ADVANCED SIMILARITY QUERIES AND THEIR APPLICATION IN DATA MINING

Xia Chenyi

NATIONAL UNIVERSITY OF SINGAPORE
2005

ADVANCED SIMILARITY QUERIES AND THEIR APPLICATION IN DATA MINING

Xia Chenyi
(Bachelor of Engineering)
(Shanghai Jiaotong University, China)

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
2005

Summary

This thesis studies advanced similarity queries and their application in knowledge discovering and data mining. The similarity queries are important in various database systems such as multimedia, biological, scientific and geographic databases. In these databases, data are usually represented by d-dimensional feature vectors. The similarity of two data points is measured by the distance between two feature vectors. In this thesis, two variants of similarity queries - the k-Nearest Neighbor join (kNN join) and the Reverse k-Nearest Neighbor query (RkNN query) have been closely investigated and efficient algorithms for their processing are proposed. Furthermore, as one illustration of the importance of such queries, a novel data mining tool - BORDER which is built upon the kNN join and utilizes a property of the reverse k-nearest neighbor is proposed.

The kNN join combines each point of one dataset with its kNNs in the other dataset. It facilitates data mining tasks such as clustering and classification and is able to provide more meaningful query results than just the range similarity join. In this thesis, an efficient kNN join algorithm, Gorder (the *G-ordering* kNN join method) is proposed. *Gorder* is a block nested loop join method which achieves its efficiency by sorting data into the *G-order* that enables effective join pruning, data blocks scheduling and distance computation filtering and reduction. It utilizes a *two-tier partitioning strategy* to optimize I/O and CPU time separately and reduces distance computational cost by pruning redundant computation based the distance of fewer dimensions. It does not require an

index for the source datasets and is efficient and scalable with regard to both the dimensionality and the size of the input datasets. Experimental studies on both synthetic and real-world datasets are conducted and presented. The experimental results demonstrate the efficiency and the scalability of the proposed method, and confirm the superiority of the proposed method to the previous solutions.

The Reverse k-Nearest Neighbor (RkNN) query aims to find all points in a dataset that have the given query point as one of their k-nearest neighbors. Previous solutions are very expensive when data points are in high dimensional spaces or the value of k is large. In this thesis, an innovative *estimation-based* approach called ERkNN (the estimation-based RkNN search) is designed. ERkNN retrieves RkNN candidates based on the *local kNN-distance estimation* methods and verifies the candidates using the efficient *aggregated range query*. Two local kNN-distance estimation methods, the PDE method and the kDE method, are provided and both work effectively on uniform as well as skewed datasets. By employing the effective estimation-based filtering strategy and the efficient refinement procedure, ERkNN outperforms previous methods significantly and answers RkNN queries in high-dimensional data spaces and of large values of k efficiently and effectively.

To the end, we show how the kNN join and RkNN query can be utilized for data mining. We introduce a novel data mining tool - BORDER (a BOundaRy points DEtectoR) for effective boundary point detection. Boundary points are data points that are located at the margin of densely distributed data (e.g. a cluster). The knowledge of boundary points can help in data mining tasks such as data preparation for clustering and classification. BORDER employs the state-of-the-art kNN join technique Gorder and makes use of a property of the RkNN. Experimental study demonstrates BORDER detects boundary points effectively and can be used to improve the performance of clustering and classification analysis considerately.

In summary, the contributions of thesis is that we have successfully provided efficient solutions to two types of advanced similarity queries - the kNN join and the RkNN query and illustrated their application in data mining with a novel data mining tool - BORDER. We hope that ongoing research in similarity query processing will continue to improve the query performance and put forward more abundant data mining tools for users.

Acknowledgements

”In every end, there is a beginning. In every beginning, there is an end. In the middle, there is a whole mess of stuff.” This describes accurately my PhD candidature time, a very precious and memorable period of my life, in which there is an end and there is a beginning, in which there are happiness and joyfulness and also depression and sadness, in which the most precious and wonderful person in my life I was given, in which the most important and joyous transformation of my life happened, during which I have met people of various types and learned different knowledge from them, and during which the thesis has been worked on and is finally materialized. I am thankful to the One who gives me this epoch of life and all who have shared this period of life with me and helped me in all kinds of ways.

First, I would like to express my thanks to my supervisor, Professor Ooi Beng Chin and Dr. Lee Mong Li and Professor Wynne Hsu. I am thankful to their extraordinary patience on me, their guidance and all kinds of supports which they have given me generously. I also want to thank the professors I have worked with, Professor Lu Hongjun, Dr. Anthony Tung and Dr. David Hsu, who gave me lots of help ranging from refining ideas to drafting and finalizing the papers.

To my beloved parents and sister, together with my best friend, who are always trusting me and having confidence in me, always caring me and missing me, and always encouraging me and supporting me, I am longing to give them a tight and warm embrace

to express my unspeakable gratitude toward them.

Finally, I would like to thank all my colleagues of database and bioinformatics laboratories for their help and friendship. We have not only worked together but also shared our leisure time together. And I hope our friendship endures in our lives.

This thesis contains three pieces of the work that I have done as a PhD candidate and have been accepted by VLDB 2004, CIKM 2005 and TKDE respectively. I dedicate the thesis to the period of life when the thesis has been worked on, as a memorization of the end and the beginning.

Contents

List of Figures

Chapter 1

Introduction

Similarity queries are important operations for databases and have received much attention in the past decades. They have numerous applications in various areas such as Multimedia Information System [36, 47, 96], Geographical Information Systems [92, 97, 98, 48], Computational Biology research [64, 63], String and Time-Series Analysis applications [110, 51, 104, 132], Medical Information Systems [80], CAD/CAM applications, Picture Archive and Communication Systems (PACS) [39, 94] and data mining tasks such as clustering and outlier detection [52, 117, 130, 55, 22, 23, 75].

A similarity query operates on a dataset containing a collection of objects (e.g., images, documents and medical profiles). Each object in the dataset is represented by a multi-dimensional feature vector extracted by feature extraction algorithms [50]. For example, the features of an image can be the color histograms describing the distribution of colors in the image [46]. The similarity or dissimilarity between two objects is determined by a distance metric, e.g., Euclidean distance. There are five types of similarity queries: the range query, the k-nearest neighbor (kNN) query, the range similarity join, the kNN similarity join and the reverse k-nearest neighbor (RkNN) query. According to their computation complexities, they can be categorized into two groups - the *basic similarity query* which includes the range query and the kNN query, and the *advanced similarity query* which includes the range similarity join, kNN similarity join and the

RkNN query.

In this thesis, we examine the problem of two advanced similarity queries - the kNN similarity join and the RkNN query. Two novel algorithms - Gorder for efficient kNN join and ERkNN for approximate RkNN search are proposed.

Moreover, we conduct an initial exploration of utilizing the kNN similarity join and RkNN query for the data mining tasks. An interesting data mining tool - BORDER has been devised. BORDER is built on top of the kNN join algorithm Gorder utilizing the property of the reverse k-nearest neighbor. It can find boundary points efficiently and effectively.

In the following sections, we first define the similarity queries and then present the motivations of our study. At last, we give a summarization of the contribution of the study and present the outline of the thesis.

1.1 Similarity Queries

In this section, the basic concepts of the similarity queries are introduced. We first present formally the concepts of *dataset* and the *similarity* and then the definitions of the range query, the k-nearest neighbor (kNN) query, the range similarity join, the kNN similarity join and the reverse k-nearest neighbor (RkNN) query and categorize them according to their search complexity.

1.1.1 Data Representation

In similarity search applications, objects are feature-transformed into vectors with fixed length. Therefore, a dataset is a set of feature vectors (or points) in a d-dimensional data space D, where d is the length of the feature vector and the data space $D \subseteq \mathbb{R}^d$. Each data point p in a dataset is in the form

$$p =< x_1, ..., x_d >.$$

Definition 1.1.1 (Dataset)*: A dataset S is a set of N points in a d-dimensional data space D,*

$$S = \{p_1, ..., p_N\}$$
$$p_i \in D, i = 1, ..., N, D \subseteq \mathbb{R}^d.$$

N is number of objects in the dataset or the cardinality of the dataset.

1.1.2 Similarity

Similarity is measured by the distance between the feature vectors of two objects according to the given distance metric. The distance metric is application-dependant - one may choose different ways of measuring distance that are appropriate for different applications. The distance metric always satisfies the following conditions:

- Given two data points p and q ($p \neq q$), $Dist(p, q) > 0$;
- Given any point p, $Dist(p, p) = 0$;
- Given two data points p and q, $Dist(p, q) = Dist(q, p)$.

The commonly-used distance metrics are:

- L_ρ metric:

$$Dist_{L_\rho}(p, q) = \left(\sum_{i=1}^{d} |p.x_i - q.x_i|^\rho\right)^{1/\rho}, \ 1 \leq \rho \leq \infty$$

  Particularly, L_1 is called the Manhattan distance. It is also known as city block distance, boxcar distance, or absolute value distance. The distance between two data points are the sum of the absolute differences between coordinates of a pair of objects. Queries using Manhattan metric are rhomboid shaped.

$$Dist_{Manhattan}(p,q) = \sum_{i=1}^{d} |p.x_i - q.x_i|$$

L_2 is the Euclidean distance, which is the most widely applied distance metric. It is the straight line distance between two points. Queries using Euclidean distance are hyper-spheres.

$$Dist_{Euclidean}(p,q) = \left(\sum_{i=1}^{d} |p.x_i - q.x_i|^2\right)^{1/2}$$

L_∞ is called the maximum metric. Queries using maximum metric are hypercubes.

$$Dist_{maximum}(p,q) = max(|p.x_i - q.x_i|),\ 1 \leq i \leq d$$

- Weighted L_ρ metric:

$$Dist_{weightedL_\rho}(p,q) = \left(\sum_{i=1}^{d} w_i \cdot |p.x_i - q.x_i|^\rho\right)^{1/\rho},\ 1 \leq \rho \leq \infty$$

where w_i is the weight assigned to dimension i. Weighted L_ρ metric is a generalized L_ρ distance. There are weighted Manhattan distance, weighted Euclidean distance and weighted maximum distance correspondingly.

In the rest of the thesis, we use the most commonly used metric - Euclidean distance for demonstration purposes. The proposed methods can be extended to other distance metrics straightforwardly.

1.1.3 Range Query

A range query specifies a query range r in the predicate clause and asks questions like "What are the set of objects whose distance (dissimilarity) to the given query object are within r ?"

Definition 1.1.2 (Range Query)*: Given a dataset S, a query object q, a positive real r and a distance metric $Dist()$, the range query, denoted as $Range(q, r, S)$, retrieves all objects p in S such that $Dist(p, q) \leq r$.*

$$Range(q, r, S) = \{p \in S | Dist(p, q) \leq r\}$$

There is a special range query called the window query. The window query specifies a rectangular region which is parallel to the axis in data space and selects all data points inside of the hyper-rectangle. The window query can be regarded as a range query using the weighted maximum metric, where the weights w_i represent the inverse of the side lengths of the window.

1.1.4 kNN Query

The kNN query specifies a rank parameter k in the predicate clause and asks questions like "What are the k objects that are closest to or most similar to the given query object?"

Definition 1.1.3 (k-Nearest Neighbor Query) *Given a dataset S, query object q, a positive integer k and a distance metric $Dist()$, k-nearest neighbor query, denoted as $kNN(q, S)$, retrieves the k closest objects to q in R.*

$$kNN(q, S) = \{A \subseteq S | \forall p \in A, p' \in S - A, Dist(p, q) \leq Dist(p', q) \wedge |A| = k\}$$

1.1.5 Range Similarity Join

The range similarity join (range join in short) is the set-oriented range query. The range join has a set of query objects (the query set R) and retrieves objects which are within range r from the dataset S for each point in query set R. The result of a range join is a set of object pairs (p, q) such that $Dist(p, q) \leq r$, where p is from data set S and q is from query set R, . Query set R and the data set S can be the same dataset. In this case, the range join is called the self range join.

Definition 1.1.4 (Range Join) *Given one data set S and one query set R, a real r and a distance metric $Dist()$, the kNN join, denoted as $R \bowtie_r S$, returns pairs of points (p, q) such that q is from the outer query set R and p from the inner data set S, and $Dist(p, q) \leq r$.*

$$R \bowtie_r S = \{(p, q) | q \in R, p \in S, Dist(p, q) \leq r\}$$

1.1.6 kNN Similarity Join

The k-nearest neighbor similarity join (kNN join in short) is the set-oriented kNN query and combines each point of the query (outer) set R with its k-nearest neighbors from the inner data set S defined firstly in [18]. When R is equal to S, the kNN join is called the self kNN join[20].

Definition 1.1.5 (kNN Join) *Given one point dataset S and one query dataset R, an integer k and a distance metric $Dist()$, the kNN join, denoted as $R \ltimes_{kNN} S$, returns pairs of points (p, q) such that q is from the outer query set R and p from the inner data set S, and p is one of the k-nearest neighbors of q.*

$$R \ltimes_{kNN} S = \{(p, q) | q \in R \wedge p \in S \wedge p \in \textit{kNN(q, S)}\}$$

1.1.7 RkNN Query

The Reverse k-Nearest Neighbors (RkNN) query retrieves all objects in a dataset S that have the given query point q as one of their k nearest neighbors. The RkNN problem was first introduced in [78] and was also known as the influence set problem. The RkNN query has the mono-chromatic case and the bi-chromatic case.

In the mono-chromatic case, there is only one input dataset - the point dataset S.

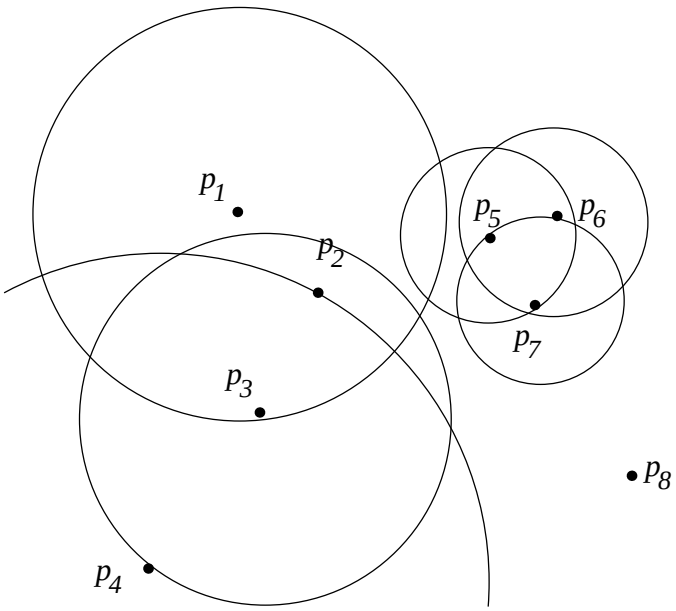


Figure 1.1: An example of mono-chromatic RkNN query.

Definition 1.1.6 (Mono-chromatic Reverse k-Nearest Neighbor Query) *Given a dataset S, query object q, a positive integer k and a distance metric $Dist(\ )$, mono-chromatic reverse k-nearest neighbor query, denoted as $RkNN(q, S)$, retrieves all objects p in S such that $Dist(p, q) \leq Dist(p, q')$, for $\forall\ q' \in kNN(p, S)$, where $kNN(p, S)$ are the k-nearest neighbors of point p in dataset S.*

$$RkNN(q, S) = \{p | p \in SDist(p, q) \leq Dist(p, q'), \forall \wedge q' \in kNN(p, S)\}.$$

In the bi-chromatic case, the RkNN query has two input datasets - the point dataset S and the query dataset R (also called *site* dataset in [115]). The query dataset R is different from the point dataset S. The query point q is from the site dataset R.

Definition 1.1.7 (Bi-chromatic Reverse k-Nearest Neighbor Query) *Given a point dataset S, a query dataset R, a query object $q \in R$, a positive integer k and a distance metric $Dist(\ )$, bi-chromatic reverse k-nearest neighbor query, denoted as $RkNN(q, R, S)$, retrieves all objects p in S such that $Dist(p, q) \leq Dist(p, q')$, for $\forall\ q' \in kNN(p, R)$, where $kNN(p, R)$ are the k-nearest neighbors of point p in dataset R.*

$$RkNN(q, R, S) = \{p | p \in SDist(p, q) \leq Dist(p, q'), \forall \wedge q' \in kNN(p, R)\}.$$

Figure 1.1 illustrates an example of the mono-chromatic RkNN query. Let dataset $S = \{p_1, p_2, ..., p_8\}$, p_2 be the query point and k=2. Since p_2 is one of the 2-nearest neighbors of p_1, p_3 and p_4, $R2NN(p_2, S) = \{p_1, p_3, p_4\}$.

1.1.8 Classification of the Similarity Queries

Both the range query and the kNN query are classified as the basic similarity query because of their comparatively low query cost. The naive solution to the range query (the sequential scan method) scans the dataset S sequentially, computes the distance of each object to the query object and then outputs the objects p such that $Dist(p, q) \leq r$. The naive solution to the kNN query maintains a sorted array of size k to store the k-nearest neighbor candidates. Similarly, it scans the dataset S sequentially. When it finds an object p that is closer to the query object q than the current k-th nearest neighbor candidate, it inserts p into the sorted array and removes the current k-th nearest neighbor from the candidates set. So both query is upper bounded by $O(N)$ and can be solved in $O(N)$ time by scanning the point dataset S sequentially. N is the cardinality of point dataset S. By utilizing the index techniques which will be introduced in Chapter 2, the complexity of both queries can be reduced to $O(logN)$ [16].

The range join and the kNN join are much more expensive than their single query counterparts. Naive approach to answer a range join or a kNN join performs the range query or the kNN query for each point in the query set R. This involves M (M is the cardinality of R) times scanning of the dataset S, which introduces tremendous distance computation and disk accesses. The query complexity of both the range join and the kNN join is upper-bounded by the $O(NM)$, where N is the cardinality of S and M is the cardinality of R. For the self range join or the self kNN join, their query complexity is upper-bounded by the $O(N^2)$, where N is the cardinality of S. Therefore, both queries

are categorized as the advanced similarity query.

Although the RkNN query only has one query point, it is also categorized as the advanced similarity query because of its high computation complexity. Note that the k-nearest-neighbor relation is not symmetric, that is, if p is one of q's k-nearest neighbors, q is not necessary to be one of p's k-nearest neighbors. Therefore, the RkNN query is much more complex than the kNN query. The naive solution for RkNN search has to first compute the k-nearest neighbors for each point p in the dataset S (for the mono-chromatic RkNN query) or R (for the bi-chromatic RkNN query). Then points p whose distance from the query point $Dist(p, q)$ is equal or less than the distance between p and its k-th nearest neighbor can be determined as q's reverse k-nearest neighbors. The complexity of the first step is equal to the kNN join, so the complexity is upper-bounded by $O(N^2)$ for mono-chromatic case and $O(NM)$ for the bi-chromatic case. The second step is a sequential scan of the dataset S. Therefore, it is also categorized as the advanced similarity query.

1.2 Motivation

In the section, we describe the interesting applications of the kNN join, the RkNN query and a specially property of the number of a point's reverse k-nearest neighbors, which motivated our research.

1.2.1 Motivation of the Study of the kNN Join

The kNN-join, with its set-oriented nature, can be used to efficiently support many important data mining tasks which have wide applications. In particular, it is identified that many standard algorithms in almost all stages of knowledge discovery process can be accelerated by including the kNN join as a primitive operation. For examples,

- Outlier analysis. Outlier analysis is to find out data objects that do not comply with the general behavior or model of the data [52]. It has important applications such as the fraud detection (detecting malicious use of credit card or mobile phone), customized marketing (identifying the spending behavior of customers with extremely low or extremely high incomes) or medical analysis (finding unusual responses to various medical treatments) [52]. In the first step of LOF [23](a density-based outlier detection method), the k-nearest neighbors for every point in the input dataset are materialized. This can be achieved by a single self kNN-join of the dataset.

- Data Classification. Data classification predicts the new data objects' categorical labels according to the model built according to a set of objects with known categorical labels (the training set). The knowledge of the new objects' category can be used for making intelligent business decisions. For example, it can be used to analyze the bank loan applicants to identify the loan is either safe or risky. It also can be used in the medical expert system to diagnose the patients. The k-nearest neighbor classifier is one of the simplest but effective classification methods which identifies the new object's category by examining that object's k-nearest neighbors in the training set. The unknown sample is assigned the most common class among its k-nearest neighbors. Given a set of unlabelled objects (the testing set), the kNN join can be used to classify them efficiently by joining the testing set with the training set.

- Data Clustering. Clustering is the process of grouping a set of physical or abstract objects into classes of similar objects so that important data distribution patterns and interesting correlations among data attributes can be identified [52]. It is also known as the *unsupervised learning* and has wide applications such as pattern recognition, image processing, market or customer analysis and biological

research. The kNN join can be used in many clustering algorithms to accelerate the process.

In each iteration of the well-known k-means clustering process [54], the nearest cluster centroid is computed for each data point. A data point is assigned to the its new nearest cluster if the previously assigned cluster centroid is different from the currently computed one. A kNN join with $k = 1$ between the data points and the cluster centroids can thus be applied to find all the nearest centroid for all data points in one operation.

In the hierarchical clustering method called Chameleon [72], a kNN-graph (a graph linking each point of a dataset to its k-nearest neighbors) is constructed before the partitioning algorithm is applied to generate clusters. The kNN-join can also be used to generate the kNN-graph.

Compared to the traditional point-at-a-time approach that computes the k-nearest neighbors for all data points one by one, the set oriented kNN join can accelerate the computation dramatically [19].

