the current data in window.

When the partition granularity turns fine, to get the value of M_p'- M_p, we should estimate the number of the added cells v_p'- v_p because M_p'- M_p is equal to (v_p'- v_p)m_c, where v_p' and v_p are the number of non-empty cells under partition P and P' respectively, and m_c is the memory cost of a cell. The number of cells in a window is related with the data distribution, and the skewer the data is, the less the number of cells is. The conventional measures (e.g. variance) for data skewness cannot be used to estimate the number of cells. Figure 3 shows that the data with small variance occupy few cells (Figure 3(a)), and the data with big variance occupy many cells (Figure 3(b)). In Figure 3(c) data occupy the most cells, which variance is between that of Figure 3(a) and that of Figure 3(b). Based on this fundamental observation, we consider the measure that can estimate the number of non-empty cells should be based on density. So we propose a new measure named *DSF* (Data Skew Factor) to estimate the skew degree of data, which is based on density.

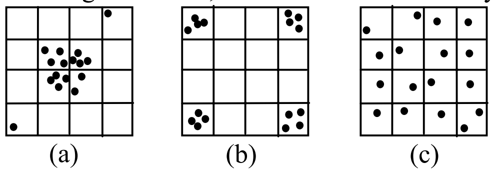

Figure 3. Variances in different data distributions

Definition 7. (*DSF*(*X*,*P*) of dataset *X* under partition *P*) The *DSF* (*X*,*P*) of dataset X under partition P is defined as $\frac{1}{2N}\left(\sum_{i=1}^{w}|p_i-\mu|\right)$ where p_i is the number of points in cell C_i, w is the number of cells and N is the total number of data points. In addition, μ is a constant which is the average of points number in each cell calculated by formula $\mu=N/w$.

DSF(*X, P*) of a dataset is a value in $[0,1)$. *DSF*(*X, P*)*=0* means that there is no skew under partition *P*. The closer to 1 *DSF* (*X,P*) is, the larger the data skewness is. Intuitively, *DSF*(*X,P*) indicates the degree of fullness of the space filled with data.

Theorem 1. *DSF* of a dataset under partition *P* is equal to $\frac{1}{N}\left(\sum_{j=1}^{v}(p_j-\mu)\right)$ where p_j is the number of points in C_j in which the number of points is larger than μ.

Proof: By Definition 7, $DSF(X,P)=\frac{1}{2N}\left(\sum_{i=1}^{w}|p_i-\mu|\right)$. Suppose the number of cells in which point's number is larger than μ be equal to *v*. Then,

$$DSF(X,P)=\frac{1}{2N}\left(\sum_{j=1}^{v}(p_j-\mu)-\sum_{i=1}^{w-v}(p_i-\mu)\right)$$

$$=\frac{1}{2N}\left(\sum_{j=1}^{v}p_j-v\mu-\sum_{i=1}^{w-v}p_i+(m-v)\mu\right)$$. Because

$\sum_{j=1}^{v}p_j+\sum_{i=1}^{w-v}p_i=\sum_{i=1}^{w}p_i=N$ and $w=N$. Thus,

$$\frac{1}{2N}\left(2\sum_{j=1}^{v}p_j-N-2v\mu+N\right)=\frac{1}{N}\left(\sum_{j=1}^{v}(p_j-\mu)\right).$$

By theorem 1, when calculating *DSF* of a dataset, we only need to visit the cells in which the number of points is larger than μ. So it simplifies the calculation of *DSF*.

Theorem 2. Given *P* of dataset *X*, if $w\geqslant N$, then *DSF(X,P)* is equal to the percentage of empty cells where *N* is the number of points, and *w* is the total number of cells.

Proof: By Definition 7, $DSF(X,P)=\frac{1}{2N}\left(\sum_{i=1}^{w}|p_i-\mu|\right)$. Suppose the number of cells in which the point's number is larger than μ be equal to *v, s* be the number of cells which selectivity is less than μ and larger than 0, and *q* be equal to the percentage of empty cells respectively. Thus, $SOD(X,P)=\frac{1}{2N}\left(\sum_{j=1}^{v}(p_j-\mu)-\sum_{i=1}^{s}(p_i-\mu)+\sum_{i=1}^{qw}\mu\right)$.

