Boundary Points Detection Using Adjacent Grid Block Selection (AGBS) kNN-Join Method

Shivendra Tiwari, Saroj Kaushik

Department of Computer Science and Engineering
IIT Delhi, Hauz Khas, New Delhi, India 110016
{shivendra , saroj}@cse.iitd.ac.in

Abstract. Interpretability, usability, and boundary points detection of the clustering results are of fundamental importance. The cluster boundary detection is important for many real world applications, such as geo-spatial data analysis and point-based computer graphics. In this paper, we have proposed an efficient solution for finding boundary points in multi-dimensional datasets that uses the BORDER [1] method as a foundation which is based on Gorder k-nearest neighbors (kNN) join [10]. We have proposed an Adjacent Grid Block Selection (AGBS) method to compute the kNN-join of two datasets. Further, the AGBS method has been used for the boundary points detection method named as AGBS-BORDER. It executes AGBS join for each element in the dataset to get the paired (qi, pi) kNN values. The kNN-graphs are generated at the same time while computing the kNN-join of the underlying datasets. The nodes maintain a list of kNN which indicate the outgoing edges of the graph; however the incoming edges denote the Reverse kNN (RkNN) for the corresponding nodes. Finally, threshold filter is applied - the node whose number of RkNN is less than the threshold values are marked as the border nodes. We have implemented both the BORDER [1] and AGBS-BORDER methods and tested the same using randomly generated geographical points. The AGBS based boundary computation over performs the existing BORDER method.

Keywords: k-Nearest Neighbor (kNN), Reverse kNN (RkNN), kNN-Join,, Cluster Boundaries, Principal Component Analysis (PCA), Adjacent Grid Block Selection (AGBS).

1 Introduction

The size of spatial database increases dramatically and the structure of data becomes more and more complicated. To discover valuable information in those databases, spatial data mining techniques have been paid significant attention during recent years. Knowledge discovery in databases is a non-trivial process of identifying valid, interesting and potentially valuable patterns in data. Due to the need for efficient and effective analysis tools to discover information from these data, many techniques have been developed for knowledge discovery in databases. Such techniques include data classification, mining association rule, clustering, outlier analysis, data cleaning and data preparation. In general, clustering is a method to separate data into groups

without prior labeling so that the elements inside the same group are most similar [1]. The cluster boundary detection is becoming more and more valuable for many real world applications, hence it is important to optimize the boundary detection algorithm as much as possible. Mostly, the boundary point detection algorithms scan the complete data space iteratively and therefore computation cost happens to be high. Even the small improvement in the memory and computation results a significant impact on the overall systems [5].

The kNN algorithm is a method for classifying objects based on closest training examples in the feature space. A kNN of a data point $r \in R$ is a set of k points $s \in S$ which are closest among the rest of the points $s' \in S$. The kNN join between two datasets R and S, returns the kNN points $p_s \in S$ for each point $p_r \in R$. The kNN-Join is important as the one-time computation of the neighbors for all data points can dramatically accelerate the operations of the traditional one-point-based similarity search. For example, data mining algorithms can be implemented on the top of one join operation, instead of many similarity queries. However computing the kNN Join iteratively for the dynamic and changing data is expensive. Essentially, the kNN-join combines each point of the outer dataset R with its kNN from the inner dataset S. The applications of the kNN-joins include kNN classification, k-means clustering, sample assessment and sample post-processing, k-distance diagrams, etc [3]. We assume a metric space and the objects are represented by a multidimensional point in the space. Therefore, we also refer to an object as a point occasionally. A data point in our context is a multidimensional feature vector corresponding to a complex object such as an image. In the rest of the paper, we use R to symbolize the outer dataset and S the inner dataset.

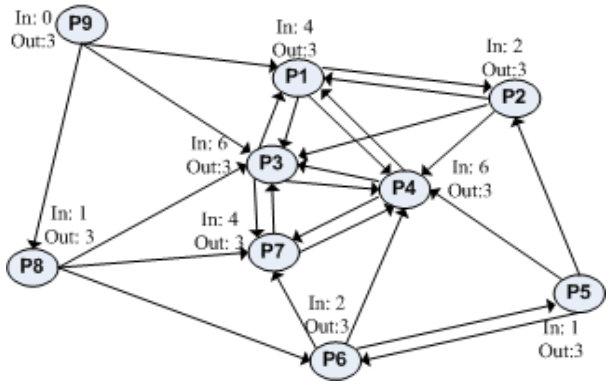

Fig.1. kNN Graph for k=3. The outgoing edges show the kNN relationships; however the incoming edges denote the RkNN relationships among the data points.

