Pattern Analysis and Applications (2020) 23:975–1009
https://doi.org/10.1007/s10044-019-00831-1

INDUSTRIAL AND COMMERCIAL APPLICATION

BISDBx: towards batch-incremental clustering for dynamic datasets using SNN-DBSCAN

Panthadeep Bhattacharjee[1] · **Pinaki Mitra**[1]

Received: 25 October 2018 / Accepted: 17 June 2019 / Published online: 1 July 2019

Abstract
Many important applications such as recommender systems, e-commerce sites, web crawlers involve dynamic datasets. Dynamic datasets undergo frequent changes in the form of insertion or deletion of data that affects its size. A naive algorithm may not process these frequent changes efficiently as it involves the entire set of data points each time a change is inflicted. Fast incremental algorithms process these updates to datasets efficiently to avoid redundant computation. In this article, we propose incremental extensions to shared nearest neighbor density-based clustering (SNNDB) algorithm for both addition and deletion of data points. Existing incremental extension to SNNDB viz. InSDB cannot handle deletion and handles insertions one point at a time. Our method overcomes both these bottlenecks by efficiently identifying affected parts of clusters while processing updates to dataset in batch mode. We propose three incremental variants of SNNDB in batch mode for both addition and deletion with the third variant being the most effective. Experimental observations on real world and synthetic datasets showed that our algorithms are up to 4 orders of magnitude faster than the naive SNNDB algorithm and about 2 orders of magnitude faster than the pointwise incremental method.

Keywords Density based clustering · Incremental · Batch · Dynamic datasets · Insertion · Deletion

1 Introduction

Clustering enables partitioning of data into groups, subsets or categories [1, 2, 9, 12]. Many popular clustering algorithms work on static snapshot of data [10, 11, 17, 23]. However, many real-world applications such as search engines and recommender systems are expected to work over dynamic datasets [5, 18, 20]. Dynamic datasets undergo frequent changes in their size upon insertion or deletion of data points. A naive method to get exact clustering over the changed dataset is to run the clustering algorithm again. However, if the changes in the dataset are minor, then change in output is also expected to be minimal. These changes cannot be ignored as they might be significant for the datapoints and their neighborhood.

Most of the computation in reclustering is going to be redundant. This problem becomes more severe with an increase in frequency of updates to dynamic datasets. For large datasets, a rerun of the algorithm might not finish before the next batch of updates arrive. Incremental algorithms [3, 4, 8] target this fundamental issue of redundant computation yet obtain identical output to their non-incremental counterpart. However, incremental algorithms perform efficiently when the changes made to the dataset is small as compared to the size of the original dataset. If significant changes are made to the dataset, then the use of incremental algorithms may not be an appropriate choice. Experimentally, we observe that the effectiveness of our incremental algorithms is more when small percentage of data points are added to or removed from the base dataset.

SNNDB [6] is a robust graph-based clustering technique that enables finding clusters of arbitrary shapes, sizes and densities. It is an amalgamation of the clustering scheme involving shared nearest neighbors [13] method and DBSCAN [7]. SNNDB uses the concept of shared nearest

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s10044-019-00831-1) contains supplementary material, which is available to authorized users.

✉ Panthadeep Bhattacharjee
panthadeep@iitg.ac.in; panthadeep.edu@gmail.com

Pinaki Mitra
pinaki@iitg.ac.in

[1] Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, North Guwahati, Amingaon, Assam 781039, India

Table 1 Tabular summary about the motivation behind our work

Motivation	Description
Redundant computation	Non-incremental algorithms fail to address the issue of redundant computation while handling dynamic datasets. They involve the entire dataset against every new update made to the dataset. It is important to design faster algorithms to get rid of redundant computation.
Small frequent Updates	When minimal number of insertions or deletions are made upon a larger base dataset (Refer Table footnote), the changes in clustering is also expected to be small. As a result, there is a need for designing intelligent algorithms to handle such frequent updates efficiently without redundant computation.
InSDB [21] handles pointwise addition	InSDB(IncSNN-DBSCAN) [21] is an incremental extension of the SNNDB(SNN-DBSCAN) [6]. InSDB handles addition of data points one at a time. The process may get slower as the size of the base dataset increases with new insertions. This is because in order to find the affected points against every insertion, a single scan of the whole dataset is required. This scanning time is bound to increase with the size of base dataset. As a result, there is a need to process updates in batch mode in order to quicken the cluster detection against new insertions.
InSDB [21] does not handle deletion	InSDB only facilitates insertion of data points. While deletion is also an integral part of dynamic datasets, the changes in clustering results can be affected by deletion of data points from the base dataset. Old links may break and clusters may split. Therefore, it is also important to design incremental algorithms for deletion of data points in batch mode.

Base dataset is the dataset before any insertion or deletion of points

Table 2 Drawbacks of the related methods

Algorithm	Key Function	Weakness	Improvement
SNNDB [6]	Non-incremental, detects clusters of arbitrary shapes	Inefficient for handling dynamic datasets with frequent updates	Pointwise incremental method InSDB [21] (IncSNN-DBSCAN)
InSDB [21]	Incremental method, pointwise addition, detects clusters dynamically	Fails to handle deletion, not suitable for large base dataset	Batch-incremental algorithms

neighbors (SNN) to determine the proximity score between two data points. The proximity score or the similarity value between data points p and q is the number of elements that p and q have in common between their corresponding K-nearest neighbor(KNN) [19] lists. Two data points are said to share a strong link if the number of overlapping points between their KNN lists exceeds a certain threshold provided both the points are present in each others' KNN list. A data point is designated as a dense or core point if it has sufficient number of adjacent strong links; otherwise, it is a non-core point (please refer to Sect. 2 for the definitions of KNN, SNN, core and non-core points). SNNDB comprises of the following key components to determine the clusters:

- KNN lists of the data points.
- Similarity matrix which holds the similarity values or the proximity score between any pair of points.
- Set of core and non-core points.

The algorithm groups connected core points into a cluster while a non-core point is allocated to a cluster containing its nearest core point. Points which do not obtain any cluster membership are classified as outliers or noise points. SNNDB incurs a quadratic time complexity of $\mathcal{O}(n^2)$. This is mainly due to the time taken to construct the similarity matrix. Table 1 provides a brief description about the motivation behind our work.

Existing incremental extension to SNNDB is IncSNN-DBSCAN(InSDB) [21] which facilitates addition of data points one at a time. Moreover, InSDB fails to detect clusters dynamically when points are deleted from the original dataset. In order to address these issues, we propose incremental extensions of SNNDB which processes updates made due to addition or deletion of data points in batch mode. Inserting or deleting points in batches will enable faster processing of updates at one attempt which is not the case with point-based processing. The main idea behind our incremental algorithms is to reduce the time taken to determine each of the components of SNNDB in order to achieve a better efficiency. (Please refer to Table 2 for a comparison of key functionalities between SNNDB and InSDB while handling

Table 3 Brief overview of our batch-incremental algorithms proposed in this paper

Algorithm	Brief working mechanism	Advantage	Improvement
Batch-Inc1	Computes the KNN list incrementally, detects same clusters as SNNDB batchwise insertion	Reduces the time taken to compute the KNN lists post-new updates	*Batch-Inc2*
Batch-Inc2	Computes the KNN list+ similarity matrix incrementally, same clusters as SNNDB batchwise insertion	Reduces the time taken to compute the KNN lists and construct K-SNN graph post-new updates	$BISDB_{add}$
$BISDB_{add}$	Computes the KNN list+ similarity matrix incrementally+ core and non-core points incrementally, same clusters as SNNDB, batchwise insertion	Reduces the time taken to compute the KNN lists, construct K-SNN graph, identify core and non-core points post-new updates	
Batch-Dec1	Computes the KNN list incrementally, detects same clusters as SNNDB batchwise deletion	Reduces the time taken to compute the KNN lists post-new removals	*Batch-Dec2*
Batch-Dec2	Computes the KNN list+ similarity matrix incrementally, same clusters as SNNDB batchwise deletion	Reduces the time taken to compute the KNN lists and construct K-SNN graph post-new removals	$BISDB_{del}$
$BISDB_{del}$	Computes the KNN list+ similarity matrix incrementally+ core and non-core points incrementally, same clusters as SNNDB, batchwise deletion	Reduces the time taken to compute the KNN lists, construct K-SNN graph, identify core and non-core points post-new updates	

dynamic datasets. Refer Table 3 for a brief overview about our proposed incremental methods). Next, we present our set of contribution(s) made in this paper.

1.1 Our contribution(s)

Our key contribution(s) in this paper can be summarized by the following list:

1. We initially propose three incremental variants of SNNDB which processes updates made due to addition of data points in batch mode. These three algorithms are *Batch-Inc*1, *Batch-Inc*2 and $BISDB_{add}$. *Batch-Inc*1 computes KNN list incrementally. *Batch-Inc*2 computes KNN list and similarity matrix incrementally while $BISDB_{add}$ computes each of the components of SNNDB incrementally. From our experimental observations, we observed that the third variant $BISDB_{add}$ is the most effective as compared to the other two variants.
2. For handling deletion, we propose three new variants: *Batch-Dec*1, *Batch-Dec*2 and $BISDB_{del}$ with the third variant $BISDB_{del}$ being the most effective. The computation scheme of each of these variants corresponds to its addition counterpart.
3. We showed the effectiveness of our fastest incremental variants: $BISDB_{add}$ and $BISDB_{del}$ over SNNDB while handling minimal changes made to the dataset.
4. We also demonstrated the fact that when the size of base dataset increases or decreases, pointwise addition or deletion no longer remains an effective solution to detect clusters dynamically. The updates made to the dataset in batch mode proves to be more efficient than both the naive and point-based incremental method as we inflict changes on a larger base dataset.
5. We provided a thorough analysis of clusters obtained from the batch-incremental algorithms for addition and deletion.

Paper Outline Section 2 provides the definitions of concepts and terminologies used throughout the paper. In Sect. 2.1, we formally present the objective of our problem. Section 4 deals with the related methods based on which the batch-incremental algorithms are built. The following Sect. 5 gives a generic structure of the batch-incremental algorithms for both addition and deletion proposed in this paper. Sections 6 and 7 describe the batch-incremental algorithms for addition and deletion of data points, respectively. The complexity analysis of the most efficient batch-incremental algorithms for addition and deletion are presented in Sect. 8. Section 10 gives a detailed description of various experiments that were

Table 4 Notations used in the paper

Notation	Description
C	Set of Clusters prior to any changes in dataset
C'	Set of Clusters after dataset is updated
D	Original (Base) dataset
D'	Changed dataset after insertions or deletions
B	Number of batches
n	No. of points per batch(inserted or deleted)
k	Total no. of points to be inserted or deleted
K	Size of the K-nearest neighbor list
δ_{sim}	Strong link formation threshold
δ_{core}	Core point formation threshold
$\mathcal{P}(.)$	Power set
KNN(.)	KNN list of any data point
Sim_Mat(.)	Similarity matrix of dataset
Core(.)	Set of core points of dataset
Non-Core(.)	Set of non-core points of dataset
|.|	Size of a set

carried on to prove the efficiency of batch-incremental algorithms. We finally present the cluster analysis followed by discussion and conclusion in Sects. 11 and 12, respectively.

2 Preliminaries and problem formulation

In this section, we define the key terms used in this article along with the problem formulation. (Please refer Table 4 for the meaning of notations used in this paper).

1. **K-nearest neighbor (KNN) list** We define the KNN list of a data point by identifying its top-K closest points. For our purpose, we adopt the Euclidean distance measure.
2. **Shared nearest neighbors (SNN)** The concept of shared nearest neighbors was first introduced by Jarvis and Patrick [13]. The SNN between two data points p and q is defined as number of points p and q have in common between their respective KNN lists.
3. **Similarity value** The SNN value between two points p and q is referred to as the similarity value of p with q or vice-versa. It is defined by the following equation:

$$similarity(p, q) = KNN(p) \cap KNN(q) \quad (1)$$

where $KNN(x)$ is the number of elements present in the KNN list of data point x.
4. **Similarity matrix or SNN graph** Similarity matrix represents the shared nearest neighbor(SNN) graph. The data points are modeled as nodes and the similarity value between any pair of nodes is the considered as the edge weight.
5. **K-SNN graph** K-Sparsified SNN(K-SNN) graph is the residual graph formed after "K-sparsification" of the original SNN graph. Here K represents the size of KNN list for each data point in D. From the original SNN graph, an edge is retained between a pair of points p and q, only if p and q are present in each other's KNN list and the edge weight between p and q is greater than or equal to certain threshold δ_{sim}(say). However, if the edge weight falls below δ_{sim}, then the link is not formed. The edges which are present in the SNN graph are identified as strong links. This method of obtaining a residual graph from the original SNN graph is known as K-sparsification of the SNN graph [6]. We refer the SNN graph containing the nodes connected by strong links as the K-sparsified SNN graph or K-SNN graph.
6. **Core and non-core points** In the K-SNN graph, if the number of strong links associated with a particular point exceeds a certain threshold value δ_{core} (say), then the point obtains a core point status. The remaining points are classified as non-core points.
7. **Noise points** The non-core points which do not share a link with any of the core points and fail obtain a cluster membership are classified as noise points or outliers.
8. **Clustering** Given a dataset D, a similarity function $sim(x,y)$, and a point density function $dense(x)$, we define clustering by a mapping f: $D \rightarrow C$, where $C = \mathcal{P}(D)$. If $x,y \in D$ and $x \neq y$ and there exists two threshold values δ_{sim} and δ_{core} then:

   i If $sim(x,y) \geq \delta_{sim}$ and $dense(x) > \delta_{core}$, $dense(y) > \delta_{core}$, then $f(x) = f(y)$.
   ii *If* $sim(x,y) \geq \delta_{sim}$ and $dense(x) > \delta_{core}$, $dense(y) \ngtr \delta_{core}$, such that $\exists\, z \in D$ where $x \neq y \neq z$ and $dense(z) > \delta_{core}$ and $sim(y,z) \geq \delta_{sim}$, if $sim(y,z) > sim(x,y)$ *then* $f(y) = f(z)$.
   iii $\forall\, x$ where $dense(x) \ngtr \delta_{core}$,here exists not y such that $sim(x,y) \geq \delta_{sim}$ and $dense(y) > \delta_{core}$, then $x \notin C$.

   As per the first point, if the degree of closeness or similarity between points x and y is greater than a threshold value δ_{sim} and both x and y are dense or core points, then both points are a part of the same cluster.

   As per the second point, the similarity between points x and y is greater than a threshold value δ_{sim} and x is core but y is non-core. There exists another core point z and the similarity between y and z is greater than δ_{sim}. In that case, if y is more similar to z than x, then points y and z belong to the same cluster.

   The third points state that if x is a non-core point, and there exists no core point y with which x has a similarity greater than δ_{sim}, then point x is a noise point.
9. **Batch-incremental Clustering** Given a data set D along with its initial clustering f : $D \rightarrow C$ where $C \subseteq D$ and an insertion or deletion sequence of B batches with 'n' points per batch. After $k \leq nB$ updates (insertions or deletions), such that $nB(\text{mod } k) \equiv 0$. Let D' be the new data set, then an incremental clustering is defined as a mapping h : f, $D' \rightarrow C'$, where $C' = \mathcal{P}(D')$ is isomorphic to the one time clustering f(D') by the non-incremental algorithm.

2.1 Problem formulation

For k number of insertions/deletions where $k \in \mathbf{N}$, let $T_{non-inc}$ be the total time taken by the non-incremental method, $T_{point-inc}$ be the total time taken by the point insertion-based method and $T_{batch-inc}$ be the total time taken by the incremental method processing data in batch mode. Let $C_{non-inc}$, $C_{point-inc}$ and $C_{batch-inc}$ be the respective set of clusters obtained after k updates, then we establish the following objectives:

1. $T_{batch-inc} < T_{non-inc} | C_{batch-inc} = C_{non-inc}$
2. $T_{batch-inc} < T_{point-inc} | C_{batch-inc} = C_{point-inc}$

In the next section, we highlight some recent works on clustering.

3 Recent works on clustering

Li et'al [16] proposed a robust structured nonnegative matrix factorization (RSNMF) framework leveraging the use of $l_{2,p}$-norm loss function [14] to deal with noises and outliers in clustering. The work focused on learning a robust discriminative representation of feature sets while handling dimensionality reduction.

In a $\mathcal{R}^d$ space, the phenomenon of 'curse of dimensionality' [6] is a major concern due to which the proximity between data points obtained through any geometric model, e.g., distance becomes unreliable. One of the building block algorithms for our proposed batch-incremental methods in this paper viz. SNN-DBSCAN [6] relies on the shared nearest neighbor (SNN) [13] scheme to alleviate the use of any distance-based function. SNN is a data dependent measure that identifies the closeness between a pair of points based on the number of data items in their shared neighborhood. The SNN-based scheme of identifying the proximity score between data points limits the disadvantages associated with distance-based methods, e.g., inability to find clusters of arbitrary densities and shapes.

High dimensional data, e.g., images may involve redundant features along with the presence of noisy elements. In order to identify a subset of useful and redundancy constrained features, a nonnegative spectral clustering scheme coupled with analysis of redundant features [15] was proposed. The nonnegative spectral analysis technique helped learning of cluster labels related to input data more accurately. The simultaneous learning of cluster labels and attribute matrix enabled selection of the most discriminative features appropriately.

Another robust clustering technique called MBSCAN [22] overcomes the limitations of distance-based clustering in $\mathcal{R}^d$ space by adopting a data dependent dissimilarity measure. MBSCAN utilizes the measure of probability mass [22] instead of any geometric model for computing pairwise dissimilarity of points.

In Sect. 4, we precisely describe the algorithms that form the background of our proposed batch-incremental methods for both addition and deletion in this paper.

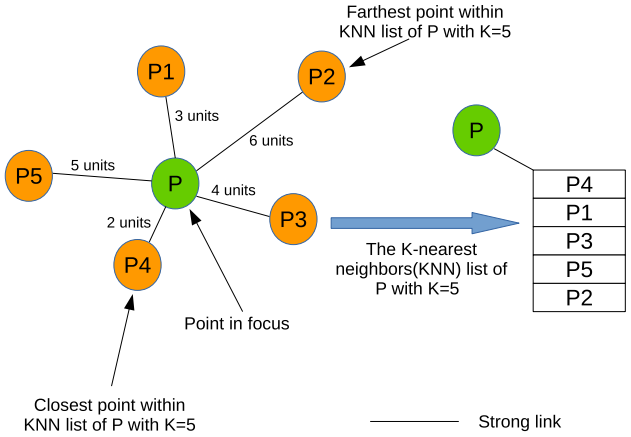

Fig. 1 KNN list for point P where $K = 5$

4 Background

In this section, we provide a brief description about algorithms:SNN-DBSCAN [7] and IncSNN-DBSCAN [21] which form the basis of our batch-incremental methods in this paper.

4.1 SNN-DBSCAN

Shared nearest neighbor density-based clustering algorithm (SNN-DBSCAN) [6] or SNNDB is a graph-based clustering technique that enables determining clusters of varying densities, sizes and shapes. The robustness of the algorithm comes in form of its ability to detect clusters from small compact regions which other density-based clustering methods usually ignore. However, due to the non-incremental nature of SNNDB, processing frequent updates made to the dataset while detecting clusters dynamically remains a major challenge. Our incremental algorithms seek to improve upon the performance of the SNNDB clustering technique while handling dynamic datasets. The building blocks of SNNDB method are:

- K-nearest neighbors (KNN).
- Shared nearest neighbors (SNN).

KNN list consists of top-K-nearest neighbors for any data point. Please refer to Fig. 1 for the representation of KNN list for the data point P(say). In this figure, let the points {P1, P2, P3, P4 ,P5} be at a distance of 3, 6, 4, 2 and 5 units, respectively, from P. Then for $K = 5$ the KNN list for P is the set {P4, P1, P3, P5 ,P2}.