From Theorem 1, *DSF*(*X,P*)= $\frac{1}{N}\left(\sum_{j=1}^{v}(p_j-\mu)\right)$,then

DSF(*X,P*)= $\frac{1}{2}DSF(X,P)-\frac{1}{2N}\sum_{i=1}^{s}(p_i-\mu)+\frac{1}{2N}qw\mu$.

So *DSF-q*= $\frac{1}{N}\left(\sum_{j=1}^{s}(\mu-p_j)\right)$. If $w\geqslant N$, then $\mu\leqslant 1$. Because the selectivity must be larger than 1 or equal to 0, so *s*=0, i.e. *DSF-q=0*.

Theorem 2 tells us that we can use *DSF* to estimate the number of non-empty cells when the total number of cells under certain partition is larger than the number of points, the condition is easy to meet in a sliding window over data streams.

Theorem 3. Given two partitions *P*' and *P* of dataset *X*, if $P'\subset P$, then *DSF*(*X, P*')$\geqslant$*DSF*(*X,P*).$\subset$

Proof: Let w_1, w_2 be the total number of cells of partition *P*'and *P* respectively, and μ_1, μ_2 be the average points number of them. Because of $P'\subset P$, then $w_1=rw_2$, $\mu_2=r\mu_1$ where *r* is a natural number. By Definition 7: $DSF(X,P)=\frac{1}{2N}\left(\sum_{i=1}^{w_2}|p_i-\mu_2|\right)$,any cell $C_i\in P$ can be divided into *r* cells in *P*'. i.e. $|p_i-\mu_2|=\left|\sum_{j=1}^{r}p_{ij}-r\mu_1\right|\leq\sum_{j=1}^{r}|p_{ij}-\mu_1|$. Then

$$DSF(X,P)\leq\frac{1}{2N}\left(\sum_{i=1}^{w_2}\sum_{j=1}^{r}|p_{ij}-\mu_1|\right)=\frac{1}{2N}\left(\sum_{i=1}^{rw_2}|p_i-\mu_1|\right),$$

i.e. *DSF*(*X, P*')$\geqslant$*DSF*(*X, P*).

Theorem 3 tells that the finer the granularity of a partition is, the larger *DSF* is. We can use *DSF* under coarse granularity to estimate the *DSF* under fine granularity. Theorem 2 shows that the *DSF* is equal to the percentage of empty cells. Suppose two partitions *P*' and *P* of dataset *X*, $P'\subset P$, and v_p' and v_p denote the non-empty cell number under *P*' and *P* respectively. Because the percentage of empty cells is equal to *DSF*, then v_p'=(1-*DSF*(*X,P*'))$\times w_1$ and v_p=(1-*DSF* (*X,P*))$\times w_2$, where w_1 and w_2 are the total number of cells under two partitions. Moreover $P'\subset P$, then *DSF* (*X, P*')$\geqslant$ *DSF* (*X, P*), thus we can estimate v_p' by v_p'$\leqslant$(1-*DSF*(*X, P*))$\times w_1$. M_p'-M_p required in Algorithm 5 is equal to ((1-*DSF*(*X, P*))$\times w_1$-(1-*DSF*(*X,P*)) $\times w_2$)m_c, where m_c is the cost memory of a cell.

4 Experimental Results

This section evaluates the performance of our algorithm. The following experiments are conducted on Microsoft Windows 2000 Server with 512MB main memory and 2.5GHz CPU. There are 3 kinds of datasets, the first one is KDD-CUP-99 Network Intrusion Detection stream dataset. It is collected by MIT Lincoln Labs, and includes five types of objects: normal link and four intrusions. The intrusion-types are further classified into 24 types. The second dataset is Image Fourier Coefficient dataset, which is widely used in multi-index tests (e.g. X-Tree [14]). In addition, in order to reflect the evolution of the stream data over time, we construct the third dataset by changing the distribution of cluster every 100000 points over the second dataset.

