Dynamic Clustering and Indexing of Multi-Dimensional Datasets *

Yong Shi and Aidong Zhang
Computer Science and Engineering Department
State University of New York at Buffalo
Buffalo, NY 14260, USA

Abstract – *Nowadays large volumes of data with high dimensionality are being generated in many fields. Most existing indexing techniques degrade rapidly when dimensionality goes higher. ClusterTree is a new indexing approach representing clusters generated by any existing clustering approach. It is a hierarchy of clusters and subclusters which incorporates the cluster representation into the index structure to achieve effective and efficient retrieval. The authors propose a new ClusterTree^{+} indexing structure which has new features from the time perspective. Each new data item is added to the ClusterTree with the time information which can be used later in the data update process for the acquisition of the new cluster structure. This approach guarantees that the ClusterTree is always in the most updated status which can further promote the efficiency and effectiveness of data insertion, query and update. This approach is highly adaptive to any kind of clusters.*

Keywords: time indexing, ClusterTree, high-dimensional datasets, time-related query, time-related deletion

1 Introduction

Recently large volumes of data with high dimensionality are being generated in many fields. Many approaches have been proposed to index high-dimensional datasets for efficient querying. Although most of them can efficiently support nearest neighbor search for low dimensional datasets, they degrade rapidly when dimensionality goes higher. Also the dynamic insertion of new data can cause original structures no longer handle the dataset efficiently since it may greatly increase the amount of data accessed for a query. Among those approaches, the ClusterTree [1] is the first work towards building efficient index structure from clustering for high-dimensional datasets. Clustering is an analysis technique for discovering interesting data distributions and patterns in the underlying dataset. Given a set of n data points in a d-dimensional space, a clustering approach assigns the data points to k groups ($k << n$) based on the calculation of the degree of similarity between data points such that the data points within a group are more similar to each other than the data points in different groups. Each group is a cluster. The ClusterTree approach builds an index stucture on the cluster structures to facilitate efficient queries.

*This research is supported by NSF, CMIF (Center for Multisource Information Fusion, and CUBRC (Calspan-UB Research Center).

Nowadays a large proportion of the dataset is time related, and the existence of the obsolete data in the dataset may seriously degrade the dataset processing. However, few approaches are proposed to implement the maintainance of the whole dataset to get rid of the obselete data and keep the dataset always in the most updated status for the convenience and effectiveness of dataset procoss such as query and insertion.

In this paper, we propose a new ClusterTree^{+} indexing structure which has new features from the time perspective. Each new data item is added to the ClusterTree^{+} with the time information which can be used later in the data update process for the acquisition of the new cluster structure. This approach guarantees that the ClusterTree^{+} is always in the most updated status which can further promote the efficiency and effectiveness of data insertion, query and update. Our approach is highly adaptive to any kind of clusters.

The rest of the paper is organized as follows. Section 2 summaries the related work on index structure design including the ClusterTree. Section 3 introduces our modified ClusterTree approach – ClusterTree^{+}. Section 4 presents the processing of ClusterTree^{+}. Section 5 gives the conclusion.

2 Related Work

Existing multidimensional tree-like indexing approaches can be further classified into two categories: space partitioning and data partitioning.

A data partitioning approach partitions a dataset and builds a hierarchy consisting of bounding regions. The bounding regions at higher levels contain the lower regions and the bounding regions at the bottom level contain the actual data points. Popularly used index structures include R-Tree, R*-tree, SS-Tree, SS^{+}-Tree and SR-Tree. An R-tree

[2] is a height-balanced tree with index records in its nodes. There are two kinds of nodes: internal and leaf nodes. All the nodes have minimum bounding rectangles (MBR) as page region. R-tree is used to store *rectangular regions* of an image or a map. R-trees are very useful in storing very large amounts of data on disk. One disadvantage of R-trees is that the bounding boxes (rectangles) associated with different nodes may overlap. The R*-tree [3] is an R-tree variant which incorporates a combined optimization of area, margin and overlap of each enclosing rectangle in the tree nodes. It reduces the overlapping between the MBRs of neighboring nodes, reduces the volume of the MBRs and improves the storage utilization.