However, after the kNN join has been proposed recently in [20], to the best of our knowledge, the MuX kNN join [20, 19] is the only algorithm that has been specifically designed for the kNN-join. The MuX kNN join algorithm is an index-based join algorithm and MuX [21] is essentially an R-tree based method. Therefore, it suffers as an R-tree based join algorithm. First, like the R-tree, its performance is expected to degenerate with the increase of data dimensionality. Second, the memory overhead of the MuX index structure is high for large high-dimensional data due to the space requirement of high-dimensional minimum bounding boxes. Both constraints restrict the scalability of the MuX kNN-join method in terms of dimensionality and data size.

As a consequence, new algorithms for efficient support of the kNN join in high-dimensional spaces are highly desired. In this thesis, we design Gorder (the G-ordering

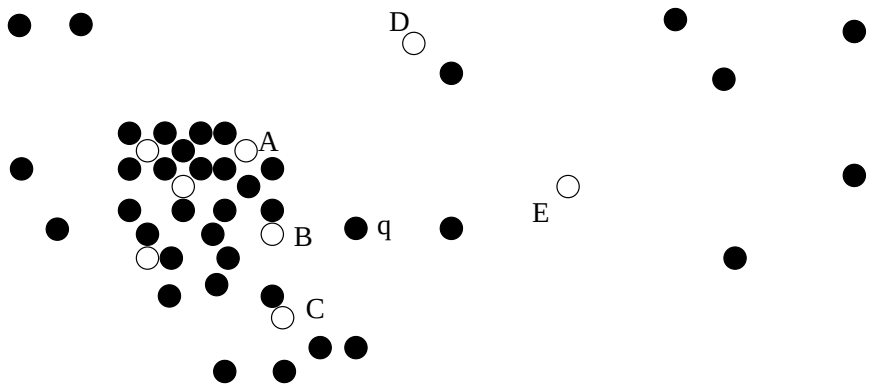


Figure 1.2: An illustration of resource allocation with quota limit.

kNN join) which is based on the block nested loop join and exploits optimization techniques such as sorting, data blocks scheduling, distance computation filtering and reduction to improve the query efficiency.

1.2.2 Motivation of the study of the RkNN Query

The RkNN query has received much attention in the recent years because of its important applications in profile-based marketing, information retrieval, decision support systems, document repositories and management of mobile devices [78, 115, 125, 114, 76]. For examples,

- Decision support. The knowledge of the reverse k-nearest neighbors enables a decision maker to arrive at the best trade-off decisions. For example, when two banks are to be merged, many branches have to be closed and services have to be redistributed. The decision as to which branches to close and how to reallocate the services requires the knowledge of the existing customers who view the branch among their top k preferred branches. For any two branches, if there is a big overlap between two such sets of customers, one of the branches can possibly be closed without sacrificing the quality of service to the customers.

- Profile-based Marketing. The RkNN query helps a company to have insights into the attractiveness of the products/services offered, and thus enable the tailored marketing. For example, a telecommunication company may offer many types of package targeting different groups of consumers. The knowledge that which customers will find the package the most suitable plan can assist the marketing department in recommending the most appropriate package tailored to the customers. These customers form the influence set of the package and can be determined by an RkNN query based on the the distance between the profiles of the customers and the feature vector representing the new package.

- Resource Allocation with Quota Limit. Consider Figure 1.2. Suppose each unfilled circle $'\circ'$ denotes a resource with a quota limit of 3. In other words, each resource can serve at most 3 filled points $'\bullet'$ which denote clients. If we wish to determine which recourse should be assigned to serve q, we may do so by looking for the nearest resources of q, e.g. the 3 nearest resources A, B, C. However, checking for quota limit, we realize that none of the A,B, nor C, can serve q because they each have 3 nearest neighbors that they are serving already. Instead, issuing a reverse 3-nearest neighbor query on the resource points, immediately we know D, E will consider q as one of their 3-nearest neighbors. Hence, we can assign either D or E to serve q.

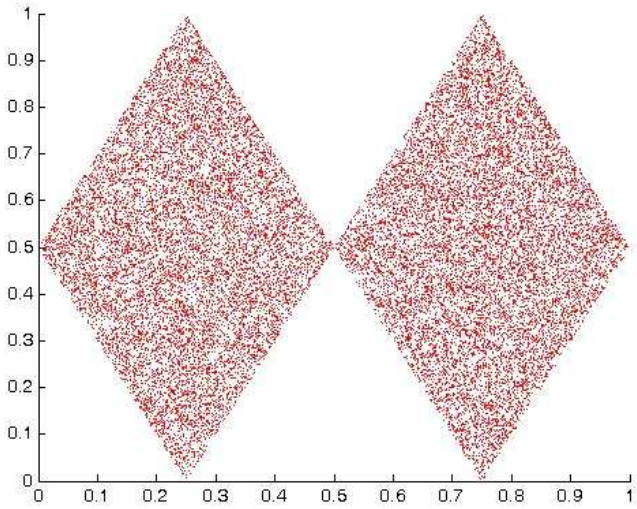
- Risk profiling in medical system [61]. It is often necessary to know the risk profile of each patient in order to recommend a most effective care strategy for the patient. One way to determining the risk profile of a patient is to classify the patient into a risk group according to the characteristics of the patient and the features characterizing different risk groups using the RkNN query.

A number of methods have been developed for the efficient processing of RkNN queries. They can be divided into two categories: *pre-computation* and *space pruning*. *Pre-computation* methods [78, 125] pre-compute the nearest neighbors of each point in the datasets and store the pre-computed information in hierarchical structures. This approach cannot answer an RkNN query unless the corresponding k-nearest neighbor information is available. *Space pruning* methods such as [112, 116, 114] utilize the geometry properties of RNN to find a small number of data points as candidates and then verify them with NN queries or range queries. However, these methods are all very expensive when data dimensionality is high or when the value k is large. Designing efficient search algorithm for the RkNN query in high-dimensional spaces is challenging and interesting. In this thesis, we overcome the difficulty of the RkNN query with estimation techniques. The ERkNN - an estimation-based RkNN search algorithm is put forward.

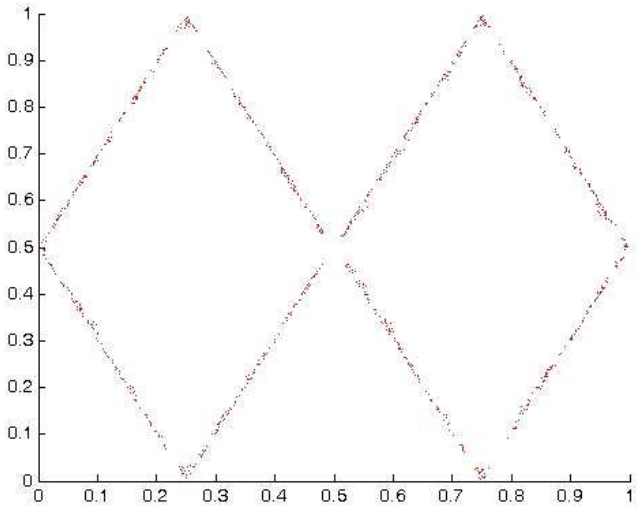
1.2.3 Motivation of BORDER

Data mining, also known as knowledge discovery in database, is the process of finding new and potentially useful knowledge from data. Advancements in information technologies have led to the continual collection and rapid accumulation of data in repositories. Turning such data into useful information and knowledge is desired. Consequently, numerous data mining technologies, including data cleaning and preparation techniques, data classification, association rules analysis, data clustering, and outlier analysis [52], have been proposed in the recent years.

In this thesis, we propose a novel data mining tool - BORDER for effective boundary point detection which is based on the finding that data points that have much fewer reverse k-nearest neighbors tend to locate at the margin of densely distributed data. As illustrated in Figure 1.3 (a), there is a 2-dimensional dataset with quadrangle-shaped



(a)



(b)

Figure 1.3: A preliminary study.

clusters. In Figure 1.3 (b), we plot the points whose reverse 50-nearest neighbors are fewer than 30 points. The plot shows that those points having fewer reverse k-nearest neighbors clearly define the boundaries of the clusters.

Boundary points are potentially useful in data mining applications since first, they represent a subset of population that are at the verge of the densely-distributed region and possibly straddle two or more classes. For example, this set of points may denote a subset of population that should have developed certain diseases, but somehow they do not. Special attention is certainly warranted for this set of people since they may reveal some interesting characteristics of the disease. Secondly, the knowledge of these points is also useful for data mining tasks such as classification and clustering [67] since these points are most likely to be mis-classified and mis-clustered. Removing such points before the classification or clustering analysis could improve the classification or clustering results.

Motivated by the usefulness of boundary points in data mining and the interesting observation of the relationship between the location of a point and its number of reverse k-nearest neighbors, we design BORDER, a data mining tool which finds the boundary points efficiently and effectively.

1.3 Contributions

The major contributions of this dissertation are three-fold:

1. A novel kNN-join algorithm, called *Gorder* (or the G-ordering kNN join method), is proposed to answer the kNN join operation efficiently. *Gorder* is a block nested loop join method which achieves its efficiency by sorting data based on an ordering that enables effective join pruning, data blocks scheduling and distance computation filtering and reduction. It utilizes a *two-tier partitioning strategy* to optimize

I/O and CPU time separately and reduces distance computational cost by pruning redundant computation based the distance of fewer dimensions. It does not require an index for the source datasets and is efficient and scalable with regard to both the dimensionality and the size of the input datasets. Experimental studies on both synthetic and real-world data sets are conducted and presented. The experimental results demonstrate the efficiency and the scalability of the proposed method, and confirm the superiority of the proposed method to the previous solutions.

2. An innovative *estimation-based* approach called ERkNN (the estimation-based RkNN search) is designed to handle RkNN queries in high-dimensional data spaces and for large values of k. ERkNN retrieves RkNN candidates based on the *local kNN-distance estimation* (kNN-distance is the distance from a data point to its k-th nearest neighbor) and verifies the candidates using an efficient *aggregated range query*. Two local kNN-distance estimation methods, the PDE method and the kDE method, are provided, which work effectively on both uniform and skewed datasets. Employing the effective estimation-based filtering strategy and the efficient refinement procedure, ERkNN outperforms previous methods by a significant margin. Extensive experiments on various datasets proves that ERkNN retrieves the reverser k-nearest neighbors efficiently and accurately.

3. A novel data mining tool, BORDER (a BOundaRy points DEtectoR) is proposed to detect boundary points. Boundary points are data points that are located at the margin of densely distributed data (e.g. a cluster). The knowledge of boundary points can help in data mining tasks such as data preparation for clustering and classification. BORDER detects boundary points according to the finding that data points that are located at the margin of densely distributed data tend to have much fewer reverse k-nearest neighbors. It transforms the expensive set-oriented RkNN query into the kNN join by utilizing the *reversal-ship* between the k-nearest neigh-

bor and the reverse k-nearest neighbor and employs the state-of-the-art kNN join technique - Gorder. Experimental study shows that BORDER finds the boundary points effectively. Moreover, the performance of the clustering and classification analysis can be improved considerably by removing the boundary points in advance.

1.4 Organization

The rest of the thesis is arranged as follows:

- Chapter 2 presents a survey of related work of similarity queries with particular focus on the kNN join and the RkNN query.
- Chapter 3 investigates the kNN join. Gorder, an efficient kNN join processing algorithm that exploits sorting, data page scheduling and distance computation filtering and reduction to reduce both I/O and CPU costs is proposed.
- In Chapter 4, we study the problem of the RkNN query . An innovative *estimation-based* solution -ERkNN (the estimation-based RkNN search) which can efficiently handle RkNN queries in high-dimensional data spaces and for large values of k is provided.
- Chapter 5 presents BORDER - a data mining tool for boundary points detection. We propose a novel method BORDER (a BOundaRy points DEtectoR) which employs the state-of-the-art kNN join technique and makes use of the property of the RkNN.
- Chapter 6 concludes the thesis with a summary of our contributions and a discussion of the future research.

Chapter 2

Related Work

In order to process similarity queries efficiently, numerous indexing techniques and search algorithms have been proposed in the recent decades. In this chapter, we first introduce the indexing techniques and algorithms for the basic similarity search with index, and then review algorithms for the advanced similarity queries, i.e., the range join, the kNN join and the RkNN query.

2.1 Index Techniques

Database Index is a mechanism to locate and access data within a database [1, 107, 91]. Given a dataset for similarity search, we build an index upon the feature vectors (which are keys) of the input dataset first and then apply the similarity search algorithms. Utilizing the index structures, the search algorithms can effectively locate data which are highly likely to be the answers, prune away those that are surely not answers, and retrieve data points that meet the query condition more efficiently. Numerous index structures have been proposed. They can be classified into three classes: data partitioning methods, space partitioning methods, and data transformation methods.

- Data partitioning methods: data partitioning methods group (or cluster) nearby (similar) data points together and organize them in multi-layered hierarchical structures. The R-tree family [49, 10, 111, 12], the A-tree [109]), the MuX index [21],

the SS-tree [121], the M-tree [131, 29], and the SR-tree [73] all belong to this category.

- Space partitioning methods: Space partitioning structures partition the data space iteratively along predefined lines regardless of the distribution of data. Space partitioning methods include the multi-dimensional hashing [83, 34, 85, 43, 86], grid-files [57, 95, 40, 120, 56, 14], kdB-trees [8, 9], hB-tree [89] etc.

- Data transformation methods: Data transformation methods transform the original d-dimensional data into single attribute values (or codes) and then index them with the one dimensional index structures such as B-trees [99] or simply stored them in a flat file. Such methods include the pyramid tree [11], iminmax [100, 127, 101], iDistance [129, 128], the space filling curves [102, 35, 66, 93], and the VA-file [119, 118].

Compared with the space partitioning methods, data partitioning methods are more adaptive to the data distribution and work more efficiently on real life and skewed distributed datasets. However, in high-dimensional space, data partitioning structures are seriously affected by the *curse of dimensionality* [11] problem and a similarity search based on an index could perform even worse than a simple search which scans the dataset sequentially (called the *sequential scan*). The data transformation methods are usually the most effective index methods for data of very high dimensionality.

Recently a number of dimensionality reduction techniques - the discrete fourier transform (DFT) [5], the discrete wavelet transform (DWT) [82, 106, 122], the principal component analysis (PCA) [53, 77, 71, 26] (also known as the single value decomposition) have been proposed. Dimensionality reduction techniques reduce data dimensionality by condensing the important information into a smaller number features. Some improved indexing methods [30, 68] utilize dimensionality reduction techniques so that they are

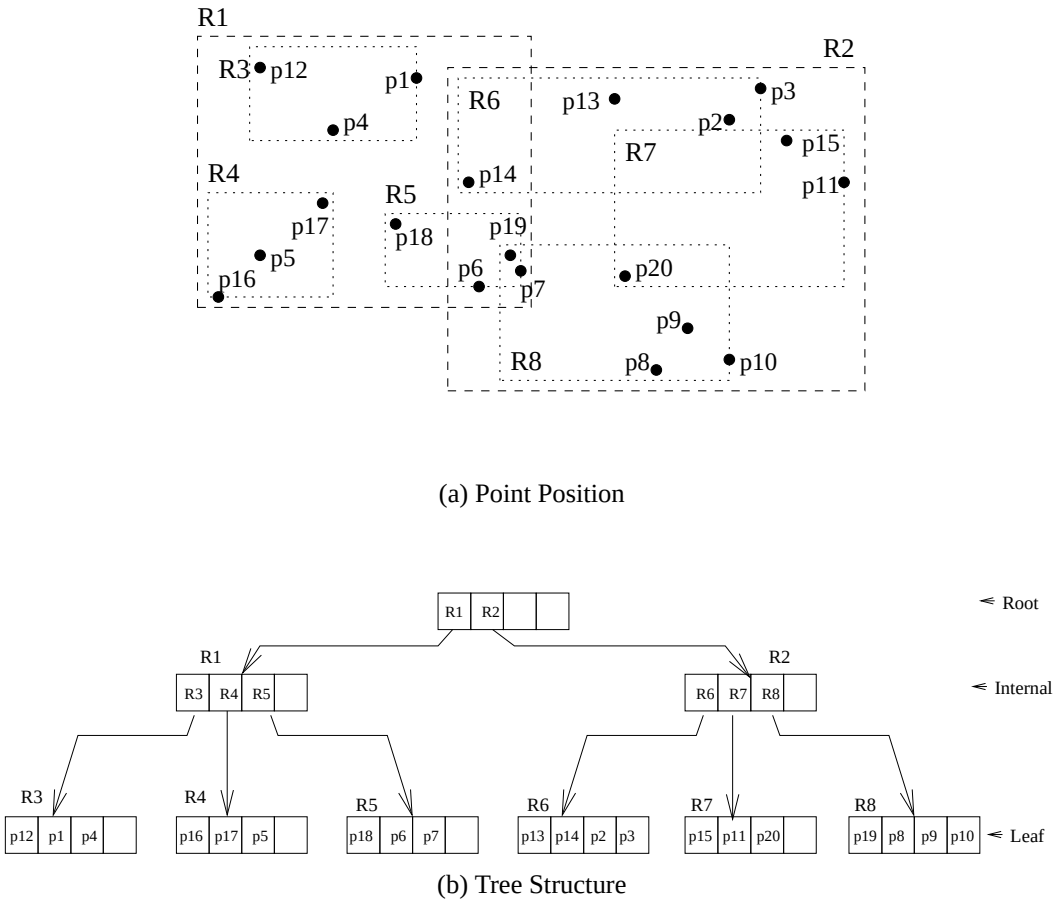


Figure 2.1: An R-tree Example

less affected by the problem of the *curse of dimensionality* and more scalable to high-dimensional spaces.

The comprehensive surveys of the multidimensional index structures can be found in [42, 16, 13, 126].

2.2 Basic Similarity Queries with Index

2.2.1 The R-tree

In the following discussions, we use the R-tree to illustrate the similarity search algorithms with index. We first give a detailed description of the R-tree [49].

Figure 2.1 illustrates an R-tree example. The R-tree belongs to data partitioning methods. As most of the other index structures, it is designed primarily for secondary storage and each tree node in the hierarchical tree corresponds to a page of the secondary storage. Nodes at the lowest level are called the *leaf nodes* or *data nodes*. Nodes at all other layers of the tree are called the *directory nodes* or *internal nodes*. The only node at the highest level of the tree is called the *root* of the tree. An R-tree is *height-balanced*, i.e., the lengths of the paths from the root to all data nodes are identical. The length of a path between the root and a data page is called the tree height.

Each entry e contained in the internal nodes are of the form of $(Rect, pointer)$. $pointer$ points to a node underneath. The node is called the *child node* of e. $Rect$ is a minimal bounding rectangle (MBR) that bounds the data objects in the subtree rooted at the child note pointed by $pointer$. The data points (or feature vectors) are stored in the data nodes of the R-tree.

The number of entries stored in every internal node of the R-tree has a lower bound m and upper bound M (except the root which has no lower bound). M is called the fanout of the tree. It is the maximal number of entries can be stored in an internal node and can be derived from the predefined page size of the R-tree and the size of an entry.

$$M = \frac{\text{page size of the R-tree}}{\text{size of an entry}}$$

m is defined to ensure efficient storage utilization.

$$m \ll \frac{M}{2}$$

The R-tree allows inserting and deleting data points dynamically. When a new data point is inserted into the tree, the insertion algorithm first routes the new data from the root node to a leaf node by picking a child node that needs least enlargement of the MBR to enclose the new data point. If the insertion causes overflow (i.e., the number of entries in a node is greater than its capacity), the node will be split. To remove a data point, the deletion algorithm traverses the tree to locate the leaf node containing the point and then removes it from the node and shrinks the MBR. The deletion of a data point may cause underflow (i.e., the number of entries stored in a node is smaller than the lower bound). In this case, the node will be removed and all data points inside will be reinserted into the tree.

The R-tree works effectively for data spaces of relatively small number of dimensions. But its performance degrades rapidly when the number of data dimensions increases. Variants methods have been proposed to improve the R-tree. The R*-tree [10] employs the *forced reinsert* policy and a sophisticated node-splitting policy to improve the storage utilization of the R-tree and minimize the combination of overlap between bounding rectangles and their total area. The R^+-tree [111] uses clipping to prevent overlap between bounding rectangle at the same tree level to overcome the problems associated with overlapping regions in the R-tree. The X-tree [12] introduces the *supernode* which are of larger page size into the R*-tree. The A-tree [109] (Approximation tree) replaces minimum bounding rectangles (MBRs) in the internal nodes with virtual bounding rectangles (VBRs) which represents MBRs approximately and compactly and thereby, increases the fanout of the tree and reduces the tree height.

Since the R-tree is the most fundamental hierarchical index structure, most similarity

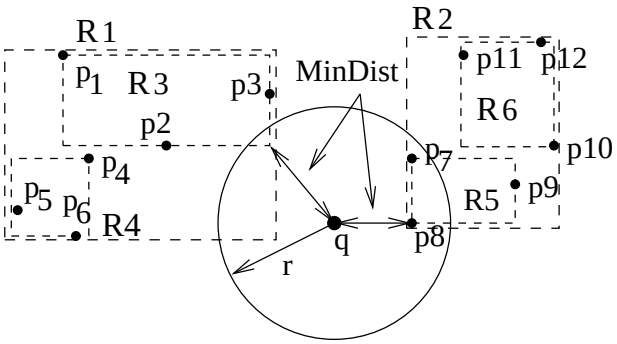


Figure 2.2: A Query Example

query algorithms are developed upon it and they can be migrated to other hierarchical index structures straightforwardly.