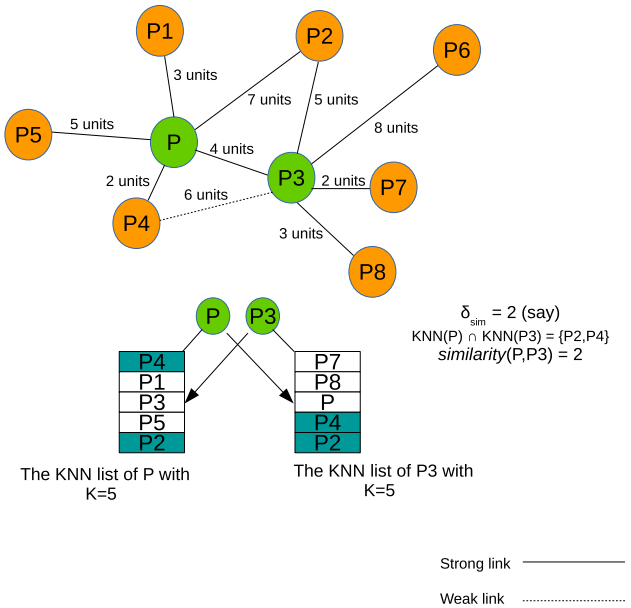

Fig. 2 Similarity value between points P and P3 in the K-SNN graph given that P ∈ KNN(P3) and P3 ∈ KNN(P) and $K = 5$

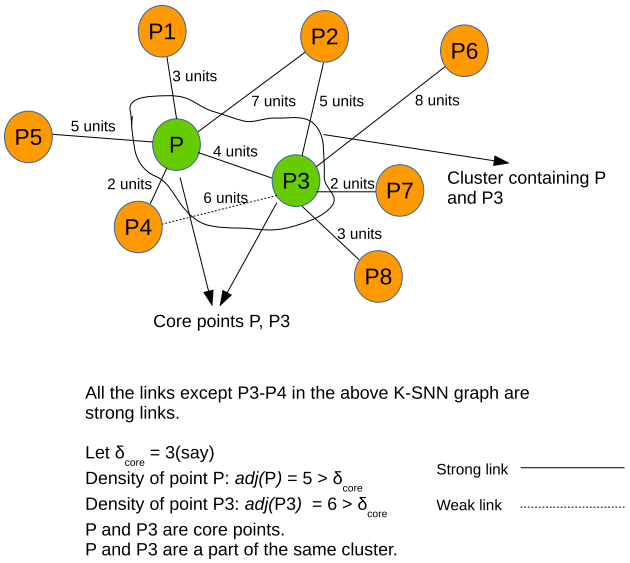

Fig. 3 Cluster containing core points P and P3 in the K-SNN graph. If δ_{core} is set as 4, then $adj(P) > \delta_{core}$ and $adj(P3) > \delta_{core}$

The concept of shared nearest neighbors or SNN is inherited from the clustering scheme proposed by Jarvis and Patrick [13]. The SNN clustering technique limits to use any distance metric for deciding the measure of closeness between any two data points. Instead it relies on the number of shared data points between the KNN lists of any pair of points (p, q) to evaluate their proximity. The proximity score obtained is treated as the similarity value between p and q. While constructing the SNN graph, the data points are treated as nodes while the edge weight is equivalent to the similarity value between p and q. This step is followed by the "K-Sparsification" [6, 13] step. In this step, an edge is formed between any two nodes p and q iff the following two conditions are satisfied:

1. Points p and q are present in each others' KNN list.
2. The similarity value between p and q is greater than or equal to a certain threshold δ_{sim}(say).

Each of the edges constructed between any pair of points (p,q) satisfying the above two conditions are considered as strong links. Figure 2 demonstrates the similarity value calculation and strong link formation between two points P and P3. The KNN list of P contains {P4, P1, P3, P5 P2} while the KNN list of P3 consists of {P7, P8, P, P4, P2}. We observe that both P and P3 are included in each others' KNN list. As a result, the number of shared elements in their KNN lists gives the similarity value between P and P3. The proximity score or the degree of closeness between P and P3 is given as 2. This is because P and P3 share two elements:{P2,P4} between their KNN lists. If the value of δ_{sim} is set to be 2, then the edge between points P and P3 is considered as a strong link since $\#\{P2,P4\} \geq \delta_{sim}$.[1]

The graph obtained by this mechanism is known as the K-sparsified SNN(K-SNN) graph [6, 13]. In the K-SNN graph, all the existing edges between any pair of nodes are strong links. While constructing a edge between p and q, if any one of the above two conditions is violated, an edge is not formed between p and q. All the connected components contained in the K-SNN graph are now treated as the final set of clusters by the SNN [13] algorithm. However, the SNNDB [6] method produces the K-SNN graph without considering its connected components as clusters. Instead, the SNNDB algorithm adopts a clustering scheme similar to the DBSCAN [7] algorithm. SNNDB identifies the dense(core) and border (non-core) points to find its final set of clusters. In the K-SNN graph, for any given point p(say), SNNDB detects the number of strong links (edges in the K-SNN graph) adjacent to p. Let it be denoted as $adj(p)$. If $adj(p) > \delta_{core}$ (a certain threshold) then p is classified

[1] Having a point in the KNN list does not guarantee the formation of a shared strong link between the concerned point and its neighbor. For a shared strong link to exist, each of the two conditions for strong link formation must be satisfied.

as a core point, else p is a non-core point. The number of strong links associated with point p provides a measure of its density.

Similar to DBSCAN [7], if p and q are two core points connected by a strong link, then both these points obtain the same cluster membership(Refer to first point under *Clustering* definition from Sect. 2). However, if one of them is a non-core point, then that point is allocated to a cluster containing its nearest core point(Refer to second point under *Clustering* definition from Sect. 2). The nearest core point is the one that shares a strong link with the non-core point having a higher edge weight as compared to other core points. The set of points which fail to obtain any cluster membership are classified as noise points or outliers (Refer to third point under *Clustering* definition from Sect. 2). Please refer to Fig. 3 for an illustration of the core point and cluster formation scheme.

In this figure (Fig. 3), let us assume that the core point formation threshold (δ_{core}) is set to 3. Now for point P, the number of adjacent strong links is five. Therefore *adj*(P) is 5. Similarly for point P3, *adj*(P3) is determined as 5. Points P3 and P4 share a weak link which is not considered as a link. It is just given for representational purpose. Since, the density of the points P and P3 exceeds the threshold value of δ_{core}, P and P3 are designated as core points. As per the DBSCAN [7] clustering scheme, points P and P3 become a part of the same cluster.

4.2 IncSNN-DBSCAN

IncSNN-DBSCAN [21] or InSDB is an incremental extension to the SNNDB algorithm. InSDB facilitates detection of clusters dynamically while points are added to the original dataset D one at a time. InSDB identifies each data point $p \in D$ with the following properties: KNN list, strengths of shared strong links, number of adjacent strong links, core or non-core status. When a new data point arrives, InSDB identifies only those points which undergo changes in their properties. The algorithm targets the affected points while the rest of the points are allowed to exist in their previous state.

Let *Npt* be a new data point entering D. Upon entry of *Npt*, D changes to D'. Now $\forall\ p \in D$, if p exhibits changes in its properties (stated above), then InSDB targets p. The changes that p incurs in its properties may lead to creation of new SNN connections and deletion of the existing ones. New SNN connections could merge the existing clusters and deletions could split them. The selective handling of data points ensures that reconstruction time of the updated KNN lists, K-SNN graph is drastically reduced. InSDB shows that a very small percentage of points ultimately gets affected due to which it becomes more efficient than SNNDB.

However, InSDB is a point-based insertion technique, which might slow down as the size of D increases. This is because when inserts are made upon a larger base dataset, the time required to find the affected points will increase along with the reconstruction time of the updated K-SNN graph against every insert.

5 Structure of our batch-incremental SNNDB clustering algorithms

The generic structure of our batch-incremental clustering algorithms is given as follows:

1. New data points are added to or removed from the original dataset in batches.
2. We characterize each data point by the following properties:
   (a) KNN list.
   (b) SNN value or the similarity value of a point with each of its adjacent points connected by a strong link.
   (c) Core, Non-core point status.

   The affected points undergo changes in their properties upon entry or removal of data points.
3. In case of addition, the algorithms compute the values of these properties for each of the newly added points. A fraction of existing points which are affected due to new updates in the dataset are targeted.

   In case of deletion, the incremental algorithms target only those points which are affected by the deletion of existing points.
4. The data points which are not affected by the entry of new points or removal of old points do not change the values of their properties.
5. Some points change their status from core to non-core while others change from non-core to core points. The strength of the shared links between data points may alter. If the link strength reduces below δ_{sim}, the link gets broken resulting in possible splitting of clusters.
6. The changed dataset consists of data points with updated property values.
7. Two connected core points are contained in the same cluster. The cluster expansion takes place by grouping the core points accordingly. The non-core points are put into a cluster of their nearest core point. Points which

fail to obtain any cluster membership are categorized as noise points.
8. The size of the base dataset increases (or decreases) in size. The whole process is repeated after a new batch of points have been added to or removed from the updated dataset.

In the following two sections, we present the proposed Batch-Incremental SNNDB (BISDB) Clustering Algorithms for Addition and Deletion.

6 Batch-incremental SNNDB clustering algorithms for addition

In this section, we present our proposed batch-incremental SNNDB clustering algorithms for addition: *Batch-Inc*1, *Batch-Inc*2 and $BISDB_{add}$. The goal of these algorithms is to identify clusters dynamically while points are added to the dataset in batch mode. Prior to executing each of these batch-incremental algorithms, the non-incremental SNNDB [6] has to be executed. We execute the SNNDB clustering algorithm in order to obtain the following information with respect to(wrt.) the original dataset D:

1. KNN list $\forall\, p \in D$ given as KNN(p).
2. Similarity matrix (K-SNN graph) wrt. D given as $Sim\_Mat(D)$.
3. Set of core and non-core points wrt. D given as $Core(D)$ and $Non - Core(D)$.
4. Set of clusters C wrt. D.
5. Set of noise points or outliers wrt. D.

We present these algorithms in a step-wise manner. Individual steps contain a detailed description of the algorithm with necessary graphical illustration leading to the detection of clusters dynamically.

6.1 The *Batch-Inc*1 algorithm

The *Batch-Inc*1 algorithm builds the updated KNN list incrementally. When new points enter D, some of the old points are affected as they may undergo change in their properties. The algorithm builds the KNN list for each of the data points incrementally by targeting only the affected old points. The points which are not affected due to entry of new points retain their existing KNN list. The new similarity matrix(updated K-SNN graph/K-$SNN_{updated}$ graph), new core and non-core points are determined non-incrementally. The steps of the *Batch-Inc*1 algorithm are as follows:

1. **Set the parameters** The algorithm takes three parameters: K, δ_{sim}, δ_{core}. The parameters have the following meanings:
   (a) K denotes the size of the KNN list for each data point.
   (b) δ_{sim} is the minimum number of shared nearest neighbors between two data points p,q required to form a strong link between them given that p and q are in each others' KNN list.
   (c) δ_{core} is the minimum number of adjacent strong links associated with a point p exceeding which p becomes a core point.
2. **Obtain the required data from prior SNNDB execution**
   (a) Get the original dataset D where $|D| = x$(say).
   (b) Get the KNN list $\forall\, p_i \in D, i = 1, 2, \ldots, x$.
   (c) Get the similarity matrix ($Sim\_Mat(D)$).
   (d) Add a batch containing n new data points to D. The size of D increases and let the changed dataset be denoted as $D\prime$ where $|D'| = x+n$.
3. **Compute the KNN list of new points** In this step, the KNN list of all the newly added points is computed. If n data points are added in a single batch, then $\forall x_i, i = 1, 2 \ldots, n$, we find KNN($x_i$).
4. **Compute the updated KNN list for existing data points incrementally** The number of existing data points in D(original dataset) prior to any insertion is x. When n new points are added to D, D changes to $D\prime$ ($|D'| = x + n$). The algorithm identifies among previously existing points in D that can accommodate any newly added point in its KNN list replacing an old point. If the size of the nearest neighbor list is K then, a maximum of K old points can be replaced by the new ones from the KNN list of an old point. Those points which contain at least one newly added point in its KNN list are categorized as $KN - S_{add}$ type affected points.

   The term $KN - S_{add}$ means that both the KNN list as well as the similarity measures of the affected data points are altered. KN stands for change(s) in the KNN list while S signifies a possible change in the similarity values (shared link strength) of the affected data point

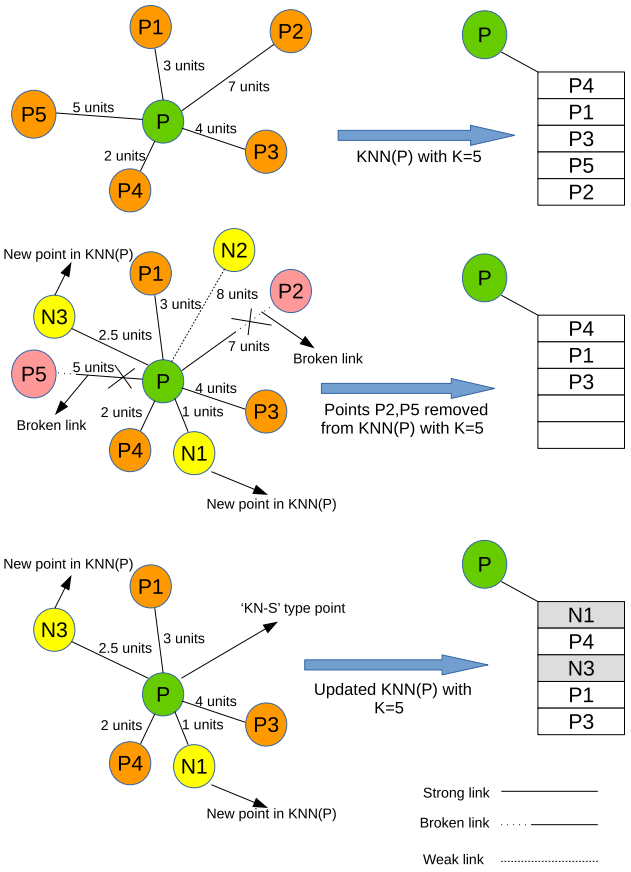

Fig. 4 The formation of $KN - S_{add}$ type affected points upon entry of new points

with points in its updated KNN list(KNN$_{updated}$(.)). If the new link strength falls below δ_{sim}, the link is not formed. The new points and the unaffected old points are not categorized as $KN - S_{add}$ type. The unaffected old points retain their previous KNN list. In this way, *Batch-Inc*1 avoids redundant KNN list computation for the unaffected points by focusing on only $KN - S_{add}$ type points.

Let us visit Fig. 4 for an illustrative example of this step(Step 4). Consider the point P, where $KNN(P) = \{P4, P1, P3, P5, P2\}$ (topmost image in Fig. 4) prior to entry of any new points in the dataset. Let three new points N1, N2 and N3 enter the dataset. For our purpose, we consider that N1 and N3 are at a distance of 1 and 2.5 units, respectively, from P. N2 is at a distance of 8 units (say). On comparing distances with other nearest neighbors of P, it is clear that the points N1 and N3 can potentially enter into the KNN list of P displacing P2 and P5. The link between the pairs of points: (P,P2) and (P,P5) gets broken[2] resulting in two vacant slots in KNN(P) (second image in Fig. 4). Consequently, points N1 and N3 occupy the two vacant slots created in KNN(P).

Between N1, N2 and N3, we consider only N1 and N3 share a strong link with P. On sorting the current set of points in increasing order of distance to point P, the updated KNN list of P(KNN$_{updated}$(P)) obtained incrementally consists of {N1,P4,N3,P1,P3} (bottom image in Fig. 4). Point P is therefore a $KN - S_{add}$ type affected point since it accommodates new points N1, N3 in its updated KNN list. New point N2 does not have any influence over the KNN list of P and is therefore not a member of KNN$_{updated}$(P). The newly entered points and the old unaffected points are not classified as $KN - S_{add}$ type. For any non-$KN - S_{add}$ type old point(say P1, P3 or P4), we have the following: KNN$_{updated}$(P1) = KNN(P1),
KNN$_{updated}$(P3) = KNN(P3),
KNN$_{updated}$(P4) = KNN(P4).

5. **Construct the updated K-SNN graph** The algorithm constructs the updated K-SNN(K-SNN$_{updated}$) graph or the new similarity matrix ($Sim\_Mat(D')$) non-incrementally. The updated dataset D' now consists of $x + n$ points. Therefore, $\forall p_i \in D', i = 1, \ldots, x + n$, *Batch-Inc*1 determines if a shared strong link can be constructed $\forall q_i \in$ KNN$_{updated}(p_i)$. If we revisit Fig. 4(bottom image), we observe that with each of the points belonging to KNN$_{updated}$(P), P shares a strong link. The edge weight or the link strength becomes the similarity value for point P with every other point in KNN$_{updated}$(P).
6. **Identify new core and non-core points** The new similarity matrix effectively represents the K-SNN$_{updated}$ graph. The algorithm checks every point in K-SNN$_{updated}$ graph in order to identify if it has sufficient number of strong links ($> \delta_{core}$) adjacent to it. Such points are referred to as core (dense) points. The remaining points are non-core. The new set of core and non-core points are denoted as $Core(D')$ and $Non - Core(D')$.
7. **Form Clusters** Two core points are grouped into the same cluster. A non-core point is pulled toward the cluster of its nearest core point.[3]
8. **Discard noise points** The non-core points which are not connected to any core point are classified as noise

[2] The link gets broken as the points are no longer in each others' KNN list (a necessary condition to construct a shared strong link).

[3] The core point with which the shared link strength is highest becomes the "nearest" core point.

points. Such points do not obtain any cluster membership.

9. **Preserve the updated values**

   (a) $D = D'$
   (b) $x = x + n$
   (c) $\forall p_i, i = 1, 2, \ldots, x + n$
   $\text{KNN}(p_i) = \text{KNN}_{updated}(p_i)$.
   (d) $Sim\_Mat(D) = Sim\_Mat(D')$
   (e) $Core(D) = Core(D')$
   (f) $Non - Core(D) = Non - Core(D')$

10. Repeat Steps 1 to 9 for the next batch of entering n points.

6.2 The *Batch-Inc*2 algorithm

The *Batch-Inc*2 algorithm constructs the updated KNN list and the new similarity matrix(K-SNN$_{updated}$ graph) incrementally in order to further improve the efficiency while detecting clusters dynamically. The steps of the *Batch-Inc*2 algorithm are as follows:

1. **Set the parameters** The algorithm takes three parameters: K, δ_{sim}, δ_{core}. The parameters have the following meanings:

   (a) K denotes the size of the KNN list for each data point.
   (b) δ_{sim} is the minimum number of shared nearest neighbors between two data points p,q required to form a strong link between them given that p and q are in each others' KNN list.
   (c) δ_{core} is the minimum number of adjacent strong links associated with a point p exceeding which p becomes a core point.

2. **Obtain the required data from prior SNNDB execution**

   (a) Get the original dataset D where $|D| = x$(say).
   (b) Get the KNN list $\forall$ p$_i \in D, i = 1, 2, \ldots, x$.
   (c) Get the similarity matrix ($Sim\_Mat(D)$).
   (d) Add a batch containing n new data points to D. The size of D increases and let the changed dataset be denoted as $D\prime$ where $|D'| = x$+n.

3. **Compute the KNN list of new points** Step 3 is similar to that of *Batch-Inc*1.