In contrast to the R-Tree, SS-Tree [4] uses hyper-spheres as region units. Each hyper-sphere is represented by a centroid point, and a radius large enough to contain all of the underlying data points. Queries on the SS-Tree are very efficient because it only needs to calculate similarity between a region and the query point. The SS$^+$-Tree [5] is a variant of the SS-Tree with a modified splitting heuristic to optimize the bounding shape for each node. SR-Tree [6] is an index structure which combines the bounding spheres and rectangles for the shapes of node regions to reduce the blank area. The region for a node in it is represented by the intersection of a bounding sphere and rectangle so that the overlapping area between two sibling nodes is reduced. It takes the advantages of both rectangles and spheres. However, the storage required for the SR-Tree is larger than the SS-Tree because the nodes in the SR-Tree need to store the bounding rectangles and bounding spheres. Consequently, the SR-Tree requires more CPU time and more disk accesses than the SS-Tree for insertions.

Space partitioning approaches recursively divide a data space into disjoint subspaces. K-D-B Tree and Pyramid-Tree are this type. The K-D-B Tree [7] partitions a d-dimensional data space into disjoint subspaces by $(d-1)$-dimensional hyper-planes which are alternately perpendicular to one of the dimension axes. Due to the disjointness among its nodes at the same level, the K-D-B Tree has the advantage that the search path for querying a data point is unique. The Pyramid-Tree [8]'s main idea is to divide the data space first into $2d$ pyramids, each sharing the center point as its peak (the tip point of a pyramid). Each pyramid is then sliced into slices parallel to the base of the pyramid, and each slice forms a data page. The range query under this index structure can be efficiently processed for both low- and high-dimensional datasets. However, the partitioning of the Pyramid-Tree can not ensure that the data points in one data page are always neighbors.

Among those approaches, the ClusterTree is the first work towards building efficient index structures from clustering for high-dimensional datasets. The ClusterTree has the advantage of supporting efficient data query for high-dimensional datasets. A ClusterTree is a hierarchical representation of the clusters of a dataset. It organizes the data based on their cluster information from coarse level to fine, providing an efficient index structure on the data according to clustering. Like many other structures, it has two kinds of nodes: internal and leaf nodes. The internal nodes include pointers to subculsters of this cluster, the bounding sphere for the subcluster and the number of data points in the subclusters. The leaf nodes contain pointers to the data points. For each cluster, the ClusterTree calculates the following parameters: the number of data points, the centroid $\mathbf{c}$, and the volume of the minimum bounding sphere S. The centroid $\mathbf{c} = \langle c_1, c_2, \cdots, c_d \rangle$ can be calculated by:

$$c_i = \frac{\sum_{j=1}^{N} \mathbf{o_j}_i}{N},\ 1 \le i \le d, \tag{1}$$

where N is the number of the data points in the cluster and $\mathbf{o_j}_i$ is the i-th value of data point $\mathbf{o_j}$ in the cluster. Thus, each cluster is represented by a hyper-sphere S. So there may be some empty regions which contain no data, and two bounding hyper-spheres of two different clusters may overlap. We can define the density of the cluster as:

$$\begin{aligned} Density_c &= \frac{number\ of\ points\ in\ C}{volume\ of\ S} \\ &= \frac{number\ of\ points\ in\ C}{\frac{2\pi^{d/2} r^d}{d\Gamma(\frac{d}{2})}}. \end{aligned} \tag{2}$$

The gamma function $\Gamma(x)$ is defined as:

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt, \tag{3}$$

where $\Gamma(x+1) = x\Gamma(x)$ and $\Gamma(1) = 1$. When the density of a cluster falls below a pre-selected threshold or the number of the data points in the cluster is larger than a pre-selected threshold, the cluster will be decomposed into several smaller clusters (subclusters).

