# DBSCAN on Grid Based R-TREE

Submitted in partial fulfillment of the requirements of
CS F366 Laboratory Project

by
**Siddharth Bhatia**
**2011B4A7680P**

Under the supervision of
**Prof. Shanmugasundaram Balasubramaniam,**
**Department Of Computer Science and Information Systems**

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
(RAJASTHAN)

December 3, 2014

# Acknowledgements

I am grateful to my advisor, Professor Shanmugasundaram Balasubramaniam for allowing me to explore my interests, while constantly guiding me whenever I was stuck.

Professor Poonam Goel was instrumental in the progress of the algorithm and reshaped my way of thinking. I also thank Dhruv Kumar, Sonal Kumari and Jagat Sesha Challa for their insightful suggestions and putting up with me.

Last, but not the least, I thank Birla Institute of Technology and Science, Pilani for giving me this opportunity to rub shoulders with real research and be a part of a huge project.

# Abstract

The objective of this report is to design a new algorithm for DBSCAN using the GR-Tree structure. The performance is compared to DBSCAN implemented on R-Tree. The bottleneck step in DBSCAN is the redundant neighborhood computations for every point but we perform this computation only once for a particular cell (group of points). The code was tested and profiled with three datasets-3D Road Network, Delucia and Bertone having 4.35 lakh, 3.2 million and 3.4 million data points respectively. We document the algorithm and the optimizations along with some directions for taking this work further.

# Contents

# Chapter 1

# Introduction

## 1.1 DBSCAN

Density-based spatial clustering of applications with noise ($DBSCAN$) is a data clustering algorithm proposed by Martin Ester, Hans-Peter Kriegel, Jrg Sander and Xiaowei Xu in 1996 [1]. It is a density-based clustering algorithm because it finds a number of clusters starting from the estimated density distribution of corresponding nodes. $DBSCAN$ is one of the most common clustering algorithms and also most cited in scientific literature.

$DBSCAN$ requires two parameters: $\varepsilon$ and the minimum number of points required to form a dense region ($minPts$). It starts with an arbitrary starting point that has not been visited. This point's $\varepsilon$-neighborhood is retrieved, and if it contains sufficiently many points, a cluster is started. Otherwise, the point is labeled as noise. Note that this point might later be found in a sufficiently sized $\varepsilon$-environment of a different point and hence be made part of a cluster. If a point is found to be a dense part of a cluster, its $\varepsilon$-neighborhood is also part of that cluster. Hence, all points that are found within the $\varepsilon$-neighborhood are added, as is their own $\varepsilon$-neighborhood when they are also dense. This process continues until the density-connected cluster is completely found. Then, a new unvisited point is retrieved and processed, leading to the discovery of a further cluster or noise.

## 1.2 Complexity of DBSCAN

$DBSCAN$ visits each point of the database, possibly multiple times (e.g., as candidates to different clusters). For practical considerations, however, the time complexity is mostly governed by the number of regionQuery invocations. $DBSCAN$ executes exactly one such query for each point, and if an indexing structure is used that executes such a neighborhood query in $O(\log N)$, an overall runtime complexity of $O(N \log N)$ is obtained.

## 1.3 GR-Tree

$GR$-tree is a grid based $R$-tree. An $n$-dimensional grid is constructed over the dataset whose non empty cells are indexed using an $R$-tree. Each cell further consists of a local cell-$R$-tree that is constructed for the points indexed in that cell. The formal construction of the $GR$-tree is given below.

Similar to the $R$-tree, $GR$-tree consists of two kinds of nodes- internal nodes and external nodes. Internal nodes store $n$-dimensional minimum bounding boxes (*Gmbbs*) which further point to other internal or external nodes. *Gmbbs* store the region of the bounding box that contain all the regions of the nodes that are in the sub tree rooted at the current *Gmbb*. External nodes consist of $n$-dimensional cells of size $c$, which are non empty cells resulting out of gridding over the entire dataset. Each node (both internal and external) has a minimum of $G_m$ entries and maximum of $G_M$ entries stored in it. The points indexed by each cell are further indexed by an $R$-tree that is local to each cell.

Each local $R$-tree also has internal and external nodes where internal nodes consist of minimum bounding boxes (*Rmbbs*) and the external nodes consists of pointers to the data points. Each node (both internal and external) has a minimum of $R_m$ entries and maximum of $R_M$ entries stored in it.