We denote different algorithms as follows: *CDS* denotes CDS-Tree-based algorithm; *CB* denotes the cell-based algorithm, which is clustering algorithm of CLIQUE on a whole space instead of on sub-space. To partition the dimension conveniently, the value in a dimension is normalized to [0,1]. Let τ denote the density threshold that is equal to the percentage of the number of points in the cells, and *d* denote the length of partition interval. We conducted such experiments as follows.

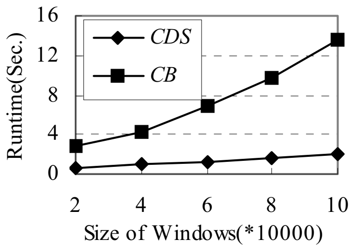

Figure 4. Runtime comparison for different size of windows (d=0.02, τ=0.005)

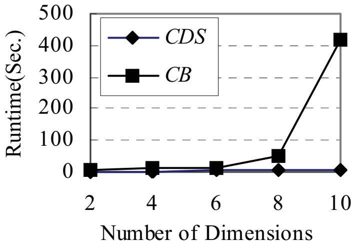

Figure 5. Scalability comparison with data dimensionality (d=0.005~0.2, τ=0.005)

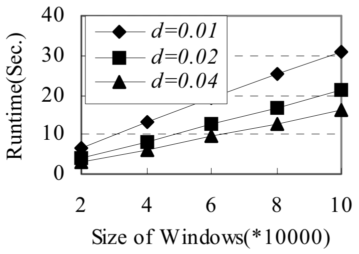

Figure 6. Scalability for different size of windows (τ=0.005)

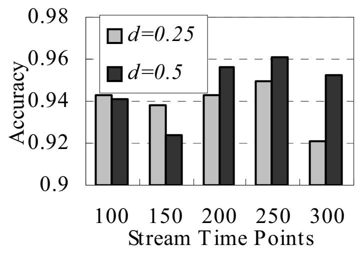

Figure 7. Accuracy of CDS in different stream time points (τ=0.02~0.06, KDD CUP99)

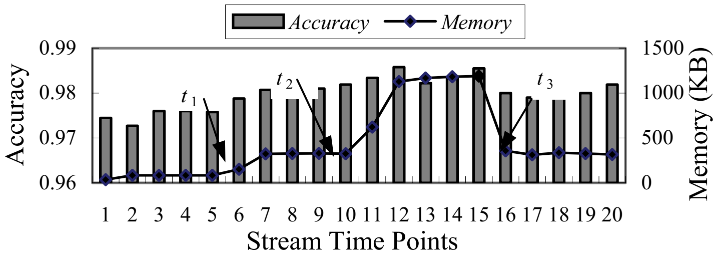

Figure 8. Granularity adjustment (M_{max}=1300KB, λ=1.5, η=0.9)

(1) Scalability. First we compare the runtime of *CDS* and *CB* on Image Fourier Coefficient datasets in Figure 4 and 5. Figure 4 shows that our algorithm based on CDS-Tree is faster than *CB* at least 6 times, where the dimensionality is 3, and the size of windows varies from 20000 to 100000 tuples. Figure 5 shows the experimental results on scalability with dimensionality, where the size of window and the time point are of the same. When larger than 8 dimensions of datasets, the time that the cell-based algorithm costs is more than 70 times than *CDS*, so we conclude that *CDS* is superior to *CB* algorithm in time efficiency when dimensionality turns higher.

To test performance of *CDS*, we test runtime under different partition granularity. As shown in Figure 6, the algorithm has linear scalability with the size of windows, and the finer granularity is, the longer the runtime is, where *d* is equal to 0.01, 0.02, and 0.04.

(2) Accuracy. Because the algorithms based on space partition are density-based clustering algorithms, they could not adopt the measures, such as the sum of square distance, used by distance-based algorithms. In this paper, for datasets labeled classes, such as KDD CUP-99, the clustering accuracy is defined as the ratio of the number of data belonging to its corresponding class to the whole cluster size. For datasets no labeled classes, we use DBSCAN [7] as the standard algorithm to label each data object, and make sure these two algorithms find the same number of outliers. 34 continuous attributes in KDD CUP-99 are normalized. As shown in Figure 7, the partition length of each attribute is either 0.25 or 0.5, and the accuracy of all the test results is above 92%.