3 ClusterTree$^+$ and its Construction

Nowadays a large proportion of a dataset may be time related, and the existence of the obsolete data in the dataset may seriously degrade the dataset processing. However few approaches are proposed to implement the maintainance of the whole dataset to get rid of the obselete data and keep the dataset always in the most updated status for the convenience and effectiveness of dataset procoss such as query and insertion. Although ClusterTree can efficiently handle the processing of datasets with high dimensionality, it doesn't consider the dynamic data update issue.

Here we present a modified version of ClusterTree: ClusterTree$^+$, which is based on the design of the ClusterTree and enhance its capability of handling dynamic data insertions, queries and deletions.

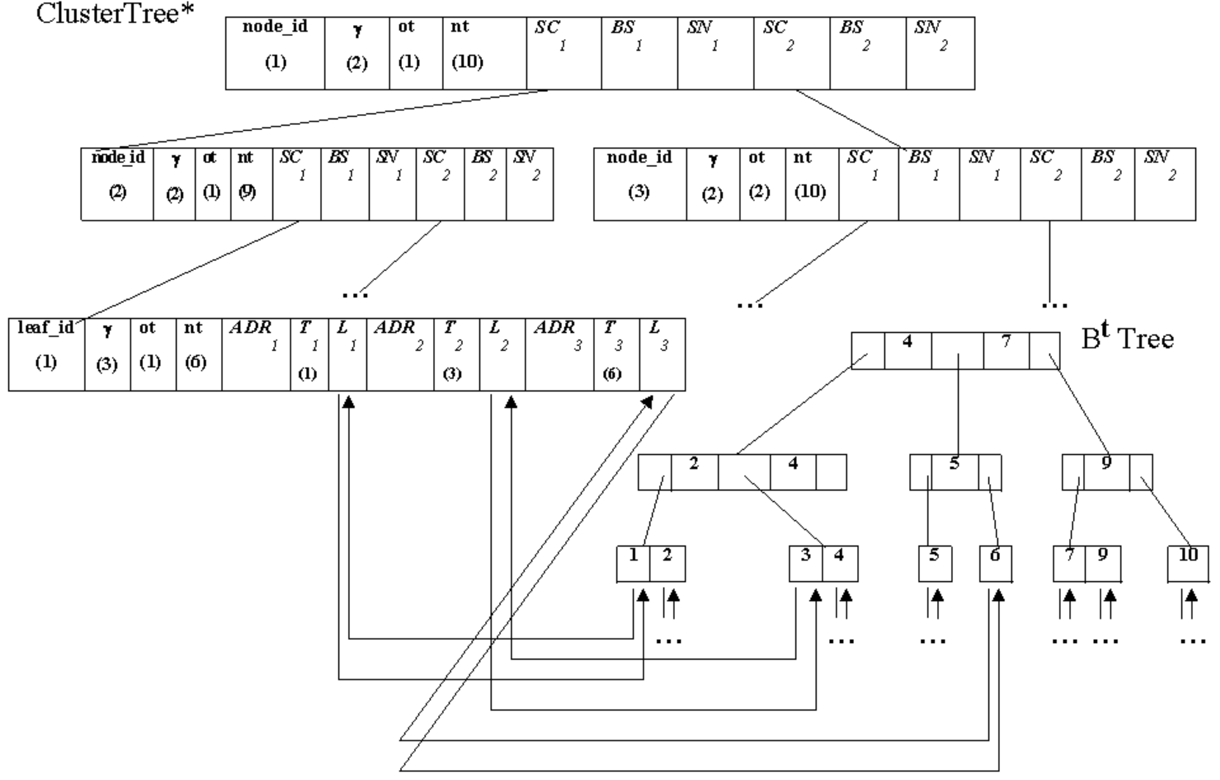

Figure 1: An example of ClusterTree$^+$.