2.2.2 Algorithms for the Range Query

Search algorithms for the range query utilizing the R-tree traverses the tree in a branch-and-bound manner. It starts from the root of the tree. Upon visiting an internal node of the R-tree, the search algorithm calculates the MinDist (Definition 3.3.3) between each entry inside and the query point and applies the following Pruning Strategy 2.2.1 to decide whether the child node pointed by this entry should be visited.

Pruning Strategy 2.2.1 *If $MinDist(R, q) > r$, then node R can be pruned from search because it cannot contain any points p such that $Dist(p, q) \leq r$.*

$MinDist(R, q)$ is the minimum distance between the minimum bounding rectangle of node R and the query point q (see Figure 2.2 as an illustration).

Definition 2.2.1 (MinDist between MBR and point) *The minimum distance between the minimum bounding rectangle of node R and a point $q(x_1, x_2, ..., x_d)$, denoted as*

$MinDist(R, q)$, *is defined as follows:*

$$MinDist(R,q) = \sum_{i=1}^{d} \left(\begin{cases} lb_i - q.x_i & if\ q.x_i < lb_i \\ 0 & otherwise \\ q.x_i - ub_i & if\ ub_i < q.x_i \end{cases} \right)^2$$

where lb_i *is the lower bound of the minimum bounding rectangle at dimension* i *and* ub_i *is the upper bound of the minimum bounding rectangle at dimension* i*.*

Upon visiting a data node, the search algorithm calculates the distances between the data points p and the query point. Data points such that $Dist(p, q) \leq r$ are output as the results of the range query.

Different range query algorithms traverse the tree nodes in different sequences. The depth-first algorithm always visits the unpruned child node first and the breadth-first algorithm always visits the qualified sibling node first. The depth-first algorithm is implemented in a recursive way and the breadth-first algorithm is implemented in an iterative way.

Figure 2.2 gives a range query example, where q is the query point and r is the query radius. The depth-first range search algorithm visits the tree nodes in the following sequence:

$$\text{tree root} \Longrightarrow R_1 \Longrightarrow R_3 \Longrightarrow R_2 \Longrightarrow R_5$$

The breadth-first range search algorithm visits the tree nodes in the following sequence:

$$\text{tree root} \Longrightarrow R_1 \Longrightarrow R_2 \Longrightarrow R_3 \Longrightarrow R_5$$

Nodes R_4 and R_6 are discarded by applying Pruning Strategy 2.2.1, which saves both I/O and CPU costs.

2.2.3 Algorithms for the kNN Query

The kNN query is more complex than the range query because the query range is unknown first. The kNN search algorithms maintain an array of size k to store the k-nearest neighbor candidates (the kNN candidate array). The distance of the kth nearest neighbor candidate to the query point $dnn_k(q)$ (called the kNN-distance of q) is used for pruning tree nodes. The following pruning strategy is adopted by the kNN query algorithms.

Pruning Strategy 2.2.2 *If $MinDist(R, q) > dnn_k(q)$, node R can be pruned from search because it cannot contain any points p that are closer to the query point than the current k-nearest neighbor candidates.*

The pruning distance $dnn_k(q) = Dist(c_k, q)$, where $\{c_1, ..., c_k\}$ are the k-nearest neighbor candidates sorted in ascending order accordingly to their distances to the query point. $dnn_k(q)$ is ∞ at the beginning of the search and converges during the search.

There are three types kNN search algorithms: the depth-first method, the best-first method and the incremental method [108, 58, 59].

Depth-first kNN Search Algorithm

The depth-first search algorithm [108] accesses a tree node in the following way:

- If the node is an internal node, the depth-first search algorithm first sorts the entries inside of the node according to their minimum distances to the query point. Then, starting from the first entry with the minimum MinDist, the algorithm calls recursively the depth-first search algorithm for the child node pointed by the entry if the entry cannot be pruned by Pruning Strategy 2.2.2.

- If the node is a leaf node, it computes the distance between each data point and the query point and inserted data points p such that $Dist(p, q) < dnn_k(q)$ into the kNN candidate array.

Visit Node	$dnn_k(q)$	Candidate Set
Tree root	∞	$\emptyset$
R_1	∞	$\emptyset$
R_3	$Dist(q, p_2)$	$\{p_3, p_2\}$
R_2	$Dist(q, p_2)$	$\{p_3, p_2\}$
R_5	$Dist(q, p_7)$	$\{p_8, p_7\}$

Table 2.1: The procedure of the depth-first 2-nearest neighbor search.

Given a kNN query (k=2) in Figure 2.2 where q is the query point, the search algorithm visits the tree nodes in the following sequence:

$$\text{Tree Root} \rightarrow R_1 \rightarrow R_3 \rightarrow R_2 \rightarrow R_5.$$

Nodes R_4 and R_6 are pruned from search and p_7 and p_8 are found as the 2-nearest neighbor of q. Table 2.1 summarizes the procedure of the depth-first search.

Best-first kNN Search Algorithm

The best-first approach [58] uses a priority queue to maintain the nodes that shall be visited. Entries in the priority queue are sorted in ascending order according to their MinDist to the query point. In the initial, only the tree root is in the priority queue. The search algorithm always dequeues the first entry in the priority queue and processes according to the node type:

- If a node is an internal node, for each entry inside, the algorithm computes its MinDist to the query point and inserts it into the priority queue if its MinDist is smaller than $dnn_k(q)$.
- If a node is a leaf node, for each data in the node, the search algorithm calculates its distance from the query point. If the distance is smaller than $dnn_k(q)$, the data point is inserted into the kNN candidate array.

Visit Node	$dnn_k(q)$	Priority Queue	Candidate Set
Tree root	∞	$\{R_1, R_2\}$	$\emptyset$
R_1	∞	$\{R_2, R_3, R_4\}$	$\emptyset$
R_2	∞	$\{R_5, R_3, R_6, R_4\}$	$\emptyset$
R_5	$Dist(q, p_7)$	$\{R_3, R_6, R_4\}$	$\{p_8, p_7\}$

Table 2.2: The procedure of the best-first 2-nearest neighbor search.

The process is stopped when $dnn_k(q)$ is smaller than the MinDist of the first entry in the priority queue or the priority queue is empty.

For the 2NN query in Figure 2.2, the nodes will be visited in the following sequence:

$$\text{tree root} \rightarrow R_1 \rightarrow R_2 \rightarrow R_5.$$

The algorithm stops after visiting R_5 and output p_8 and p_7 as results because $MinDist(R_3, q)$ is greater than $dnn_k(q) = Dist(q, p_7)$. Nodes R_3, R_6, R_4 which are remained in the priority queue are neglected. Compared with the depth-first method, the best-first approach is more efficient because it accesses fewer tree nodes. In this example, the depth-first approach accesses 5 nodes in total, while the best-first approach accesses only 4 nodes. Table 2.2 summarizes the procedure of the best-first kNN search.

Incremental kNN Search Algorithm

The incremental k-nearest neighbor search [59] is very similar to the best-first approach and also employs a priority queue to maintain the nodes that shall be visited. However, the incremental search algorithm stores both the tree nodes and the data points in the priority queue and therefore, need not maintain a separate kNN candidate array.

The priority queue is initialized with the tree root. Items in the queue is sorted in ascending order according to their distance or MinDist to the query point. At each iteration, the incremental kNN search algorithm dequeues the first entry from the queue and

Entry Dequeued	Priority Queue	Output
$root$	$\{R_1, R_2\}$	
R_1	$\{R_2, R_3, R_4\}$	
R_2	$\{R_5, R_3, R_6, R_4\}$	
R_5	$\{p_8, p_7, R_3, R_6, p_9, R_4\}$	
p_8	$\{p_7, R_3, R_6, p_9, R_4\}$	p_8 as the 1st nearest neighbor
p_7	$\{R_3, R_6, p_9, R_4\}$	p_7 as the 2nd nearest neighbor
R_3	$\{p_3, R_6, p_2, p_9, R_4, p_1\}$	
p_3	$\{R_6, p_2, p_9, R_4, p_1\}$	p_3 as the 3rd nearest neighbor

Table 2.3: The procedure of an incremental k-nearest neighbor search.

checks its type.

- If the item is a data point, it is reported as a next nearest neighbor.
- If the item is an internal node, for each entry inside, the algorithm computes its MinDist to the query point and inserts it into the priority queue.
- If the item is a leaf node, for each data point inside, the algorithm computes its distance to the query point and inserts it into the priority queue.

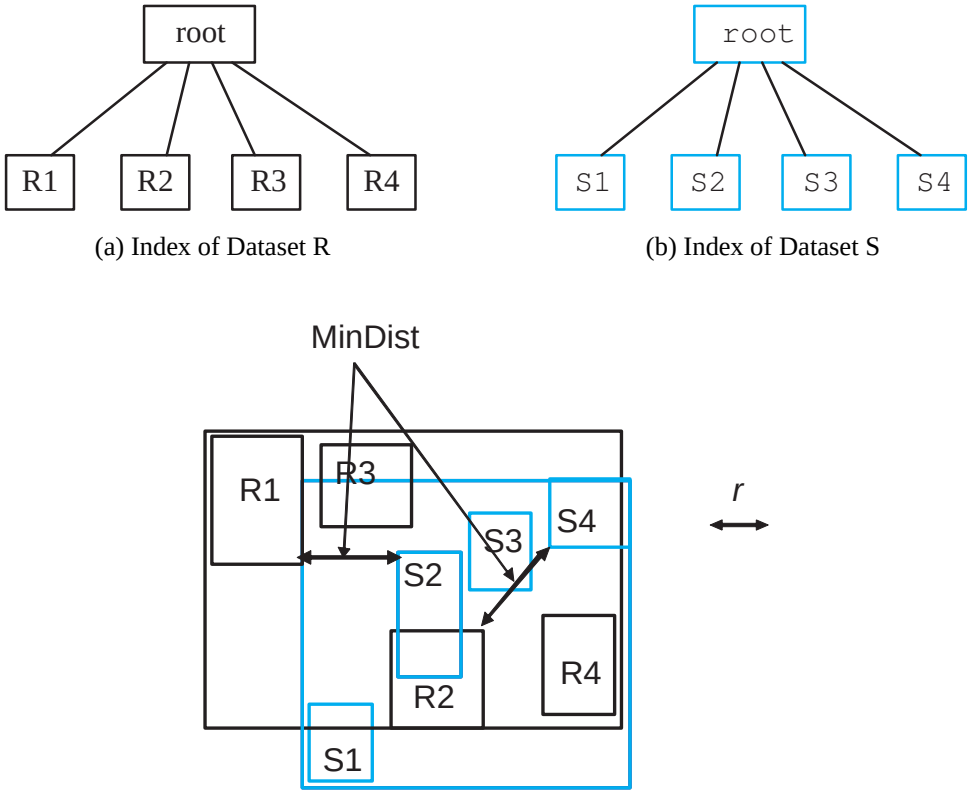
The algorithm stops when there are k points being reported as k-nearest neighbors.

Given the 2NN query in Figure 2.2, the incremental search algorithm traverses the following nodes:

$$\text{tree root} \rightarrow R_1 \rightarrow R_2 \rightarrow R_5.$$

Table 2.3 summarizes the search procedure of the incremental 2NN query in Figure 2.2.

The incremental approach accesses the same number of tree nodes as the best-first approach in the same sequence. Both are proved to be optimized in terms of the number of node accesses [16]. An advantage of the incremental approach over the best-first approach is that after the k nearest neighbors are found, the $(k+1)$th nearest neighbor can be produced immediately without reinvoking the query algorithm and processing the query from scratch by utilizing the information in the priority queue. As shown in



(a) Index of Dataset R

(b) Index of Dataset S

(c) The Planar Presentation

Figure 2.3: An RSJ Join Example

Table 2.3, after finding the 2-nearest neighbors of q, the incremental algorithm can find the third nearest neighbor p_3 quickly.

2.3 Algorithms for the Range Similarity Join

The range similarity join has been well-studied and a large number of techniques have been proposed for it. They can be broadly classified into three categories: the index-based algorithms, the hash-based algorithms and the sort-based algorithms.

Note that some join algorithms which we will introduce in the following sections are designed originally for the spatial join [1]. They can be applied to the range similarity join

[1]The spatial join computes the pairs of the spatial objects that intersect with each other.

by treating the data points p as hypercubes centered at p and of length r, where r is the query radius for the range join.

2.3.1 Index-based Similarity Range Join Algorithms

In the first category, the join algorithms utilize hierarchical index structures pre-constructed upon the datasets. The R-tree Spatial Join (RSJ) [24], the breadth-first R-tree join [62], the incremental distance join [59] and the MuX range-join [21] all belong to this category.

These methods first build the hierarchical indexes upon the input datasets R and S and then traverse the indexes of R and S synchronously to form joining node pairs according to the following pruning strategy.

Pruning Strategy 2.3.1 *Given a range similarity join $R \bowtie_r S$, if MinDist between a node (page) of R R_i and a node (page) of S S_i $MinDist(R_i, S_j) > r$, node pair (R_i, S_j) can be excluded from being examined because they cannot contain any point pairs (q_n, p_m) such that are $Dist(q_n, p_m) \leq r$.*

$MinDist(R_i, S_j)$ is the minimum distance between the MBRs of the nodes R_i and S_i (see Figure 2.3 for illustration).

Definition 2.3.1 (MinDist between two MBRs) *The minimum distance between two minimum bounding rectangles of node R and S, denoted as $MinDist(R, S)$, is defined as follows:*

$$MinDist(R, S) = \sum_{i=1}^{d} \left(\begin{cases} R.lb_i - S.lb_i & if\ S.lb_i < R.lb_i \\ 0 & otherwise \\ S.lb_i - R.lb_i & if\ R.lb_i < S.lb_i \end{cases} \right)^2$$

where $R.lb_i$ ($S.lb_i$) is the lower bound of the minimum bounding rectangle of $R(S)$ at

dimension i and $R.ub_i$ (or $S.ub_i$) is the upper bound of the minimum bounding rectangle of $R(S)$ at dimension i.

For example, given two datasets R and S in Figure 2.3 and the radius r illustrated at the right side of Figure 2.3 (c), the nodes pairs to be joined together are (R_2, S_1), (R_2, S_2), (R_2, S_3), (R_3, S_2), (R_3, S_3), and (R_4, S_3).

R-tree Spatial Join

The R-tree Spatial Join (RSJ)[24] is the simplest index-based join method which traverses the indexes in the depth-first manner. The R-trees are built on the input datasets R and S beforehand. The RSJ algorithm starts from the root nodes of R and S and traverses the indexes of R and S synchronously. For each internal node pairs (R_i, S_j) being under consideration, the algorithm calculates the minimum distance between all pairs of the child nodes of R_i and S_j and forms the pairs having distances equal to or smaller than r. For each unpruned child node pairs, the RSJ algorithm is called recursively so that it traverses the R-trees *depth-firstly*. If R_i and S_j are leaf nodes, it calculates the distance between all pairs of points inside R_i and S_j and point pairs such that $Dist(q_n, p_m) \leq r$ are output as join results.

Breadth-First R-tree Join

The breadth-first R-tree join (BFRJ) [62] is an improvement of RSJ which traverses the R-trees in the breadth-first manner. BFRJ computes join pairs one level at a time and creates an *intermediate join index* (IJI) which are node pairs to be joined at the next lower level at each level. Based on IJIs, the BFRJ algorithm applies the following optimization strategies to speed up the join processing: i) the *ordering optimization* that orders the nodes by the space-filling curve (e.g. the Z-order [102]) to minimize page faults during join computation at the next level; ii) the *buffer management* optimization that schedules

the buffer paging so that the nodes to be processed in the near future are more likely hit in the memory. These optimizations makes BFRJ more efficient than RSJ in terms of the I/O cost.

Incremental Distance Join

The incremental distance join (IDJ) [59] works in the similar way of the incremental k-nearest neighbor algorithm [58]. It traverses the indexes of R and S synchronously from the tree roots and employs a priority queue to maintain pairs (e_1, e_2), where e_1 and e_2 can be either a tree node or a data point. The priority queue is initialized with the pair of the tree roots of R and S. Items in the queue is sorted in ascending order according to the MinDist or distance between them. At each iteration, IDJ dequeues the first entry from the queue and processes it according to the types of e_1 and e_2.

- If both e_1 and e_2 are points, this pair is reported as a join result.
- If one of e_1 and e_2 is a point and the other is leaf nodes, the algorithm computes the distance between the point to all points in the leaf nodes and inserts the point pairs into the priority queue.
- If both e_1 and e_2 are leaf nodes, for each point in e_1, the algorithm computes its distance to all points in e_2 and inserts the point pairs into the priority queue.
- If one of e_1 and e_2 is an internal node and the other is a point, the algorithm computes the minimum distance between the point to all entries in the internal nodes and inserts the pairs into the priority queue.
- If both e_1 and e_2 are node (either internal node or leaf node) and one of them is an internal node, for each item in e_1, the algorithm computes its minimum distance to all items in e_2 and inserts the pairs into the priority queue.

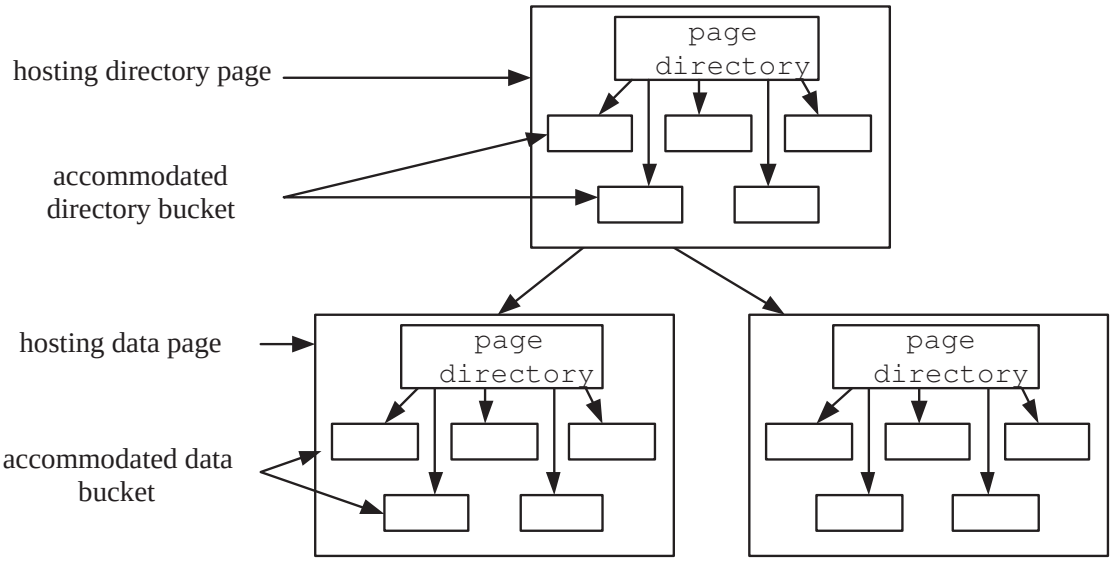


Figure 2.4: Multipage Index (MuX)

The process is stopped when r is smaller than the MinDist or distance of the first entry in the priority queue or the priority queue is empty.

MuX Range Join

The MuX range join is based on the *Multipage index* (MuX) [21, 20]. MuX is motivated by the observation that the fine-grained index with small node (page) size benefits the CPU performance, whereas index with large node (page) size has better I/O performance, especially for data in high-dimensional spaces. MuX solves this confliction between the optimization of the CPU cost and the optimization of the I/O cost by with a two-tiered structure.

Figure 2.4 illustrates the MuX index. Same as the R-tree, MuX is a height-balanced hierarchical structure and uses minimum bounding rectangles as the bounding objects in the internal nodes. Each node of the MuX is called the *hosting page*. In each hosting page, a hash-table-liked secondary structure is maintained for the purpose of the optimization of the CPU cost. The secondary structure consists of a flat directory (called the

page directory) and the *accommodated buckets*. The page directory consists of an array of MBRs and pointers. Each pointer points to an accommodated bucket. If the hosting page is a directory page, the accommodated buckets are called *directory buckets* and store entries of the form $(Rect, pointer)$. $Rect$ is an MBR and $pointer$ points to a child hosting page. If the hosting page is a data node, the accommodated buckets are called *data buckets*. The data buckets contains feature vectors of the objects. The hosting pages are usually of large size in order to optimize I/O cost. The accommodated buckets are tuned to partition the data finely in order to optimize the CPU cost. By this way, MuX achieves optimized I/O cost and CPU cost at the same time.

The MuX range join is very similarity to the RSJ algorithm. It starts from the roots of the MuX indexes built on R and S in advance and traverses the indexes synchronously in the depth-first manner. For each pair of hosting pages (R_i, S_j) being under consideration, the MuX range join algorithm examines each pair of the accommodated buckets inside and forms the accommodated buckets pairs that should be examined according to Pruning Strategy 2.3.1. Then for each generated accommodated buckets pair (r_i, s_j), the algorithm calculates the minimum distance between all pairs of the entries inside of s_i and r_j and forms the hosting page pairs that have MinDist equal to or smaller than r. For each generated hosting page pairs, the MuX range join is called recursively. If R_i and S_j are leaf hosting pages, the algorithm first generates accommodated buckets pairs (r_i, s_j) such that $MinDist(r_i, s_j) \leq r$, where r_i (s_j) is accommodated bucket inside of R_i (S_j). Then for each unpruned accommodated buckets pair, it calculates the distance between all pairs of points inside r_i and s_j. Point pairs such that $Dist(q_n, p_m) \leq r$ are output as join results, where q_n (p_m) is a point inside of r_i (s_j).