The kNN-Graph for a set of objects is a directed graph with vertex set $R \cup S$ and an edge from each $r \in R$ to its k most similar objects in S under a given similarity measure as shown in Fig.1. The nodes in the graph represent the data points labeled as P_i. Each node keeps record of their incoming and outgoing edges. The kNN-graphs are useful on the various cluster analysis processes.

The kNN cluster can be divided into three parts i.e. Core, Ourlier, and Boundary points. A Core Point is defined as a point that has at least k RkNN points: i.e. $|RkNNs| \geq k$. An Outlier Point is a point that has less than k number of RkNNs i.e. $|RkNNs| < k$. Lesser the number of RkNNs, the more distant it is from its neighbors. A Boundary Point is an outlier, classified to be boundary point, if majority of its RkNNs belong to one cluster. Boundary points are data points that are located at the margin of densely distributed cluster. The difference between Boundary Point and Outlier Point is that a boundary point is one that is usually near the boundaries of a cluster or amidst more than one cluster; however an outlier point has $|RkNNs| < k$ and most of its RkNNs are outliers. But for a boundary point, its $|RkNNs| < k$ and most of its RkNNs are core-points [1]. In order to filter unnecessary distance computation in the kNN-join processing, popular algorithms based on an index such as the R-tree compute the MinDist (i.e. minimum distance between the query point and a node of the R-tree) and choose to traverse the node with the minimum MinDist first. The MinDist is also compared with the pruning distance (the distance between the query point and its k^{th} nearest neighbor candidate). Our work-in-progress AGBS's efficient grid creation, grid block searches, nested block join scheduling have been used for the boundary points detection in this paper.

This paper is divided into six sections. Section 2 and 3 contain the related work and AGBS based kNN-join computation method separately. Section 4 explains the efficient AGBS-BORDER algorithm for the boundary points detection. The sections 5 and 6 talk about the cost analysis, limitations & future work respectively. Finally the paper has been concluded in the section 7.

2 Related Work

There has been a considerable amount of work on efficient nearest neighbor finding methods [10, 12, 14, and 15]. The reverse k-nearest neighbor (RkNN) problem has been first proposed in 2000 [4] and has been paid increasing attention to in the recent years. Existing studies all focus on the single RkNN query. Various methods have been proposed for its efficient processing and can be divided into two categories: pre-computation methods and space pruning methods. The shortcoming of pre-computation methods is that they cannot answer an RkNN query unless the corresponding k-nearest neighbor information is available. The preprocessing of k-nearest neighbor information is actually a kNN join. Space pruning methods utilize the geometry properties of RkNN to find a small number of data points as candidates and then verify them with nearest neighbor queries or range queries. Space pruning methods are flexible because they do not need to pre-compute the nearest neighbor's information. However, they are very expensive when data dimensionality is high or the value k is big.

In 2008, Dongquan Liu et al proposed a boundary detection algorithm known as TRICLUST based on the Dalaunay Triangulation [5]. TRICLUST algorithm treats clustering task by analyzing statistical features of data. For each data point, its values of statistical features are extracted from its neighborhood which effectively models

the data proximity. By applying specifically built criteria function, TRICLUST is able to effectively handle data set with clusters of complex shapes and non-uniform densities, and with large amount of noises. One additional advantage of TRICLUST is the boundary detection function which is valuable for many real world applications such as geo-spatial data processing, point-based computer graphics, etc. Since we are mainly interested in spatial data which is usually low dimensional, the analysis here is for two dimensional cases. The time complexity of constructing Delaunay triangulation graph is O(N.logN), where N is the number of data points.

Olga Sourina et al proposed a clustering and boundary detection method for the time dependent data in 2007. The method is based on geometric modeling and visualization. Datasets and boundaries of clusters are visualized as 3D points and surfaces of reconstructed solids changing over time. This method applies the concepts of geometric solid modeling and uses density as clustering criteria that comes from traditional density-based clustering techniques. Visual clustering allows the user to analyze results of clustering the data changing over time and to interactively choose appropriate parameters [11].