There can be several different ways to associate the time information with the original ClusterTree structure. The typical three approaches are as follows:

- Directly add time information into the ClusterTree as another dimension.
- Use a simple queue to process the time issue.
- Use an individual B$^+$ tree-similar structure to handle the time information.

The advantage of the first approach is that it simplifies the implementation. We can slightly adjust the original algorithm to support the queries which may include the time information and deletions based on the time period which is specified by users. The disadvantage is that if we directly take time information as an extra dimension of the dataset, the clustering results will significantly change. Two data points which are very close to each other in the original space may not even exist in the same cluster after adding the time dimension, since the times the data are inserted into the indexing structure may be quite distant to each other. So it will dramatically degrade the result of the queries which are simply based on the data itself without considering the time issue.

The idea of using a simple queue to process the time issue is straightforward. But as a linear structure, the efficiency is the major problem.

Using an individual B$^+$ tree [9]-similar structure to handle the time information can support both time-related queries and time-irrelevant queries. For time-irrelevant queries, we can use algorithms similar to those of the original ClusterTree. As for time-related queries including range queries and p-Nearese Neighbors queries, we can use the intersection of the searching result of both modified ClusterTree structure and B$^+$ tree-similar structure to get the final result. This approach can also efficiently support user-specified periodic deletions to get rid of the obsolete data in the dataset.

So we choose the last kind of approach to set up the ClusterTree$^+$ indexing structure as our solution to solve the data update problem of high dimensional datasets. The ClusterTree$^+$ has two separate structures, one is a modified ClusterTree structure – ClusterTree*, the other is a modified B$^+$ tree – B^t tree.

The ClusterTree* is a hierarchical representation of the clusters of a dataset. It has two kinds of nodes: internal and leaf nodes. The internal nodes are defined as:

$$Node : Node_id, \gamma, ot, nt, (Entry_1, Entry_2, \cdots, Entry_\gamma)$$
$$(m_{node} \leq \gamma \leq M_{node}),$$
$$Entry_i : (SC_i, BS_i, SN_i),$$

where *Node_id* is the node identifier, γ is the number of the entries in the node, *ot* is the oldest time when data are in-

serted into that node or its descendants, nt is the newest time when data are inserted into that node or its descendants, and m_{node} and M_{node} define the minimum and maximum numbers of entries in the node. An entry is created for each subcluster of the cluster for which the current non-leaf node represents. In entry $Entry_i$, $\mathcal{SC}_i$ is a pointer to the i-th subcluster, BS_i is the bounding sphere for the subcluster and SN_i is the number of data points in the i-th subcluster.

The leaf nodes are defined as:

$$Leaf : Leaf_id, \gamma, ot, nt, (Entry_1, Entry_2, \cdots, Entry_\gamma)$$
$$(m_{leaf} \leq \gamma \leq M_{leaf}),$$
$$Entry_i : (ADR_i, T_i, L_i),$$

where γ is the number of data points contained in the leaf node, and m_{leaf} and M_{leaf} are the minimum and maximum numbers of entries. $Entry_i$ contains the address of the data point residing at the secondary storage (ADR_i), the time information when the data point is inserted into the structure (T_i), and the link to the time data point in the B^t tree (L_i).

The B^t tree indexes on the time data which corresponds to the times the data were inserted into the structure. It originates from the B^+ tree with some modifications as follows:

- There is no minimum number requirement of entries in internal and leaf nodes so that there will be no cases of underflow. This corresponds to the nature of the B^t tree that the time data in it will be batchly deleted for the user-specified deletion requirement.
- In the leaf nodes, each entry has an extra field which is a link to the data point it is associated with in the ClusterTree*. In this case we can traverse from the B^t tree back to the ClusterTree* efficiently.