2.3.2 Hash-based Similarity Range Join Algorithms

The second category of techniques are the hash-based algorithms which partition the data space into buckets which can be fit in allocated memory and perform the join on pairs of joinable buckets to produce results.

The hash-based methods have the advantage over the index-based methods that indexes are not necessary to be built on the input datasets in advance. The major drawback of such techniques is that they may replicate a data item into multiple buckets and the data replication rates usually are high in high-dimensional spaces, which makes such methods very expensive in terms of both CPU cost and I/O cost when data dimensionality is high. Typical hash-based similarity range join algorithms are the partition-based spatial merge join [105], the spatial hash join [88] and the ϵ-kdB-tree range join [70]. The spatial merge join [105] and the spatial hash join [88] are proposed originally for the spatial join.

Partition-Based Spatial Merge Join

The partition-based spatial merge join (PBSM) [105] algorithm divides the space into a set of cells by applying a regular grid to it. Each partition (bucket) is a set of tiles and the partitions are disjoined. The PBSM algorithm first decomposes dataset R into the partitions and then inserts each object of dataset S into every bucket which intersects with that object. The number of partitions depends on the size of the inputs datasets and the size of the available memory. If the partition does not fit in the available memory, it is subdivided in a repartition process. In a second step, the PBSM algorithm loads each partition into main memory and joins them using a computational geometry based plane-sweeping algorithm.

Spatial Hash Join

The spatial hash join (SHJ) [88] provides a general two-step framework for the spatial hash joins. The first step of SHJ partitions the input datasets into buckets using partition functions which comprises two components - a set of *bucket extents* which describes the buckets and an *assignment function* which assigns data item to buckets. The partition functions for the two datasets may different. A data item may be mapped into multiple buckets. The algorithm assumes after the data partitioning step, data in each bucket can be fit in memory. The second step then joins inner and outer buckets to obtain results with either nested-loop join (that is, for each point in in bucket of R, it scans the bucket of S and outputs pairs intersecting with each other) or the indexed nested loop join (that is, it constructs an R-tree on the S data in the memory first and then for each point in R, performs an intersection query on the R-tree).

The ϵ-kdB-tree Range Join

The ϵ-kdB-tree range join [70] partitions the data space on one selected dimension into stripes of the width ϵ (ϵ is equal to the range join radius r). Thus, the join operation is restricted to the subsequent stripes. The algorithm assumes that the database cache is large enough to hold all the data points of two subsequent stripes so that it is possible to join two stripes in a single pass. When joining two stripes of data, it constructs a main memory data structure called the ϵ-kdB-tree for each stripe. The ϵ-kdB-tree structure partitions the data in memory into stripes of width ϵ according to the other dimensions until a defined node capacity is reached. Again, only adjacent partitions are needed to be joined together. The ϵ-kdB-tree [70] is particularly suited for the self range join. However, this method is not scalable to large dataset according to the study in [17].

2.3.3 Sort-based Similarity Range Join Algorithms

The third category is the sort-based techniques. The ORE algorithm [103], Multi-dimensional Spatial Join (MSJ) [81], GESS [31], and the Epsilon Grid Order (EGO)[17] all belong to this category. ORE and MSJ are original proposed for the spatial join.

ORE Join

The ORE algorithm [103] is based on the Z-curves [102]. For each input dataset, the algorithm imposes a binary recursive partitioning of the data space until a specified granularity is reached and obtains a set of hypercubes. Each hypercube has its corresponding Z-value (a bit string). ORE sorts the hypercubes into the nondecreasing order with regard to their Z-values and after that, merges the hypercubes intersecting with each other in main memory utilizing two main-memory stacks Stack_R and Stack_S. Making use of the property of Z-curves, it detects the intersection by checking if the Z-values of two hypercubes have a prefix-suffix relationship. A deficiency of ORE is that it allows to decompose the spatial objects into several pieces which leads to substantial data replication especially in high-dimensional spaces and this increases both space and CPU overhead for sorting and joining. In addition, replication causes duplicates in the result set and ORE has not dealt with this problem.

MSJ Join

MSJ imposes a dynamic hierarchical decomposition of the space into level files. It scans each input dataset and place each point $p(x_1, x_2, ..., x_d)$ in a level file l,

$$l = \min_{1 \leq i \leq d} = ncb\left(x_i - \frac{r}{2}, x_i + \frac{r}{2}\right)$$

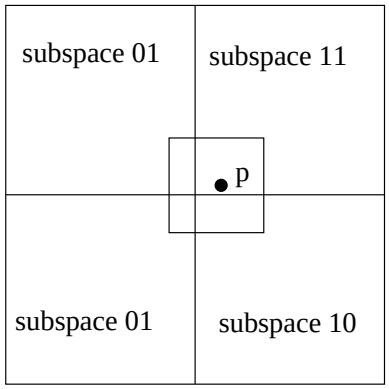


Figure 2.5: Replication of GESS.

where $ncb(b_1, b_2)$ denotes the number of most significant common bits in bit sequences of b_1 and b_2. Each point is then assigned to a Hilbert value based on the level file to which it belongs. MSJ sort the level files into nondecreasing order of Hilbert [66] values and perform a multi-way merge of all level files. Deficiency of this method is that in high-dimensional spaces a high fraction of the input data will be in level 0 [16]. Points in level 0 need to be joined with the other dataset entirely in a nested-loop manner, so it is very expensive for high-dimensional data [32].

Generic External Space Sweep Join

The Generic External Space Sweep (GESS) [31] has three steps. In the first step, each point (vector) of the input datasets p is transformed into a hyper-cube centered at p and of length r, where r is the query range radius. The hypercubes are then passed to the replication algorithm which generates codes (Z-values or Hilbert values) that represent the subspaces of each hypercube. In order to avoid the problem of MSJ in high-dimensional spaces, GESS allows replication at this step. A hypercube can be split and assigned to different subspaces. As illustrated in Figure 2.5, the hypercube is split into 4 parts and assigned to 4 subspaces. The second step of GESS is similar to the ORE algorithm. It sorts the hypercubes into nondecreasing order based on their Z-values or Hilbert values

and merges the hypercubes intersecting with each other in main memory utilizing two main-memory stacks Stack_R and Stack_S. In the last step, GESS removes duplicates which are caused by the replication from the result set using a method called Reference Point Method (RPM). GESS improves both ORE and MSJ and is more scalable to high-dimensional data.

The Epsilon Grid Order Join

The epsilon grid order join (EGO) [17] was proposed for the self range join, that is, the inner and outer datasets of the range join are same. It applies an equidistance grid with cell length $\epsilon = r$ over the data space and sorts the data points into the *epsilon grid order* [17] according to the grid cells where they are located. It divides the sorted data points into the *I/O units* and loads them with a sophisticated I/O scheduling strategy which is made up of two scheduling modes - the gallop mode and crab-step mode. For two I/O units (each corresponding to an ordered data sequence) in memory, EGO joins them following the divide and conquer paradigm. The algorithm selects one sequence and divides it into two subsequences of approximately same length recursively until the *minimum sequence capacity* is reached or the pair of sequence does not join (their distance exceeds r). For two subsequences with less than *minimum sequence capacity* data points, EGO computes the distances between each point pair and reports the point pairs whose distances are smaller than r. The EGO works very efficient for massive datasets.

2.4 Algorithms for kNN Similarity Join

There are not many works on the kNN join operation. We introduce the incremental semi-distance join algorithm [59] and the MuX kNN join [18, 20, 19] algorithm in this

section.

2.4.1 Incremental Semi-distance Join

The incremental semi-distance join [59] is proposed for the semi-distance join which is essentially same to the kNN join. It is an R-tree based solution and almost same as the incremental distance join algorithm for the range similarity join that we described in Section 2.3.1. The only difference is the termination condition. The incremental semi-distance join algorithm maintains a counter for each point in dataset R p to record the number of pairs which have been already output and whose first element is p. It terminates when the counters for all points in dataset R reaches k. The shortcoming of this method is the size of the priority queue could be very large if the kNN join is performed on a large dataset and the k value is big. In addition, because the R-tree is not scalable to datasets in high-dimensional spaces, the incremental semi-distance join also works only efficiently in low-dimensional spaces.

2.4.2 Mux kNN Join

The MuX kNN join [18, 20, 19] is the most up-to-date method specifically designed for the kNN join in high-dimensional spaces. The algorithm works on the MuX indexes pre-constructed on the datasets R and S and iterates over the R hosting pages. In each iteration, it loads an R hosting page PR (which has not been processed) into the memory and joins it with the hosting pages of S PS such that $MinDist(PR, PS) <$ pruning distance(PR). The pruning distance of PR is the maximal kNN-distance of points in PR. It joins the hosting page PR in memory with the hosting page PS with the highest *quality* first. The quality of a hosting page PS $Q(PS)$ is computed as the

following:

$$Q(PS) = \max_{BR \in PR} \left\{ \frac{\text{pruning distance}(BR)}{MinDist(PS, BR)} \right\}$$

where BR is a bucket in hosting page PR. The pruning distance of BR is the maximal pruning distance of points in BR.

When joining two hosting page PR and PS, it employs a priority queue to sort the bucket pairs (BR, BS) according to their *quality* $Q(BR, BS)$ which is computed as the following:

$$Q(BR, BS) = \frac{\text{pruning distance}(BR)}{MinDist(BS, BR)}$$

where BR (BS) is a bucket in hosting page PR (PS).

Bucket pairs such that $Q(BR, BS) < 1$ are pruned. For each joinable pair (BR, BS), for each point p in BR, the algorithm computes its distance to each point q in BS. If $Dist(p, q)$ is smaller than the pruning distance of p - distance between p and its current kth nearest neighbor candidate, q is inserted as one of p's k-nearest neighbor candidate.

Though the MuX index has much better performance than the R-tree in high-dimensional spaces, its performance is still expected to degenerate with the increase of data dimensionality. In addition, the memory overhead of the MuX index structure is high for large high-dimensional data due to the space requirement of high-dimensional minimum bounding boxes. Both constraints restrict the scalability of the MuX kNN-join method in terms of dimensionality and data size.

2.5 Algorithms for the RkNN Query

Algorithms for the RkNN query can be classified into two categories: the *pre-computation* methods and the *space pruning* methods. Most of them have been proposed initially for the RNN query (i.e., RkNN query when $k = 1$).

2.5.1 Pre-computation RkNN Search Algorithm

The *pre-computation* methods [78, 125] pre-compute and store the nearest neighbor information of each point in the dataset in advance. The RNN-tree[78] and the Rdnn-tree [125] provided two index-based solutions for the RNN query.

The RNN-tree[78] is virtually a R-tree storing the MBRs (minimal bounding rectangle) of the hyper-spheres (p, dnn) centered at p and of radius dnn. dnn is the distance between p and its nearest neighbor (NN-distance).

The Rdnn-tree [125] modifies the structure of the R-tree slightly. It augments the leaf entries of R*-tree with dnn and the internal entries with max_dnn respectively. The *leaf* nodes of the Rdnn-tree contain the data points and their NN-distance. Each entry e of the leaf node is of the form

$$(p,\ dnn(p)),$$

where p is the data point and $dnn(p)$ is the NN-distance of p.

In the *internal* nodes of the Rdnn-tree, each entry e is of the form

$$(ptr,\ max\_dnn,\ mbr).$$

ptr points to a child-node (sub-node) N'; mbr is the minimum bounding rectangle (MBR) of N'; max_dnn is the maximal dnn of all data points in the subtree rooted at N'.

$$max\_dnn = Max_{i=1}^{m} dnn(p_i) \tag{2.1}$$

where $p_1, ..., p_m$ are all points within N'.

With the nearest neighbor information stored RNN-tree or the Rdnn-tree, the RkNN query is transformed into the *point enclosure query* [78] which checks if a query point q falls within the circle (p, dnn). Points p that have q within that circle are reported as the

answers to the RNN query. The Rdnn-tree is more efficient than the RNN-tree for both static and dynamic data sets[125].

The RNN-tree and the Rdnn-tree can also be used to support the RkNN query. We only need to pre-compute the k-nearest neighbor information of the whole dataset, store the information in these data structures and apply the same point enclosure search.

The drawback of *pre-computation* methods is that they cannot answer an RkNN query unless the corresponding k-nearest neighbor information is available. Since the values of k may vary greatly in many applications, storing the k-nearest neighbor information for all possible values of k is expensive and sometimes infeasible, and maintaining such a large amount of k-nearest neighbor information in the presence of frequent updates is even more costly.

2.5.2 Space Pruning RkNN Search algorithms

Space pruning methods [112, 116, 114], include SAA [114], SRAA [115], SFT [112] and TPL [116], utilize the geometry properties of RNN to first retrieve a small number of data points as candidates and then verify them with NN queries or range queries. These approaches are useful in dynamic environments since they do not require pre-computed nearest neighbor information.

SAA Algorithm

SAA [114] makes use of the *bounded output* property, e.g. for an RNN query in the 2-dimensional space, a query point q has at most 6 RNNs [113]. Thus, SAA divides the data space into six equal regions by straight lines that intersect at the query point q as illustrated in Figure 2.6. SAA retrieves the nearest neighbors of q in each region as RNN candidates and then verified them with NN queries.

SAA is useful in two-dimensional space and has been adopted to answer the re-

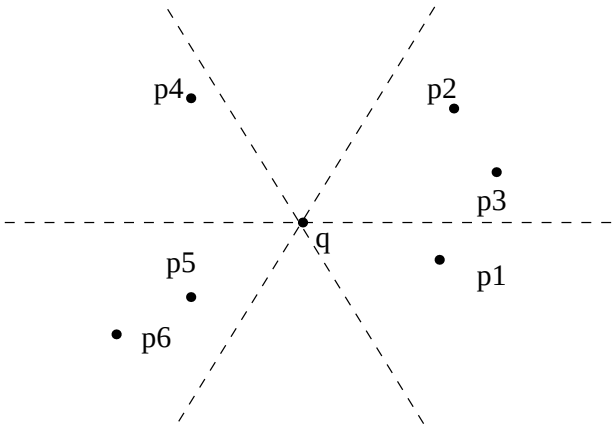


Figure 2.6: Illustration of SAA algorithm.

verse nearest neighbor query for moving objects [76]. However, it is costly for high-dimensional data because the bounding number increases exponentially with respect to data dimensionality[113]. It is also expensive for the RkNN queries. The number of reverse k-nearest neighbors of a point in two-dimensional space is bounded by $6 \cdot k$ [116]. Therefore, when k is big, a large number of candidates need to be retrieved and verified.

SRAA Algorithm

[115] investigates the bi-chromatic RNN problem and proposes SRAA. The bi-chromatic RNN query has two input datasets - the *site* dataset where the query points are located and the *point* dataset where the answers of the RNN query are found. The main idea of SRAA is to calculate the Voronoi cell V_q of the query site with respect to other *sites* in the *site* dataset and then retrieve objects within V_q from the *point* dataset as answers (see Figure 2.7 for illustration). To speed up the query processing, SRAA employs an approximate-and-refine procedure which computes the approximate Voronoi cell first and then refines the RNN candidates which are retrieved according to the approximate Voronoi cell.

The method is not scalable to large k values because computation of the k-degree

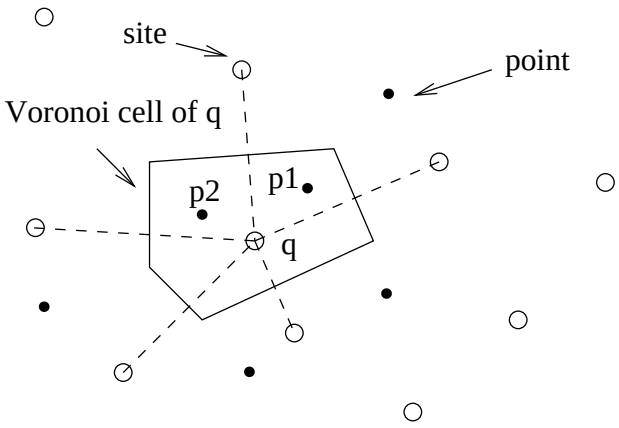


Figure 2.7: Illustration of SRAA algorithm.

Voronoi cell is expensive. It is not scalable to high-dimensional data either because computation of Voronoi cell in high-dimensional spaces is very complex.

SFT Algorithm

SFT [112] is based on the assumption that the reverse k-nearest neighbors are close to the query point and are expected to be among the K-nearest neighbors of q, where K is a value bigger than k. It retrieves K nearest points to q as candidates and then verifies the candidates with boolean range queries. The boolean range query is basically a range query that does not retrieve the answer points but only counts the number of points within the query range and stops whenever there are k points being found within the query range. Since there are multiple candidates and each candidate should be evaluated by the boolean range query, SFT uses a batch boolean range query which traverses the R-tree once in order to reduce the I/O cost. However, as we illustrated in Figure 1.1, the correlation between RkNN and KNN is not strong and a reverse k-nearest neighbor of query point q can lie far from the q. Hence, K should be set to be sufficiently big in order to reduce *false misses* (points that are RkNN but missed from the found answer set), which makes SFT expensive for RkNN queries of large k values.

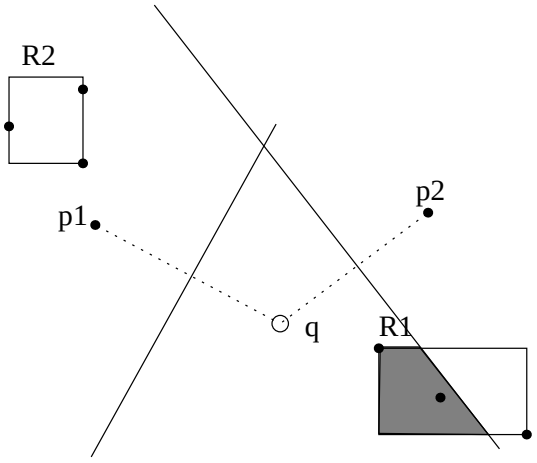


Figure 2.8: Illustration of half-plane pruning.

TPL Algorithm

The most recent work TPL [116] makes use of the *half-planes* pruning strategy, that is, if we divide the data space into two half-planes by the perpendicular bisector between q and an arbitrary data point p, any point in the half plane of p cannot be a reverse nearest neighbor of q. For example, in Figure 2.8, the points lying within the half-plane left to the line perpendicular to line $\overline{p_1 q}$ will not contains any reverse nearest neighbors of q, so node R_2 can be pruned safely. Similarly, the points lying within the half-plane right to the line perpendicular to line $\overline{p_2 q}$ will not contains any reverse nearest neighbors of q and the un-shaded area of R_1 can also be pruned away. The shaded area is called the *residual area* and points lying there are retrieved as candidates and then refined with a refinement procedure.

TPL traverse the R-tree to retrieve nearest neighbors incrementally (in the same way as the incremental k-nearest neighbor search algorithm) as RkNN candidates and uses the candidates to prune tree nodes using the *trim* algorithm according to the *half-planes* pruning strategy. The node is pruned when it is entirely trimmed by the half-planes produced by q and multiple candidates. Therefore the pruning procedure renders expensive

computation cost. The candidate retrieval procedure stops when all nodes of R-tree are either pruned or visited. The retrieved candidates are then verified by an I/O optimized refinement algorithm using range queries.

For the RkNN query, the *k-trim* algorithm is used to prune R-tree nodes based on the extended *half-planes* strategy. TPL is the most efficient and effective *space pruning* method for RkNN queries in low-dimensional spaces and of small values of k. However, its performance degrades rapidly with the increase of data dimensionality and k. The major reason is the *k-trim* algorithm that TPL used to prune a node has the complexity $\binom{n_c}{k} \cdot d$, where n_c is the number of RkNN candidates ($n_c > k$), and d is the number of dimensionality[2]. Moreover, the number of node to be trimmed also increases linearly with data dimensionality. As a result, the trimming cost becomes prohibitive when k is large or data dimensionality is high.

2.6 Summary

In this chapter, we reviewed the indexing techniques and search algorithms being proposed to efficient similarity query processing. The review shows that intensive research has been conducted on the basic similarity queries and the range join. However, studies on the kNN join and the RkNN query are not sufficient. More studies are required to improve their query performances.

[2] *k-trim* calls $\binom{n_c}{k}$ times $clipping$ algorithm in the worst case [116] and the $clipping$ algorithm has complexity of $O(d)$ [44].

Chapter 3

Gorder: An Efficient Method for kNN Join Processing

3.1 Introduction

The kNN join is a wildly-recognized important and expensive primitive operation of high-dimensional databases. The operation combines each point of one dataset with its k-nearest neighbors in another dataset. With its set-a-time nature, the kNN join can be used to efficiently support various applications where multidimensional data is involved and in particular, many data mining tasks such as the density-based outlier detection (LOF) [23], the k-means clustering [54] and the hierarchical clustering method (Chameleon) [72].

Most existing work focuses on the other similarity join - the range joins [17, 105, 87, 70, 81, 21]. However, there is an increasing need to study the kNN join in view of the observation that the parameter *r* of the range join can not be easily estimated in most cases. A oversized or undersized ε results in either a much larger or smaller answer set, hence affecting the meaningful of the result. On the contrary, the kNN similarity join returns a predefined number of answers and the parameter k is definitely much easier to determine than the *r* in most cases.

To the best of our knowledge, the MuX kNN join algorithm [20, 19] is the only up-

to-date method specifically designed for the kNN join in high-dimensional space. Since MuX [21] is essentially an R-tree based method, like the R-tree, its performance is expected to degenerate with the increase of data dimensionality. Second, the MuX index should be built in advance before the join operation. Third, the memory overhead of the MuX index structure is high for large high-dimensional data due to the space requirement of high-dimensional minimum bounding boxes. All these constraints restrict the scalability of the MuX kNN join method in terms of dimensionality and data size.