The set oriented kNN-join can accelerate the computation dramatically. The MuX kNN-join [12] and the Gorder kNN-join [10] are two up-to-date methods specifically designed for kNN-join of high-dimensional data. MuX is essentially an R-tree based method designed to satisfy the conflicting optimization requirements of CPU and I/O cost. It employs large-sized pages (the hosting page) to optimize I/O time and uses the secondary structure, the buckets which are MBRs (Minimum Bounding Rectangles) of much smaller size, to partition the data with finer granularity so that CPU cost can be reduced. MuX iterates over the R pages, and for R page in the memory, potential kNN-joinable pages in S are retrieved through MuX index on S and searched for k-nearest neighbors. Since MuX makes use of an index to reduce the number of data pages retrieved, it suffers as an R-tree based algorithm and its performance degrades rapidly when the data dimensionality increases. Gorder kNN-join [10] is a non-index approach. It optimizes the block nested loop join with efficient data scheduling and distance computation filtering by sorting data into the G-order. The dataset is then partitioned into blocks that are amenable for efficient scheduling for join processing and the scheduled block nested loop join is applied to find the k-nearest neighbors for each block of R data points [1].

The block selection in Gorder [10] is a brute force search and also the size of the blocks does not consider the value of k. Chenyi Xia et al further used Gorder for boundary point's detection in 2006 known as BORDER [1]. It processes a dataset in three steps. First, it executes Gorder kNN join to find the k-nearest neighbors for each point in the dataset. Second, it counts the number of reverse k-nearest neighbors (RkNN number) for each point according to the kNN-file produced in the first step. Third, it sorts the data points according to their RkNN number and finally, the boundary points whose RkNN number is smaller than a user predefined threshold can be incrementally output [1]. However it still goes through the multiple steps that need additional memory and computation cost. We have studied latest kNN-Join algorithms and are proposing a better algorithm based on the adjacent block selection and nested block scheduling methods in the following section.

3 Computing kNN-Join with Adjacent Grid Based Selection

The adjacent grid block selection based kNN join method is based on the fact that the grid cells are surrounded by the adjacent neighboring cells which are the priority candidates of the kNNs for the query points in a block. There are two steps of the algorithm i.e. preprocessing of the datasets and the kNN-join operation. The preprocessing of the data includes two major operations – the principal component analysis (PCA) [10, 20] as shown in Fig. 2 and the grid block generation. In the kNN-join phase, we use the physical adjacency of the grids to select the block for the computation.

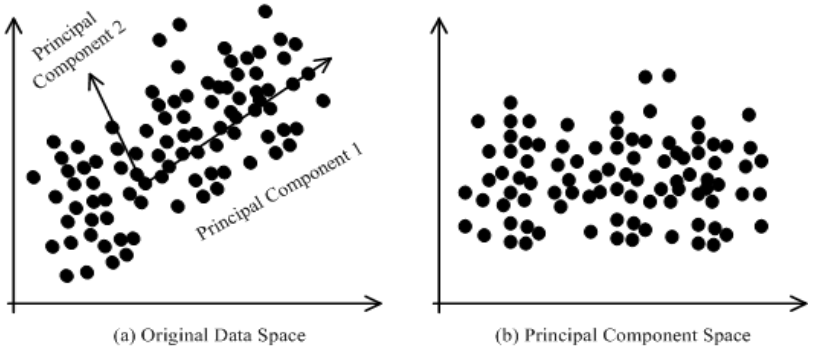

Fig.2. Illustration of the Principal Component Analysis

3.1 Preprocessing of Datasets

Before the kNN-join operation takes place, the dataset is prepared in such a way that the I/O and CPU costs are minimized. The PCA (principal component analysis) transformation and the Grid Block Generation are the two major preprocessing steps in the AGBS method. In Algorithm 1, the grid blocks are generated for both of the datasets if the operation is not self-join i.e. both the datasets are not the same.

```
Algorithm 1 Preprocessing (R, S)
Input: R and S are two data sets.
Description:
Principal-Component-Analysis (R, S);
Grid-Block-Gen(R);
If R!=S then  Grid-Block-Gen(S);
```

3.1.1 Principal-Component-Analysis

Principal component analysis (PCA) is a standard tool in modern data analysis and data mining. It is a simple and non-parametric method for extracting relevant information from confusing data sets. With minimal effort PCA provides a roadmap for how to reduce a complex data set to a lower dimension to reveal the sometimes hidden, simplified structures that often underlie it [20]. After applying PCA processing, most of the information in the original space is condenses into the first few dimensions along which the variances in the data distribution are the largest. The first principal component (or dimension) accounts for as much of the variability in the data as possible, and each succeeding component accounts for as much of the remaining variability as possible as shown in Fig 2 [10].

3.1.2 Grid Block Generation

The grid block generation for given dataset is based on the two major considerations i.e. the bigger block should be used for the I/O and the smaller blocks should be used for distance computation. Since the most of the computation cost arise in the distance computation at the data level, the data blocks should be of optimal size on the runtime memory. If we assume each block as a kNN cluster, it will need at least k data points into it to satisfy all the points. It needs to compute the distance between at least k points in order to find the kNN of a single point. Hence the complexity of the kNN join can not be less than the O(nk). We propose to ensure the average number of points in a single lowest level block to be O(k). Since the distance computation has been minimized at the data point's level, the computation cost is expected to increase at the grid block selection. The optimization in the block selection technique will improve the overall performance of the system.