An example of the ClusterTree$^+$ structure is shown in Figure 1. It shows that currently there are 9 data which are inserted at time 1, 2, 3, 4, 5, 6, 7, 9 and 10. The root node of the ClusterTree* has two entries. The first entry also has two entries. At the third level, the first leaf node has three data entries which are inserted at time 1, 3 and 6. The L_i fields of these entries of the leaf node are connected with the leaf nodes of the B^t tree which is on the bottom-right part of the Figure 1.

3.1 Construction

The construction of ClusterTree$^+$ includes the construction of ClusterTree* and the construction of B^t tree in parallel. The construction of the ClusterTree* is similar to the construction of ClusterTree except that each internal node and leaf node should set the *nt* (newest time) and *ot* (oldest time) as the current time. This is because we don't know the insertion times of the original data points in the dataset, so we have to set the insertion times of all the original data points as the current one. Later when new data points are inserted into the structure, different times can be recorded into the structure. Meanwhile, we create a leaf node in B^t tree which includes the current time data. All L field of the entries in the leaf node of ClusterTree* should be pointed to the created leaf node in B^t tree.

4 Processing of the ClusterTree$^+$

There are three major processing of the ClusterTree$^+$: insertion, query and deletion.

4.1 Insertion

For a new coming data point, we classify it into one of three catagories:

- *Cluster points*: are either the duplicates of or very close to some data points in a cluster within a given threshold.
- *Close-by points*: are the data points which are neighbors to some points in the clusters within a given threshold.
- *Random points*: are the data points which are either far away from all of the clusters and can not be bounded by any bounding sphere of the ClusterTree, or might be included in the bounding spheres of the clusters at each level, but they do not have any neighboring cluster points within a given threshold.

Thus depending on the type of the new coming data point, we can apply the insertion algorithm of ClusterTree to recursively insert data point to a certain leaf node of ClusteterTree* accompanied by the insertion time information in the T(time) field of the new entry in that leaf node, while inserting a new entry which includes the insertion time info into the B^t tree. We then point the L(link) field of the new entry in the certain leaf node in ClusterTree* to the new entry in the leaf node of B^t tree, and point the L(link) field of the new entry in the certain leaf node in B^t tree to the new entry in the leaf node of ClusterTree*.

4.2 Query

There are two kinds of queries, one is the time-irrelevant queries which include the range queries and P Nearest Neighbor queries, and the other is the time-related queries which include certain time period requirement. For example, some users may ask for the neighbors to a certain data point which are inserted into the structure in almost the same time as the insertion time of that data point.

We can depend on the original ClusterTree query algorithm to solve the former kind of queries. As for the latter one, we can solve it as follows:

Algorithm: time-related queries

Input: a data point, a time stamp;

Output: the set of data points in ClusterTree which satisfy the query requirement;*

Algorithm steps:

- **find** *the set a of candidate data points in the ClusterTree***;*
- *at the same time* **search** *the entry x in* B^t *tree whose time data is closest to the query time stamp;*
- **find** *the set of the entries in* B^t *tree which are in a certain threshold in time distance to the entry x;*
- **find** *the set b of corresponding data points in ClusterTree** *using the L field in the entries in* B^t *tree;*
- **get** *the result set of data sets by the intersection of set a and b.*

4.3 Deletion

In many systems the obsolete data should be deleted periodically. Managers may want to delete those data which are inserted into the system a certain time ago, or they want to delete those data inserted during a certain peroid. Also they can simply indicate to the data system to automatically adjust itself. The ClusterTree$^+$ can support such user-specified deletions.

Algorithm: time-related deletion1
Input: a time stamp ts;
Output: the new ClusterTree$^+$ *which has got rid of the obsolete data (the data inserted before the specified time stamp);*
Algorithm steps:

- **find** *the entry x in* B^t *tree whose time data is left-closest to the time stamp ts ("left-closest" means that it is the newest one which is older than the specified time stamp ts);*
- **get** *the set a of entries in the* B^t *tree which are older than the entry x;*
- **search** *for the set b of corresponding data points in ClusterTree** *using the L field in the entries in* B^t *tree;*
- **cut** *those entries in the* B^t *tree recursively which are older than entry x;*
- **cut** *set a in the* B^t *tree;*
- **cut** *set b in the ClusterTree**.