In this chapter, we present a novel algorithm *Gorder* (or the G-ordering kNN join method). *Gorder* is a block nested loop join method which achieves its efficiency by sorting data points based on an ordering that enables effective join pruning, data blocks scheduling and distance computation filtering and reduction. It first sorts input datasets into the *G-order* (an order based on grid), so that the the dataset can be partitioned into blocks that are amenable for efficient scheduling for join processing. Then, it applies the *scheduled block nested loop join* to find the k-nearest neighbors for each block of R data points.

Gorder is efficient due to the following factors:

1. It inherits the strength of the block nested loop join in being able to reduce random reads.

2. It prunes away unpromising data blocks from probing to save both I/O and similarity computation costs by exploiting the property of the G-ordered data.

3. It utilizes a *two-tier partitioning strategy* to optimize I/O and CPU time separately.

4. It reduces distance computational cost by pruning redundant computation based the distance of fewer dimensions.

The remainder of the chapter is organized as follows.

- Section 3.2 investigates the properties of the kNN-join problem.
- Section 3.3 presents the algorithm Gorder, including its data scheduling and distance computation pruning and reduction techniques to optimize the both I/O and CPU time. A cost analysis is also given.
- Section 3.4 describes a performance study and presents the experimental results.
- Section 3.5 concludes this chapter with a summarization.

3.2 Properties of the kNN Join

The kNN join has the following properties:

- It is asymmetric, that is,

$$R \ltimes_{kNN} S \Leftrightarrow S \ltimes_{kNN} R.$$

  The reason is that the k-nearest neighbor relationship is asymmetric. If a point p is one of q's k-nearest neighbors, q is not necessary to be one of p's k-nearest neighbors.

- The cardinality of the answer set of a the kNN join is predictable, since a the kNN join returns k-nearest neighbors for each point of R.
- The distance from each point in R to its nearest neighbors is unknown apriori.

Property 2 makes the kNN join more useful than another similarity join – the range join in situations where a good range *r* cannot be determined easily. The range join returns pairs of points from two datasets with their similarity distance not exceeding a given value. One of the difficulties to use the range join in real application is that the distribution of data points are often unknown and pre-defining an appropriate similarity

distance threshold between points is rather difficult, if not impossible. As such the results of the range join are somehow unpredictable and applications are subject to run on trial-and-error basis. The kNN join overcomes this difficulty by employing the rank predicate as the selection condition. The cardinality of the query result of the kNN join is always $N \cdot k$, where N is the cardinality of the query (outer) dataset R. The kNN join has the following advantages over the range join:

- The kNN join returns a predefined number of answers, that is, the size of its answer set is controllable.
- The parameter k of the kNN join is much easier to determine than the parameter r of the range join.

Therefore, the kNN join is a more practical operation than the range join in many situations.

Property 3 inherits the difficulty of the k-nearest neighbor query and makes the kNN join more complex than the range join. Given the k-nearest neighbor query, in order to filter unnecessary distance computation and page (node) access, the search algorithms based on an index such as the R-tree [49] (the RKV [60, 108] and the HS [108]) schedule the loading of data page by computing MinDist and choosing to traverse the node with the minimum MinDist first. MinDist is also compared with the pruning distance (the distance between the query point and its kth nearest neighbor candidate) to prune away nodes with MinDist greater than the pruning distance. The page scheduling and pruning strategies are very important for the kNN query processing and affect the query efficiency significantly. In the same way, they affect the efficiency of the kNN join processing substantially. It is important to consider the data schedule and the pruning strategy when we design the kNN join algorithm.

There are two starting points as the devising of the kNN join algorithm based on exiting kNN query methods.

- indexed-based multiple kNN query (index nested loop join)
- block sequential search (block nested loop join).

Both have its strength and weakness. The index-based multiple kNN query is optimized for the CPU cost, however, introduces tremendous I/O time because of large number of random accesses[20]. In addition, it is widely recognized that most high-dimensional indexes do not scale up well, and in fact, many perform worse than sequential scan when the dimensionality is high. kNN join further escalates the complexity and search cost of a high-dimensional index.

On the contrary, the block sequential search is optimized for I/O time. However, without any distance computation pruning, the CPU cost is enormous since the number of distance computation is $|R| \cdot |S|$.

Gorder therefore is developed based on the block nested loop join with the sorting, data scheduling, and distance computation filtering and reduction technologies to achieve good the kNN join performance. For ease of discussion, in the following, the data space in our discussion is a unit hypercube $[0..1]^d$.

3.3 Gorder

Gorder kNN join is a simple yet efficient kNN join algorithm based on an ordering according to grid – the *G-ordering*. It is a block nested loop method which achieves its efficiency by exploiting sorting, data scheduling and distance computation reduction. As shown in Algorithm 1, it consists two phases. In the first phase (line 1), it sorts the input datasets R and S based on the *G-ordering*. In the second phase (line 2), it performs the *scheduled block nested loop join* on the G-ordered data and outputs the join results. The algorithm is described in detail in this section.

Algorithm 1 Gorder_kNN(R, S)

Input:

R and S are two data sets.

Description:

1: G_Ordering R and S;
2: Join_Grid_Ordered_Data(R, S);

3.3.1 G-ordering

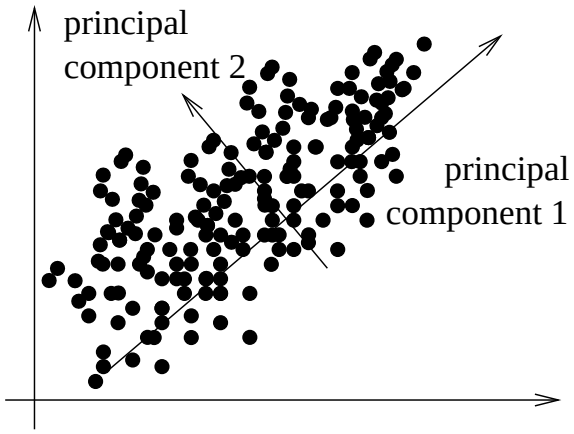
In relational databases, sorting is used not only to arrange the tuples according to an order, but to group tuples with the same value on the joining attribute together to facilitate processing based on partitions. Similarly in Gorder, an ordering based on grid called the *G-ordering* is designed to group nearby data points together, so that in the *scheduled block nested loop join* phase the G-ordered data can be partitioned into blocks and scheduled for join.

As illustrated in Figure 3.1, the G-ordering has two steps – the PCA (principal component analysis) transformation and the *Grid Order* sorting.

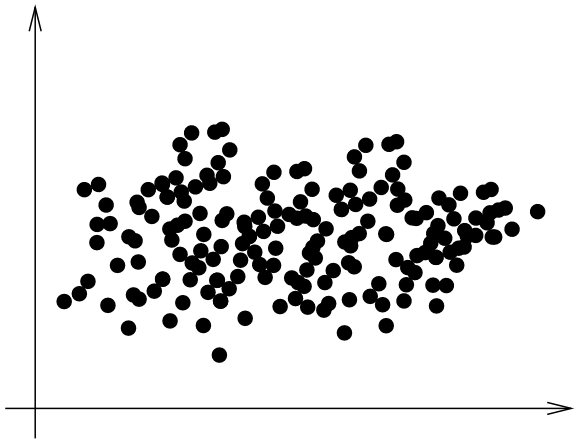
PCA Transformation

The first step of G-ordering performs the principal component analysis [69] on the input datasets R and S together and transforms the original data into the principal component space.

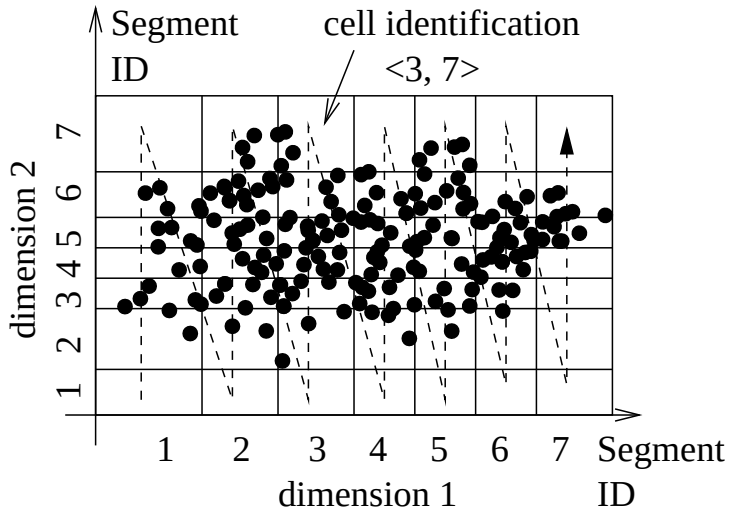
The principal component analysis (PCA) is a mathematical procedure that transforms a number of (possibly) correlated variables into a (smaller) number of uncorrelated variables called principal components. The principal components are defined as a set of variables (features) that define a projection that encapsulates the maximum amount of variation in a dataset and are orthogonal (and therefore uncorrelated) to the previous principal component of the same dataset. The first principal component accounts for as much of the variability in the data as possible, and each succeeding component accounts



(a) Original Data Space



(b) Principal Component Space



(c) Grid Order

Figure 3.1: Illustration of G-ordering.

for as much of the remaining variability as possible. PCA captures the variance in the dataset and determines the directions along which the data exhibit high variance. After PCA processing, most of the information in the original space is condensed into the first few dimensions along which the variances in the data distribution are the largest.

The PCA transformation takes two steps. Let the dataset be regarded as a $N \times d$ matrix where N is the cardinality of the dataset and d is the dimensionality of data. In the first step, the mean and covariance matrix of the dataset are first calculated to get the $d \times d$ eigenmatrix[30]. Each row of the eigenmatrix is an eigenvector and each eigenvector has its corresponding eigenvalue. The eigenmatrix is sorted according to the eigenvalues of the eigenvectors. The first principal component is the eigenvector with the largest eigenvalue and the second principal component corresponds to the eigenvector with the second largest eigenvalue and so on. In the second step, data points are transformed into a new space by multiplying the feature vector of each point with the eigenmatrix.

Figure 3.1 (a) - (b) illustrates the transformation of the data from the original data space to the principal component data space.

Grid Order

The secondary step of G-ordering sorts R and S into the *Grid Order*. The Grid Order applies a grid onto the data space and partitions it into l^d rectangular cells, where l is the number of segments per dimension of the grid. Figure 3.1 (c) is an illustration of a two-dimensional space partitioned by a 7x7 grid. Cell length of the grid can be equal or variable. In the following discussions, the cells are assumed to be of same length $\frac{1}{l}$ for the simplicity of presentation, while the methods can be easily generalized to the grid with variable cell length.

The *identification vector* of cell is defined as a d-dimensional vector $\nu = < s_1, ..., s_d >$, where s_i is the segment number to which the cell belongs on the ith dimension. Based

on the identification vector of the cell, the cells can be ordered lexicographically as illustrated in Figure 3.1.

The *Grid Order* is defined as below.

Definition 3.3.1 (Grid order $\prec_g$) *Given a grid which partitions the d-dimensional data space into l^d rectangular cells, points $p_m \prec_g p_n$ if and only $\nu_m \prec \nu_n$, where ν_m (ν_n) is the cell surrounding point p_m.*

$\nu_m \prec \nu_n$ if and only if a dimension k exists that, $\nu_m.s_k < \nu_n.s_k$ and $\nu_m.s_j = \nu_n.s_j$, for $\forall j < k$.

Essentially, the grid order is to sort the data points according to the cell surrounding the point, so after the second phase of G-ordering, points within the same cell are grouped together.

Properties of the G-ordered Data

The G-ordered data exhibit two interesting properties:

1. Suppose there are two points p and q in the dataset in the original d-dimensional space. Let p_k(q_k) denote the projection of the point p (q) on the first k dimensions after G-ordering. Because the first few dimensions are most important, $dist(p_k, q_k)$ can be very near to the actual distance between p and q [27].

2. Given a block of G-ordered data B containing m points p_1,...,p_m, a *bounding box* which covers all points in that block can be calculated by examining the first point p_1 and last point p_m of the ordered data.

Before the *bounding box* is computed, the *active dimension* [17] of the G-ordered data is first calculated.

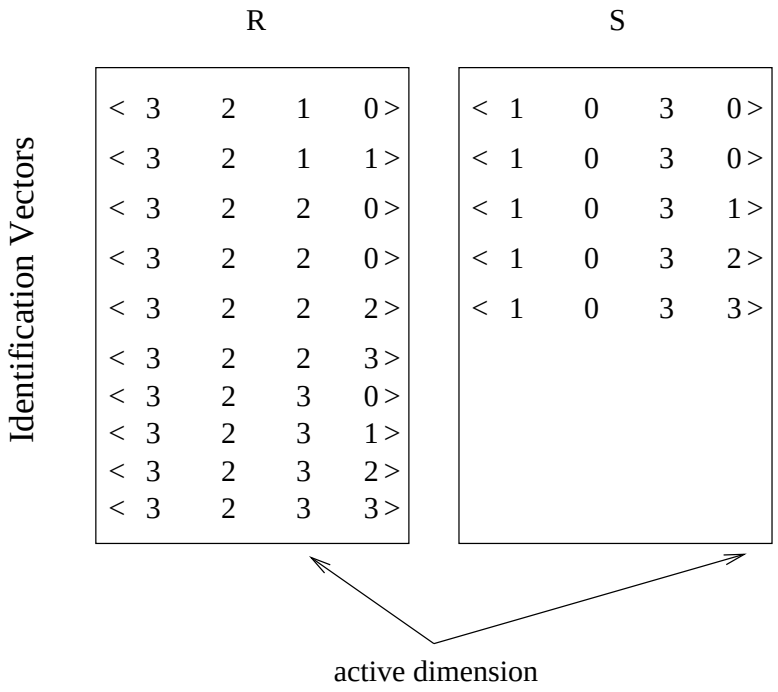


Figure 3.2: Illustration of the active dimension of the G-order data

Definition 3.3.2 (Active Dimension of the G-order Data) *Let ν_1 (ν_m) be the identification vector of the cell surrounding p_1(p_m), dimension α is the* active dimension *of the G-ordered data B, if*

$$(1) \quad \nu_1.s_\alpha < \nu_m.s_\alpha$$

$$(2) \quad \nu_1.s_j = \nu_m.s_j \qquad \forall j < \alpha.$$

Literally, α is the first dimension that $\nu_1.s_j < \nu_m.s_j$ ($1 \leq j \leq d$). Figure 3.2 illustrates an example of the active dimension for two G-ordered datasets R and S. The active dimension is 3 and 4 for dataset R and S respectively.

The bounding box of B is represented by the low-left point E = $< e_1, ..., e_d >$ and high-right point T = $< t_1, ..., t_d >$.

$$e_k = \begin{cases} (\nu_1.s_k - 1) \cdot \frac{1}{l} & if \quad 1 \leq k \leq \alpha \\ 0 & if \quad k > \alpha \end{cases}$$

$$t_k = \begin{cases} \nu_m.s_k \cdot \frac{1}{l} & if \quad 1 \leq k \leq \alpha \\ 1 & if \quad k > \alpha \end{cases}$$

The properties of the G-ordered data are used effectively in Gorder for join scheduling and distance computation reduction. Property 1 implicates that the partial distance of the first k dimensions between two points can approximate the real distance effectively and Property 2 will be used to measure the similarity of two blocks of G-ordered data and schedule the data for joining.

3.3.2 Scheduled Block Nested Loop Join

In the second phase of Gorder, G-ordered data of R and S are examined for joining. The join stage of Gorder is characterized by two properties. First, Gorder employs the *two-tier partitioning strategy* to optimize the I/O time and CPU time separately. Secondarily, it schedules the data for joining in order to optimize the kNN processing.

Two-tier partitioning

The *first-tier partitioning* is optimized for I/O time. Gorder partitions the G-ordered input datasets into blocks consisting of several physical pages. Suppose we allocate n_r and n_s buffer pages for the data of R and S respectively, we partition R and S into blocks of the allocated buffer sizes. The blocks of R are loaded into memory sequentially and iteratively one block at a time and the S blocks are loaded into memory in the sequence scheduled based on their similarity to the block of R data in buffer. The similarity of two blocks of G-ordered data of R and S is measured by the distance between their *bounding boxes*. This loading of multiple pages 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 schedules 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 k 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 checking of tuples for the kNN join. To overcome such a problem, here the second-tier partitioning in memory is introduced. The *second-tier partitioning* segments the R and S data in memory into blocks of much smaller size (the sub-blocks). The optimized size of the sub-block is $2 \cdot k - 5 \cdot k$ data points according to our experiment results. Again, similarity of two blocks data of R and S is used to schedule the join sequence and filter distance computation between blocks of data.

Similarity of G-ordered Data

The similarity of two blocks of G-ordered data is measured by the minimum distance (MinDist) between their *bounding boxes*. As presented in Section 3.3.1, the *bounding box* of a block of G-ordered data can be computed by examining the first and last points of the G-ordered data.

Definition 3.3.3 (MinDist of G-ordered Data) *The minimum distance of two blocks of G-ordered data B_r and B_s, denoted as MinDist(B_r, B_s) is defined as the minimum distance between their bounding boxes.*

$$MinDist(B_r, B_s) = \sum_{k=1}^{d} d_k^2$$

$$d_k = max(b_k - u_k, 0) \tag{3.1}$$

$$b_k = max(B_r.e_k, B_s.e_k);\ u_k = min(B_r.t_k, B_s, t_k)$$

For blocks with same MinDist, they are sorted by the MaxDist.

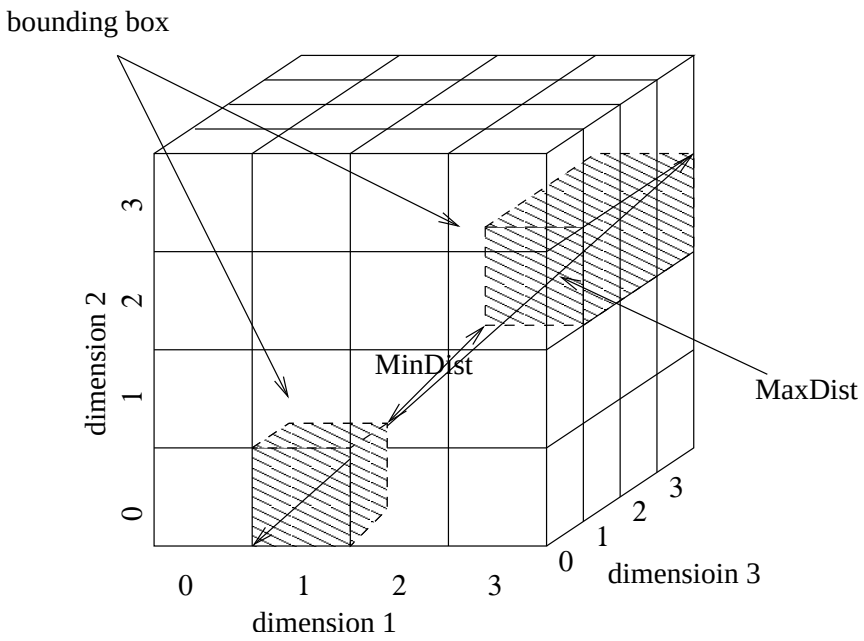


Figure 3.3: Illustration of MinDist and MaxDist.

Definition 3.3.4 (MaxDist of G-ordered Data) *The maximum distance of two blocks of G-ordered data B_r and B_s, denoted as MaxDist(B_r, B_s) is defined as the maximum distance between their bounding boxes.*

$$MaxDist(B_r, B_s) = \sum_{k=1}^{d} (u_k - b_k)^2$$

$$b_k = min(B_r.e_k, B_s.e_k);\ u_k = max(B_r.t_k, B_s, t_k)$$

Pruning Strategy

A direct observation is that MinDist is a lower bound to the distance of any two points from blocks of R and S respectively. The following corollary follows this observation directly.

Corollary 3.3.1 *For point p_r in block B_r and point p_s in block B_s, $MinDist(B_r, B_s)$ is*

Algorithm 2 Join_Grid_Ordered_Data(R, S)

Input:

R and S are two G-ordered data sets that have been partitioned into blocks.

Description:

```
1: for each block B_r ∈ R do
2:    ReadBlock(B_r);
3:    SortBlocks(S, B_r);
4:    for each B_s ∈ NotPruned(S, B_r) do
5:       ReadBlock(B_s);
6:       MemoryJoin(B_r, B_s);
7:    OutputkNN(B_r);
```

a lower bound to the distance between p_r and p_s, that is,

$$\forall p_r \in B_r, p_s \in B_s, \quad MinDist(B_r, B_s) \leq dist(p_r, p_s)$$

Based on Corollary 3.3.1, we have the following pruning strategies:

1. If $MinDist(B_r, B_s) >$ pruning distance of p, B_s does not contain any points belonging to the k-nearest neighbors of the point p, and therefore the distance computation between p and points in B_s can be filtered. Pruning distance of a point p is the distance between p and its k-th nearest neighbor candidate. Initially, it is ∞.

2. If $MinDist(B_r, B_s) >$ pruning distance of B_r, B_s does not contain any points belonging to the k-nearest neighbors of any points in B_r, and hence the join of B_r and B_s can be pruned away. The pruning distance of an R block is the maximum pruning distance of the R points inside.