The idea is to create multilevel grid block hierarchy of the input datasets. First, divide the data into the bigger blocks in the secondary storage. The blocks have specific identity that can be used for the search at the time of join operation. A multi-level index file can be created separately to store the block id and their offset in the file mappings. The secondary storage blocks are read at the time of kNN-join operation selectively. Second, further more levels of grid block hierarchy are created on the memory to reduce the distance computation. The global grid block matrix is loaded on the memory which is lightweight information of the block structure that can be implemented on a multidimensional array. Let's assume that the Min_i and Max_i are the minimum and the maximum values of the i^{th} dimension in the dataset R and the total number of blocks is N. δ_i is the maximum difference of values between any two points of a grid block in the i^{th} dimension. The point values in the n^{th} block of the i^{th} dimension are from $(Min_i + n * \delta_i)$ to $(Min_i + (n+1) * \delta_i - \varepsilon)$. Here, ε is the threshold value to handle the grid block boundary. Considering the total point count in the dataset R is |R|, then N and δ_i is calculated as below:

$$N = \left(\frac{|R|}{k} \right) \quad \textbf{(1)}$$

$$\delta_i = \left(\frac{Max_i - Min_i}{N} \right) \quad (2)$$

Where, k is the value in k-Nearest Neighbor join query. The calculation of the cell id is simply based on the range division of the dataset in question. The blocks are partitioned it into N^d rectangular cells, where N is the number of block per dimension of the grid. Fig. 3 is an illustration of a two-dimensional space partitioned by a 9x9 grid. A vector of block ids for each dimension makes the unique identity of a block or a cell. The block id of the point $p_r \in R$ for the i^{th} dimension can be calculated as below:

$$b_i = \left(\frac{p_i - Min_i}{\delta_i} \right) \quad (3)$$

In Algorithm 2, the blocks are generated for the dataset R. All the points in the dataset are iterated, and the grid block identity is calculated. The procedure Compute-Block-id() in line 2 is used to compute the grid block or the cell id as per the equation (3). There are separate blocks for each dimension in consideration after the PCA processing. Assuming that the block number for the i^{th} block is b_i, the cell id is a vector of block numbers i.e. $<b_1, b_2, b_3, \ldots, b_d>$. In line 3, Point-to-Block() procedure is used to divide the data and store the points into the cell at the secondary storage. A multidimensional matrix structure can be used to physically store the points into the grid blocks on the runtime memory. As soon as the runtime memory reaches to the limit the cell data is written into the secondary storage. We propose to create separate files for each data page (I/O optimization block) so that the file seek operations are minimized. Once all the data points are written into the secondary storage blocks, the cells are merged in the sorted order of their block id vector. The search of the appropriate cell is fast being the direct memory access in the matrix.

```
Algorithm 2 Grid-Block-Gen(R)
Input: R is a dataset
Description:
for each point p_r ∈ R do
   cell-id = Compute-Block-id (p_r);
   Point-to-Block (p_r, cell-id);
```

3.2 Adjacent Grid Blocks Selection Join

In the second phase of this method, the grid blocks of R and S are examined for joining. The two-tier partitioning strategy has been applied to optimize the I/O time

and CPU time separately. The first-tier partitioning is optimized for I/O time. We partition the grid oriented input datasets into blocks consisting of several physical pages. The blocks in dataset R are loaded into memory sequentially and iteratively one block at a time and the S blocks are loaded into memory in the surrounding adjacent sequence based on their similarity to the R data in buffer. The loading of bigger block at a time is efficient in terms of I/O time as it significantly reduces seek overhead. In addition, in order to optimize the kNN processing, it selects the S blocks so that the S blocks that are most likely to yield k-nearest neighbors can be loaded into memory and joined with R data in buffer early. The large block size reduces disk seek time, however, as a side effect; it may introduce additional CPU cost due to redundant pair-wise distance computation of tuples for kNN-join. To overcome such a problem, we use the two-tier partitioning in memory. The two tier hierarchy on memory is of smaller data size called Blocks and Sub-blocks.