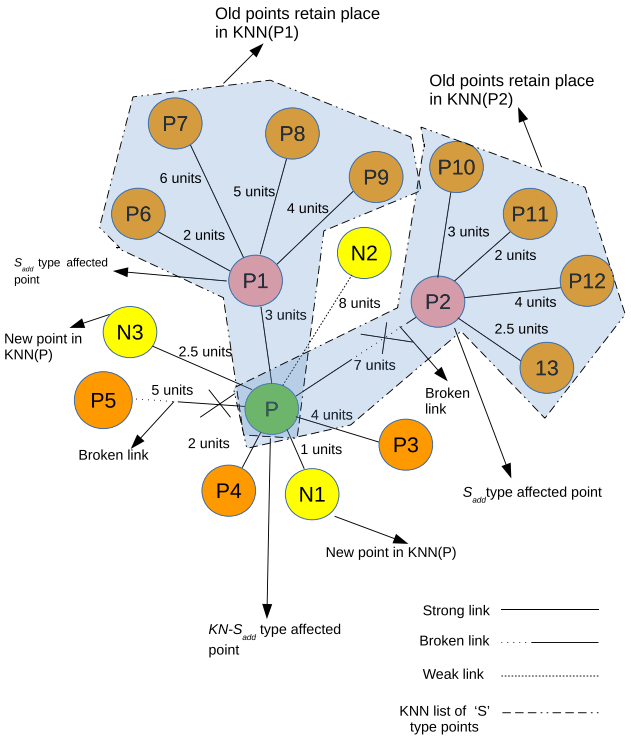

Fig. 5 The formation of S_{add} type affected points upon entry of new points

4. **Compute the updated KNN list for existing data points incrementally** Step 4 is similar to that of *Batch-Inc*1.
5. **Construct the updated K-SNN graph incrementally** This step focuses on reconstruction of K-SNN$_{updated}$ graph incrementally by building new similarity matrix($Sim\_Mat(D')$). Similar to *Batch-Inc*1, the algorithm targets only the affected points while building the K-SNN$_{updated}$ graph. However, *Batch-Inc*2 introduces a new type of affected point known as S_{add} type point. The S_{add} type points are those which do not change their old KNN list upon entry of new data points. However, they are a part of the updated KNN list of $KN - S_{add}$ type points. S_{add} type points may contain at least one $KN - S_{add}$ type point in their KNN list. The non-$KN - S_{add}$ type points displaced by the new points from the updated KNN list of a $KN - S_{add}$ type point also belong to S_{add} type. The term S_{add} means that for any point $p \in D'$(updated dataset), only the value of shared strong link(edge weight) with p's adjacent points may change but KNN(p) remains the same. For S_{add} type affected point p, we have KNN$_{updated}(p)$ = KNN(p).

S_{add} type points can be determined from the updated KNN list of any $KN - S_{add}$ type point. On scanning the updated KNN list of any $KN - S_{add}$ type point, the old points which are non-$KN - S_{add}$ type are classified as S_{add} type. The old points which are non-$KN - S_{add}$ type and are replaced by the newly entered points are also classified as S_{add} type. New data points are neither $KN - S_{add}$ nor S_{add} type. The old unaffected points retain their original KNN list as well as the similarity values and are neither $KN - S_{add}$ nor S_{add} type. *Batch-Inc*2 therefore avoids rebuilding of updated KNN lists and K-SNN$_{updated}$ graph((*Sim_Mat*(D'))) without involving D' in its entirety.

Figure 5 illustrates the formation of S_{add} type affected points upon entry of new data points. Similar to *Batch-Inc*1, let N1, N2 and N3 be the three new points entering the dataset. N1 and N3 find a place in KNN$_{updated}$(P) due to which P is classified as a $KN - S_{add}$ type point. KNN$_{updated}$(P) consists of {N1,P4,N3,P1,P3}(from *Batch-Inc*1). However, if we focus within the two shaded polygons in Fig. 5, we observe the following:

(a) Within the left polygon:
KNN$_{updated}$(P1) = KNN(P1) = {P6,P,P9,P8,P7}
(b) Within the right polygon:
KNN$_{updated}$(P2) = KNN(P2) = {P11,P13,P10,P12,P}

From the updated nearest neighbor list of P1 and P2, we observe that both these points have retained their original KNN list even after the entry of N1, N2 and N3(new points). Another notable observation is that P1 $\in$ KNN$_{updated}$(P) and P1 is non-$KN - S_{add}$ type, therefore P1 can be categorized as S_{add} type point. However, point P2 $\notin$ KNN$_{updated}$(P) but P2 $\in$ KNN(P) (refer to Step 4 of *Batch-Inc*1). This means that P2 is a displaced point from the updated KNN list of P. P2(non-$KN - S_{add}$ type) also qualifies as a S_{add} type point.

As per *Batch-Inc*2, the S_{add} type points can be determined from scanning the updated KNN list of a $KN - S_{add}$ type point. Let us revisit Fig. 4 where we observe that for the $KN - S_{add}$ type point P, KNN$_{updated}$(P) = {N1,P4,N3,P1,P3} while before entry of new points KNN(P) = {P4,P1,P3,P5,P2}. This means that in KNN$_{updated}$(P), the displaced points are P2 and P5 while the retained points are P1, P3 and P4. The new points are N1 and N3. Among the retained points (please refer to Fig. 5), point P1 sustains its original KNN list post-entry of new points. In addition, the presence of P1 in KNN$_{updated}$(P) means that P1 continues to share a strong link with P. Since P1(a non-$KN - S_{add}$ type point) $\in$ KNN$_{updated}$(P), P1 qualifies as S_{add} type point with a shared strong link with P. For the displaced point P2 (refer Fig. 5), KNN$_{updated}$(P2) = KNN(P2) and P2(a non-$KN - S_{add}$ type point), therefore P2 is also categorized as a S_{add} type point without a shared strong link since P2 $\notin$ KNN$_{updated}$(P).

6. **Identify new core and non-core points** Step 6 is similar to that of *Batch-Inc*1.
7. **Form Clusters** Step 7 is similar to that of *Batch-Inc*1.
8. **Discard noise points** Step 8 is similar to that of *Batch-Inc*1.
9. **Preserve the updated values**
   (a) $D = D'$
   (b) $x = x + n$
   (c) $\forall p_i, i = 1, 2, \ldots, x + n$ KNN(p_i) = KNN$_{updated}(p_i)$.
   (d) $Sim\_Mat(D) = Sim\_Mat(D')$
   (e) $Core(D) = Core(D')$
   (f) $Non - Core(D) = Non - Core(D')$
10. Repeat Steps 1 to 9 for the next batch of entering n points.

6.3 The $BISDB_{add}$ algorithm

The $BISDB_{add}$ algorithm builds the updated KNN list, the updated K-SNN graph, the new core and non-core points incrementally. With each of the components of the SNNDB being handled incrementally, $BISDB_{add}$ attempts to reduce the computational cost as compared to *Batch-Inc*1 and *Batch-Inc*2. The steps of the $BISDB_{add}$ algorithm are as follows:

1. **Set the parameters** The algorithm takes three parameters: K, δ_{sim}, δ_{core}. The parameters have the following meanings:
   (a) K denotes the size of the KNN list for each data point.
   (b) δ_{sim} is the minimum number of shared nearest neighbors between two data points p,q required to form a strong link between them given that p and q are in each others' KNN list.
   (c) δ_{core} is the minimum number of adjacent strong links associated with a point p exceeding which p becomes a core point.

2. **Obtain the required data from prior SNNDB execution**

   (a) Get the original dataset D where $|D| = x$(say).
   (b) Get the KNN list $\forall\ p_i \in D, i = 1, 2, \ldots, x$.
   (c) Get the similarity matrix ($Sim\_Mat(D)$).
   (d) Get the set of core ($Core(D)$) and non-core($Non - Core(D)$) points.
   (e) Add a batch containing n new data points to D. The size of D increases and let the changed dataset be denoted as $D\prime$ where $|D'| = x+n$.

3. **Compute the KNN list of new points** Step 3 is similar to that of *Batch-Inc*1 and *Batch-Inc*2.
4. **Compute the updated KNN list for existing data points incrementally** Step 4 is similar to that of *Batch-Inc*1 and *Batch-Inc*2.
5. **Construct the updated K-SNN graph incrementally** Step 5 is similar to that of *Batch-Inc*2.
6. **Identify new core and non-core points incrementally:** In the K-SNN$_{updated}$ graph, for each of the total number of $KN - S_{add}$ and S_{add} type points, $BISDB_{add}$ checks whether its number of adjacent strong links exceeds δ_{core}. If this happens, the concerned point is treated as a core point or else it is a non-core point. The remaining points retain their existing core or non-core status from the previous iteration.
7. **Form Clusters** Step 7 is similar to that of *Batch-Inc*1 and *Batch-Inc*2.
8. **Discard noise points** Step 8 is similar to that of *Batch-Inc*1 and *Batch-Inc*2.
9. **Preserve the updated values**

   (a) $D = D'$
   (b) $x = x + n$
   (c) $\forall p_i, i = 1, 2, \ldots, x + n$
   $\mathrm{KNN}(p_i) = \mathrm{KNN}_{updated}(p_i)$.
   (d) $Sim\_Mat(D) = Sim\_Mat(D')$
   (e) $Core(D) = Core(D')$
   (f) $Non - Core(D) = Non - Core(D')$

10. Repeat Steps 1 to 9 for the next batch of entering n points.

6.4 Summary: batch-incremental SNNDB clustering algorithms for addition

In this section, we presented three incremental clustering algorithms for addition: *Batch-Inc*1, *Batch-Inc*2 and $BISDB_{add}$. Each of these algorithms is an incremental extension of the SNNDB [6] method processing updates in batch mode. The main idea behind developing these methods revolves around reducing the time taken to compute the individual components of SNNDB: KNN list, K-SNN graph(similarity matrix), core and non-core points.

In an attempt to improve the efficiency of SNNDB while handling dynamic datasets, we initially propose *Batch-Inc*1. *Batch-Inc*1 computes the updated KNN list of all the data points incrementally while rest of the components are computed similar to SNNDB. In order to improve upon *Batch-Inc*1, we propose *Batch-Inc*2 which rebuilds both the updated KNN lists and the K-SNN$_{updated}$ graph upon entry new data points incrementally. The third method $BISDB_{add}$ computes all the three components which includes detection of core and non-core points incrementally.

The SNNDB method takes $\mathcal{O}(N^2)$ time toward completion where N is the total number of data points at any given time. This is mainly due the construction of similarity matrix and the KNN lists which involves quadratic time complexity. *Batch-Inc*1 provides marginal improvement by building the updated KNN lists incrementally in $\mathcal{O}(N)$ time. However, building the K-SNN$_{updated}$ graph involves quadratic time complexity. *Batch-Inc*2 aims to address this issue by reconstructing the K-$_{updated}$ graph incrementally upon entry of new data points. While building the updated K-SNN graph, *Batch-Inc*2 only updates the shared strong link strength of $KN - S_{add}$ and S_{add} type points leaving rest of the points. For identifying the new core and non-core points, *Batch-Inc*2 involves all the data points in D'(updated dataset) to detect the dense and non-dense points. This results in *Batch-Inc*2 running in linear time.

$BISDB_{add}$ identifies the core and non-core type points incrementally and therefore improves upon the previous two methods. $BISDB_{add}$ also runs in linear time. We shall provide a detailed time complexity analysis of the $BISDB_{add}$ method in Sect. 8. Please refer to Algorithm 1 for the pseudo-code representation of the $BISDB_{add}$ algorithm. In the next section, we present our incremental algorithms for deletion of data points in batch mode.

Algorithm 1: The $BISDB_{add}$ algorithm (pseudo-code)

Initialize parameters K, δ_{sim}, δ_{core};
// Set *nrow* as the total no. of data points after increment of *nrow2* points upon *nrow1*
$nrow \leftarrow nrow1 + nrow2$;
// Update dataset after increment
for $i1 \leftarrow 1$ **to** $nrow2$ **do**
    Append new data point $i1$ to base dataset data_matrix[];
    $i \leftarrow$ i+1;
// Find KNN list of new points
for $i \leftarrow 1$ **to** $nrow1$ **do**
    for $i1 \leftarrow nrow1$ **to** $nrow$ **do**
        Compute the distance between data points $i1$ and i ;
        if *i <= K-1* **then**
            Insert data point i to the KNN list of data point $i1$;
        else
            Insert data point i next to the KNN list of data point $i1$;
            sort($KNN\_matrix[i1]$);
            $KNN\_matrix[i1].pop()$;
// Find points that can be potentially affected
for $i1 \leftarrow nrow1$ **to** $nrow$ **do**
    for $j1 \leftarrow nrow1$ **to** $nrow$ **do**
        if $distance(i1, j1|i1 \neq j1) <= distance(i1, KNN\_matrix[i1][K])$ **then**
            Insert data point k to the KNN list of data point i ;
            sort($KNN\_matrix[i1]$);
            $KNN\_matrix[i1].pop()$;
        else
            —
for $i \leftarrow 1$ **to** $nrow1$ **do**
    for $k \leftarrow nrow1$ **to** $nrow$ **do**
        Compute distance between i and k;
        if $distance(i, k|i \neq k) <= distance(i, KNN\_matrix[i][K])$ **then**
            Insert data point k to the KNN list of data point i ;
        else
            Do nothing;
// Identify $KN - S_{add}$ *and* S_{add} type affected points
for $i \leftarrow 1$ **to** $nrow$ **do**
    if $KNN\_list[i].size() > K$ **then**
        $i \in KN - S_{add}$ type points;
    else
        —
for *each* $i \in KN - S_{add}$ **do**
    for *each* $j \in KNN\_matrix[i] \cup$ *points displaced from* $KNN\_matrix[i]$ **do**
        if $j \notin KN - S_{add} \wedge j$ *is not a new point* **then**
            $j \in S_{add}$ type points;
        else
            —
// Construct the updated K-SNN graph and detect core, non-core points incrementally
for *each* $i \in KN - S_{add} \cup S_{add}$ **do**
    for *each* $j \in KNN\_matrix[i]$ **do**
        if $similarity(i, j) > \delta_{sim}$ **then**
            An edge is formed between pints i and j;
        else
            —
    if $similarity\_matrix[i].size() > \delta_{core}$ **then**
        $i \in CORE$ points set;
    else
        $i \in$ *Non-CORE* points set;
Cluster formation is similar to the SNNDB algorithm;
Repeat $BISDB_{add}$ for the next batch of entering points.;

7 Batch-incremental SNNDB clustering algorithms for deletion

In this section, we present three BISDB clustering algorithms for deletion: *Batch-Dec*1, *Batch-Dec*2 and $BISDB_{del}$. The goal of these algorithms is to determine clusters dynamically while points are deleted from the dataset in batches. Prior to executing each of the three variants for deletion, the non-incremental SNNDB [6] has to be executed. However, while executing the SNNDB method, we increase the window size of the KNN list for each data point. Each of the algorithms for deletion now maintains an additional space for $(w-1)$*K extra points in their expanded KNN(e-KNN) list, where $w \in \mathbf{N}$(set of natural numbers), $w \geq 2$ and K is the size of the original KNN list. The total size of e-KNN list for each data point turns out to be K+$(w-1)$*K or w*K. For a given point $p \in D$, each of the points in e-KNN(p) including the additional w*K points are placed in increasing order of their distance from p.

Contrary to the proposed three variants for addition, no new data point enters the original dataset D. Instead points are removed from D. As a result, the original KNN list for some of the data points may shrink in size. In order to maintain a valid size of original KNN list, e-KNN list becomes functional. In case of shrinkage, data points from the additional $(w-1)$*K space migrate to the top-K slots in the e-KNN list to fill the void.

The following set of information is obtained by executing the SNNDB method with respect to(wrt.) D (original dataset)

1. e-KNN list $\forall\, p \in D$ given as e-KNN(p).
2. Similarity matrix (K-SNN graph) for D given as $Sim\_Mat(D)$.
3. Set of core and non-core points wrt. D given as $Core(D)$ and $Non-Core(D)$.
4. Set of clusters C wrt. D.
5. Set of noise points or outliers wrt. D.

In the following subsections, we present the deletion algorithms in a step-wise manner. Individual steps contain a detailed description with necessary graphical illustration.

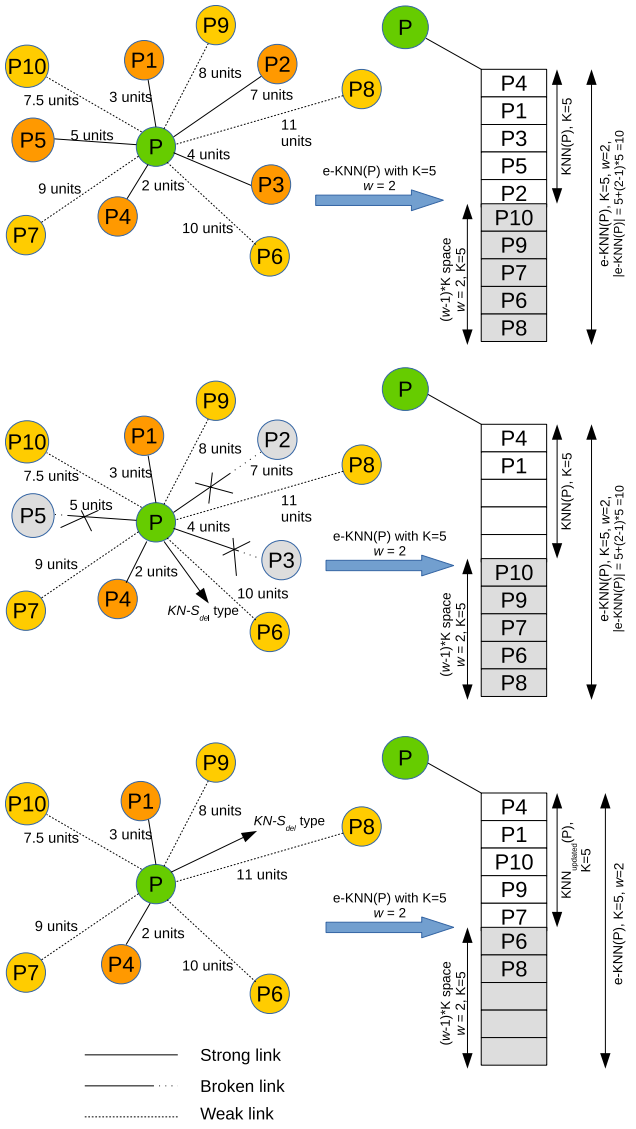

Fig. 6 The formation of $KN-S_{del}$ type affected points upon deletion of existing points

7.1 The *Batch-Dec*1 algorithm

The *Batch-Dec*1 algorithm builds the updated e-KNN list for each of the existing data points incrementally. The new similarity matrix(K-$SNN_{updated}$ graph) and the new core and non-core points are determined non-incrementally.

The steps of the *Batch-Dec*1 algorithm are as follows:

1. **Set the parameters** The algorithm takes four parameters: K, w, δ_{sim}, δ_{core}. The parameters have the following meanings:

   (a) K denotes the size of the original KNN list for each data point.

(b) w is the multiplier that decides the number of times the original KNN list to be expanded to form the e-KNN list.
(c) δ_{sim} is the minimum number of shared nearest neighbors between two data points p,q required to form a strong link between them given that p and q are in each others' KNN list.
(d) δ_{core} is the minimum number of adjacent strong links associated with a point p exceeding which p becomes a core point.

2. **Obtain the required data from prior SNNDB execution**

(a) Get the original dataset D where $|D| = x$(say).
(b) Get the e-KNN list $\forall\ p_i \in D, i = 1, 2, \ldots, x$.
(c) Get the similarity matrix ($Sim\_Mat(D)$).
(d) Delete a batch containing n existing data points from D. The size of D decreases and let the changed dataset be denoted as D' where $|D'| = x - n$.

3. **Remove the components associated with the deleted points**

(a) Remove the e-KNN list of the deleted points.
(b) Remove all the associated shared strong links with the deleted points from the K-SNN graph.
(c) The deleted points lose their cluster membership(if not a noise point).

4. **Compute the updated e-KNN list for existing data points** The number of existing points in D(original dataset) prior to any deletion is x. When n data points are removed from D, D changes to $D'(|D'| = x - n)$. The algorithm identifies among the remaining points in $D\prime$ that has removed any already deleted data point from its e-KNN list. While investigating, *Batch-Dec*1 only checks the top-K slots of the concerned point excluding the additional $(w - 1)$*K space. If the concerned point loses any data point from its top-K window due to prior deletion, it is categorized as a $KN - S_{del}$ type affected point. Consequently, the first additional point from the $(w - 1)$*K space migrates to fill the emptied slot in order to maintain the exact size(length K) of original KNN list.