Join Algorithm

Algorithm 2 outlines the scheduled block nested loop join algorithm of Gorder. It loads blocks of R into memory sequentially (lines 1-2). For the R block in memory B_r, S

Algorithm 3 MemoryJoin(B_r, B_s)

Input:

B_r and B_s are two blocks from R and S respectively.

Description:

```
1: Divide B_r, B_s into sub-blocks;
2: for each sub-block B'_r ∈ B_r do
3:    SortBlocks(B_s, B'_r);
4:    for each sub-block B'_s ∈ NotPruned(B_s, B'_r) do
5:       for each point p_r ∈ B'_r do
6:          if MinDist(B'_r, B'_s) ≤ PrunDist(p_r) then
7:             for each point p_s ∈ B'_s do
8:                ComputeDist(p_s, p_r, d_α^2);
```

blocks are sorted in the increasing order of their distance to B_r (line 3). Note that this sorting does not require any disk accesses because there are only a small number of blocks and the bounding box for each block of S can be kept in in memory. At the same time, blocks with $MinDist(B_r, B_s)$ greater than the pruning distance of B_r are pruned (pruning strategy 2). That is, only the remaining blocks are loaded into memory one by one (lines 4-5). With each pair of R and S block, we join them in memory by calling function $MemoryJoin$ (line 6). After all unpruned S blocks are processed with B_r, the kNN candidate sets for points in B_r are output as the join results (line 7).

The memory join algorithm is shown in Algorithm 3. Both R-block and S-block are divided into sub-blocks (line 1). For each R sub-block B'_r, the S sub-blocks are arranged according to their distance to B'_r. Pruning strategy 2 is again used to pruning those S sub-blocks with MinDist(B'_r, B'_s) greater than the pruning distance of B'_r. Those unpruned S sub-blocks participate the join with R sub-blocks one by one (lines 4-5). To join R and S sub-block B'_r and B'_s, each data point p_r in B'_r is compared with B'_s. For each point p_r in B'_r, we examine whether MinDist(B'_r, B'_s) is greater than the pruning distance of p_r. If true, by pruning strategy 1, B'_s cannot contain any points that are k-nearest neighbors of p_r and so the B'_s can be skipped (lines 6-7). Otherwise, function *Compute_Dist* is called for p_r and each data point p_s in B'_s (line 8). Function *Compute_Dist*, as described in the

following subsection, inserts those p_s with $dist(p_r, p_s)$ smaller than the pruning distance of p_r into the kNN candidate set of p_r. ${d_\alpha}^2$ is the distance between the bounding boxes of B'_r and B'_s on the α-th dimension,[1] where $\alpha = min(B'_r.\alpha, B'_s.\alpha)$ and $B'_r.\alpha$ and $B'_s.\alpha$ are active dimension of B'_r, B'_s respectively.

3.3.3 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.

The bounding boxes of the G-ordered data has some special properties which can be utilized for distance computation reduction.

Property 3.3.1 *The edge of the bounding box of a block G-ordered data B extends the full domain from 0 to 1 on dimension j ($j > B.\alpha$), where $B.\alpha$ is the active dimension of B.*

This property is directly observable from the computation of *bounding box*. Therefore, when we compute the similarity of two blocks of G-ordered data, we only need to take the first α dimensions into account, where α=min($B_1.\alpha$, $B_2.\alpha$) and $B_1.\alpha$ ($B_2.\alpha$) is the active dimension B_1 (B_2). As a result, the computation of MinDist and MaxDist are reduced to:

$$MinDist(B_1, B_2) = MinDist(B_{1,\alpha}, B_{2,\alpha})$$
$$MaxDist(B_1, B_2) = MaxDist(B_{1,\alpha}, B_{2,\alpha}) + d - \alpha$$

$B_{1,\alpha}$ ($B_{2,\alpha}$) is the projection of B_1 (B_2) on the first α dimensions.

The next important property of the *bounding box* is as follows:

[1] Refer to Equation 3.1 in Definition 3.3.3.

Property 3.3.2 *The projection of the bounding box of a block of G-ordered data B containing m points p_1,...,p_m on the first $B.\alpha - 1$ dimensions is corresponding to a grid cell in the first $B.\alpha - 1$ dimensions.*

The reason is, according to the definition of *Grid Order*, $p_1 \prec_g ... \prec_g p_m \Leftrightarrow \nu_1 \prec ... \prec \nu_m$, where ν_k is the cell surrounding point p_k. Based on the definition of *active dimension*, $\nu_1.s_j = \nu_m.s_j$ ($\forall j < B.\alpha$), so we have $\nu_1.s_j = ... = \nu_m.s_j$ ($\forall j < B.\alpha$).

This property indicates that the projection of all points in a block of G-ordered data B on the first $B.\alpha - 1$ dimensions are within one grid cell in the first $B.\alpha - 1$ dimensions. Hence, for any points p and q from B_1 and B_2 respectively, $MinDist(B_{1,\alpha-1}, B_{2,\alpha-1})$ can be used to approximate the distance between the projection of p and q on the first $\alpha - 1$ dimensions when the grid is of fine granularity. The approximated distance is the low bound of the real distance. That is,

$MinDist(B_{1,\alpha-1}, B_{2,\alpha-1}) \approx dist(p_{\alpha-1}, q_{\alpha-1})$. $p_{\alpha-1}$ ($q_{\alpha-1}$) is the projection of p (q) on the first $\alpha - 1$ dimensions.

Based on the above two properties, we now are able to define the pruning strategy based on the approximate distance as formalized by the following corollary.

Corollary 3.3.2 *For any point p and q from the G-ordered blocks B_r and B_s respectively, if $MinDist(B_{r,\alpha-1}, B_{s,\alpha-1}) + dist(p_{\{\alpha,k\}}, q_{\{\alpha,k\}})(\alpha \leq k \leq d)$ is greater than the pruning distance of p, q cannot be a k-nearest neighbor candidate of p, where α =min($B_r.\alpha$, $B_s.\alpha$) and $p_{\{i,j\}}$ ($q_{\{i,j\}}$) is the projection of p (q) on the dimensions from i to j.*

Algorithm 4 outlines the algorithm in reducing distance computation. It calculates $MinDist(B_{r,\alpha-1}, B_{s,\alpha-1})$ from $MinDist(B_r, B_s)$ first (line 1). Then, it accumulates the distance between p and q from dimension α, where α=min($B_r.\alpha$, $B_s.\alpha$) (lines 2-5). Whenever $pdist$ is greater than the pruning distance of p, q cannot be one of the

Algorithm 4 Compute_Dist (p, q, d_α^2)

Input:

p, q are two data points from B_r and B_s respectively. d_α^2 is the distance between the bounding boxes of B_r and B_s on the α-th dimension.[2]

Description:

1: $pdist := MinDist(B_r, B_s) - {d_\alpha}^2$;
2: **for** $k := \alpha$ to d **do**
3: pdist :=pdist+ $(p.x_k - q.x_k)^2$;
4: **if** $pdist >$ pruning distance of p **then**
5: Prune q;
6: $pdist := pdist - (MinDist(B_r, B_s) - {d_\alpha}^2)$;
7: **for** k:=1 to α-1 **do**
8: pdist :=pdist+ $(p.x_k - q.x_k)^2$;
9: **if** $pdist >$ pruning distance of p **then**
10: Prune q;
11: Insert q into the kNN candidate set of p;

k-nearest neighbors of p and can be pruned away (lines 4-5). If q cannot be pruned by the approximation distance, we remove the approximation factor (line 6) and calculate their real distance (lines 7-10). If $dist(p, q)$ is smaller than the pruning distance of p, q is inserted into the kNN candidate set of p.

According to the algorithm of Gorder, Gorder produces the kNN join results correctly. Firstly, the MinDist of two blocks of G-ordered data is the low bound to the distance of any two points from these two blocks respectively (Corollary 3.3.1). Secondly, Gorder only skips the S blocks (sub-blocks) whose MinDist from the R block (sub-blocks) is greater than the pruning distance of R block (sub-blocks). Finally, the reduced distance computation only prunes away S data points that are not one of the k-nearest neighbors of a R point (Corollary 3.3.2). Hence, for all blocks of R data, Gorder finds the correct k-nearest neighbors.

3.3.4 Analysis of Gorder

The I/O and CPU cost of Gorder is analyzed in the following. Suppose the number of R (S) data pages is N_r (N_s). In the G-ordering phase, the PCA transformation needs to perform the sequential scan of R and S twice. The cost is $2(N_r + N_s)$. Suppose that there are B buffer pages available in memory, the sorting step of the G-ordering requires

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

page accesses using the external merge sort algorithm [107].

In the *scheduled block nested loop 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

$$N_r + \frac{N_r}{n_r} \cdot N_s \cdot \gamma_1$$

where γ_1 is the selectivity of the S blocks. Consequently, the total I/O cost in terms of the number of page accesses is:

$$\begin{aligned} &2(N_r + N_s) + +N_r + \tfrac{N_r}{n_r} \cdot N_s \cdot \gamma_1 \\ &+2N_r\left(\left\lceil log_{B-1}\tfrac{N_r}{B}\right\rceil + 1\right) + 2N_s\left(\left\lceil log_{B-1}\tfrac{N_s}{B}\right\rceil + 1\right) \end{aligned}$$

The major CPU cost of Gorder is the distance computation in the *scheduled block nested loop join* phase. The number of distance computation is:

$$P_r \cdot P_s \cdot \gamma_2$$

where P_r (P_s) is the number of points of R (S), γ_2 is the selectivity of distance computation. The PCA processing of G-ordering performs $(P_r + P_s) \cdot d^2$ multiply [45]. However, the multiply and comparison operations incurred in the G-ordering phase are

comparatively much less significant.

We estimate the selectivity ratio γ_1 and γ_2 using the Minkowski Sum model proposed in [15] and [21] which has been shown to be effective in high-dimensional data.

$$\gamma = \sum_{l=0}^{d} \left(\sum_{\{i_1 \ldots i_l \in 2^{\{0 \ldots d-1\}}\}} \left(\prod_{j=1}^{l} a_{i_j} \right) \right) \cdot V_{sphere}^{d-l}(\varepsilon) \tag{3.2}$$

$$V_{sphere}^{d-l}(\varepsilon) = \frac{\sqrt{\pi^{d-l}}}{\Gamma\left(\frac{d-l}{2}+1\right)} \cdot \varepsilon^{d-l} \tag{3.3}$$

$$\varepsilon = \sqrt[d]{\frac{k \cdot \Gamma(d/2+1)}{P_S}} \cdot \frac{1}{\sqrt{\pi}} \tag{3.4}$$

where, $\Gamma(x+1) = x\Gamma(x), \;\; \Gamma(1) = 1, \;\; \Gamma(1/2) = \sqrt{\pi}$.

Following the analysis in [15], we simplify Equation 3.2 by approximating the *bounding boxes* with the hypercube. Therefore,

$$\gamma = \sum_{l=0}^{d} \binom{d}{l} \left(\sqrt[d]{\frac{M_r}{P_r}} + \sqrt[d]{\frac{M_s}{P_s}} \right)^l \cdot V_{sphere}^{d-l}(\varepsilon) \tag{3.5}$$

where M_r (M_s)is the number of points in the block of R (S) data. When we replace M_r and M_s with the number of points in the block of data R (p_r) and S (p_s), we get γ_1.

$$\gamma_1 = \sum_{l=0}^{d} \binom{d}{l} \left(\sqrt[d]{\frac{p_r}{P_r}} + \sqrt[d]{\frac{p_s}{P_s}} \right)^l \cdot V_{sphere}^{d-l}(\varepsilon) \tag{3.6}$$

where,

$$p_r = \frac{n_r \cdot \text{page size}}{\text{size of data vector}} \text{ and } p_s = \frac{n_s \cdot \text{page size}}{\text{size of data vector}}$$

Parameter	Default Setting
page size	8192Byte
number of nearest neighbors (k)	10
buffer size	8% of total size of R and S
maximum buffer size for R	20% of buffer size
number of points in sub-block	30
number of segments per dimension	100

Table 3.1: Default parameter values.

n_r and n_s are the number of buffer pages allocated to R data and S data.

When we replace M_r and M_s with the number of points in the sub-block of data R (p'_r) and S (p'_s), we get γ_2.

$$\gamma_2 = \sum_{k=0}^{d} \binom{d}{l} \left(\sqrt[d]{\frac{p'_r}{P_r}} + \sqrt[d]{\frac{p'_s}{P_s}} \right)^l \cdot V_{sphere}^{d-l}(\varepsilon) \tag{3.7}$$

3.4 Performance Evaluation

We conducted extensive experimental study to evaluate the performance of Gorder and present the results in this section. In the study, we used both synthetic cluster datasets and real life datasets. The synthetic cluster datasets were generated using the method described in [68]. The real life datasets are the Corel dataset from UCI KDD data repository [3] which contains 32 dimensional feature vectors of around 60K images.

We compared Gorder with MuX and simple block nested loop join (NLJ). The MuX join [21, 20] is the current state-of-art method for the kNN join processing, which has been shown to be optimized for both CPU and I/O time and that it outperforms the join algorithm based on the R-tree (RSJ) significantly.

The experiments were conducted on a Pentium 4 2.6GHz PC running WinXP. The buffer allocated for all methods is around 8% of the datasets of R and S. Extra memory

was allocated to MuX for storing the internal nodes. The number of nearest neighbor (k) is 10 by default. The default settings of Gorder are summarized in Table 4.2.

Performance is presented in terms of the elapsed time (which includes I/O and CPU time), the I/O time and the distance computation selectivity. The elapsed time and I/O time of Gorder includes the time for both G-ordering and joining phases. Time of MuX does not include the index building time. Distance computation selectivity is calculated by the following equation:

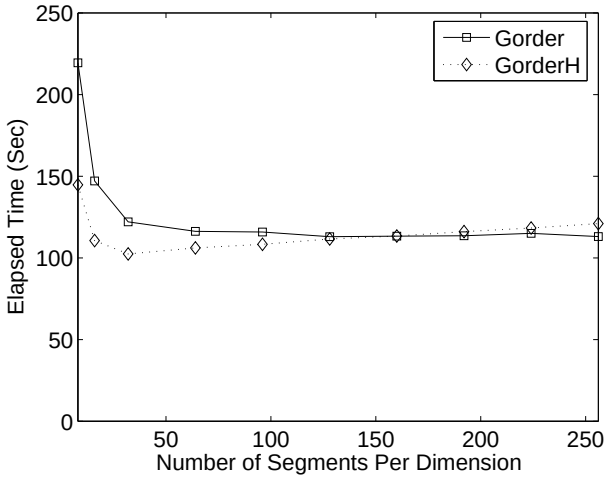
$$\frac{number\ of\ point\ distance\ computations}{|R| \cdot |S|}.$$

3.4.1 Study of Parameters of Gorder

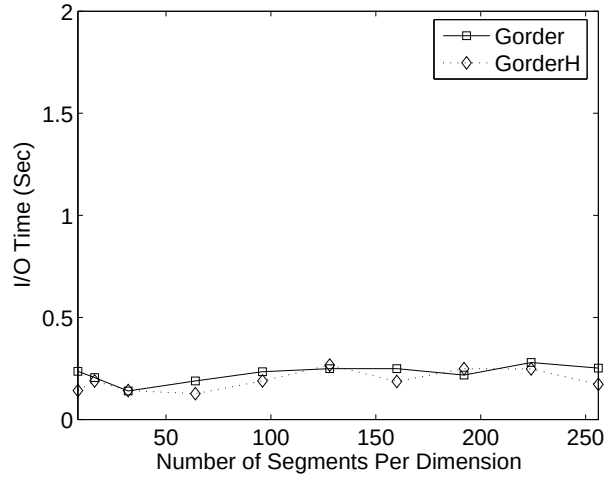
In this set of experiments, we study the performance of Gorder using the real life KDD dataset.

The first set of experiments evaluates the effect of various parameters on the performance of Gorder. With the expectation that the real life dataset is usually skewed, we implemented the GorderH for comparison purposes. GorderH applies a grid with variable cell length onto the data space during the G-ordering phase. We compute an equi-width histogram for each dimension in the PCA transformation stage and partition each dimension into segments with equal number of points inside. We performed the self the kNN join on the datasets. The measured time for GorderH includes the time for histogram processing.

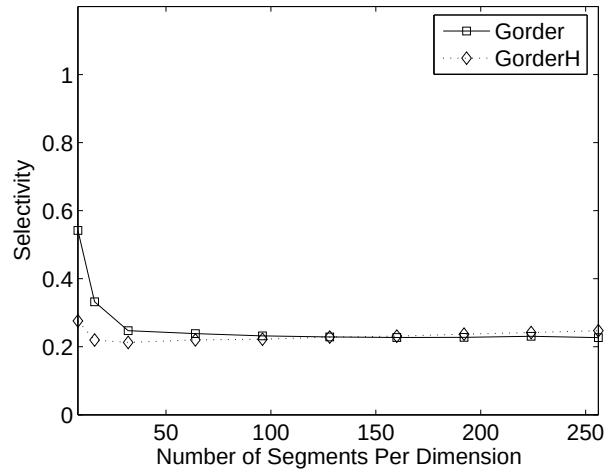
Effect of grid granularity We first evaluate the effect of the granularity of the grid by varying the number of segments per dimension of the grid from 8 to 256. Figure 3.4 presents the results of on the Corel dataset. From the results, we observe that when we increase the number of segments from 8 to 32, the performance of Gorder improves



(a) Elapsed Time



(b) I/O Time



(c) Distance Computation Selectivity

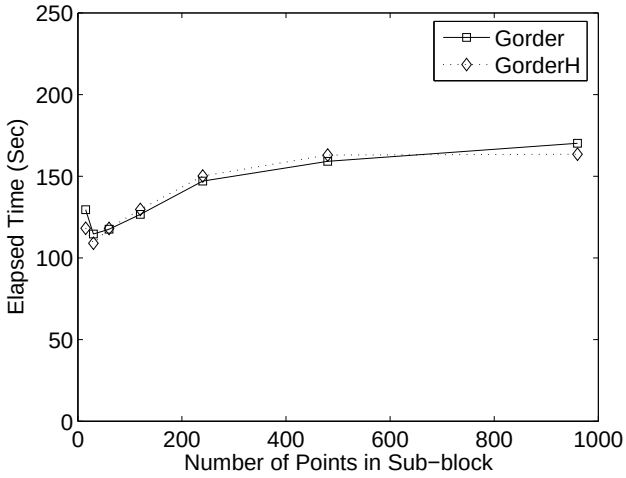
Figure 3.4: Effect of grid granularity (Corel dataset)

noticeably with a speed-up factor of 0.88. The speed-up factor of GorderH is 0.12. The reason is that with finer granularity grid, the *bounding box* bounds the data points more tightly. Hence, the MinDist low bound becomes more accurate and more effective in pruning. An interesting observation is that when we further increase the number of segments per dimension, Gorder (which uses the equi-length grid) becomes as efficient as and even better than the GorderH (which uses the variable length grid based on histogram). This indicates the fine-granularity grid makes Gorder adaptive to the data distribution and eliminates the need to maintain the histogram.

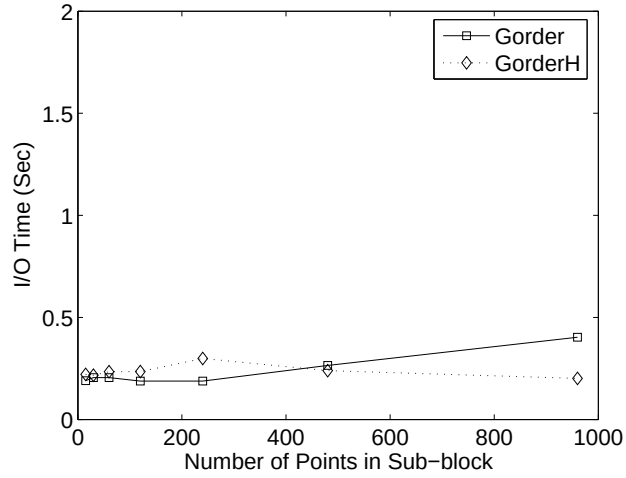
Comparing the I/O time with the total elapsed time, we notice that the I/O time is much less significant than the CPU time (less than 1% of the total response time), which confirms the benefit of using the block accessing and that the kNN join is CPU critical due to the large number and the complexity of the distance computations.