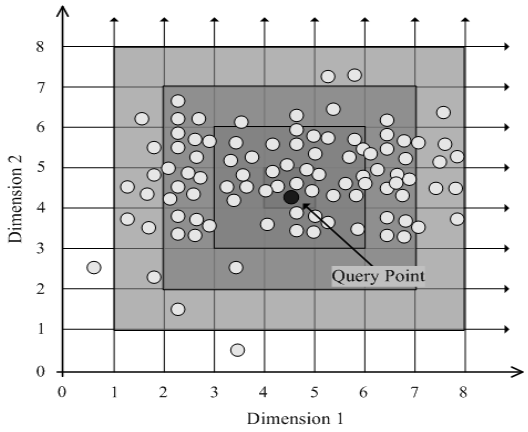

Fig.3. AGBS Block Traversal in their adjacency neighboring order.

Algorithm 3 Grid-Block-Selection-Join (R, S)

Input: R and S are two grid oriented data sets that have been partitioned into blocks.

Description:

```
for each block B_r ∈ R do
   Read-Block(B_r);
   Find-Closest-Block(S, B_r);
   for each B_s ∈ Surrounding-Adj-Block(S, B_r) do
       Read-Block(B_s);
       Block-Join(B_r, B_s);
   Output-KNN(B_r);
```

Algorithm 3, outlines the join algorithm of AGBS method. It loads secondary storage data blocks of R into memory sequentially. For the R block in memory B_r, S blocks are selected in the increasing order of their adjacency to B_r. With each pair of R and S block, we join them in memory by calling function Block-Join(). After all un-pruned S blocks are processed with B_r, the kNN points are written into the output files.

Algorithm 4 Block-Join (B_r, B_s)

Input: B_r and B_s are the grid blocks that have been partitioned into more sub-blocks further on memory.

Description:

```
for each sub-block SBr ∈ Br do
   Find-Closest-Block(Bs, SBr);
   for each SBs ∈ Surrounding-Adj-Block(Bs, SBr) and SBs
   ∈ NotPruned(Bs, SBr)do
      Sub-Block-Join(SBr, SBs);
```

In Algorithm 4, the data is available on memory to be processed. $SB_r \in B_r$ is the sub-blocks divided based on the data density and the k value in the query. For each sub-blocks SB_r, the Find-Closest-Block() method finds the closes sub-block $SB_s \in B_s$. Then the surrounding sub-blocks in B_s are selected in the increasing order of their adjacency to SB_r. With each pair of SB_r and SB_s block, the join operation takes place in runtime memory by calling function Sub-Block-Join().

Algorithm 5 Sub-Block-Join(SB_r, SB_s)

Input: SB_r and SB_s are two sub-blocks from R and S respectively.

Description:

```
for each point pr ∈ SBr do
  if(MinDist(pr,SBr) <= pruningDist(pr))
         for each point ps ∈ SBs do
            ComputeDist(ps, pr);
```

The lowest level block join algorithm is shown in Algorithm 5. In order to join R and S sub-block SB_r and SB_s, each data point $p_r \in SB_r$ is paired with $p_s \in SB_s$ for the distance computation. Distance computation reduction is important for optimization of CPU time because of the complexity of the distance metric and the high-dimensional data. We have taken the distance computation method used in Gorder [10]. The bounding boxes of the grid based data have some special properties which can be utilized for distance computation reduction. While computing the similarity of two blocks of grid oriented data, only first few dimensions are needed to take into account.

4 Boundary Points Detection with AGBS kNN-join

In this section, we describe the details of the proposed AGBS based boundary detection method. It considers the observation that boundary points tend to have fewer RkNN [1]. The new and improved method considers the basic idea of marking the smaller RkNN nodes as the boundary nodes. Fig. 4 shows a processing procedure of the AGBS-BORDER algorithm for the boundary nodes detection in a given dataset. It has mainly two steps – First, generate the kNN Graph using the AGBS kNN-Join operation. Instead of just generating the paired nodes denoting the kNN of a node, we generate a kNN digraph in which each node keeps records of its kNN and RkNN neighbors. Second, the threshold filter is applied on the resulted RkNN stored in the individual nodes of the kNN digraph. Finally, the results of the threshold filter is stored and used for the appropriate applications.