The movement of a single point from the extra $(w - 1)$*K space in the e-KNN list prevents the shrinkage of top-K space in the original KNN list. In a worst case scenario, for any given point $p \in D'$(changed dataset), a maximum of K points might be removed from the top-K slots of e-KNN(p) due to prior deletion. In this case, the next set of K points placed in succession in the additional $(w - 1)$*K space($w \geq 2$), fills the emptied top-K slots. However, the migration creates vacant slot(s) within the additional $(w - 1)$*K space. These vacant slots are filled by other points in D' which are closer to p to rebuild e-KNN$_{updated}(p)$[4]. In this way, *Batch-Dec*1 avoids redundant e-KNN list computation for the unaffected points by focusing on only $KN - S_{del}$ type points.

Through Fig. 6, we illustrate this step(Step 4) from *Batch-Dec*1. We present this example by assuming $K = 5$ and $w = 2$. Consider the point P(topmost image from Fig. 6), for which we have the following prior to any deletion:

(a) KNN(P) = {P4,P1,P3,P5,P2}
(b) e-KNN(P) = {P4,P1,P3,P5,P2,P10,P9,P7,P6,P8}

With parameters w and K assuming values 2 and 5, respectively, the total size of e-KNN(P) is 10(since |e-KNN(.)| = K + $(w - 1)$*K). The first five points of e-KNN(P):{P4,P1,P3,P5,P2} are a part of KNN(P), while the remaining five points are stored in order to prevent shrinkage of KNN(P) in case of deletion.

Let P2,P5 and P3 be the deleted points (middle image from Fig. 6). As a result, these points are removed from KNN(P) and consequently from e-KNN(P). KNN(P) now shrinks in size and is reduced to points P4, P1. The strong links that P shared with P2,P5 and P3 are broken. P is therefore categorized as $KN - S_{del}$ type point.

As per *Batch-Dec*1, the size of additional $(w - 1)$*K space is 5 and it consists of points:{P10,P9,P7,P6,P8} placed in increasing order of distance from P. The removal of P2,P5 and P3 creates three vacant slots within KNN(P) (middle image from Fig. 6). As a result, points P10, P9 and P7 from the additional $(w - 1)$*K space migrate toward KNN(P) to overcome the shrinkage[5]. The updated KNN list and the current e-KNN list of P are given as:

(a) KNN$_{updated}$(P) = {P4,P1,P10,P9,P7}
(b) e-KNN(P) = {P4,P1,P10,P9,P7,P6,P8}

We observe that the additional $(w - 1)$*K space in e-KNN(P) currently consists of points P6 and P8 (bottom image from Fig. 6) and three empty slots. Prior to

[4] When no more points remain to prevent the shrinkage of top-K window due to further deletion, *Batch-Dec*1 involves entire D' to rebuild e-KNN$_{updated}(p)$.

[5] We assume that P does not share a strong link with P10, P9 and P7 yet they can be present in KNN$_{updated}$(P).

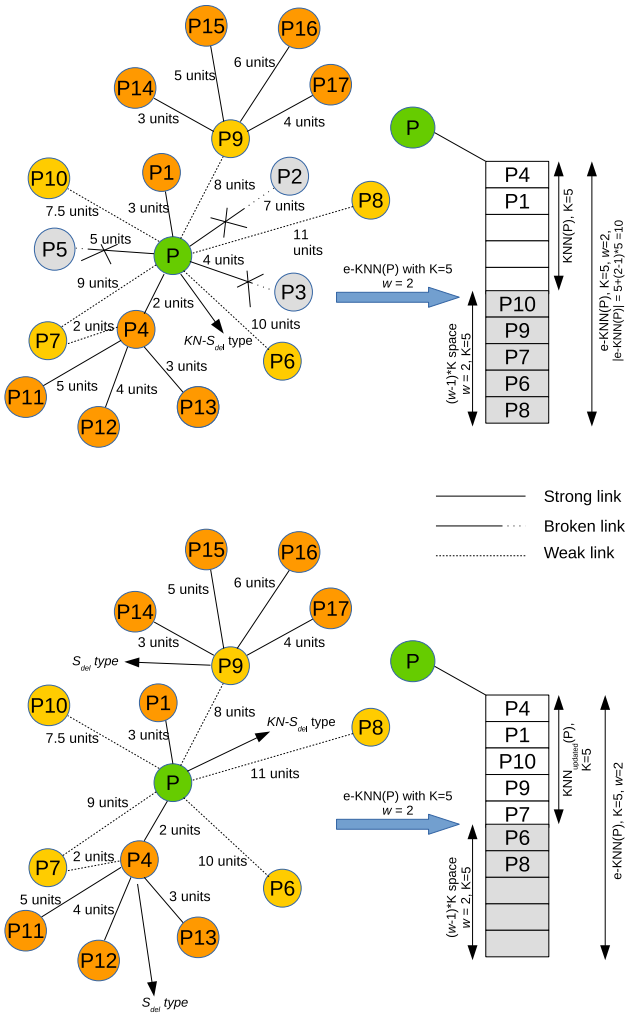

Fig. 7 The formation of S_{del} type affected points upon deletion

the next batch of deletion, *Batch-Dec*1 fills these empty slots in e-KNN(P) to produce e-KNN$_{updated}$(P). This is done in order to prevent shrinkage of KNN(P) in case of deletion from subsequent iterations.

5. **Construct the updated K-SNN graph** The algorithm constructs the K-SNN$_{updated}$ graph or the new similarity matrix ($Sim\_Mat(D')$) non-incrementally based on the remaining points in D'. D' now consists of $x - n$ points. Therefore, $\forall p_i \in D', i = 1, .., x - n$, *Batch-Dec*1 determines if a shared strong link can be constructed $\forall q_i \in$ KNN$_{updated}(p_i)$ where KNN$_{updated}(p_i) \subset$ e-KNN$_{updated}(p_i$).
6. **Identify new core and non-core points** $Sim\_Mat(D')$ represents the K-SNN$_{updated}$ graph. The algorithm scans through the K-SNN$_{updated}$ graph in order to identify points with sufficient number of strong links ($> \delta_{core}$) adjacent to it. Such points are designated as core (dense) points. The remaining points are non-core. The new set of core and non-core points are denoted as $Core(D')$ and $Non - Core(D')$.
7. **Form Clusters** Two core points are grouped into the same cluster. A non-core point is pulled toward the cluster of its nearest core point.
8. **Discard noise points** The non-core points which are not connected to any core point are classified as noise points. Such points do not obtain any cluster membership.
9. **Preserve the updated values**

   (a) $D = D'$
   (b) $x = x - n$
   (c) $\forall p_i, i = 1, 2, \ldots, x - n$
   e-KNN(p_i) = e-KNN$_{updated}(p_i)$.
   (d) $Sim\_Mat(D) = Sim\_Mat(D')$
   (e) $Core(D) = Core(D')$
   (f) $Non - Core(D) = Non - Core(D')$

10. Repeat Steps 1 to 9 for the next batch of deleted n points.

7.2 The *Batch-Dec*2 algorithm

The *Batch-Dec*2 algorithm constructs the updated e-KNN list and the new similarity matrix(K-SNN$_{updated}$ graph) incrementally. The new core and non-core points are determined non-incrementally. The steps of *Batch-Inc*2 algorithm are as follows:

1. **Set the parameters** The algorithm takes four parameters: K, w, δ_{sim}, δ_{core}. The parameters have the following meanings:

   (a) K denotes the size of the original KNN list for each data point.
   (b) w is the multiplier that decides the number of times the original KNN list to be expanded to form the e-KNN list.
   (c) δ_{sim} is the minimum number of shared nearest neighbors between two data points p,q required to form a strong link between them given that p and q are in each others' KNN list.
   (d) δ_{core} is the minimum number of adjacent strong links associated with a point p exceeding which p becomes a core point.

2. **Obtain the required data from prior SNNDB execution**

   (a) Get the original dataset D where $|D| = x$(say).
   (b) Get the e-KNN list $\forall$ p$_i \in D, i = 1, 2, \ldots, x$.

Table 5 Summary of the batch-incremental SNNDB clustering algorithms for addition/deletion

Components-Algorithm	*Batch-Inc*1/*Batch-Dec*1	*Batch-Inc*2/*Batch-Dec*2	$BISDB_{add}$/$BISDB_{del}$
Updated KNN list	Incrementally	Incrementally	Incrementally
Updated K-SNN graph	Non-Incrementally	Incrementally	Incrementally
Updated core and non-core points	Non-Incrementally	Non-Incrementally	Incrementally

(c) Get the similarity matrix (*Sim_Mat*(D)).
(d) Delete a batch containing n existing data points from D. The size of D decreases and let the changed dataset be denoted as D' where $|D'| = x\text{-}n$.

3. **Remove the components associated with the deleted points**

(a) Remove the e-KNN list of the deleted points.
(b) Remove all the associated shared strong links with the deleted points from the K-SNN graph.
(c) The deleted points lose their cluster membership(if not a noise point).

4. **Compute the updated e-KNN list for existing data points incrementally** Step 4 is similar to that of *Batch-Dec*1.
5. **Construct the updated K-SNN graph incrementally** In this step, reconstruction of $\text{K-SNN}_{updated}$ graph happens incrementally by building new similarity matrix(*Sim_Mat*(D')) after deletion of n data points. Similar to *Batch-Dec*1, the algorithm targets only the affected points while building the $\text{K-SNN}_{updated}$ graph. In addition to $KN - S_{del}$ type points, *Batch-Dec*2 introduces S_{del} type affected point. The S_{del} type points do not change their old KNN list upon deletion of data points. However, an S_{del} type point is a part of the updated KNN list of at least one $KN - S_{del}$ type point. The non-$KN - S_{del}$ type points which migrate from the additional $(w-1)$*K space into the top-K window of a $KN - S_{del}$ type point are also categorized as S_{del} type. For any S_{del} type point $p \in D'$(updated dataset), only the value of the shared strong link(edge weight) with p's adjacent points may change but KNN(p) remains the same. As a result, we have $\text{KNN}_{updated}(p)$ = KNN(p). Since KNN(P) $\subset$ e-KNN(P), $\text{e-KNN}_{updated}$ (p) = e-KNN(p). *Batch-Dec*2 therefore avoids rebuilding of updated e-KNN lists and $\text{K-SNN}_{updated}$ graph without involving D' in its entirety.

Figure 7 illustrates the formation of S_{del} type affected points. We present this example by assuming $K = 5$ and $w = 2$. Let us consider the point P(top image of Fig. 7) having {P4,P1,P3,P5,P2} in its original KNN list(KNN(P)) prior to any deletion. Let P2, P3 and P5 be the deleted points from KNN(P). Consequently, the strong link that P shared with each of the deleted points is broken resulting in three empty slots. P is therefore categorized as a $KN - S_{del}$ type point. The three vacant slots in KNN(P) are filled by points P10, P9 and P7, respectively. These three points were a part of the additional $(w-1)$*K space in e-KNN(P). The updated KNN list of P($\text{KNN}_{updated}$(P)) now comprises of {P4,P1,P10,P9,P7} (bottom image of Fig. 7).

For points P4 and P9 in Fig. 7, we make the following observations:

(a) $\text{KNN}_{updated}$(P9) = KNN(P9) = {P14,P17,P15,P16,P}
(b) $\text{KNN}_{updated}$(P4) = KNN(P4) = {P,P7,P13,P12,P11}

We observe that P4 and P9 have retained their original KNN list post-deletion. P4 was originally present in KNN(P) while P9 has moved toward KNN(P) from the additional $(w-1)$*K space. Since P is a $KN - S_{del}$ type point, therefore P4 and P9 qualify as S_{del} type points.

6. **Identify new core and non-core points** Step 6 is similar to *Batch-Dec*1.
7. **Form Clusters** Step 7 is similar to that of *Batch-Dec*1.
8. **Discard noise points** Step 8 is similar to that of *Batch-Dec*1.
9. **Preserve the updated values**

(a) $D = D'$
(b) $x = x - n$
(c) $\forall p_i, i = 1, 2, \ldots, x-n$ e-KNN(p_i) = $\text{e-KNN}_{updated}(p_i)$.
(d) *Sim_Mat*(D) = *Sim_Mat*(D')
(e) *Core*(D) = *Core*(D')
(f) *Non* − *Core*(D) = *Non* − *Core*(D')

10. Repeat Steps 1 to 9 for the next batch of deleted n points.

7.3 The $BISDB_{del}$ algorithm

The $BISDB_{add}$ algorithm computes the updated KNN list, the updated K-SNN graph, the new core and non-core points incrementally aiming to improve upon *Batch-Dec*1 and *Batch-Dec*2. The steps of the $BISDB_{add}$ algorithm are as follows:

1. **Set the parameters** The algorithm takes four parameters: K, w, δ_{sim}, δ_{core}. The parameters have the following meanings:
   (a) K denotes the size of the original KNN list for each data point.
   (b) w is the multiplier that decides the number of times the original KNN list to be expanded to form the e-KNN list.
   (c) δ_{sim} is the minimum number of shared nearest neighbors between two data points p,q required to form a strong link between them given that p and q are in each others' KNN list.
   (d) δ_{core} is the minimum number of adjacent strong links associated with a point p exceeding which p becomes a core point.
2. **Obtain the required data from prior SNNDB execution**
   (a) Get the original dataset D where $|D| = x$(say).
   (b) Get the e-KNN list $\forall\ \mathrm{p}_i \in D, i = 1, 2, \ldots, x$.
   (c) Get the similarity matrix ($Sim\_Mat(D)$).
   (d) Get the set of core ($Core(D)$) and non-core($Non - Core(D)$) points.
   (e) Delete a batch containing n existing data points from D. The size of D decreases and let the changed dataset be denoted as D' where $|D'| = x$-n.
3. **Remove the components associated with the deleted points**
   (a) Remove the e-KNN list of the deleted points.
   (b) Remove all the associated shared strong links with the deleted points from the K-SNN graph.
   (c) The deleted points lose their cluster membership(if not a noise point).
4. **Compute the updated e-KNN list for existing data points incrementally** Step 4 is similar to that of *Batch-Dec*1 and *Batch-Dec*2.
5. **Construct the updated K-SNN graph incrementally** Step 5 is similar to that of *Batch-Dec*2.
6. **Identify new core and non-core points** In the K-SNN$_{updated}$ graph, for each of the total number of $KN - S_{del}$ and S_{del} type points, $BISDB_{del}$ checks whether its number of adjacent strong links exceeds δ_{core}. If this happens, the concerned point is treated as a core point or else it is a non-core point. The remaining points retain their existing core or non-core status from the previous iteration.
7. **Form Clusters** Step 7 is similar to that of *Batch-Dec*1 and *Batch-Dec*2.
8. **Discard noise points** Step 8 is similar to that of *Batch-Dec*1 and *Batch-Dec*2.
9. **Preserve the updated values**
   (a) $D = D'$
   (b) $x = x - n$
   (c) $\forall p_i, i = 1, 2, \ldots, x - n$ e-KNN(p_i) = e-KNN$_{updated}(p_i)$.
   (d) $Sim\_Mat(D) = Sim\_Mat(D')$
   (e) $Core(D) = Core(D')$
   (f) $Non - Core(D) = Non - Core(D')$
10. Repeat Steps 1 to 9 for the next batch of deleted n points.

7.4 Summary: batch-incremental SNNDB clustering algorithms for deletion

In this section, we presented three incremental clustering algorithms for deletion: *Batch-Dec*1, *Batch-Dec*2 and $BISDB_{del}$. These three incremental variants for deletion focuses on reducing the time taken to compute the individual components of SNNDB: KNN list, K-SNN graph(similarity matrix), core and non-core points while points are removed from the dataset in batches.

Existing incremental extension to InSDB [21] fails to handle deletion. In order to address this issue, we initially propose *Batch-Dec*1. *Batch-Dec*1 builds the updated KNN list of all the data points incrementally by targeting only the $KN - S_{del}$ type affected points. *Batch-Dec*2 reconstructs both the updated KNN lists and the K-SNN$_{updated}$ graph incrementally. The third method $BISDB_{add}$ computes all the three components which includes detection of core and non-core points incrementally.

*Batch-Dec*1 provides a marginal improvement by building the updated KNN lists incrementally in $\mathcal{O}(N)$ time where N is the total number of data points at any given time. However, building the K-SNN$_{updated}$ graph involves quadratic time complexity. While reconstructing the K-SNN$_{updated}$ graph, *Batch-Dec*2 only updates the shared strong link strength of $KN - S_{del}$ and S_{del} type points leaving rest of the points. For identifying the new core and non-core points, *Batch-Dec*2 conducts a complete scan of all the data points in D'(updated dataset). This results in *Batch-Dec*2 running in linear time.

$BISDB_{del}$ identifies the core and non-core points incrementally and therefore improves upon the previous two methods. $BISDB_{del}$ also runs in linear time. We shall provide a detailed time complexity analysis of the $BISDB_{del}$ method in Sect. 8. Please refer to Algorithm 2 for the pseudo-code representation of the $BISDB_{del}$ algorithm. Table 5 presents a brief summary of the proposed three incremental variants of the SNNDB method for addition/deletion.

Algorithm 2: The $BISDB_{del}$ algorithm (pseudo-code)

```
Initialize parameters K, δ_sim, δ_core, w;
Set 'nrow' as original no. of data points;
// Update dataset after decrement
for i ← 1 to n do
    Remove existing data point i from the base dataset data_matrix[];
    i ← i+1;
// Update KNN list of the affected points from existing dataset
for i ← 1 to nrow do
    for each j ∈ KNN_matrix[i] do
        if j is a deleted point then
            Remove j from KNN_matrix[i] and shrink KNN(i);
        else
// Identify KN − S_del type points
for i ← 1 to nrow do
    if KNN_matrix[i].size() < K ∧ i ∉ Deleted_Set then
        i ∈ KN − S_del points;
    else
// Fill the empty slots in e-KNN list of KN − S_del type points
for each i ∈ KN − S_del do
    for each j ∈ KNN_matrix[i] ∧ j is empty do
        Select a point closest to i from (w-1)*K space and fill the empty slot in KNN_matrix[i];
    Fill the empty slots of (w-1)*K space with other points in data_matrix[];
    sort(e-KNN(i));
// Identify S_del type points
for each i ∈ KN − S_del do
    for each j ∈ KNN_matrix[i] do
        if j ∉ KN − S_del then
            j ∈ S_del points;
        else
// Construct the updated K-SNN graph and detect core, non-core points incrementally
for each i ∈ KN − S_del ∪ S_del do
    for each j ∈ KNN _matrix[i] do
        if similarity(i, j) > δ_sim then
            An edge is formed between pints i and j;
        else
    if similarity_matrix[i].size() > δ_core then
        i ∈ CORE points set;
    else
        i ∈ Non-CORE points set;
Cluster formation is similar to the SNNDB algorithm;
Repeat BISDB_del for the next batch of removed points.;
```

8 Complexity analysis

In this section, we briefly discuss the time complexity of $BISDB_{add}$ and $BISDB_{del}$ algorithms. Prior to presenting the running time analysis of the batch-incremental methods, we must note that the non-incremental SNNDB [6] algorithm has a time complexity of $\mathcal{O}(N^2)$ for an input data of size N. The quadratic time complexity of SNNDB algorithm is mainly due to its pairwise similarity calculation between data points while constructing the SNN graph.

For batch-incremental methods, we assume that the size of new updates is smaller as compared to the size of the base dataset. Next, we present the running time analysis of $BISDB_{add}$. (Refer Pseudo-code 1): $BISDB_{add}$ runs in linear time. Let us assume that a total of B batches are inserted with k points per batch. The size of each batch is

significantly smaller than the size of the base dataset. As a result, we have $k << N$. Let D be the base dataset where $|D| = n(n \in \mathbf{Z}^+)$ and D' be the updated dataset after new insertions. For Bth batch arrival at any point in time, we have $|D'| = n + kB = N(say)$. Next, we analyze the running time $BISDB_{add}$ by identifying the complexity of its subcomponents.