Algorithm: time-related deletion2
Input: time stamp ts1, time stamp ts2(ts1 $\leq$ *ts2);*
Output: the new ClusterTree$^+$ *which has got rid of the data inserted during the period between specified time stamps ts1 and ts2;*
Algorithm steps:

- **find** *the entry x in* B^t *tree whose time data is right-closest to the time stamp ts1 ("right-closest" means that it is the oldest one which is newer than the specified time stamp);*
- **find** *the entry y in* B^t *tree whose time data is left-closest to the time stamp ts2;*
- *if the time data of entry x is newer than that of entry y,* **exit***;*
- **get** *the set a of entries in the* B^t *tree which are between entry x and y;*
- **find** *the set b of corresponding data points in ClusterTree** *using the L field in the entries in* B^t *tree;*
- **cut** *those entries in the* B^t *tree recursively which are antecedents of set a;*
- **cut** *set a in the* B^t *tree;*
- **cut** *set b in the ClusterTree**.

Algorithm: automatic adjustment
Output: the new ClusterTree$^+$ *which has been automatically adjusted;*
Algorithm steps:
recursively check each subcluster:
a) if the gap between the subcluster's nt filed (new time inserted) is over certain threshold, delete the whole subculster or move it to a secondary memory because it means that it contains no new data;
Also delete the corresponding entries in the B^t *tree;*
b) if the old data density of a subcluster is over certain threshold, then subcluster should be reorganized in order to get rid of the old part;
c) if there are two subclusters which are "close" enough to each other and their time "nature" are similar enough to each other, they can be merged into one subcluster for a more compact and reasonable vision.

5 Conclusion

In this paper, we have proposed a modified version of ClusterTree – ClusterTree$^+$ which can efficiently support the time-related queries and user-specified deletions. The ClusterTree$^+$ can keep the dataset always in the most updated status to promote the efficiency and effectiveness of data insertion, query and update. Further experiments will be available to support the analysis of the ClusterTree$^+$.

This approach can be helpful in the fields of data fusion where the data evolve dynamically and regular approaches often fail to solve the problem of keeping a certain system always containing the most updated data. This approach can dynamically supervise the data status of the system and efficiently get rid of obsolete data, and at the same time, reorganize the structure of the dataset.

References

[1] Dantong Yu and Aidong Zhang. ClusterTree: Integration of Cluster Representation and Nearest Neighbor Search for Image Databases. In *IEEE International Conference On Multimedia and Expo*, New York City, July 2000.

[2] A. Guttman. R-Trees: A Dynamic Index for Geometric Data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[3] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 322–331, Atlantic City, NJ, May 1990.

[4] D.A. White and R. Jain. Similarity Indexing with the SS-tree. In *Proceedings of the 12th Intl. Conf. on Data Engineering*, pages 516–523, New Orleans, Louisiana, February 1996.

[5] R. Kurniawati, J. S. Jin, and J. A. Shepherd. The SS+-tree: An Improved Index Structure for Similarity Searches in a High-Dimensional Feature Space. In *Proceedings of SPIE Storage and Retrieval for Image and Video Databases*, pages 13–24, February 1997.

[6] N. Katayama and S. Satoh. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 369–380, Tucson, Arizona, 1997.

[7] J.T. Robinson. The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 10–18, Ann Arbor, MI, April 1981.

[8] S. Berchtold, C Bohm, and H. Kriegel. The Pyramid-Technique: Towards Breaking the Curse of Dimensionality. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 142–153, Seattle, Washington, 1998.

[9] B. Salzberg. *File Structures: An Analytic Approach*. Prentice-Hall, 1988.