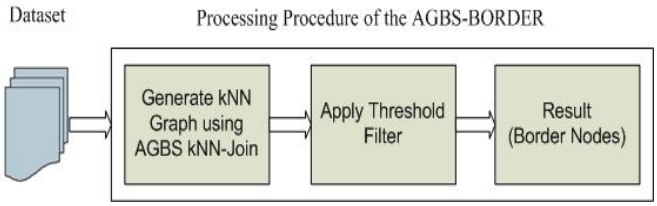

Fig. 4. The Processing Procedure of the AGBS-BORDER

4.1 kNN Graph Generation using AGBS

The kNN-graph is created at the same time the AGBS procedure runs for generating kNN-join pairs with an additional traversal of the AGBS's output. Suppose $r \in R$ has a list of kNN points $s_i \in S$ where i = 1 to k. It means that r has access pointer to the s_i data points. The list of kNN points are treated as the outgoing edges for its kNN-graph. The RkNN denote the incoming edges of the kNN graph that needs to be identified separately. All we need is, iterate the nodes on the runtime memory, and update the counter to the kNN nodes i.e. RkNN. A pair of nodes and corresponding RkNN counter should be written in the output files. Since we read the data in multiple chunks from the secondary storage, this is possible that the RkNN counters for a data point are known at the multiple I/O and kNN-join operations. The nodes have to store the RkNN counter and keep them writing in the output file so that they can be accumulated. Finally, the output pair files are ready and the RkNN values are accumulated from the secondary storage files. Algorithm 6 shows a kNN-graph generation algorithm using AGBS based kNN-join method.

Algorithm 6. Generate-kNN-Graph (R, S)

Input: R and S are the Input Datasets.

Output: kNN Graph having nodes with incoming and outgoing edges

```
Description:
Call procedure Grid-Block-Selection-Join(R, S);
Read the file with <node, RkNN> pairs;
Accumulate all the RkNN for each point;
Store the RkNN for each point as the incoming edges;
```

4.2 Marking Boundary Nodes through Threshold Filter

The basic idea of the proposed solution is based on the observation that boundary points tend to have fewer RkNN [1]. In order to identify the boundary nodes in the datasets, apply the RkNN threshold filter that have been calculated and stored in the kNN-graph. The nodes are traversed sequentially, and if the RkNN value of the data points are less than a suitable threshold value, then write them into a boundary point's output file. Finally, the boundary nodes in the output files are ready to use data for various applications.

5 Performance Analysis

The performance of the new algorithm and the old BORDER method has been shown by their mathematical analysis and the runtime comparison graphs separately. Since the majority of the computation cost lies in the kNN-join procedure, we have analyzed the AGBS algorithm in the subsection below. However we have implemented and compared the runtime performance in two stages i.e. kNN-join procedure and the overall boundary detection computation.

5.1 Analysis of AGBS Algorithm

In order to analyze the I/O and CPU cost of the overall algorithm, we suppose the number of R (S) data pages is N_r (N_s). In the preprocessing phase, the PCA transformation needs to perform the sequential scan of R and S twice i.e. $2(N_r + N_s)$. Suppose that there are B buffer pages available in memory, the total I/O cost of sorting the pages using the external merge sort algorithm is given as below[1, 10]:

$$2N_r\left(\left\lceil \log_{B-1}\frac{N_r}{B}\right\rceil+1\right)+2N_s\left(\left\lceil \log_{B-1}\frac{N_s}{B}\right\rceil+1\right) \tag{4}$$

In the block join phase, suppose we allocate n_r buffer pages to R data and n_s buffer pages to S data. The I/O cost is [10]:

$$N_r + \frac{N_r}{n_r}.N_s \tag{5}$$

The total cost I/O cost can be represented as below:

$$2N_r\left(\left\lceil \log_{B-1}\frac{N_r}{B}\right\rceil+1\right)+2N_s\left(\left\lceil \log_{B-1}\frac{N_s}{B}\right\rceil+1\right) + N_r + \frac{N_r}{n_r}.N_s + 2(N_r+N_s) \tag{6}$$

In order to analyze the CPU computation cost, we need to consider the iteration of the grid blocks and the distance computation among the two points in the datasets. The CPU computation can be represented as given below:

$$B_r.\sum_{l=0}^{n}\sum_{b=0}^{2l+1}(|B_s|.C) \tag{7}$$

Where n is the level of surrounding adjacent iteration in data S when all the points in block page N_r have their corresponding kNN values. C is the cost required to compute the distance between p_r and p_s.

5.2 Runtime Performance Comparisons

5.2.1 kNN-Join: Gorder vs AGBS

Gorder[10] and BORDER[1] have already compared the performance of the various kNN join and boundary points detection algorithms where the they seem to be out performing the other algorithms. We have implemented and compared the Gorder based kNN-join with AGBS based kNN-join algorithms. Also we compared the BORDER and AGBS-BORDER based computation separately. We implemented these methods on a MS Windows XP v2002 OS with the Intel Core(TM)2 Duo T8100 @2.10GH processor, and 2GB RAM hardware settings using CPP as the programming language. We used the randomly generated geographical points using an online tool [22]. We created multiple input data files with of 500 points to 20000 points in order to create the test cases.