[Line 9–17]: While updating the KNN list dynamically, the inner loop varies through total number of newly added points in a single batch(k). The inner loop finds the KNN for each of the k newly added points. The time taken to compute the distance between a newly added point and an existing point is $\mathcal{O}(1)$. Initially, for each iteration, a running time of $\mathcal{O}(k^2)$ is incurred. Since $k << N$, k^2 may be treated as a constant. However, we cannot guarantee that the current set of points which occupy the KNN window for each of the new points are its actual top-K points. This is because the current top-K slots in KNN list for the new points are filled by k old data points. In order to detect the actual KNN list for a newly entered point, $BISDB_{add}$ scans through the remaining $(N - k)$ old data points along with $(k - 1)$ newly entered points. With K being the size of KNN list, K$<<N$, the KNN list of the newly added points is constructed in linear time.

[Line 18–32]: For computing the updated KNN list of the older points, $BISDB_{add}$ identifies each point $p \in D$ that are affected by entry of new points $q \in (D' - D)$. The affected points update their new KNN list while rest of the old points retain their previous KNN list. This particular operation incurs a running time of $\mathcal{O}((N - k) * k)$.

[Line 33–42]: The affected points which alter their KNN list are the $KN - S_{add}$ points. $BISDB_{add}$ identifies the S_{add} type points from the KNN list of $KN - S_{add}$ points. While scanning the $KN - S_{add}$ points, at most K number of comparisons are done to determine the S_{add} points. This operation takes $2K*\mathcal{O}(|KN - S_{add}|) \simeq \mathcal{O}(1)$ time given that both K and $KN - S_{add}$ are constants and are considerably smaller than $|D'|$.

[Line 43–53]: The shared nearest neighbor graph is constructed by targeting the $KN - S_{add}$ and S_{add} type points while involving them to create new links or break the existing ones. This segment of $BISDB_{add}$ has a running time of $O((|KN - S_{add}|+|S_{add}|)*2K(K+c' \log_2 K) \simeq \mathcal{O}(1)$ since $|KN - S_{add}|, |S_{add}|, c'$ and K are small constant entities. The detection of core and non-core points happens in constant time while clusters are extracted in linear time.

$BISDB_{del}$(Refer Pseudo-code 2): $BISDB_{del}$ has a linear time complexity. Let us assume that a total of B batches were deleted with k points per batch. Let D be the base dataset where $|D| = n(n \in \mathbf{Z}^+)$ and D' be the updated dataset after new deletions. For B^{th} batch removal at any point in time, we have $|D'| = (n - k) * B = N(say)$.

[Line 8–13]: This segment of $BISDB_{del}$ updates the KNN list of data points dynamically. We clearly observe that for each existing point $i \in D'$, the algorithm checks for the presence of an already deleted point in i's KNN list. For a single point, the algorithm performs at most K comparisons. Deleting a data point j from i's KNN list and replacing by its' immediate next point or a point present in the additional $(w - 1)$*K space of e-KNN(i) list takes $\mathcal{O}(1)$ time. As a result, the steps required to compute the updated KNN list of existing points has a worst case time complexity of $K*\mathcal{O}(n - k) \simeq \mathcal{O}(N)$. For determining the number of $KN - S_{del}$ type points a complete scan of the updated dataset(D') is made which takes $K*\mathcal{O}(n - k) \simeq \mathcal{O}(N)$ time. $BISDB_{del}$ identifies core and non-core points incrementally along with the dynamic reconstruction of SNN graph post-deletion identical to the $BISDB_{add}$ algorithm.

Next, we compare the time complexity proofs of the proposed batch-incremental algorithms and the SNNDB method.

9 Comparing time complexity proofs of SNNDB, $BISDB_{add}$ and $BISDB_{del}$

9.1 SNNDB [6]

Let T_{KNN} = time taken to compute the KNN lists of N data points in base dataset D, then

$$T_{KNN} = N(N - 1) \tag{2}$$

Let T_{Edge} = time taken to construct edge between any pair of points in D, then

$$T_{Edge} = 2c'[\log_2 K + \frac{K}{2}] \quad [c' \in \mathcal{R}^+, c' << N, K << N] \tag{3}$$

where K is the size of the KNN list. The term $2 * c' \log_2 K$ is the time taken to search the presence of both data points in each others' KNN list while $c'K$ time is needed to find the similarity measure between any pair of data points[6].

Let T_{K-SNN} = time taken to construct the K-SNN(K-sparsified SNN) graph.

$$\begin{aligned} &\therefore T_{K-SNN} = \frac{N^2}{2} * T_{Edge} \\ &\implies T_{K-SNN} = \frac{N^2}{2} * 2 * c' \left[\log_2 K + \frac{K}{2}\right] \\ &\implies T_{K-SNN} = N^2 * c' \left[\log_2 K + \frac{K}{2}\right] \end{aligned} \tag{4}$$

[6] The KNN list of any data point is sorted in increasing order of distances with its top-K neighboring points.

Table 6 Datasets description

Dataset	Size	#Attributes	Description
Mopsi12	13000	2	Search locations in Finland
5D	100000	5	Synthetic dataset
Birch3	100000	2	Gaussian clusters
KDDCup'04	60000	70	Identifying homologous proteins to native sequence
KDDCup'99	54000	41	Network intrusion detection data

Let $T_{core}, T_{non-core}$ be the time taken to identify core and non-core points.

$$\therefore T_{Core+Non-Core} = N \tag{5}$$

A linear scan of the dataset is sufficient to identify points with more than δ_{core} number of strong links adjacent to it.

Let $T_{Cluster}$ be time taken to extract clusters.

$$\therefore T_{Cluster} = N * K \tag{6}$$

Now if T_{SNNDB} be the total time taken by the non-incremental SNNDB [6] algorithm, then we obtain the following:

$$\begin{aligned} T_{SNNDB} &= T_{KNN} + T_{K-SNN} + T_{Core+Non-Core} \\ &\quad + T_{Cluster} \\ &= cN^2 + KN[K << N] \\ &\quad \left[c = 1 + c'\frac{K}{2} + c' \log_2 K\right] \\ &\quad [c', K \text{ are constants}, \therefore c \text{ is constant}] \\ \implies T_{SNNDB} &\simeq \mathcal{O}(N^2) \end{aligned} \tag{7}$$

The term $c' \log_2 K$ is used for searching the presence of a point in another point's KNN list. This operation is mandatory in order to validate the strong link formation criterion as discussed in the algorithm.

9.2 $BISDB_{add}$

Let n be the size of base dataset and k be the number of points added per batch. We deduce the running time of $BISDB_{add}$ when a single batch insertion has been made. Let N be the total number of points after a batch insertion ($\therefore N = n + k$).

$$\begin{aligned} T_{KNN} &= k(n + k - 1) + kn \\ &= k(N - 1) + k(N - k) \quad [\text{where } k << N] \\ &= k(2N - k - 1) \\ &= 2kN - k(k + 1) \end{aligned} \tag{8}$$

The term $k(n + k - 1)$ is required for filling the KNN lists of each k newly entered point. While the term kn is required to detect if any of the new points penetrate into the KNN list of old points. Effectively kn amount of time is required to find the $KN - S_{add}$ type points.

Let $T_{S_{add}}$ = time taken to find the S_{add} type points. Since S_{add} type points are determined from the updated KNN list of the $KN - S_{add}$ type points, we have the following equation:

$$\begin{aligned} T_{S_{add}} &= |KN - S_{add}| * 2K \\ &\quad [\text{where } |KN - S_{add}| << N] \end{aligned} \tag{9}$$

The term $2K$ is used because the displaced points from the updated KNN list of any $KN - S_{add}$ type point are also checked for their S_{add} status. At most K number of points can be displaced from the updated KNN list of a $KN - S_{add}$ point.

$$\begin{aligned} T_{K-SNN} &= (|KN - S_{add}| + |S_{add}|) * 2c'\left(\frac{K}{2} + \log_2 K\right) \\ &\quad [\text{where } |S_{add}| << N] \end{aligned} \tag{10}$$

For calculating T_{K-SNN}, only $KN - S_{add}$ and S_{add} type points are taken into account, while rest of the points remain untouched. The term $2c'(\frac{K}{2} + \log_2 K)$ depicts the time required to construct or break a possible strong link for a $KN - S_{add}$ or S_{add} type point with each data item in their updated KNN list.

$$T_{Core+Non-Core} = |KN - S_{add}| + |S_{add}| \tag{11}$$

For finding the core and non-core points incrementally, the algorithm checks only the $KN - S_{add}$ and S_{add} type points while the unaffected points retain their previous core or non-core property.

$$T_{Cluster} = K * N \tag{12}$$

Now if $T_{BISDB_{add}}$ be the total time taken by the $BISDB_{add}$ algorithm, then we obtain the following:

$$\begin{aligned} T_{BISDB_{add}} &= T_{KNN} + T_{K-SNN} + T_{Core+Non-Core} \\ &\quad + T_{Cluster} \\ &= N(K + 2k) - c_1 + c_2 \\ &\quad [\because k, K, |KN - S_{add}|, |S_{add}| \text{ are constants}] \\ &\quad [\therefore c_1, c_2 \text{ are constants}] \\ \implies T_{BISDB_{add}} &\simeq \mathcal{O}(N) \end{aligned} \tag{13}$$

Here $c_1 = k(k + 1)$ and $c_2 = (|KN - S_{add}| + |S_{add}|) * [1 + 2c'(\frac{K}{2} + \log_2 K)]$.

9.3 $BISDB_{del}$

Let n be the size of base dataset and k be the number of points deleted per batch. We deduce the running time of

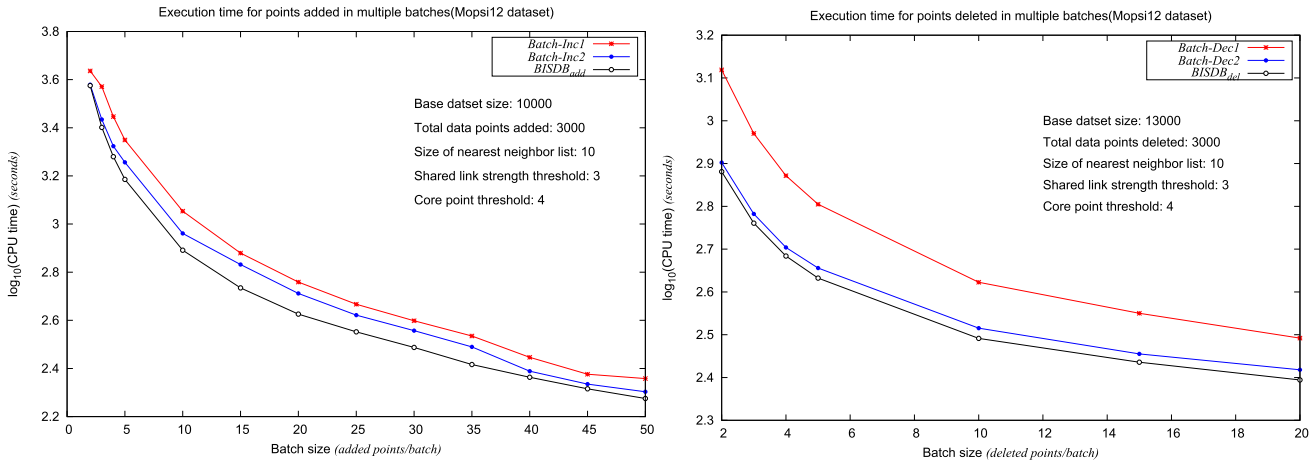

Fig. 8 Mopsi12 dataset: (Left) Efficiency comparison between batch-incremental addition algorithms; (Right) Efficiency comparison between batch-incremental deletion algorithms

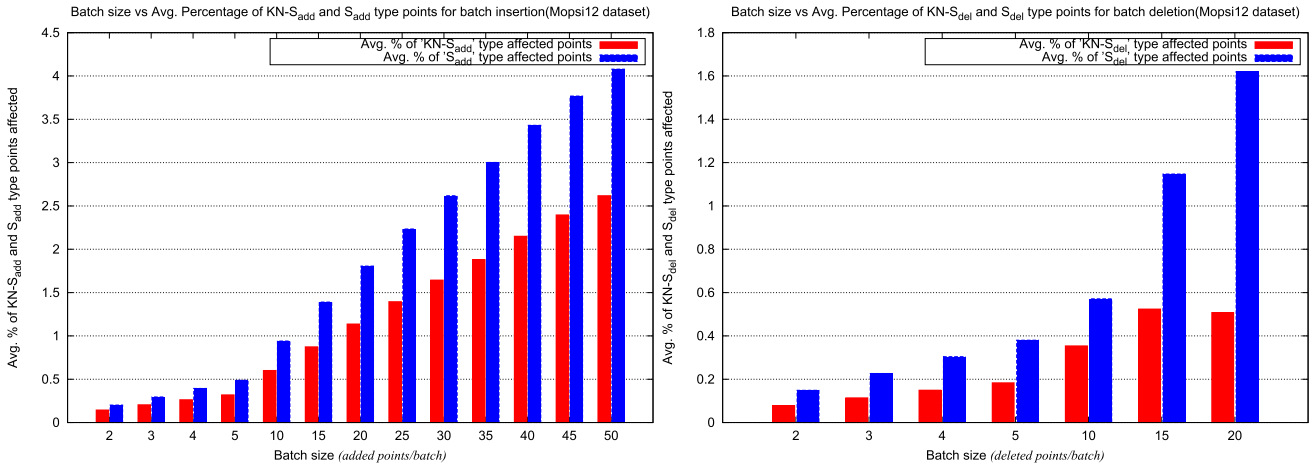

Fig. 9 Mopsi12 dataset: (Left) Avg. percentage of $KN - S_{add}$ and S_{add} type points for multiple batch insertion of varying batch size ($BISDB_{add}$); (Right) Avg. percentage of $KN - S_{del}$ and S_{del} type points for multiple batch deletion of varying batch size ($BISDB_{del}$)

$BISDB_{del}$ when a single batch deletion has been made. Let N be the total number of points after deletion of a batch ($\therefore N = n - k$). Let T_{e-KNN} be the time taken to compute the updated e-KNN list of existing data points in D'(updated dataset after deletion).

$$T_{e-KNN} = K * N \tag{14}$$

In the new e-KNN list of existing points, the top-K slots are checked for the presence of any deleted point. If an already removed point is found, it is replaced by the immediate next point. The vacant slot(s) are filled by the points in $(w-1) * K$ space. Effectively KN amount of time is required to find the $KN - S_{del}$ type points.

Let $T_{S_{del}}$ be the time taken to find the S_{del} type points. Since S_{del} type points are determined from the updated e-KNN list of $KN - S_{del}$ type points, we have the following equation:

$$T_{S_{del}} = |KN - S_{del}| * K[|KN - S_{del}| << N] \tag{15}$$

The term K is used instead of $2K$ because the removed points from the updated e-KNN list of any $KN - S_{del}$ type point are not taken into consideration.

$$T_{K-SNN} = (|KN - S_{del}| + |S_{del}|) * 2c'\left(\frac{K}{2} + \log_2 K\right) \quad [|S_{del}| << N] \tag{16}$$

$$T_{Core+Non-Core} = |KN - S_{del}| + |S_{del}| \tag{17}$$

$$T_{Cluster} = K * N \tag{18}$$

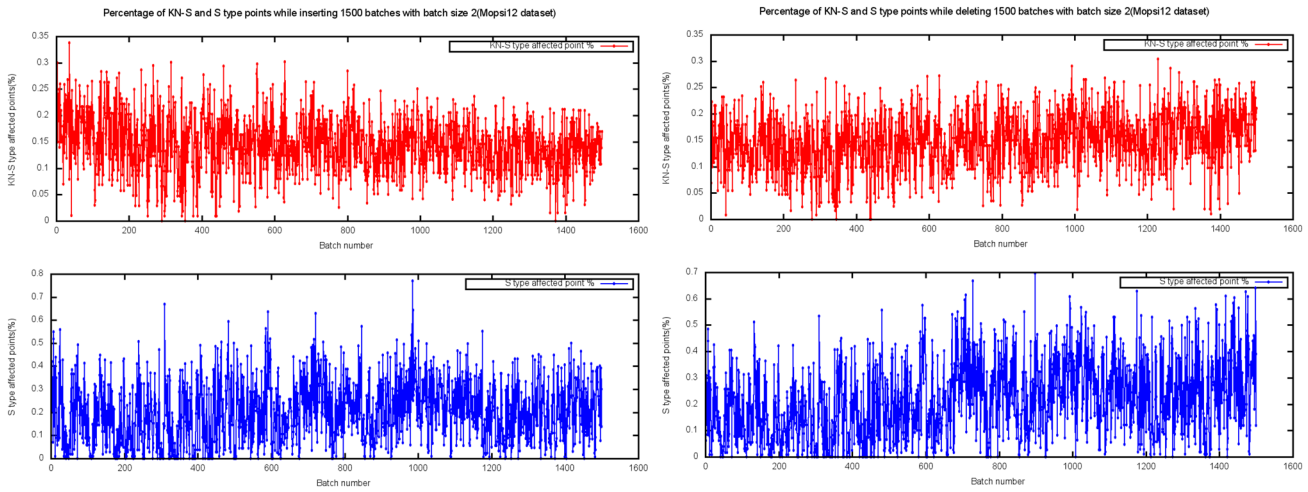

Fig. 10 Mopsi12 dataset: (Left) Percentage of $KN - S_{add}$ and S_{add} type points created while adding 1500 batches ($BISDB_{add}$); (Right)Percentage of $KN - S_{del}$ and S_{del} type points created while deletion of 1500 batches ($BISDB_{del}$)

Let $T_{BISDB_{del}}$ be the total time taken by the $BISDB_{del}$ algorithm, then we obtain the following:

$$\begin{aligned} T_{BISDB_{del}} &= T_{KNN} + T_{K-SNN} + T_{Core+Non-Core} \\ &\quad + T_{Cluster} \\ &= KN + c_1 \\ &\quad [K, KN - S_{del}, S_{del} \text{ are constants}] \\ &\quad [\therefore c_1 \text{ is constant}] \\ \implies T_{BISDB_{del}} &\simeq \mathcal{O}(N) \end{aligned} \tag{19}$$

Here $c_1 = (|KN - S_{del}| + |S_{del}|)[1 + 2c'(\frac{K}{2} + \log_2 K)]$

On comparing Eqs. 7, 13 and 19, we observe that $BISDB_{add}$ and $BISDB_{del}$ are more efficient than SNNDB in terms of CPU execution time.

In the next section, we describe the experimental procedures that were carried out in order to compare the efficiency of the proposed batch-incremental algorithms over point-based incremental methods and the SNNDB [6] algorithm.

10 Experimental evaluation

We performed experiments on real world and synthetic datasets to prove the efficiency of our batch-incremental algorithms over SNNDB [6], InSDB [21] and point-based deletion($BISDB_{del}$ with a batch size of one). We simulated our algorithms in C++ on a Linux platform(Intel(R) Xeon(R) CPU E5530 @ 2.40GHz) with 32GB RAM. The experiments were conducted in following phases.

1. Phase-1: Finding the most effective batch-incremental variant (addition/deletion).
2. Phase-2: Prove the efficiency of most effective batch-incremental variant(addition/deletion) over InSDB [21] and point-based deletion[7], respectively.
3. Phase-3: Show that InSDB [21] or point-based deletion becomes ineffective when larger updates (insertion or deletion of points) are made to the dataset.
4. Phase-4: Prove the efficiency of most effective batch-incremental variant(addition/deletion) over SNNDB [6].

10.1 Datasets

We used real world and synthetic datasets for evaluating our proposed algorithms. The real-world datasets used were Mopsi2012 https://cs.joensuu.fi/sipu/datasets/, KDDCup'99 http://archive.ics.uci.edu/ml/index.php and KDDCup'04 http://archive.ics.uci.edu/ml/index.php while the synthetic datasets were: 5D points set and Birch3 https://cs.joensuu.fi/sipu/datasets/. Please refer to Table 6 for a description of the datasets used along with their size and number of dimensions involved in the experiments.

Next, we describe the individual phases of the experimental evaluation. In each of these phases, we present a set of key observations and the reasons that lead to the obtainment of results.