(3) Granularity adjustment for the changes of data streams. To show the effect of adjustment of partition granularity, we choose a time interval of data stream, and track the analysis accuracy and the cost of memory. As shown in Figure 8, the granularity adjustments are executed at time point t_1, t_2 and t_3 respectively. Before time point t_1, the granularity is coarse and the analysis accuracy is low relatively too. There is more memory left, so the granularity turns finer at time point t_1. We can find that the analysis accuracy increases, but the memory cost also increases. At time point t_2, the same adjustment happened too. At time point t_3, the algorithm finds the memory cost will reach limit, so it changes current granularity to coarse. In addition, Figure 8 shows the trend of accuracy with different granularities, i.e., the larger the granularity of partition is, the lower the accuracy is.

5 Conclusions

This paper presents a novel index structure to improve the cell-based algorithm for clustering arbitrary shapes in data streams. Since the traditional algorithm is low efficient with high dimensionality and fine granularity partition, CDS-Tree is designed to only store non-empty cells, which keeps the position relationships among cells, so a sliding window is easy to maintain and clustering procedure is easy to implement. Moreover, in order to get the clustering accuracy as high as possible, we propose a new measure for data skew to be used to adjust partition granularity automatically for the change of data streams. The experimental results show the clustering algorithm based on CDS-Tree has a good scalability with the size of window and data dimensionality comparing with typical algorithms in trivial style, and it can maximize the clustering accuracy for the available amount of memory space.

There are two goals in our future works. The first one is to study how to choose good density threshold in clustering algorithm based on CDS-Tree. The second one is to investigate evolution analysis of clusters based on CDS-Tree and provide some effective algorithms.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. Proc. of 21st ACM Symposium on Principles of Database Systems (PODS 2002), Madison, Wisconsin, 2002, pp. 30-45.
[2] M. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look. Proc of 2002 ACM SIGMOD International Conference on Management of Data. Madison, Wisconsin, 2002, pp. 635-645.
[3] R. T. Ng and J. Han. Efficient and Effective Clustering Methods for Spatial Data Mining. In Proceedings of the 20th VLDB Conference, 1994, pp. 144-155.
[4] G. Karypis, E.-H. Han, and V. Kumar. CHAMELEON: A hierarchical clustering algorithm using dynamic modeling. COMPUTER, 32:68-75,1999.
[5] S. Guha, R. Rastogi, and K. Shim. CURE: An efficient clustering algorithm for large databases. In Proc. SIGMOD, Seattle, WA, 1998, pp. 73–84.
[6] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. In proc. 1996 ACM SIGMOD, Montreal, Canada, June 1996, pp. 103-114.
[7] M. Ester, H. P. Kriegel, J. Sander, X. Xu,. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases. Proc. of KDD Conf., Portland OR, USA, 1996, pp. 226-231.
[8] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. Proc. of ACM SIGMOD Conf. Seattle, WA, 1998, pp. 94-105.
[9] S. Guha, N. Mishra, R. Motwani, L. O'Callaghan. Clustering Data Streams. In Proc. IEEE FOCS Conference, 2000, pp. 359-366.
[10] S. Guha, A. Meyerson, N. Mishra, R. Motwani. Clustering Data Streams: Theory and Practice. TKDE special issue on clustering, vol. 15, 2003, pp. 515-528.
[11] B. Babcock, M. Datar, R. Motwani, L. O'Callaghan. Maintaining Variance and k–Medians over Data Stream Windows. Proc. of PODS 2003, San Diego, California, 2003, pp. 234-243.
[12] C. C. Aggarwal, J. Han, J. Wang, P. S. Yu. A Framework for Clustering Evolving Data Streams. Proc. of VLDB 2003, Berlin, Germany, 2003, pp. 81-92.
[13] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. Proc. of ACM SIGMOD 1990 Conf., Atlantic City, NJ, 1990, pp. 322–331.
[14] S. Berchtold, D. A. Keim and H. Kriegel. The X-tree: An Index Structure for High-Dimensional Data. Proc. of the 22nd VLDB Conf. Bombay, India, 1996, pp. 28-39.