Fig. 5 shows comparison graphs in two parts – (a) 2000 data points with the fixed lowest level block size of 1.5 x 2 km rectangular size. The page size in the secondary storage was 13 x 24 km in the rectangular shape. Since the I/O cost is similar in both of the methods, we focused on doing iterations of the blocks and distance computation cost comparisons in these two methods. With the variable k settings ranging from 1 to 350, the AGBS method works better consistently through out all the test cases. Further, in the Fig. 5 (b), a comparison graph is shown when the value of

the k and the block sizes are fixed, however the data size is variable. The data points belong to the same geographical location, but with the greater density. The AGBS's performance is consistently better than the Gorder method.

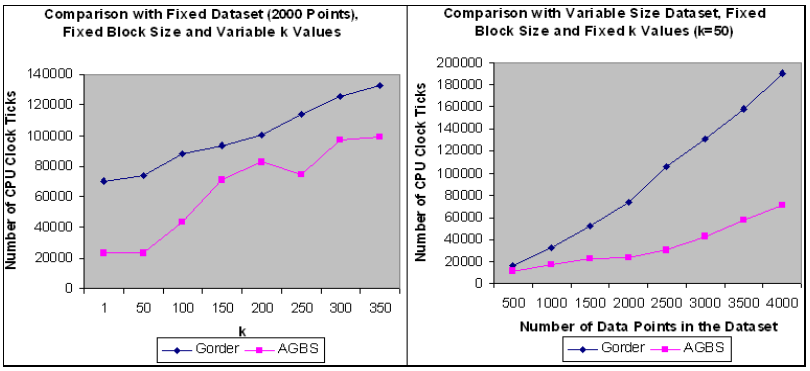

Fig.5. (a) Comparison between Gorder and AGBS with variable k values. (b) Comparison between Gorder and AGBS with variable dataset size.

5.2.2 Boundary Points Detection: BORDER vs AGBS-BORDER

The boundary points detection algorithms have been implemented and compared with the different data size and the variable k values. The AGBS based boundary points detection method i.e. AGBS-BORDER performs better than the Gorder based BORDER method.

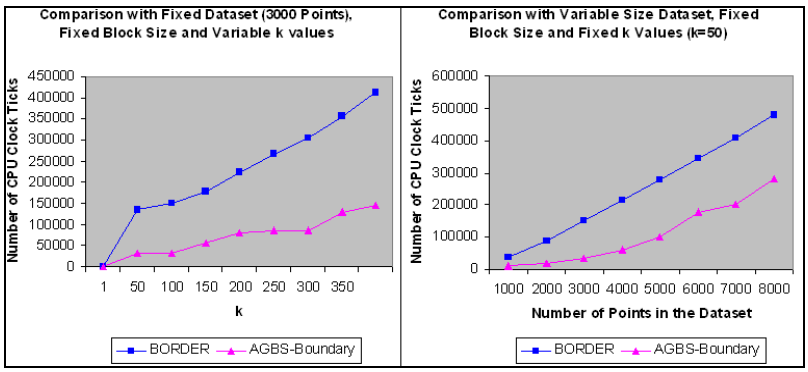

Fig.6. (a) Comparison between BORDER and AGBS-BORDER with variable k values. (b) Comparison between BORDER and AGBS-BORDER with variable dataset size.

Fig. 6 shows comparisons between BORDER and AGBS-BORDER in two aspects. The first part shows the runtime performance comparison with 3000 points and

variable number of k values. However the second part shows the comparison with different settings of fixed k (=50), fixed block size and variable data size. In all the scenarios the proposed AGBS based boundary points detection method performs better than the Gorder based BORDER method.

6 Limitations and Future Work

We have implemented both BORDER [1] and AGBS-BORDER methods for join of two dimensional geographical coordinates. We can easily extend the implementation and testing to high dimension dataset; however the results do not change much as the solution is generic to the high dimensional data. The runtime performance has been measured based on the CPU computation time needed for these methods separately. The experiments are planned to compare the I/O cost measurements in the future extensions. Since the AGBS-join iterates all the blocks, the ideal dataset for this method is the equally distributed dataset. The dataset that are already clustered and the distance between clusters is long, the AGBS's performance goes down. We need to extend the proposed solution in order to make it more robust against the dataset distribution in the data spaces. Finally, we are on the way testing the algorithm with bigger data size and different other environmental settings. The proposed method is based on the observation that boundary points tend to have fewer reverse k-nearest neighbors. This assumption is usually true when the dataset contains well-clustered data. However, for some real-world dataset such that the data are not well-clustered and the boundary is not so clear, this assumption may be invalidated and the solution might fail to find correct boundary points.