[7] Point-based deletion signifies the execution of $BISDB_{del}$ (most effective batch-incremental deletion algorithm) with batches of size 1.

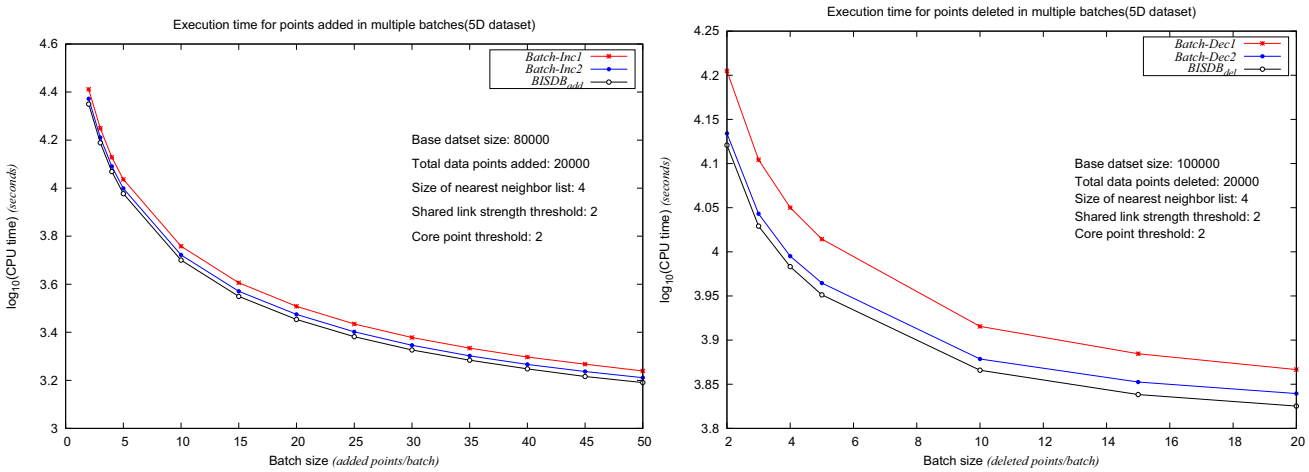

Fig. 11 5D dataset: (Left) Efficiency comparison between batch-incremental addition algorithms; (Right) Efficiency comparison between batch-incremental deletion algorithms

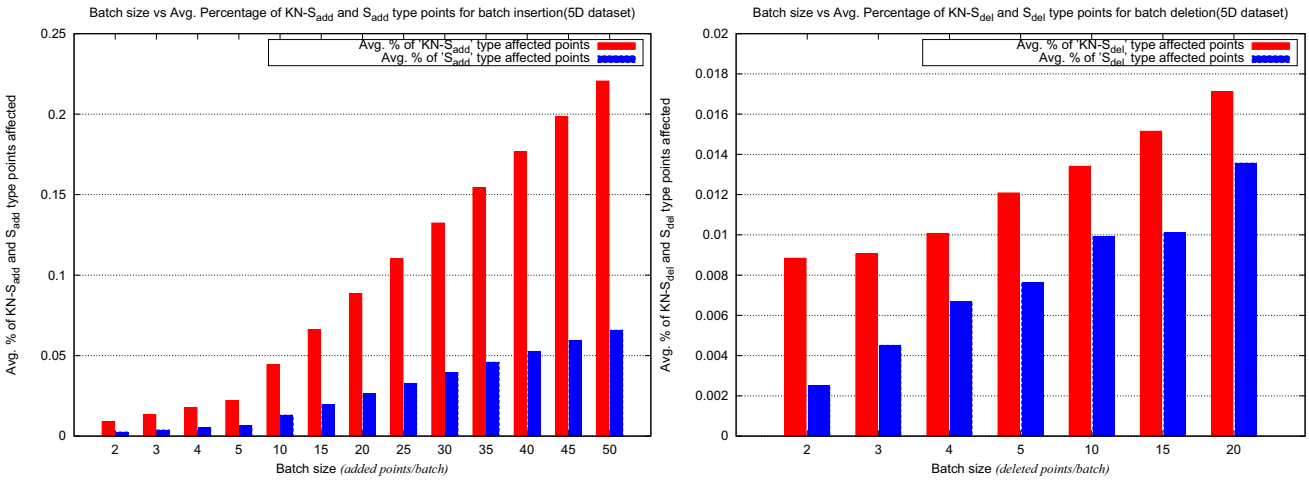

Fig. 12 5D dataset: (Left) Avg. percentage of $KN - S_{add}$ and S_{add} type points for multiple batch insertion of varying batch size($BISDB_{add}$); (Right)Avg. percentage of $KN - S_{del}$ and S_{del} type points for multiple batch deletion of varying batch size ($BISDB_{del}$)

10.2 Phase-1: Finding the most effective incremental variant

In Phase 1, we compared the efficiency for each of the three batch-incremental algorithms for addition and deletion independently to find the most effective variant. We adopted three datasets: Mopsi2012, 5D points set and Birch 3 to conduct the experiments in this phase. For experimental purpose, we defined a new term called Algorithm-Components (AlgoComp). AlgoComp consisted of following components:

- Base dataset
- KNN lists
- Similarity matrix
- Core and non-core points
- Clusters

A base dataset is one taking which SNNDB [6] is executed in order to set the values of other components in AlgoComp. Initially, the same base dataset is fed as input to the batch-incremental algorithms. After running the incremental algorithms (addition/deletion) for processing a certain batch of updates, the base dataset alters its size. Points are added to or deleted from the previous base dataset while producing clusters dynamically. The new set of points becomes the updated base dataset over which the next batch of incoming or outgoing points is processed. The new KNN lists, similarity matrix (SNN graph), core and non-core points along with clusters become a part of the updated AlgoComp.

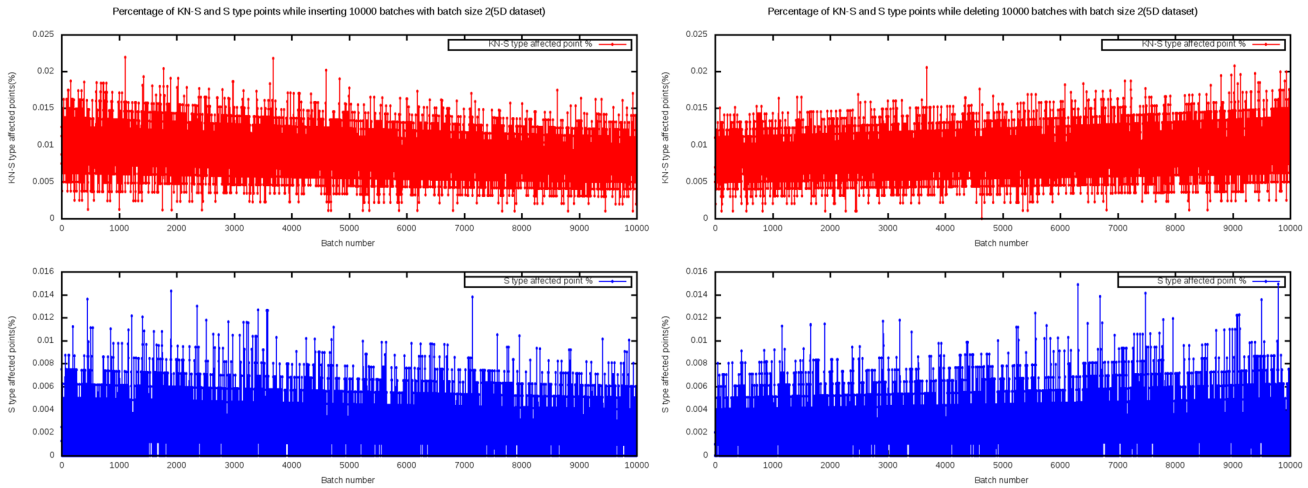

Fig. 13 5D dataset: (Left) Percentage of $KN - S_{add}$ and S_{add} type points created while adding 10000 batches ($BISDB_{add}$); (Right)Percentage of $KN - S_{del}$ and S_{del} type points created while deletion of 10000 batches ($BISDB_{del}$)

Next, we describe the experiments carried out on three individual datasets: Mopsi12, 5D, Birch3 for comparing the proposed batch-incremental clustering algorithms in both addition/deletion category. The comparisons for addition were independent from that of the deletion in order to decide the most effective variant in either category. The following are the batch-incremental algorithms for addition and deletion involved in Phase-1.

Addition: $Batch - Inc1, Batch - Inc2, BISDB_{add}$

Deletion: $Batch - Dec1, Batch - Dec2, BISDB_{del}$

10.2.1 Mopsi2012

Addition The values of parameters were taken as: K = 10, $\delta_{sim} = 3$ and $\delta_{core} = 4$. Initially, the base dataset size was taken to be 10000. We executed the SNNDB algorithm on the base dataset to set the AlgoComp values. We inserted 3000 points upon the base dataset in multiple batches. The minimum batch size was 2 and a maximum batch size was 50 processing 1500 and 60 batches, respectively. With the increase in batch size, the number of batches processed was reduced.

For example, for a batch of size 2, we executed each of the three incremental algorithms (addition) independently 1500 times to insert 3000 points. After every two points were inserted, the algorithms computed the new clusters incrementally using the pre-computed AlgoComp values from previous batch insertion. The final set of clusters that were produced after inserting 1500 batches were identical to those of the SNNDB method which processed 13000 points at once. The same set of clusters were also produced by InSDB after inserting 3000 points one at a time. Similarly, for a batch consisting 50 points, each of the three incremental variants (addition) processed 600 batches sequentially. Out of the three incremental addition algorithms, $BISDB_{add}$ performed most efficiently (Fig. 8). For various batch sizes (varying from 2 to 50), $BISDB_{add}$ maintained a better efficiency than the other two incremental variants for addition.

The incremental algorithms targeted only the $KN - S_{add}$ and S_{add} type points while retaining the properties of remaining points prior to any updates. This selective handling of data points is the main reason behind improving the efficiency of batch-incremental methods over SNNDB (as shown in Phase-4). We experimentally observed the percentage of $KN - S_{add}$ and S_{add} type affected points for the step where 1500 batches were inserted with a batch of size 2. A maximum of around 0.34% $KN - S_{add}$ type and about 0.76% of S_{add} type points were produced while processing batch number 36 and 985, respectively (Fig. 10). An insignificant percentage of $KN - S_{add}$ and S_{add} type affected points against every batch insertion meant that the incremental algorithms would outperform the SNNDB [6] method. We additionally observed that the mean percentage of $KN - S_{add}$ and S_{add} type points per batch reached a maximum of around 2.6% and 4%, respectively, for batches of size 50 while adding 600 batches sequentially (Fig. 9).

Deletion For deletion, the algorithms retained the same parameters while w is chosen as 2. We set the AlgoComp values by running SNNDB over 13000 points, and then deleted 3000 points in multiple batches incrementally. Similar to addition, we started by removing a minimum of 2 points/batch, and continued the experiments up to a batch of size 20. Each of the three deletion algorithms used the AlgoComp values before performing incremental deletion. $BISDB_{del}$ was the most effective deletion algorithm.

While identifying the affected points, we observed that a maximum of around 0.30% $KN - S_{del}$ type and about 0.69% of S_{del} type points were produced while processing

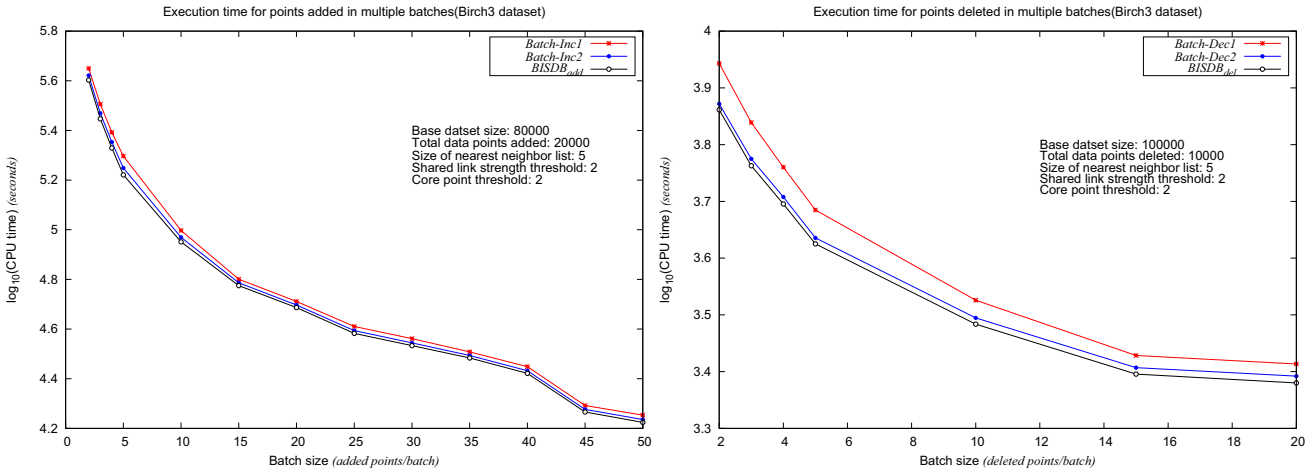

Fig. 14 Birch3 dataset: (Left) Efficiency comparison between batch-incremental addition algorithms; (Right) Efficiency comparison between batch-incremental deletion algorithms

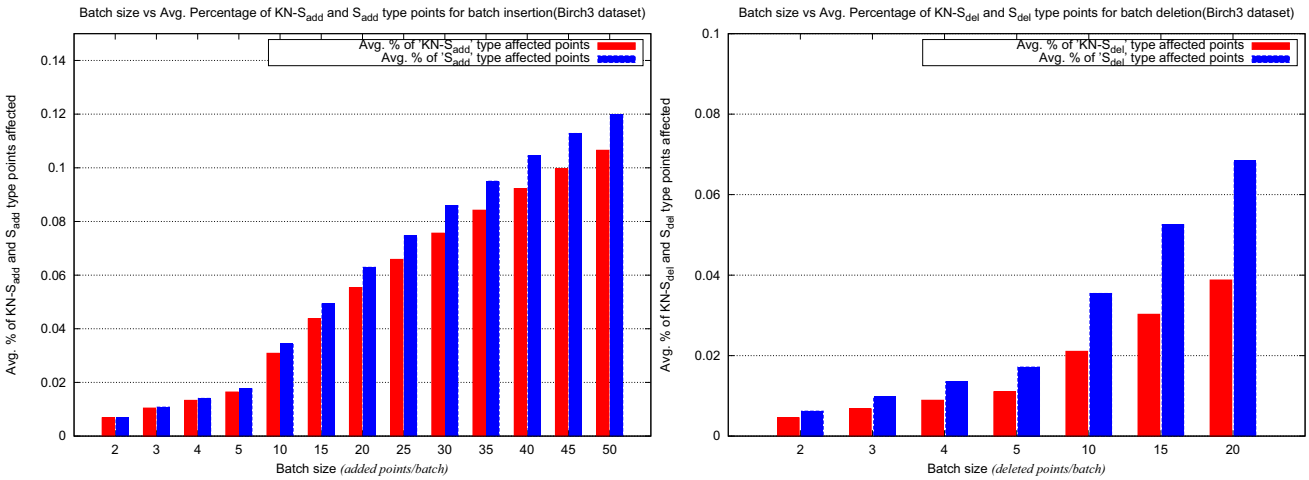

Fig. 15 Birch3 dataset: (Left) Avg. percentage of $KN - S_{add}$ and S_{add} type points for multiple batch insertion of varying batch size($BISDB_{add}$); (Right) Avg. percentage of $KN - S_{del}$ and S_{del} type points for multiple batch deletion of varying batch size ($BISDB_{del}$)

batch number 1229 and 898, respectively, out of a total of 1500 batches (Fig. 10). The mean percentage of $KN - S_{del}$ and S_{del} type points per batch reached only around 0.5% and 1.6%, respectively, for batches of size 20 while deleting 150 batches sequentially (Fig. 9).

List of key observations and analyzing the reasons (Addition/Deletion) for Mopsi12 dataset

- **Key Observation** CPU time for batch-incremental algorithms reduces with increasing batch size.

  Analysis/Reason(s) With an increase in batch size, the total number of batches to be processed decreases. The overall reconstruction time for SNN graph, KNN lists detecting core and non-core points incrementally reduces.

- **Key Observation** $BISDB_{add}$/$BISDB_{del}$ achieves the best efficiency.

  Analysis/Reason(s) Constructing KNN lists, SNN graph, finding core and non-core points incrementally.

- **Key Observation** Avg. percentage of $KN - S_{add}$/ $KN - S_{del}$ and S_{add}/S_{del} points increases with increasing batch size. However, the avg. percentage of $KN - S_{add}$ /$KN - S_{del}$ points is less than that of S_{add}/S_{del} points.

  Analysis/Reason(s) With an increase in batch size, the number of old points affected due to any batch insertion or deletion increases.

  Given any batch update, the number of points changing their previous KNN list is less than the those points which don't change their KNN list but contains at least one $KN - S_{add}/KN - S_{del}$ type point in their unchanged

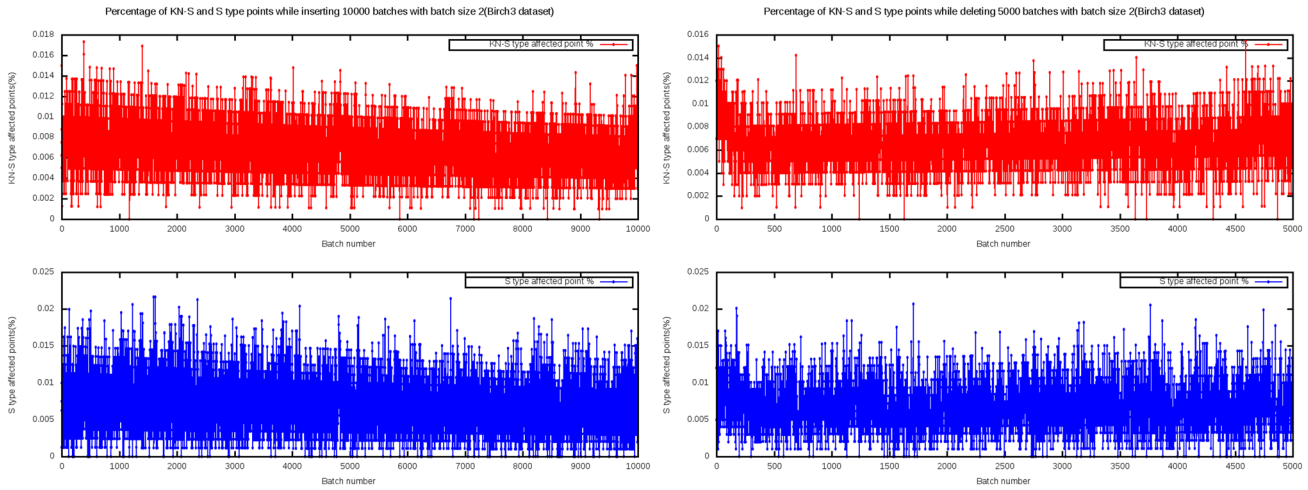

Fig. 16 Birch3 dataset: (Left) Percentage of $KN\text{-}S_{add}$ and S_{add} type points created while adding 10000 batches ($BISDB_{add}$); (Right)Percentage of $KN\text{-}S_{del}$ and S_{del} type points created while deletion of 5000 batches ($BISDB_{del}$)

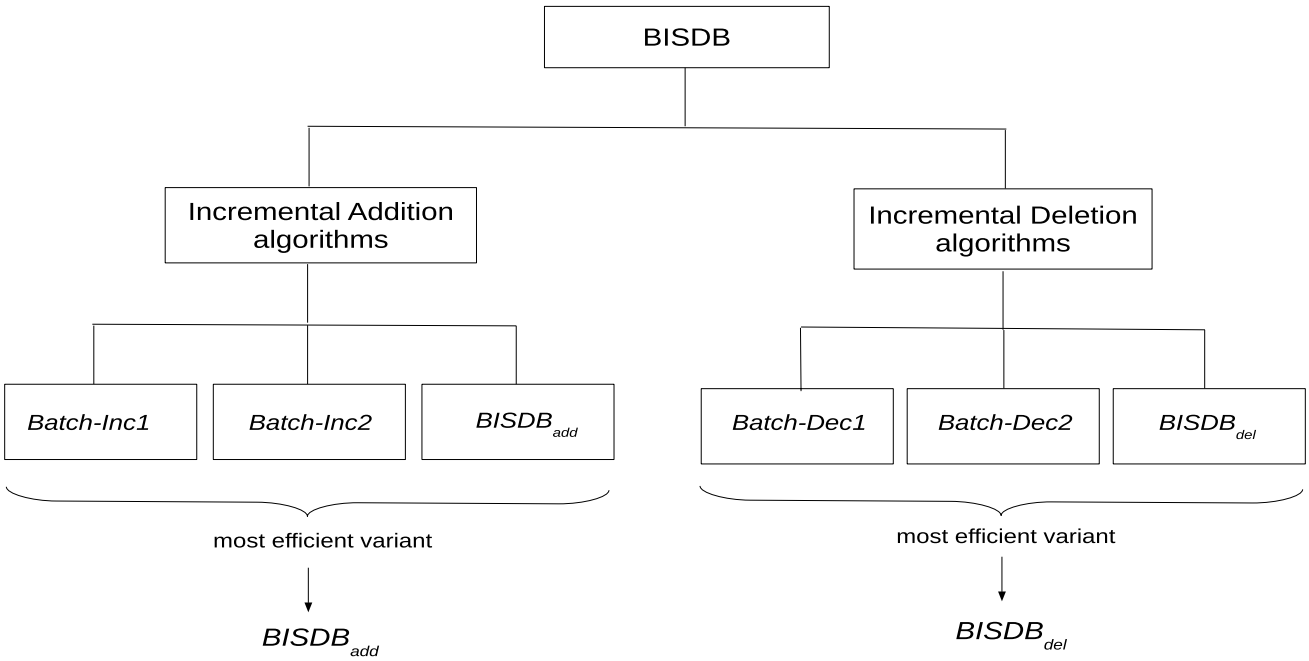

Fig. 17 Summary of Phase-1 experiments

KNN list. Therefore the avg. percentage of $KN - S_{add}$ $/KN - S_{del}$ points is less than that of S_{add}/S_{del} type points.