Effect of sub-block size

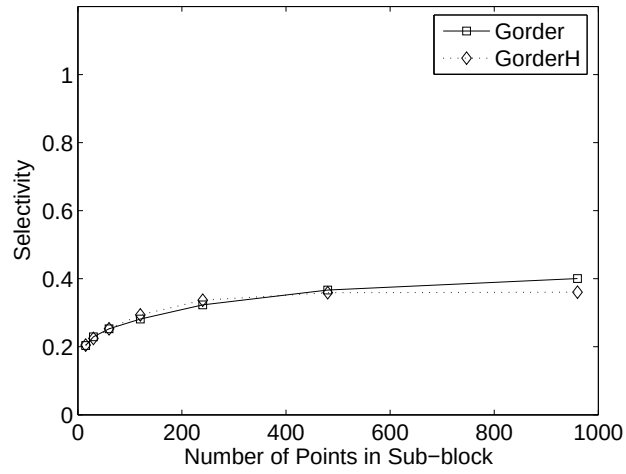
Figure 3.5 summarizes the effect of the size of the sub-block on the kNN join processing. In this experiment, the size of the sub-block is varied from 15 to 960 and we conducted the experiment on the Corel dataset. As can be observed, the selectivity of distance computation degrades when the number of points in the sub-block grows. The volume of the sub-block increases when there are more points in it, and consequently, its pruning ability become ineffective. This is consistent with the cost analysis. However, on the other hand, smaller sub-blocks do not necessarily lead to better elapsed time. We observed that when the size of the sub-block increases from 15 to 30, the performance of Gorder in terms of the elapsed time improves around 10% despite the slight degeneration of the distance computation selectivity. The reason is that the decrease of sub-block size increases the number of sub-blocks and therefore, introduces more MinDist computations. So there is a trade-off between the MinDist computation and the point distance computation. The results indicate that the best setting of the size of sub-block is around



(a) Elapsed Time



(b) I/O Time



(c) Distance Computation Selectivity

Figure 3.5: Effect of sub-block size (Corel dataset)

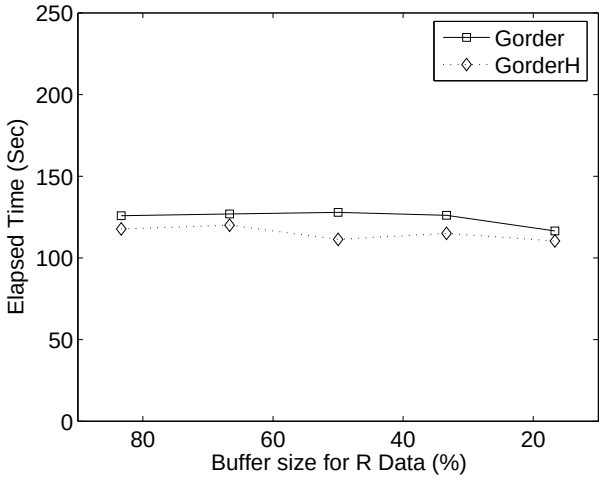
30, that is $3 \cdot k$.

Effect of buffer size for R data Next we study the effect of buffer size allocated to R data and present our study in Figure 3.6. We fixed the buffer size at around 10% of input data set and decreased the number of buffer pages for R from around 85% of buffer to around 16% of buffer. Figure 3.6 shows that as we reduce the buffer size for R, the I/O time increases quickly with the drop of the number of R buffer pages because the reduction in R buffer size causes the loading time of the S blocks to increase. However, the overall performance of Gorder with regard to the elapsed time has not been influenced a lot. The reason is when R buffer size shrinks, more S data can be loaded in buffer and hence, the R data in memory are more likely to join with the S data that yield real k-nearest neighbors first. Therefore the selectivity is improved and the increase of the I/O time is absorbed by the decrease of CPU time.

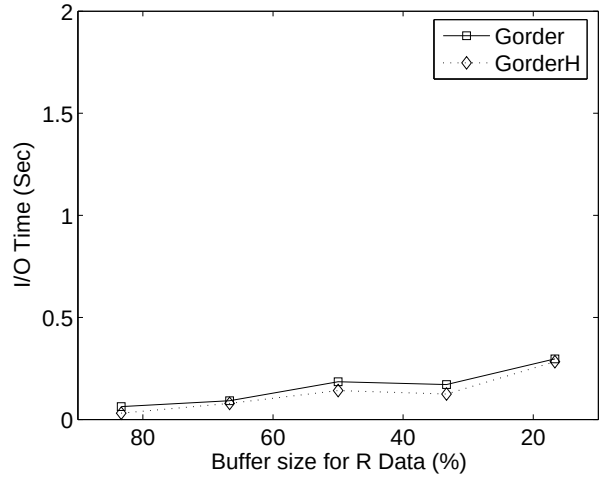
3.4.2 Effect of k

We now study the effect of k and compare the performance of Gorder with MuX and NLJ. Figure 3.7 presents the results on the Corel dataset when we varied the number of nearest neighbors k form 5 to 50.

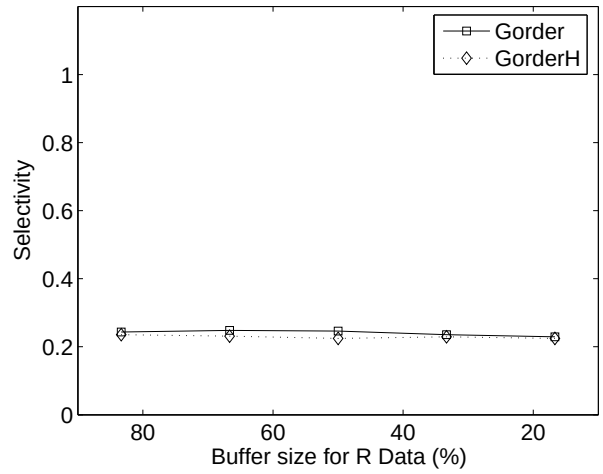
From the results, we observe that with the increase of number of nearest neighbors, the elapsed time of Gorder increases moderately. Comparatively, MuX is more affected by the increase of k and even becomes worse than NLJ. The gap of the elapsed time between MuX and Gorder widens while the value of k increases. On average, Gorder outperforms MuX with the speed-up factor of around 2 with regard to the elapsed time. In terms of distance computation selectivity, selectivity of Gorder keeps lower than the selectivity of MuX. Note that the speed-up of the elapsed time is more significant than the improvement of selectivity. This is due to the distance computation reduction technique Gorder employs. Gorder uses a subset of dimensions for block similarity computation



(a) Elapsed Time

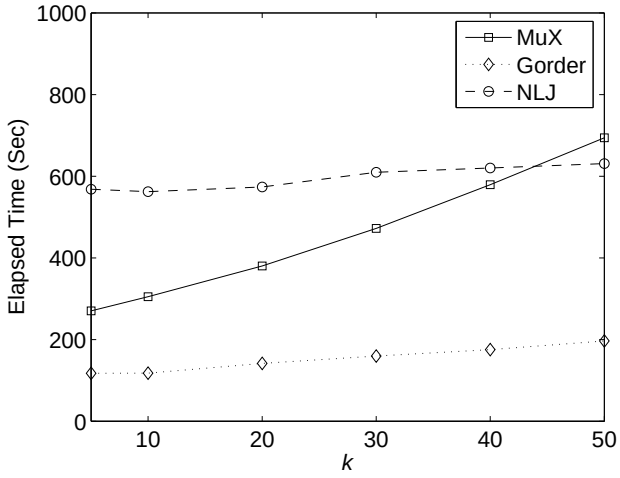


(b) I/O Time

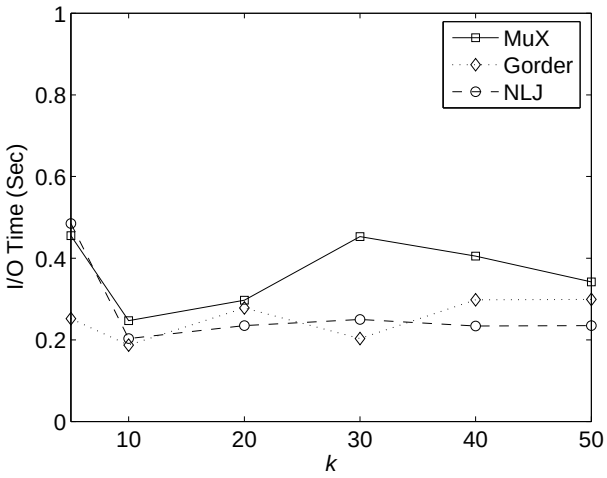


(c) Distance Computation Selectivity

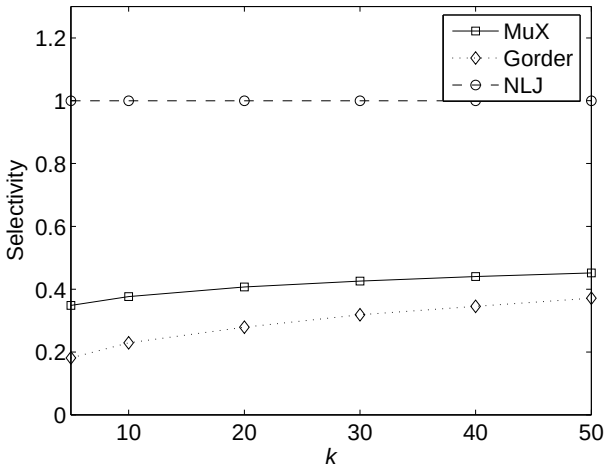
Figure 3.6: Effect of buffer size for R data (Corel dataset)



(a) Elapsed Time



(b) I/O Time



(c) Distance Computation Selectivity

Figure 3.7: Effect of k (Corel dataset)

and the block similarity is also used to reduce point distance computation; hence the speed-up in terms of elapsed time is even better than the reduction of selectivity. Figure 3.7(b) presents the I/O time incurred by different methods. Memory allocation of NLJ is the same as Gorder. That is, around 20% for R data and 80% for S data. Gorder outperforms MuX due to its one time accessing one block of data so that the expensive disk seeking time is saved. The I/O cost of Gorder is similar to the I/O cost of NLJ because Gorder filters out S blocks that will not yield kNNs with the pruning strategy but it also has more disk seek time because the S blocks are not loaded into memory sequentially.

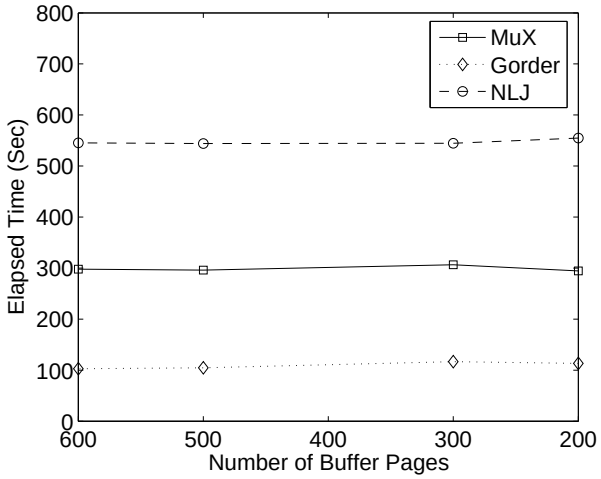
The reason that the cost of MuX increases significantly mainly because of the significant increase of the distance computation between the internal nodes and buckets.

3.4.3 Effect of Buffer Size

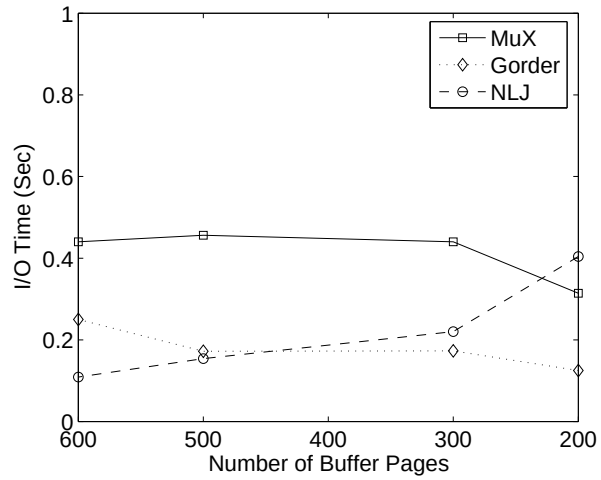
In dealing with large datasets, the kNN join algorithm must be efficient in utilizing the limited buffer space. In this experiment, we study the behavior of the join methods with respect to buffer sizes.

The study is performed on the Corel dataset and we reduced the buffer size from around 600 pages (30% of the dataset size) to around 200 pages (10% of the dataset size). The buffer size for R was kept at 50 pages. In Figure 3.8, we compare the performance of Gorder and MuX. The result shows that buffer size does not affect the overall performance the kNN join much because the major cost of the kNN join is the CPU cost. The speed-up factor of Gorder over MuX and NLJ keeps steadfastly at around 1.8 and 4 respectively.

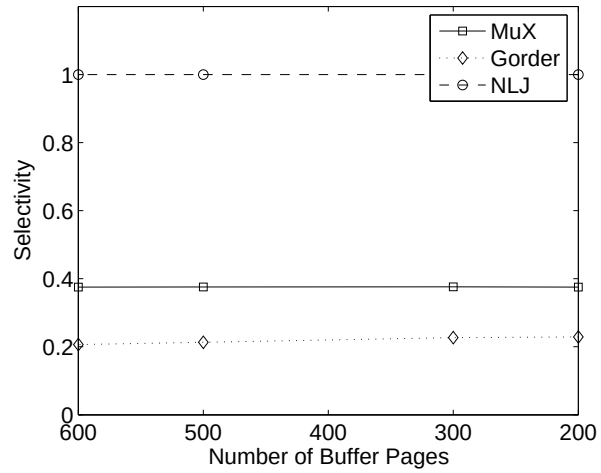
We also observe that the reduction in buffer space does not lead to the degeneration of the I/O performance of Gorder. The reason is that the reduction in buffer size reduces the volume of the bounding box and consequently, improves the effectiveness of the filtering



(a) Elapsed Time



(b) I/O Time



(c) Distance Computation Selectivity

Figure 3.8: Effect of buffer size (Corel dataset)

of S blocks. Therefore, more S blocks are filtered from being loaded into memory. Hence, the I/O time of Gorder reduces instead.

3.4.4 Evaluation Using Synthetic Datasets

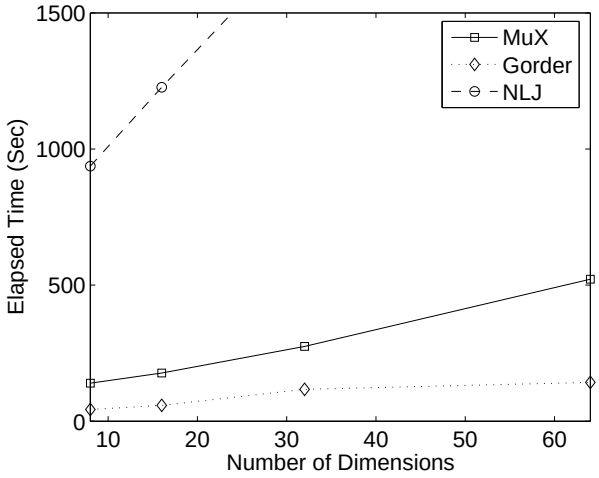
We study the scalability of Gorder on the synthetic datasets of various sizes and dimensions. Since real life data set are often clustered and correlated, we utilized method in [68] to generate clustered datasets containing 10 clusters.

Effect of Dimensionality

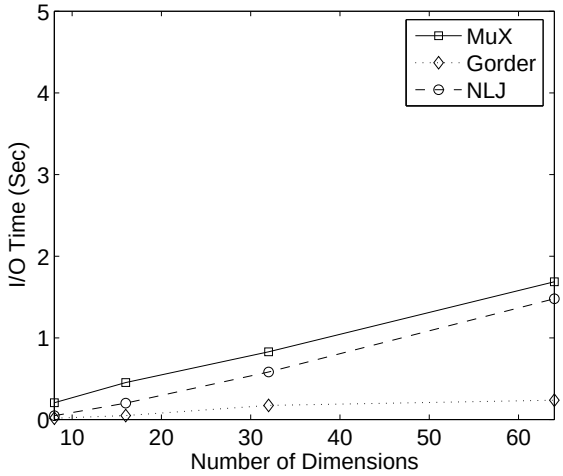
In this experiment, we evaluate the effect of data dimensionality on the join performance by varying the number of dimensions from 8 to 64. Figure 4.12 presents the results on the 100K clustered datasets. We observe that the efficiency of MuX is more affected by the increasing dimensionality. The reason is that MuX, like the R-tree, its performance degenerates with the increase of data dimensionality. The performance gain of Gorder over MuX widens as the dimensionality grows. Figure 4.12(c) shows that the distance computation selectivity of both MuX and Gorder degenerates with the increase of the number of dimensions. However, Gorder employs the distance computation reduction technique to alleviate the increase of distance computation cost for high dimensional data. Therefore, the deterioration of the elapsed time of Gorder with the increasing dimensionality is moderate. So Gorder is more scalable to high-dimensional data than MuX.

Effect of Size of Dataset

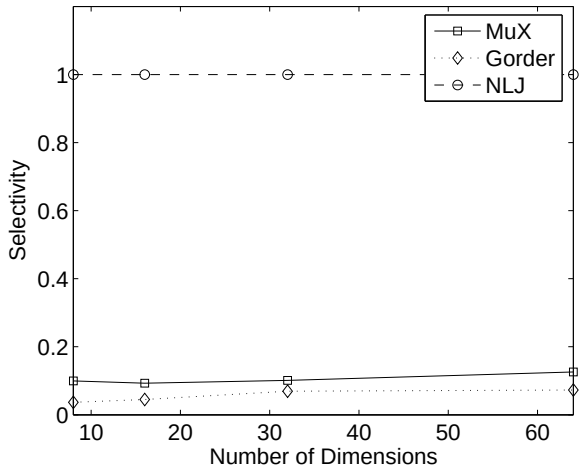
In the second experiment, we study the performance behavior with varying size of datasets. We performed the self kNN join of the clustered data in the 16-dimensional space and varied the dataset size from 10,000 to 500,000 objects. The results are sum-



(a) Elapsed Time

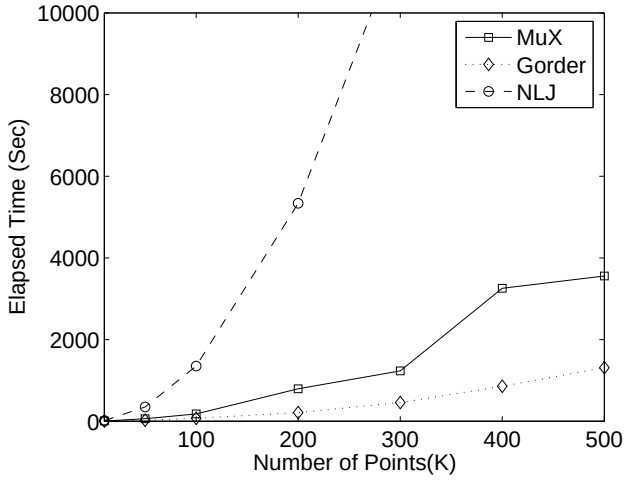


(b) I/O Time

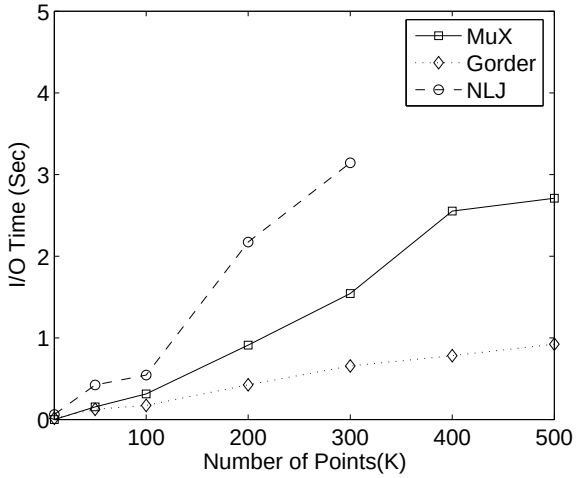


(c) Distance Computation Selectivity

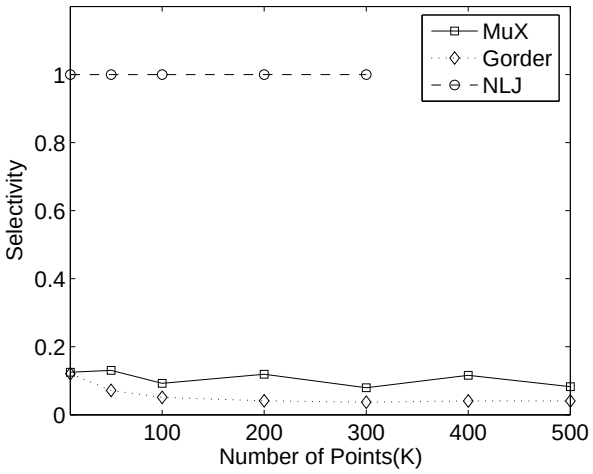
Figure 3.9: Effect of dimensionality (100k clustered dataset)



(a) Elapsed Time

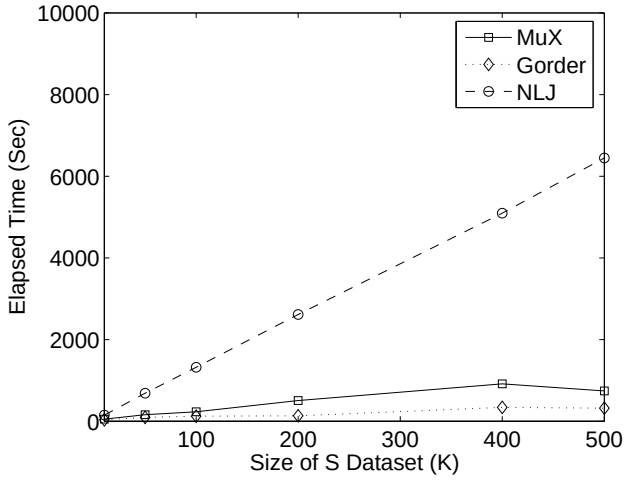


(b) I/O Time

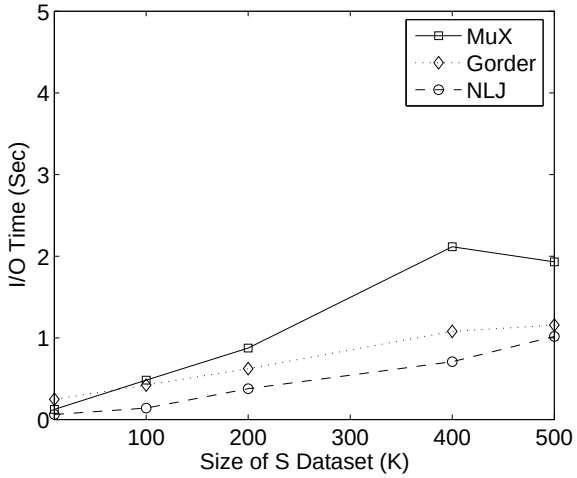


(c) Distance Computation Selectivity

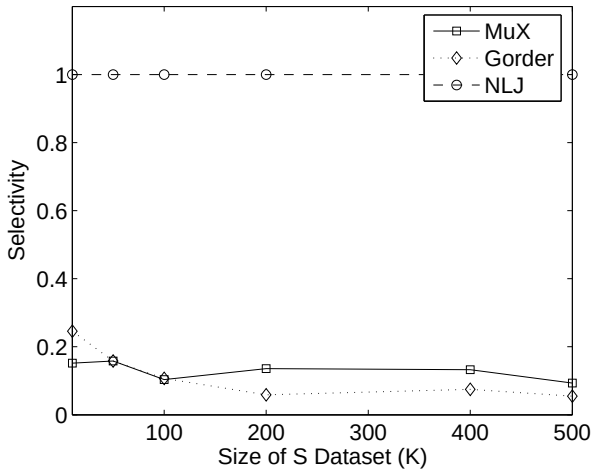
Figure 3.10: Effect of data size (16-dimensional clustered datasets)



(a) Elapsed Time



(b) I/O Time



(c) Distance Computation Selectivity

Figure 3.11: Effect of relative size of datasets (16-dimensional clustered datasets).

marized in Figure 3.10. From the result, Gorder is noted to be the most efficient method for datasets of various sizes. With the increase of dataset size, the elapsed time of MuX grows faster than Gorder. The speed-up factor of Gorder over MuX ranges from 1.2 to 2.8. Note that even for small datasets where the distance computation selectivity of Gorder is similar to MuX, the elapsed time of Gorder is still much lower than MuX due to the use of distance computation reduction technique.