This design of $GR$-tree has resulted in the following advantages:-

1. It has resulted in an efficient data structure for storing sparse grid structure where only non empty cells are indexed in an $R$-tree giving efficiency in space utilization.

2. The cells are totally disjoint, minimizing the overlap between the data points and the minimum bounding boxes of the $R$-tree, which is very crucial for any kind of query over it.

3. It enables one to choose lower values of min entries and max entries parameters for the $GR$-tree as well as the local $R$-trees increasing the query performance of both $\varepsilon$ neighborhood queries as well as $k$-nearest neighbor queries.

# Chapter 2

# Algorithm

1. First construct a *GR*-tree on the input data list. We get a list of cells along with the *GR*-tree. Each grid cell contains the list of data points which lie in its space.

2. Traverse through the cell list once and determine the cell type for each cell.

   (a) If the cell contains greater than *minPts* number of points, it's cell type is $CORE$

   (b) Else, if the sum of number of points in the immediate level cells including the cell itself is greater than or equal to *minPts*, it's cell type is $DENSE$.

   (c) Else, if the sum of number of points in the cells lying in the $\varepsilon$-extended region of the current cell, including the cell itself, is less than *minPts*, it's cell type is $NOISE$. Here, the notion of noise is not to be confused with the original definition of noise in $DBSCAN$. The points in the cell marked as $NOISE$ can be border points but surely, not core points.

   (d) Else, the cell type is $SPARSE$. Here, we will do point to point processing. Even though the cell is marked as $SPARSE$, the cell can still contain noise or border or even core points.

3. Traverse through the cell list again and process each cell $C$.

   (a) If $C$ is core, then merge the cluster id of cell with that of its immediate level cells. All the immediate level cells are surely going to be $DENSE$. They may be $CORE$ as well.

   (b) If $C$ is dense, then merge the cluster id of $C$ with those immediate level cells which have at least one core point. This includes core, dense and sparse cells containing core points.

For other cells(i.e. we dont know whether they have any core point or not), we check the cluster id of every point in that cell. If no cluster id has been assigned, we assign the cluster id of $C$ to that point.

(c) If $C$ is sparse or noise, then process every point $p$ in $C$ individually to identify every point's property or nature.

   i. Compute $\varepsilon$-neighborhood of $p$ from an auxillary $GR$-tree of List $\varepsilon$.

   ii. If $p$ is core, merge the cluster id of $p$ with every core point in $Nbh(p)$. Set the isProcessed tag of $p$ to be $TRUE$. Assign cluster id of $p$ to every non-core point which does not have any cluster id.

   iii. If $p$ is not core, merge it with a core point in its neighborhood. Else, leave. Set the isProcessed tag of $p$ to be $TRUE$.

## 2.1 Merging

Union-Find has been implemented for merging two cells, two points or a point and a cell. An implementation has been provided in the appendix. Arrays have been used since we know both the number of cells and the number of points from the dataset before the processing and therefore there is no memory overhead.

## 2.2 Bottleneck

$GgetCellsInRegion$ which is used for neighborhood query and $populateAuxGridRTree$ which is used for populating an Auxiliary $GR$-tree are the time-taking functions and therefore optimum number of calls to these has been done.

# Chapter 3

# Conclusions and Further Work

An outline of the parallel implementation of the algorithm has been discussed and will be implemented in Shared, Distributed and Hybrid Model. Some optimizations which are being looked into:

1. Creation of one auxiliary $GR$-Tree in the beginning and addition and deletion of the nodes as we move from one cell to the other.

2. Comparison of 1 Global $GR$-Tree vs $N$ Local $GR$-Trees. If we have 1 $GR$-Tree, although there is no problem with merging, we may run into trouble because of contention.

3. $kd$-Tree vs Random Distribution.