- **Key Observation** Percentage of $KN - S_{add}/KN - S_{del}$ and S_{add}/S_{del} points while processing 1500 batches (addition/deletion) is less than 1%.

  Analysis/Reason(s) The number of points altering their KNN list upon new insertion or deletion and the points forming new links or breaking existing ones due to any batch update is less than 1% of the number of points in updated base dataset.

10.2.2 5D points set

Addition The parameters for the addition algorithms were set as: K = 4, $\delta_{sim} = 2$ and $\delta_{core} = 2$. The base dataset consisted of 80000 points over which 20000 points were inserted. The AlgoComp values were computed by performing SNNDB on the base dataset. Initially, we added 2 points per batch while processing 10000 batches sequentially. Similarly, a maximum of 50 points per batch were added to process 400 batches. We observed that with increase in batch size the time required to process all the updates was reduced.

Table 7 Performance comparison of $BISDB_{add}$ and InSDB [21]

Dataset	Size	Base dataset size	Added	InSDB (s)	$BISDB_{add}$ 50 p/b (s)	Speedup ratio
Mopsi2012	13000	10000	3000	9289.24	188.50	49.27
5D points set	100000	80000	20000	44773.57	1551.82	28.85
Birch3	100000	80000	20000	681310.62	16751.17	40.67

Table 8 Performance comparison of $BISDB_{del}$ and pointwise deletion

Dataset	Base dataset size	Deleted	Remaining	$BISDB_{del}$ 1 p/b (s)	$BISDB_{del}$ 20 p/b (s)	Speedup ratio
Mopsi2012	13000	3000	10000	1360.02	247.88	5.48
5D points set	100000	20000	80000	19728.95	6686.86	2.95
Birch3	100000	10000	90000	12016.71	2399.10	5.01

Our observation illustrates the efficiency of $BISDB_{add}$ over the other two incremental methods for various batch insertions: *Batch-Inc1* and *Batch-Inc2* (Fig. 11).

While adding 10000 batches, a maximum of around 0.0218% of $KN - S_{add}$ type and about 0.0143% of S_{add} type affected points were identified while processing batch number 1104 and 1902, respectively (Fig. 13). The mean percentage of $KN - S_{add}$ and S_{add} type points per batch reached only around 0.22% and 0.06% for batches of size 20 (Fig. 12).

Deletion The deletions were performed with the same parameter values and w is set as 2. The size of the base dataset in case of deletion for 5D dataset was taken as 100000. We deleted 20000 points from the base dataset in multiple batches to evaluate our algorithms. While deleting points, we varied the batch size from 2 to 20. $BISDB_{del}$ performed most efficiently out of the three deletion algorithms. When the batch size was 2, $BISDB_{del}$ processed 10000 batches sequentially. A maximum of 0.0207% of $KN - S_{del}$ type points and about 0.0149% of S_{del} affected points were identified (Fig. 13) while processing batch number 9030 and 9790, respectively. The average percentage of $KN - S_{del}$ and S_{del} type points per batch was around 0.017% and 0.013%, respectively, for a batch of size 20 involving 10000 batches (Fig. 12).

List of key observations and reasons (Addition/Deletion) for 5D dataset

- **Key Observation** CPU time for batch-incremental algorithms reduces with increasing batch size.

  Analysis/Reason(s) With an increase in batch size, the total number of batches to be processed decreases. The overall reconstruction time for SNN graph, KNN lists, detecting core and non-core points incrementally reduces.
- **Key Observation** $BISDB_{add}$/$BISDB_{del}$ achieves the best efficiency.

  Analysis/Reason(s) Constructing KNN lists, SNN graph, detecting core and non-core points incrementally.
- **Key Observation** Avg. percentage of $KN - S_{add}$/$KN - S_{del}$ and S_{add}/S_{del} type points increases with increasing batch size. However, the avg. percentage of $KN - S_{add}$/$KN - S_{del}$ points is more than that of S_{add}/S_{del} points.

  Analysis/Reason(s) With an increase in batch size, the number of old points affected due to any batch insertion or deletion increases.

  Given any batch update, the number of points changing their previous KNN list is greater than the those points which don't change their KNN list but contains at least one $KN - S_{add}$/$KN - S_{del}$ type point in their unchanged KNN list. Therefore the avg. percentage of $KN - S_{add}$ /$KN - S_{del}$ points is more than that of S_{add}/S_{del} type points.
- **Key Observation** Percentage of $KN - S_{add}$/$KN - S_{del}$ and S_{add}/S_{del} points while processing 10000 batches (addition/deletion) is less than 0.05%.

  Analysis/Reason(s) The number of points altering their KNN list and forming or breaking new or existing links due to any batch update is less than 0.05% of the number of points in updated base dataset.

10.2.3 Birch3

Addition The parameters for the addition algorithms were set as: $K = 5$, $\delta_{sim} = 2$ and $\delta_{core} = 2$. The base dataset size was taken to be 80000. A total of 20000 points were added in multiple batches to extract the clusters dynamically. The batch size varied from 2 to 50 while computing the efficiency of each of the incremental addition algorithms. $BISDB_{add}$ proved to be the most efficient method out of the three addition variants (Fig. 14).

We computed the percentage of affected points while inserting 10000 batches with a batch of size 2. A

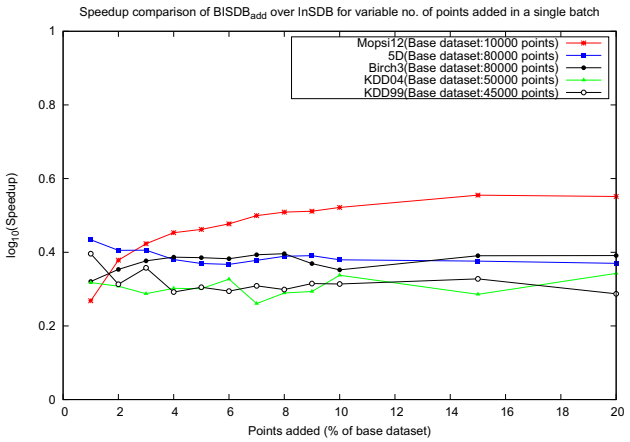

Fig. 18 Speedup comparison of $BISDB_{add}$ with InSDB for variable number of points added in a single batch

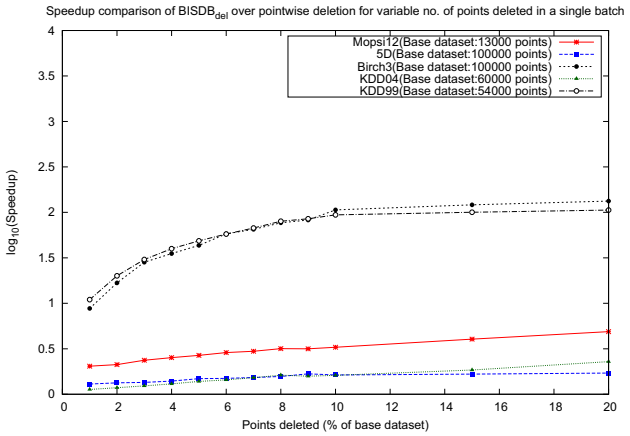

Fig. 19 Speedup comparison of $BISDB_{del}$ with point-based deletion for variable number of points deleted in a single batch

maximum of about 0.0173% of $KN - S_{add}$ type and about 0.0216% of S_{add} type affected points were identified (Fig. 16) while processing batch number 386 and 1599, respectively. The mean percentage of $KN - S_{add}$ and S_{add} type points per batch was found to be around 0.11% and 0.12%, respectively, for a batch of size 50 involving 400 batches (Fig. 15).

Deletion Retaining the same parameter values and choosing w as 2, we deleted 10000 points from a base dataset of size 100000. We varied the batch size from 2 to 20. For a batch of size 2, 10000 points were deleted in 5000 batches sequentially. $BISDB_{del}$ proved to be the most efficient method out of the three deletion variants. A maximum of around 0.0154% of $KN - S_{del}$ type points and 0.0207% of S_{del} type affected points were identified (Fig. 16) while processing batch number 4593 and 1707, respectively. The average percentage of $KN - S_{del}$ and S_{del} type points per batch was

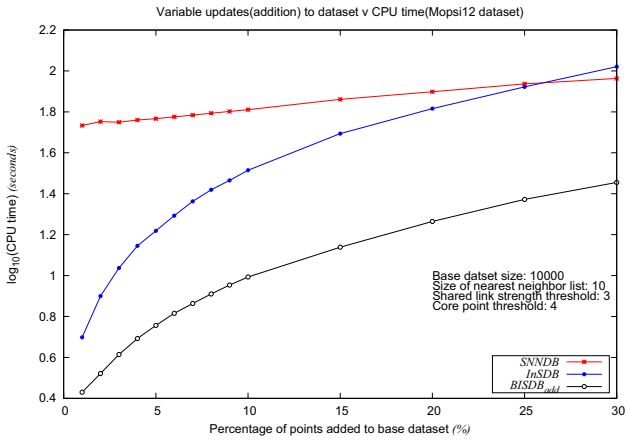

Fig. 20 Execution time of SNNDB, InSDB and $BISDB_{add}$ for variable updates (addition) in a single batch

around 0.04% and 0.07%, respectively, for a batch of size 20 involving 500 batches (Fig. 15).

List of key observations and reasons (Addition/Deletion) for Birch3 dataset

- **Key Observation** CPU time for batch-incremental algorithms reduces with increasing batch size.

  Analysis/Reason(s) With an increase in batch size, the total number of batches to be processed decreases. The overall reconstruction time for SNN graph, KNN lists detecting core and non-core points incrementally reduces.
- **Key Observation** $BISDB_{add}/BISDB_{del}$ achieves the best efficiency.

  Analysis/Reason(s) Constructing KNN lists, SNN graph, detecting core and non-core points incrementally.
- **Key Observation** Avg. percentage of $KN - S_{add}/KN - S_{del}$ and S_{add}/S_{del} type points increases with increasing batch size. However, the avg. percentage of $KN - S_{add}/KN - S_{del}$ points is less than that of S_{add}/S_{del} points.

  Analysis/Reason(s) With greater batch size, the number of existing points affected due to any batch insertion or deletion increases.

  Given any batch update, the number of points changing their previous KNN list is less than the those points which don't change their KNN list but contains at least one $KN - S_{add}/KN - S_{del}$ type point in their unchanged KNN list. Therefore the avg. percentage of $KN - S_{add}/KN - S_{del}$ points is less than that of S_{add}/S_{del} type points.
- **Key Observation** Percentage of $KN - S_{add}/KN - S_{del}$ and S_{add}/S_{del} points while processing 10000/5000 batches (addition /deletion) is less than 0.05%.

  Analysis/Reason(s) The number of points altering their KNN list and forming or breaking new or exist-

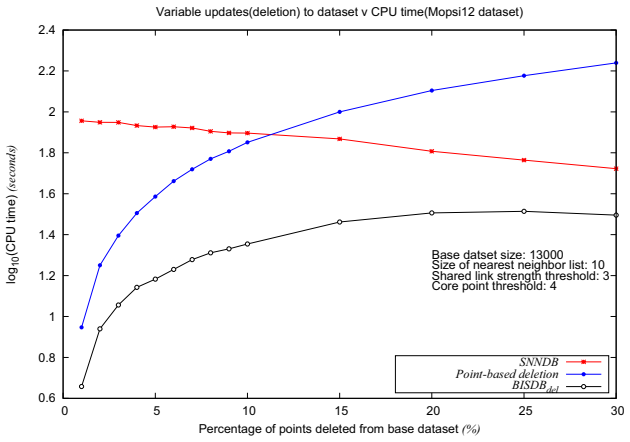

Fig. 21 Execution time of SNNDB, point-based deletion and $BISDB_{del}$ for variable updates (deletion) in a single batch

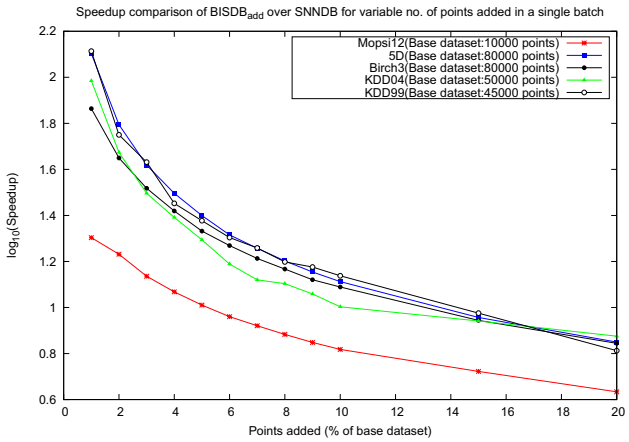

Fig. 22 Speedup of $BISDB_{add}$ over SNNDB for variable addition

ing links is less than 0.05% of the number of points in updated base dataset.

Inference drawn from Phase-1 experiments $BISD_{add}$ and $BISDB_{del}$ prove to be the two most efficient batch-incremental algorithms out of the three proposed variants in their respective category of addition and deletion (Fig. 17). These algorithms are able to detect clusters dynamically with minimal interference on the base dataset leading to greater efficiency. Also, we observed that for each dataset, the execution time of the batch-incremental algorithms (*addition* / *deletion*) follows a decreasing trend with an increase in batch size. This is because as the batch size increases, the number batches to be processed decreases. This means that the number of times reconstruction happens for K-SNN graph, KNN lists for points, identification of core and non-core points decreases.

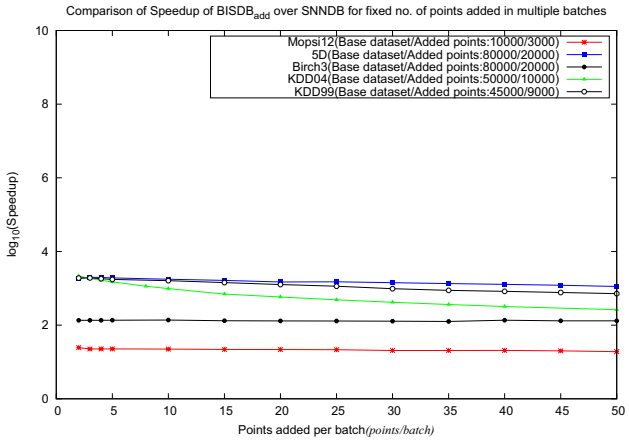

Fig. 23 Speedup of $BISDB_{add}$ over SNNDB for constant addition

10.3 Phase-2: Proving the efficiency of the most effective batch-incremental variant(addition/deletion) over InSDB [21] and point-based deletion

As shown in Phase 1, $BISDB_{add}$ and $BISDB_{del}$ are the two most efficient batch-incremental algorithms for addition and deletion. Phase-2 of experimental evaluation aims to achieve the following objectives:

1. $BISDB_{add}$ is more efficient than InSDB [21] for constant and variable updates made to the base dataset.
2. $BISDB_{del}$ is more efficient than point-based deletion for constant and variable updates made to the base dataset.[8]

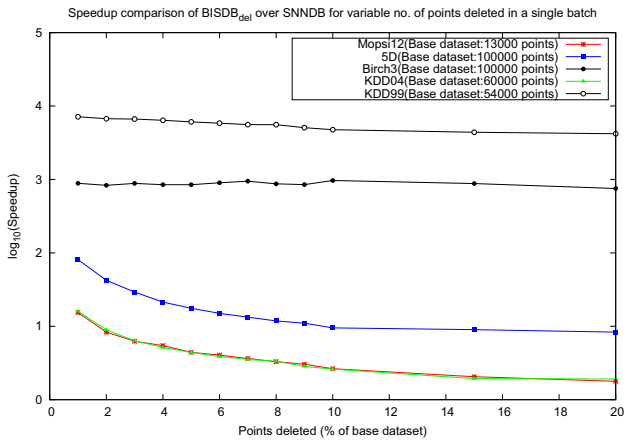

Fig. 24 Speedup of $BISDB_{del}$ over SNNDB for variable deletion

[8] We provide the detailed results pertaining to constant updates for $BISDB_{add}$ vs InSDB and $BISDB_{del}$ vs point-based deletion in the supplementary material "Online Resource.pdf" beyond this article.

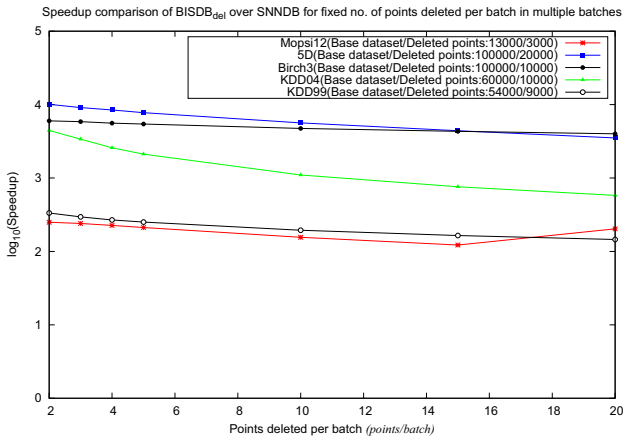

Fig. 25 Speedup of $BISDB_{del}$ over SNNDB for constant deletion

Constant updates In case of constant updates, a fixed number of points were inserted to or deleted from the base dataset in multiple batches. We executed $BISDB_{add}$/$BISDB_{del}$ to add or delete the requisite number of points in multiple batches of identical batch size. For $BISDB_{add}$, we compared its efficiency with InSDB [21] performing same number of point-based insertions while $BISDB_{del}$ was compared with point-based deletion method. Next, we provide some key observations and reasons based on the experimental results presented in Tables 7 and 8 for constant updates made to the dataset.