7 Conclusion

The AGBS-join based boundary detection method is based on the grid order based method BORDER [1]. The global grid structure is helpful on selectively reading the data blocks. A multi-level of blocks hierarchy has been used for efficient distance computation. The blocks are selected by picking the surrounding adjacent grids. We have implemented both the BORDER and AGBS-BORDER methods and tested on randomly generated geographical points. The AGBS based algorithm performs better than the existing Gorder based method. Since this method picks the closest block first, it picks the close data points first to compute the kNN neighbors. The experimental results reveal that the ADBS-BORDER is well suited method for the self-join.

References

1. Chenyi Xia Hsu et al (2006): BORDER: Efficient Computation of Boundary Points. In: Knowledge and Data Engineering, IEEE Transactions. Volume-18, Issue- 3, pp: 289 – 303. March 2006.

2. Bin Yao et al (2010): K Nearest Neighbor Queries and KNN-Joins in Large Relational Databases (Almost) for Free. In: Proccedings of Data Engineering (ICDE), 2010 IEEE 26th International Conference, pp: 4 -15, Long Beach CA, 2010.
3. Cui Yu et al (2006): Efficient index-based KNN join processing for high-dimensional data. In: Elsevier B.V. - Information and Software Technology Journal, 2006.
4. F. Korn et al (2000). Influence sets based on reverse nearest neighbor queries. In Proc. of ACM SIGMOD, pages 201–212, 2000.
5. Dongquan Liu et al (2008): Effective clustering and boundary detection algorithm based on Delaunay triangulation. Elsevier B.V., ScienceDirect Pattern Recognition Letters 29 (2008) 1261–1273.
6. Tiwari, S. et al (2011): A Survey on LBS: System Architecture, Trends and Broad Research Areas. In: DNIS 2011, LNCS 7108, pp. 223–241, 2011.
7. Guha et al (1998). CURE: An efficient clustering algorithm for large databases. In: Proc. 1998 ACM SIGMOD Internat. Conf. on Management of Data. ACM Press, New York, NY, USA, pp. 73–84.
8. Kaushik, S. et al (2011): Reducing Dependency on Middleware for Pull Based Active Services in LBS Systems. In: ICWCA 2011, LNICST 72, pp. 90–106, 2011.
9. Jagwani, P., et al (2011): Using Middleware as a Certifying Authority in LBS Applications. In: DNIS 2011, LNCS 7108, pp. 242–255, 2011.
10. Chenyi Xia wt al (2004): GORDER: An Efficient Method for KNN Join Processing. In: Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004.
11. Olga Sourina et al (2007): Visual Clustering and Boundary Detection of Time-Dependent Datasets. In: Proceedings of International Conference on Cyberworlds, 2007.
12. C. B¨ohm and H.-P. Kriegel (2001): A cost model and index architecture for the similarity join. In Proc. of ICDE, pages 411–420, 2001.
13. Sankaranarayanan, J. et al (2006): A Fast k-Neighborhood Algorithm for Large Point-Clouds. In: Proceedings of Eurographics Symposium on Point-Based Graphics (2006).
14. C. Bohm et al (2004): The k-nearest neighbor join: Turbo charging the KDD process. In: Knowledge and Information Systems (KAIS), London, UK, Nov, 2004.
15. Shim K. et al (1997): High-Dimensional Similarity Joins. In: Proceedings of IEEE Int. Conf. on Data Engineering, 1997.
16. Koudas N. et al (1997): Size Separation Spatial Join. In: Proceedings of Int. Conf. on Management of Data, 1997.
17. Koudas N. et al (1998): High Dimensional Similarity Joins: Algorithms and Performance Evaluation. In: Proceedings of IEEE Int. Conf. on Data Engineering (ICDE), Best Paper Award, 1998.
18. Tobias Emrich et al (2010): Optimizing All-Nearest-Neighbor Queries with Trigonometric Pruning. In: Proceeding of SSDBM'10 - the 22nd international conference on Scientific and statistical database management, 2010.
19. Cui Yu et al (2010): High-dimensional kNN joins with incremental updates. In: Journal of Geoinformatica, Volume 14 Issue 1, January 2010. Kluwer Academic Publishers Hingham, MA, USA.
20. Jonathon Shlens (2009): A Tutorial on Principal Component Analysis. Center for Neural Science, New York University New York City, NY. April 22, 2009.
21. Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data. In: Proceedings of SIGMOD '01 Proceedings of the 2001 ACM SIGMOD international conference on Management of data.
22. GeoMidPoint - Random Point Generator with Maps: http://www.geomidpoint.com/random/ Last accessed on 24 November 2011.