# Appendix

## UNION-FIND

```
int find2(Cluster subsets, int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find2(subsets, subsets[i].parent);
    return subsets[i].parent;
}

void Union2(Cluster subsets, int x, int y)
{
    int xroot = find2(subsets, x);
    int yroot = find2(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}
```

## MERGE

```
void mergeCellClusterIds(int cell1, int cell2)  //need to see
{
  Union2(cellCluster,cell1+totalPoints,cell2+totalPoints);
  return;
}

void mergePointClusters(int point1, int point2,int cell1, int cell2 )
{
  if(cellCluster[cell1+totalPoints].parent!=cell1 || cellCluster[cell1+
    totalPoints].rank!=0)
  {
    point1=cell1+totalPoints;
  }
  if(cellCluster[cell2+totalPoints].parent!=cell2 || cellCluster[cell2+
    totalPoints].rank!=0)
  {
    point2=cell2+totalPoints;
  }
  Union2(cellCluster,point1, point2);
  return;
}
```

## CORE CELL PROCESSING

```
if(currentCell->cellDataList->cellType == CORE)
{
  Region currentCellRegion = createRegionofCell(currentCell);
  Region cellExtendedOfCurrentCell = getEpsExtendedRegion(currentCellRegion,
    CELLSIZE);
  BCellListHd cellExtendedCellsOfCurrentCell;
  cellExtendedCellsOfCurrentCell = GgetCellsInRegion2(GRTree,
    cellExtendedOfCurrentCell, currentCellRegion);
  BCellListNode twoEpsCursor = cellExtendedCellsOfCurrentCell->first;
  if( cellExtendedCellsOfCurrentCell->count != 0)
  {
    while(twoEpsCursor != NULL)
    {
      BCell twoEpsCursorCell = twoEpsCursor->bCellElem;
      mergeCellClusterIds(currentCell->id, twoEpsCursorCell->id);
      twoEpsCursor = twoEpsCursor->next;
    }
  }
  freeRegion(currentCellRegion);
  freeRegion(cellExtendedOfCurrentCell);
  freeCellsList(cellExtendedCellsOfCurrentCell);
}
```

## DENSE CELL PROCESSING

```
else if(currentCell->cellDataList->cellType == DENSE)
{
    Region currentCellRegion = createRegionofCell(currentCell);
    Region cellExtendedOfCurrentCell = getEpsExtendedRegion(currentCellRegion,
      CELLSIZE);
    BCellListHd cellExtendedCellsOfCurrentCell;
    cellExtendedCellsOfCurrentCell = GgetCellsInRegion2(GRTree,
      cellExtendedOfCurrentCell, currentCellRegion);
    BCellListNode twoEpsCursor = cellExtendedCellsOfCurrentCell->first;
    if( cellExtendedCellsOfCurrentCell->count != 0)
    {
      while(twoEpsCursor != NULL)
      {
        BCell twoEpsCursorCell = twoEpsCursor->bCellElem;
        if(twoEpsCursorCell->cellDataList->cellType == CORE|| twoEpsCursorCell->
      cellDataList->cellType == DENSE || twoEpsCursorCell->cellDataList->
      hasCorePoint ==TRUE)
        {
          mergeCellClusterIds(currentCell->id, twoEpsCursorCell->id);
        }
        else
        {
          CellData currentNbhDataPoint = twoEpsCursorCell->cellDataList->first;
          while(currentNbhDataPoint != NULL)
          {
            if((cellCluster[twoEpsCursorCell->id+totalPoints].parent==
      twoEpsCursorCell->id+totalPoints && cellCluster[twoEpsCursorCell->id+
      totalPoints].rank==0 )&& (cellCluster[currentNbhDataPoint->data->iNum].parent
      ==currentNbhDataPoint->data->iNum && cellCluster[currentNbhDataPoint->data->
      iNum].rank==0))
            {
              Union2(cellCluster,currentCell->id+totalPoints,currentNbhDataPoint->
      data->iNum);
```

```
          }
          currentNbhDataPoint = currentNbhDataPoint->next;
        }
      }
      twoEpsCursor = twoEpsCursor->next;
    }
  }
  freeRegion(currentCellRegion);
  freeRegion(cellExtendedOfCurrentCell);
  freeCellsList(cellExtendedCellsOfCurrentCell);
}
```

# Bibliography

[1] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996. 1

[2] Yaobin He, Haoyu Tan, Wuman Luo, Huajian Mao, Di Ma, Shengzhong Feng, and Jianping Fan. Mr-dbscan: An efficient parallel density-based clustering algorithm using mapreduce. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 473–480. IEEE, 2011.

[3] Md Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. A new scalable parallel dbscan algorithm using the disjoint-set data structure. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.