Key observation(s) In Table 7, we made a comparison between $BISDB_{add}$ and InSDB. $BISDB_{add}$ inserted 50 points per batch for the three datasets: Mopsi12, 5D and Birch3 to make up for the total number of points to be added. For each of the three datasets, the batch-incremental method proved to be more efficient than the point-based InSDB algorithm.

Analysis/Reason(s) In InSDB the data points are added one at a time. Although the clusters are detected incrementally, the construction of KNN lists and the SNN graph takes place after every point insertion. The detection of core and non-core points along with the clusters becomes repetitive of the number of points added. This makes the overall process slow contrary to $BISDB_{add}$ where the reconstruction of KNN lists, SNN graph and detection of core, non-core points and clusters is only repetitive of the number of batches.

Each time a new point is added, the affected points are determined by scanning the base dataset. The size of the base dataset increases after every insertion. As a result, the task of finding affected points from the base dataset becomes computationally intensive with every new insertion. If $T_{BISDB_{add}}$ and T_{InSDB} are the final CPU execution times after

Table 9 Cluster details of SNNDB for all datasets to compare with the incremental addition algorithms

Dataset	Algorithm	Input size	#Clusters	#Core points	#Non-core points	#Outliers
Mopsi12	SNNDB	13000	408	10533	1372	1095
5D		100000	2451	6332	4316	89352
Birch3		100000	11293	68653	19585	11762
KDD'99		54000	979	7086	3172	43742
KDD'04		60000	729	1486	3810	54704

Table 10 Cluster details of InSDB and $BISDB_{add}$ for all datasets

Dataset	Algorithm	Base dataset/ Added points	#Clusters	#Core points	#Non-core points	#Outliers
Mopsi12	$BISDB_{add}$/InSDB	10000/3000	408	10533	1372	1095
5D		80000/20000	2451	6332	4316	89352
Birch3		80000/20000	11293	68653	19585	11762
KDD'99		45000/9000	979	7086	3172	43742
KDD'04		50000/10000	729	1486	3810	54704

Table 11 Cluster details of SNNDB for all datasets to compare with the incremental deletion algorithms

Dataset	Algorithm	Input size	#Clusters	#Core points	#Non-core points	#Outliers
Mopsi12	SNNDB	10000	815	3999	1249	4752
5D		80000	807	1339	1023	77638
Birch3		90000	10302	61491	17639	10870
KDD'99		45000	428	2312	744	41944
KDD'04		50000	108	153	224	49623

Table 12 Cluster details of point-based deletion and $BISDB_{del}$ for all datasets

Dataset	Algorithm	Base dataset/ Deleted points	#Clusters	#Core points	#Non-core points	#Outliers
Mopsi12	$BISDB_{del}$/point-based deletion	13000/3000	815	3999	1249	4752
5D		100000/20000	807	1339	1023	77638
Birch3		100000/10000	10302	61491	17639	10870
KDD'99		54000/9000	428	2312	744	41944
KDD'04		60000/10000	108	153	224	49623

all the points are added, then experimentally we showed that the following relation holds (refer results of Table 7):

$$T_{BISDB_{add}} < T_{InSDB} \tag{20}$$

Key observation In Table 8, we made a comparison between $BISDB_{del}$ algorithm and pointwise deletion. The pointwise deletion method is nothing but executing $BISDB_{del}$ with batches of size 1. We observed that the $BISDB_{del}$ algorithm outperforms pointwise deletion method upon execution with batches of size 20.

Analysis/Reason(s) In pointwise deletion, the data points are deleted one at a time. The construction of KNN lists and the SNN graph happens after every deleted point. The detection of core and non-core points along with the clusters happen as many times as the number of points deleted. Although the size of base dataset keeps on decreasing, the time required to remove existing links, compute the altered weights of retained links and split clusters after every point removed makes the overall process slow as compared to $BISDB_{del}$. If $T_{BISDB_{del}}$ and $T_{point-del}$ are the final CPU execution times after requisite points have been deleted, then experimentally we showed that the following relation holds (refer results of Table 8):

$$T_{BISDB_{del}} < T_{point-del} \tag{21}$$

Variable updates A variable number of insertions or deletions were made to the base dataset in a single batch. Contrary to making constant updates, we executed $BISB_{add}$ /$BISB_{del}$ by making insertions or deletions ranging from 1 to 20% of the base dataset in one batch. Each time an update was inflicted, the efficiency of $BISB_{add}$ was compared with InSDB [21] while efficiency of $BISB_{del}$ was compared with that of point-based deletion. Figures 18 and 19 demonstrate the speedup comparison of the batch-incremental methods with the point-based methods for both addition and deletion.

Key Observation On measuring the speedup of $BISDB_{add}$ with InSDB (Fig. 18) and $BISDB_{del}$ with point-based deletion (Fig. 19), we observed a tendency of increasing speedup with an increase in percentage of updates (addition/deletion) being made to the dataset.

Analysis/Reason(s) In case of addition, with greater size of updates, the process of finding clusters dynamically becomes repetitive of the number of points inserted for InSDB. This makes InSDB slow as compared to the $BISDB_{add}$ algorithm, where updates are processed in batch mode. Consequentially, more the size of insertion, slower is the InSDB algorithm.

In case of deletion, as more points are deleted from the base dataset, the process of deleting old links, splitting clusters, finding new link strengths of existing ones happen as many times as there are total number of deletions. Due to these computations against each deletion, $BISDB_{del}$ scores on efficiency over point-based deletion as the percentage of update increases.

Inference drawn from Phase-2 experiments The efficiency of $BISD_{add}$ and $BISDB_{del}$ increases wrt. point-based incremental methods when the size of the batch update increases.

10.4 Phase-3: Non-incremental SNNDB is more effective than InSDB [21] or point-based deletion when large changes are made to the base dataset

In this phase, through our observations, we aim to establish the fact that when large changes are made to the dataset, the naive method and the batch-incremental algorithms outperform the point-based incremental technique. For illustration purpose, we used the Mopsi12 dataset to demonstrate this property.

We executed $BISDB_{add}$ and InSDB by taking a base dataset of 10000 points. For $BISDB_{add}$ we inserted points in a single batch upon the base dataset. The batch size varied from 1 to 30% of the base dataset. Corresponding to every batch addition the CPU execution time of the algorithms were compared.

For deletion, we executed $BISDB_{del}$ and point-based deletion($BISDB_{del}$ with batch size of one) by taking a base dataset of 13000 points. For $BISDB_{del}$ we deleted points in a single batch from the base dataset by varying the batch size from 1 to 30% of the base dataset.

We also executed the SNNDB [6] algorithm and compared its efficiency with the batch-incremental methods (addition/deletion) and point-based methods (InSDB /

point-based deletion). The total input size for SNNDB is a combination of base dataset and the added points in a batch or the remaining points in the base dataset after deletion.

From Fig. 20, we identify that both $BISDB_{add}$ and InSDB [21] maintain a better efficiency than SNNDB [6]. However, when the extent of insertion exceeded 25% of the base dataset, the non-incremental SNNDB started achieving a better efficiency than the point-based InSDB method. The $BISDB_{add}$ method consistently outperformed both the InSDB and SNNDB algorithm for insertions of all batch sizes.

In case of deletion (refer Fig. 21), SNNDB started achieving better efficiency than point-based deletion when around 11% points of base dataset had been deleted. The $BISDB_{del}$ method consistently outperformed both the point-based deletion and SNNDB for deletions of all batch sizes.

Inference drawn from Phase-3 experiments Point-based incremental algorithms fails to achieve better efficiency when larger changes are inflicted upon the base dataset. Experimental results in this phase have shown that along with batch-incremental methods even the non-incremental SNNDB outperformed InSDB and point-based deletion for larger updates.

10.5 Phase-4: Prove the efficiency of $BISDB_{add}$ and $BISDB_{del}$ over SNNDB [6]

We executed $BISD_{add}$ and $BISD_{del}$ while comparing their efficiency with SNNDB [6]. In this phase, both constant and variable updates were made to the base dataset. For constant updates, a fixed number of points were inserted to or deleted from the base dataset. The variable updates were made in a single batch with the batch size varying from 1 to 20% of the base dataset. We then measured the speedup of batch-incremental methods with the SNNDB algorithm.

Key observation(s) For both constant and variable updates, the speedup of $BISDB_{add}$ over SNNDB decreases with increasing variable updates being made to the base dataset (Figs. 22, 23).

Similarly, in case of deletion, $BISDB_{del}$ achieved a maximum speed up for smaller updates while the speed up gradually diminishes with an increase in size of batch deletion (Figs. 24, 25).

Analysis/Reason(s) While processing smaller updates, both $BISDB_{add}$ and $BISDB_{del}$ deal with insignificant percentage of $KN - S_{add}/KN - S_{del}$ and S_{add}/S_{del} type points. As the rest of the points retain their AlgoComp values post-new insertions or deletions, the time required to reconstruct KNN lists, SNN graph and detect clusters incrementally is very less.

The batch-incremental methods achieve the efficiency in CPU execution time over the SNNDB at the cost of memory overhead usage. For addition, $BISDB_{add}$ takes a maximum of about 60% more memory than SNNDB in case of 5D dataset while in case of deletion $BISDB_{del}$ consumes a maximum of 60% more memory than SNNDB for Mopsi12 dataset.

11 Cluster analysis

In this section, we present the details of clusters that were obtained after executing the naive method(SNNDB [6]), point-based methods (InSDB [21] and point-based deletion) and the batch-incremental algorithms ($BISDB_{add}$ and $BISDB_{del}$). We compared the number of clusters, core and non-core points along with the outliers that were obtained from executing respective algorithms. Tables 9 and 10 provide a comparison of clusters that were obtained after executing SNNDB and the addition incremental methods:$BISDB_{add}$ and InSDB. While Tables 11 and 12 compare the clustering results that were obtained after executing SNNDB and the deletion incremental methods: $BISDB_{del}$ and point-based deletion ($BISDB_{del}$ with 1 point/batch). Based on the tabular results, it is evident that the set of clusters obtained from running the naive method and the incremental methods are identical for both addition and deletion. Next, we analyze the correctness of clusters obtained from our proposed batch-incremental algorithms ($BISDB_{add}$ and $BISDB_{del}$) wrt. SNNDB [6], InSDB [21] or point-based deletion.

11.1 Proof of correctness for clusters obtained

We present the proof of correctness for the BISDBx clustering algorithms, where x is either *add* or *del* wrt. SNNDB [6], InSDB [21] or point-based deletion method ($BISDB_{del}$ with 1 point/batch) based on the following Lemmas.

Lemma 1 *Post k updates (insertion/deletion), the base dataset $D(|D| = n)$ changes to D'. $KNN(D')$ obtained from BISDBx, SNNDB* [6], *InSDB* [21] *or point-based deletion are identical.*

Proof After k^{th} update, only the affected points change their KNN lists for the BISDBx algorithms by retaining the top-K closest points. Similarly, if all the $n \pm k$ points are taken into consideration, only the top-K closest points would occupy appropriate positions $\forall x \in D'$. InSDB and point-based deletion adopt a similar method of KNN list computation post any update. □

Lemma 2 $\forall(x, y) \in D'$*(updated dataset), sim*(x, y) *obtained from BISDBx, SNNDB* [6], *InSDB* [21] *or point-based deletion are identical.*

Proof $\because$ KNN(D') post k updates are identical for each class of aforementioned algorithms (Lemma 1). Therefore, the number of shared data points between the KNN lists $\forall(x, y) \in D'$ for BISDBx, SNNDB, InSDB or point-based deletion algorithms remain identical. □

Lemma 3 $\forall x \in D'$, the $Core(x), Non-core(x)$ *obtained from BISDBx, SNNDB* [6], *InSDB* [21] *or point-based deletion are identical.*

Proof $\because \forall(x, y) \in D'$, $sim(x, y)$ is identical for each class of aforementioned algorithms (Lemma 2). As a result, the number of strong links adjacent to a point is also identical across these algorithms. Therefore, $\forall x \in D'$, x will retain its same core or non-core status for BISDBx, SNNDB, InSDB or point-based deletion. □

On the basis of above Lemmas, it can be concluded that the set of clusters obtained from BISDBx, SNNDB [6], InSDB [21] or point-based deletion are identical. Therefore we may infer the following observations.

$$C_{SNNDB} = C_{InSDB} = C_{BISDB_{add}} \tag{22}$$

$$C_{SNNDB} = C_{point-based\ deletion} = C_{BISDB_{del}} \tag{23}$$

where C represents the set of clusters for a given algorithm.

11.2 Clustering results in brief

The Mopsi12 dataset for addition with 13000 points consisted of 408 clusters. The largest cluster contains 1373 points while the average cluster size is 29.171. Around 81% of data are core points. As a result, 91.4% of data points obtain a cluster membership while rest are treated as noise points. Similarly, for deletion the Mopsi dataset contained about 40% core points. However, post-deletion in batch mode nearly 47.5% points remained as outliers. This is because deletion leads to splitting of clusters and simultaneous breaking of strong links between data points. As a result, most non-core data points fail to attach with a core point so as to obtain a cluster membership.

For 5D synthetic dataset, nearly 89.3% of points are outliers in case of addition while for deletion about 97% of points are categorized as noise points. This indicates the sparse distribution of the 5D dataset resulting in only a small fraction of clusters being generated. In Birch3 dataset (addition) about 88% of points obtain a cluster membership. The maximum cluster size is 35 while on an average a cluster contains about 7.81 data points. For deletion, about 87.9% of data points belong to a cluster with an average cluster size of 7.68.

In KDD'99 and KDD'04 dataset, a large share of data points constitute of outliers for both addition and deletion. For KDD'99 (addition), the maximum cluster size is 104 while for KDD'04 it is 18. For KDD'04(deletion) around 99% of points are outliers with a maximum cluster size of eight. The clusters obtained from the algorithms are parameter dependent. All the results that we provided in this paper are based on the parameter values set prior to execution and any change in the parameter values may alter the clustering results along with the share of core, non-core and noise points. The parameter values were specified while conducting experimental evaluation for individual datasets (Refer Sect. 10).

12 Discussion and conclusion

Incremental algorithms provide an effective solution to deal with dynamic datasets. In this article, we designed an incremental alternative to the SNNDB [6] clustering algorithm. InSDB [21] is an existing incremental extension of SNNDB. However, InSDB relies on point-based insertion which makes the process extremely slow when updates are made on larger base dataset. Moreover, InSDB does not handle deletion of data points. Also when the size of updates increases, InSDB fails to detect clusters efficiently as compared to SNNDB. This is a major flaw on the part of InSDB algorithm which acts as a motivation behind designing of batch-incremental algorithms handling both addition and deletion.

The SNNDB algorithm computes the KNN list, similarity matrix, core and non-core points while detecting clusters of arbitrary shapes, sizes and densities. In order to produce an incremental extension to SNNDB in batch mode, each of these components needs to be computed incrementally. In our first line of proposed batch-incremental methods for addition, we proposed $Batch-Inc1$ which finds the KNN list of the data points incrementally. Next, we developed $Batch-Inc2$ finding both KNN lists and SNN graph(similarity matrix) incrementally. However, $BISDB_{add}$ proved to be the most efficient of all as it computes KNN lists, similarity matrix, core and non-core points incrementally. Similarly, for deletion $Batch-Dec1$ and $Batch-Dec2$ corresponds to the first two batch-incremental extensions (addition) while $BISDB_{del}$ proved to be most effective out of the three batch-incremental deletion methods.

The incremental algorithms are designed to work for smaller updates to the base dataset. In this paper, we showed that for small updates the batch-incremental methods maintain a higher efficiency. The efficiency gradually diminishes with an increase in the size of data updates. We also observed that for variable updates made to the base dataset, the batch-incremental methods consistently maintain a better efficiency than both SNNDB and point-based incremental methods: InSDB and point-based deletion.

The set of clusters obtained by SNNDB and the incremental methods are identical. For each of the five datasets, we computed the clusters, core and non-core points along with the outliers. While Mopsi12 dataset involves a significant percentage of data points in formation of clusters, KDD'99 and KDD'04 consist mostly of outliers.

As each naive algorithm comes with its own components and techniques to deal with data, it may not be feasible to design a common incremental framework for all the density-based clustering algorithms. In future, we aim to provide incremental extensions to prominent density-based clustering methods especially those dealing with high dimensional clustering.

References

1. Aggarwal CC, Reddy CK (2013) Data clustering: algorithms and applications. CRC Press, Boca Raton
2. Berkhin P (2006) A survey of clustering data mining techniques. In: Kogan J, Nicholas C, Teboulle M (eds) Grouping multidimensional data. Springer, Berlin, Heidelberg, pp 25–71
3. Can F (1993) Incremental clustering for dynamic information processing. ACM Trans Inf Syst (TOIS) 11(2):143–164
4. Charikar M, Chekuri C, Feder T, Motwani R (2004) Incremental clustering and dynamic information retrieval. SIAM J Comput 33(6):1417–1440
5. Crespo F, Weber R (2005) A methodology for dynamic data mining based on fuzzy clustering. Fuzzy Sets Syst 150(2):267–284
6. Ertöz L, Steinbach M, Kumar V (2003) Finding clusters of different sizes, shapes, and densities in noisy, high dimensional data. In: Proceedings of the 2003 SIAM international conference on data mining. SIAM, pp 47–58
7. Ester M, Kriegel HP, Sander J, Xu X et al (1996) A density-based algorithm for discovering clusters in large spatial databases with noise. KDD 96:226–231
8. Ester M, Kriegel HP, Sander J, Wimmer M, Xu X (1998) Incremental clustering for mining in a data warehousing environment. VLDB, Citeseer 98:323–333
9. Fahad A, Alshatri N, Tari Z, Alamri A, Khalil I, Zomaya AY, Foufou S, Bouras A (2014) A survey of clustering algorithms for big data: taxonomy and empirical analysis. IEEE Trans Emerg Top Comput 2(3):267–279
10. Han J, Pei J, Kamber M (2011) Data mining: concepts and techniques. Elsevier, Amsterdam
11. Jain AK (2010) Data clustering: 50 years beyond k-means. Pattern Recognit Lett 31(8):651–666
12. Jain AK, Dubes RC (1988) Algorithms for clustering data. Prentice-Hall, Inc, Englewood Cliffs, NJ
13. Jarvis RA, Patrick EA (1973) Clustering using a similarity measure based on shared near neighbors. IEEE Trans Comput 100(11):1025–1034
14. Kong D, Ding C, Huang H (2011) Robust nonnegative matrix factorization using l21-norm. In: Proceedings of the 20th ACM international conference on information and knowledge management. ACM, New York, NY, USA, CIKM '11, pp 673–682. https://doi.org/10.1145/2063576.2063676
15. Li Z, Tang J (2015) Unsupervised feature selection via nonnegative spectral analysis and redundancy control. IEEE Trans Image Process 24(12):5343–5355
16. Li Z, Tang J, He X (2018) Robust structured nonnegative matrix factorization for image representation. IEEE Trans Neural Netw Learn Syst 29(5):1947–1960. https://doi.org/10.1109/TNNLS.2017.2691725
17. Liao TW (2005) Clustering of time series dataa survey. Pattern Recognit 38(11):1857–1874
18. Lühr S, Lazarescu M (2009) Incremental clustering of dynamic data streams using connectivity based representative points. Data Knowl Eng 68(1):1–27
19. Peterson LE (2009) K-nearest neighbor. Scholarpedia 4(2):1883
20. Reed JW, Jiao Y, Potok TE, Klump BA, Elmore MT, Hurson AR (2006) TF-ICF: a new term weighting scheme for clustering dynamic data streams. In: 5th international conference on machine learning and applications, 2006. ICMLA'06. IEEE, pp 258–263
21. Singh S, Awekar A (2013) Incremental shared nearest neighbor density-based clustering. In: Proceedings of the 22nd ACM international conference on information & knowledge management. ACM, pp 1533–1536
22. Ting KM, Zhu Y, Carman M, Zhu Y, Zhou ZH (2016) Overcoming key weaknesses of distance-based neighbourhood methods using a data dependent dissimilarity measure. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 1205–1214
23. Xu R, Wunsch D (2005) Survey of clustering algorithms. IEEE Trans Neural Netw 16(3):645–678