From Figure 3.10 (c), we observe that the distance computation selectivity of Gorder improves slightly when the number of data points grows. The reason is that the increase of the number of data points makes the clusters denser and reduces the distance between a point and its k-nearest neighbors. Therefore, more points can be filtered from distance computation. The study demonstrates that Gorder is scalable to large size of data and has even better performance than MuX for large datsets.

Effect of Relative Size of Dataset

In the last set of experiments, we joined two datasets of different sizes and studied the effect of the relative sizes on the performance of the join algorithms. To study such an effect, we fixed the size of R at 100K points and varied the size of S from 10K to 1,000K so that the relative size of R:S is changed from 10:1 to 1:5. Figure 3.11 shows the results.

Both the elapsed time and I/O time of Gorder increase moderately with the increase in S data size. The cost of MuX goes up comparatively faster, which leads to the wider performance gap between Gorder and MuX as S dataset size increases. Furthermore, note that even at S size of 10K and 50k, where the selectivity of MuX is better than Gorder, Gorder is still much faster. With regard to the elapsed time, the average speed-up factor of Gorder over MuX is 1.3, which confirms the scalability of Gorder with respect to the data size again.

3.5 Summary

This chapter investigates the kNN join problem. The k-nearest neighbor (kNN) similarity join is an operation that combines each point of one data set with its kNNs in the other dataset, and it can be used to facilitate data mining tasks such as clustering, classification and outlier detection. It is also capable of providing more meaningful query results than just the range similarity join. We propose *Gorder*, an efficient kNN join processing algorithm that exploits sorting, data page scheduling and distance computation filtering and reduction to reduce both I/O and CPU costs. We prove that Gorder is efficient and scalable with regard to both data dimensionality and size with the intensive performance study on both synthetic cluster and real life datasets. The comparative study also confirms that Gorder outperforms existing methods by a significant margin.

Chapter 4

ERkNN: Efficient Reverse k-Nearest Neighbors Retrieval with Local kNN-Distance Estimation

4.1 Introduction

The reverse k-Nearest Neighbors (RkNN) query aims to find points in a dataset that have the given query point as one of their k-nearest neighbors (kNN). It has many applications in profile-based marketing, information retrieval, decision support and data mining systems and has received considerable attention in the recent years[78, 115, 125, 114, 76, 79]. The RkNN query is much more complex than the traditional one point similarity queries such as the kNN query and the range query because the reverse k-nearest neighbors are not necessary to localize to the neighborhood of the query point.

The naive solution for the RkNN search is very expensive. A number of methods have been proposed to process the RkNN query efficiently[78, 125, 112, 116, 114]. They can be divided into two categories - *pre-computation* methods[78, 125] and *space pruning* methods [112, 116, 114]:

- *Pre-computation* methods (the RNN tree [78] and the Rdnn-tree [78, 125]) precompute the nearest neighbors of each point in the datasets and store the pre-

computed information in hierarchical structures. This approach cannot answer an RkNN query unless the corresponding k-nearest neighbor information is available.

- *Space pruning* methods (SFT [112], TPL [116] and SAA [114]) utilize the geometry properties of RNN to find a small number of data points as candidates and then verify them with kNN queries or range queries.

A shortcoming of *pre-computation* methods is that they cannot answer an RkNN query unless the corresponding k-nearest neighbor information is available. Since the values of k of RkNN queries may vary greatly in many applications, storing the k-nearest neighbor information for all possible values of k is expensive and sometimes infeasible, and maintaining such a large amount of k-nearest neighbor information in the presence of frequent updates is even more costly. *Space pruning* methods can answer an RkNN query without the priori knowledge of the k-nearest neighbor information. However, all these methods become very expensive when data dimensionality is high or when the value k is large.

Our work, motivated by the deficiencies of previous methods, aims to design a search algorithm which works efficiently for the RkNN query in the high-dimensional spaces and need not store k-nearest neighbor information for all possible values of k. We overcome the difficulty of the RkNN query with the estimation techniques. The ERkNN - an estimation-based RkNN search algorithm is proposed. ERkNN employs the filter-and-refine framework. It retrieves RkNN candidates based on the estimated kNN-distance (kNN-distance is the distance from a data point to its kth-nearest neighbor). This estimation-based filter has the advantage that its computation cost is much lower than the filtering strategies employed by space pruning methods, which improves the RkNN query speed by orders of magnitude. We provide two local kNN-distance estimation methods - the PDE method and the kDE method, which enable ERkNN to answer an RkNN query even the corresponding kNN-distance information is unavailable. Ex-

Symbol	Definition
d	data dimensionality
k or $\mathcal{K}$	an integer, number of nearest neighbors
p and q	data point and query point
$Dist(p,q)$	distance between points p and q
$dnn_k(p)$	kNN-distance - distance between p and its kth-nearest neighbor
$ednn_k(p)$	estimated kNN-distance
$dnn_{\mathcal{K}}(p)$	$\mathcal{K}$NN-distance - distance between p and its $\mathcal{K}$th-nearest neighbor

Table 4.1: Symbols and definitions.

tensive experiments on both synthetic and real-world datasets demonstrate that ERkNN finds RkNN efficiently and effectively and is scalable with respect to data dimensionality, the value of k, and data size.

The remainder of the chapter is organized as follows.

- Section 4.2 investigates the RkNN problem and presents its interesting properties.
- Section 4.3 introduces the local kNN-distance estimation methods, describes the algorithm of ERkNN and provides an accuracy and cost analysis of the ERkNN algorithm.
- Section 4.4 presents the extensive performance study on both synthetic and real life datasets which compares the query performance of the ERkNN with other RkNN query algorithms.
- Section 4.5 concludes this chapter with a summarization.

Table 4.1 gives a summarization of the symbols being used frequently in this chapter.

4.2 Properties of the RkNN Query

The RkNN query has the following properties:

1. The points in the answer set of an RkNN query are not necessary to localize to the query point's neighborhood.

2. If we know apriori the *kNN-distance* (the distance between a point and its k-th nearest neighbor) of each point in the dataset, the RkNN query can be transformed into a point enclosure query [78].

3. The monotonicity property: if $k < K$, $RkNN(q, R) \subseteq RKNN(q, R)$.

The first property is unique to the RkNN query and makes the RkNN query more difficult than other *one point* similarity queries - the range query and the kNN query. The example in Figure 1.1 illustrates this behavior. Let p_2 be the query point and k=2. We observe p_2 is one of the 2-nearest neighbors of p_1, p_3, and p_4. Hence, p_2's reverse 2-nearest neighbors are p_1, p_3, and p_4. Note that p_4 is an R2NN of p_2 although it is far from the query point p_2. In contrast, p_5 and p_7 are not answers of the R2NN query of p_2 although they are close to p_2.

This property has serious impact on the design of the RkNN search algorithm. In order to retrieve all the correct answers, algorithms that zoom in to the query point and iteratively expand the search region to look for promising answers cannot be terminated until the k-nearest neighbors of all the points in the dataset are evaluated. Therefore, the complexity of the RkNN query is upper-bounded by $O(N^2)$, where N is the cardinality of the input dataset.

However, as stated in Property 2, if we know the *kNN-distance* of each point in advance, the RkNN query can be simplified to a point enclosure query [78] (also known as the point containment query). The point enclosure query checks the distance between the query point q and data points in the dataset and retrieves data point p such that the distance between p and q, $Dist(p, q)$ is less than or equal to p's *kNN-distance* (denoted as $dnn_k(p)$). The reason is if $Dist(q, p) \leq dnn_k(p)$, then q is one of p's k-nearest

neighbors. Therefore, p is one of the reverse k-nearest neighbors of q. By this way, the searching speed of the RkNN query is improved significantly and an RkNN query can be answered in at most $O(N)$ time. The *pre-computation* methods work according to this mechanism. An obvious shortcoming is that for each possible k value, the corresponding *kNN-distance* information of each point in the dataset should be calculated in advanced and stored for the RkNN query.

The estimation-based RkNN search algorithm (ERkNN) makes use of this property of the RkNN query as well but overcomes the shortcoming of the pre-computation methods by utilizing the *local estimation methods* to estimate the *kNN-distance* of data points in the dataset. It uses the point enclosure query to retrieve RkNN candidates according to the estimated kNN-distance in the filter step of ERkNN. Thus, it is able to give approximate answers to an RkNN query when the corresponding kNN-distance information is unknown.

Property 3 - the monotonicity property is obvious for the kNN query but not so apparent for the RkNN query because of Property 1 - the non-locality property. We prove it as the following:

Proof: For any data point p in the the answer set of $RkNN(q, R)$, we have

$$Dist(q, p) \leq dnn_k(p).$$

where $Dist(q, p)$ is distance between p and q, and $dnn_k(p)$ is point p' *kNN-distance*.

Since $k < K$, we have

$$dnn_k(p) \leq dnn_K(p).$$

where $dnn_K(p)$ is point p' *KNN-distance*.

So,

$$Dist(q, p) \leq dnn_K(p).$$

Algorithm 5 ERkNN(T, q, k)

Input:

T is the index tree, q is the query point, k is an integer.

Output:

RkNN answers.

Description:

1: $\mathcal{A} = \emptyset$; /*$\mathcal{A}$ is the RkNN candidate set*/
2: **Filter**(T, q, k, $\mathcal{A}$);
3: **Refinement**(T, q, k, $\mathcal{A}$);

Therefore, p is an answer to the query $RKNN(q, R)$ too.

Thus, we have

$$RkNN(q, R) \subseteq RKNN(q, R)$$

■

4.3 Estimation-Based RkNN Search

The ERkNN algorithm employs the filter-and-refine framework. Algorithm 5 outlines the algorithm. The first step *Filter* retrieves a set of points p whose distance to the query point q is equal to or greater than p's estimated kNN-distance as RkNN candidates. The second step *Refinement* verifies the candidates with the *aggregated range query*.

The novelty of ERkNN lies in its efficient candidate retrieval based on the *local kNN-distance estimation* which accurately approximates each point's kNN-distance. It answers RkNN queries of arbitrary k efficiently without requiring the corresponding *kNN-distance* information. This estimation-based filter outperforms the filter methods used by methods significantly especially when data dimensionality is high and k is big. In addition, the refinement step of ERkNN employs the aggregated range query to efficiently reduce the CPU and I/O cost and makes ERkNN more efficient.

In the following subsections, we first introduce the *local kNN-distance estimation* methods, then give the details of each procedure and provide an analysis of the ERkNN

algorithm.

4.3.1 Local kNN-Distance Estimation Methods

Previous studies on the kNN-distance estimation [15] employ the *global uniform assumption*, that is, the data are uniformly distributed over the whole data space. We call these approaches the *global kNN-distance estimation*. The kNN-distance computed by the *global kNN-distance estimation* is the *average* of the kNN-distance of all the points in the dataset. However, the local density of each point in a dataset varies considerately and so does the kNN-distance of each point. RkNN candidate retrieval that is based on the average kNN-distance tends to produce a large number of false misses (data points that are answers but missed) and false hits (data points that are not answers but retrieved), which decreases the recall of the correct RkNN answers and as well as increases the refinement cost.

In this work, we introduce the idea of the *local kNN-distance estimation* which is based on the nonparametric density estimation [41]. The nonparametric density estimation estimates a point's local density function by a small number of neighboring samples around the point. The resulting local density function gives a much better approximation of the data distribution compared to the global uniform assumption. Hence, the *local kNN-distance estimation* approximates the kNN-distance of each point much more accurately than the global approach.

We develop two *local kNN-distance estimation* methods - the PDE method and the kDE method. The PDE method is based on the parzen density estimator with uniform kernel [41], which is a most commonly-used non-parametric density estimator. The kDE method, as an alternative to the PDE method, is based on the interesting finding which is deduced from the kNN density estimator [41]. Our experiment study show that these methods produce similar estimation results, work effectively on both synthetic and real

life datasets and outperform the global approach significantly.

The PDE method

The PDE method is based on the parzen density estimator with uniform kernel [41]. Let $L(p)$ be a small sphere region centered at p. The Parzen Density Estimator counts the number of points falling in $L(p)$ and estimates the local probability density function at p, $\hat{\mathrm{X}}(p)$ as follows:

$$\hat{\mathrm{X}}(p) = \frac{k/N}{V} \tag{4.1}$$

where N is the cardinality of dataset, k is the number of points within $L(p)$ and V is the volume of $L(p)$. $L(p)$ is a d-dimensional hyper-sphere of radius $r = dnn_k(p)$, so

$$V = V_{d,r}(p) = \frac{\sqrt{\pi^d} \cdot r^d}{\Gamma(d/2+1)} = \frac{\sqrt{\pi^d} \cdot dnn_k(p)^d}{\Gamma(d/2+1)} \tag{4.2}$$

where

$$\Gamma(x+1) = x\Gamma(x),\ \Gamma(1) = 1,\ \Gamma(1/2) = \sqrt{\pi}.$$

Combining Equation 4.1 and 4.2, we have

$$\hat{\mathrm{X}}(p) = \frac{k/N \cdot \Gamma(d/2+1)}{\sqrt{\pi^d} \cdot dnn_k(p)^d} \tag{4.3}$$

Applying the uniform kernel assuming that the data density is uniform over p's kNN vicinity to $\mathcal{K}$NN vicinity (k and $\mathcal{K}$ are two integers - the number of data points in $L(p)$, the hyper-sphere region centered at p, $k \neq \mathcal{K}$), we obtain

$$\hat{\mathrm{X}}(p) = \frac{k/N \cdot \Gamma(d/2+1)}{\sqrt{\pi^d} \cdot dnn_k(p)^d} = \frac{\mathcal{K}/N \cdot \Gamma(d/2+1)}{\sqrt{\pi^d} \cdot dnn_{\mathcal{K}}(p)^d} \tag{4.4}$$

Hence we have

$$dnn_k(p) = dnn_{\mathcal{K}}(p) \cdot \sqrt[d]{\frac{k}{\mathcal{K}}} \tag{4.5}$$

Thus, we have the PDE method which estimates kNN-distance of p using the following equation:

$$ednn_k(p) = dnn_{\mathcal{K}}(p) \cdot \sqrt[d]{\frac{k}{\mathcal{K}}} \tag{4.6}$$

where $ednn_k(p)$ is the estimated kNN-distance of p and d is data dimensionality. $dnn_{\mathcal{K}}(p)$ is the $\mathcal{K}$NN-distance (the distance between p and its $\mathcal{K}$th nearest neighbor). It works as a base of estimation and is pre-computed in advance. For the *kNN-distance* of different values of k, the same $\mathcal{K}$NN-distance is used for estimation.

Note that although the PDE method employs the uniform kernel, the uniform assumption is applied only in the local region of a point p, that is, the hyper-sphere $(p, kdist)$ which is centered at p and of radius $kdist = max(dnn_k(p), dnn_{\mathcal{K}}(p))$. Since the $\mathcal{K}$NN-distance of each point p captures the local density of a point p, the PDE method estimates a point's kNN-distance much more accurately than the global approach does.

The kDE method

The kDE method is based on an interesting finding which is deduced from the kNN density estimator [41, 74] [1], that is, the ratio of (k+1)NN-distance to the kNN-distance is as follows [41, 74]:

$$\frac{dnn_{k+1}}{dnn_k} \cong 1 + \frac{1}{kd}$$

[1] Refer [41] for the detail of deduction

Thus, we have,

$$\begin{aligned} dnn_{k+1} &\cong dnn_k \cdot (1 + \tfrac{1}{kd}) \\ dnn_{k+2} &\cong dnn_{k+1} \cdot \left(1 + \tfrac{1}{(k+1)d}\right) \\ &\cong dnn_k \cdot \left(1 + \tfrac{1}{kd}\right) \cdot \left(1 + \tfrac{1}{(k+1)d}\right) \\ &\dots \end{aligned}$$

So for any two integers k_1 and k_2 ($k_1 \neq k_2$),

$$\begin{aligned} if \quad k_1 < k_2 \quad dnn_{k_2} &\cong dnn_{k_1} \cdot \textstyle\prod_{i=k_1}^{k_2-1}(1 + \frac{1}{i \cdot d}) \\ if \quad k_1 > k_2 \quad dnn_{k_2} &\cong \frac{dnn_{k_1}}{\prod_{i=k_2}^{k_1-1}(1+\frac{1}{i \cdot d})} \end{aligned}$$

Therefore, we have the kDE method which estimates kNN-distance using Equation 4.2:

$$ednn_k(p) = \begin{cases} dnn_{\mathcal{K}}(p) \cdot \prod_{i=\mathcal{K}}^{k-1}(1 + \frac{1}{i \cdot d}) & \text{if } k > \mathcal{K} \\ dnn_{\mathcal{K}}(p) & \text{if } k = \mathcal{K} \\ \dfrac{dnn_{\mathcal{K}(p)}}{\prod_{i=k}^{\mathcal{K}-1}(1+\frac{1}{i \cdot d})} & \text{if } k < \mathcal{K} \end{cases} \tag{4.7}$$

where $ednn_k(p)$ is the estimated kNN-distance of p and d is data dimensionality. $dnn_{\mathcal{K}}(p)$ is the $\mathcal{K}$NN-distance of p.

Discussions

The kNN-distance estimated by the PDE or the kDE methods is an approximation of the real kNN-distance, so the candidate set retrieved by the filter procedure of ERkNN may contain false hits and miss true answers due to the estimation error. The false hits will be removed with the refinement procedure of ERkNN. The problem of false misses will be discussed in Section 4.3.3.

For RkNN queries of different k values, ERkNN uses the same $\mathcal{K}$NN-distance as the basic for estimation. It is observed that when k is far from $\mathcal{K}$, the approximation becomes less accurate. This problem can be alleviated by a multiple $\mathcal{K}$s version of ERkNN. That is, we store several $\mathcal{K}$NN-distances ($\mathcal{K}_1$NN-distance, $\mathcal{K}_2$NN-distance,... $\mathcal{K}_m$NN-distance) and estimate a point's kNN-distance according to the $\mathcal{K}_i$NN-distance such that $\mathcal{K}_i$ is closest to k. ERkNN with multiple $\mathcal{K}$s is a straightforward extension of the single $\mathcal{K}$ case, so we will focus on the single $\mathcal{K}$ version ERkNN here.

The data dimensionality d can be evaluated by either the *embedded* dimensionality or the *intrinsic* dimensionality [123]. The *embedded* dimensionality is the length of the feature vector of data and the *intrinsic* dimensionality is the number of the *effective* features of data. Studies in query cost analysis and pattern recognition show that cost estimation and data analysis based on intrinsic dimensionality are more accurate. This is same for the *local kNN-distance estimation* according to our experimental study. The PDE and kDE methods estimate the kNN-distance more accurately when the *intrinsic* dimensionality is used in Equation 4.6 and 4.7. Approaches for intrinsic dimensionality computation are in [123].

4.3.2 The Algorithm

We now present the filter and refinement procedures of ERkNN. We use the Rdnn-tree [125] data structure for the search.

Data Structure: The Rdnn-tree is basically an R-tree that is augmented with the nearest neighbor *distance* (NN-distance). We store the data points and the $\mathcal{K}$NN-distance in the leaf nodes of the Rdnn-tree. Each leaf node entry e has the form (p, $dnn_{\mathcal{K}}(p)$), where p is the data point and $dnn_{\mathcal{K}}(p)$ is the $\mathcal{K}$NN-distance of p.

Each entry e in the *internal* nodes of the Rdnn-tree has the form (ptr, $MaxDnn_{\mathcal{K}}$, mbr). ptr points to a sub-node N'; mbr is the minimum bounding rectangle (MBR) of

Algorithm 6 Filter(T, q, k,$\mathcal{A}$)

Input:

T is the Rdnn-tree, q is the query point, k is an integer, $\mathcal{A}$ is the set of RkNN candidates.

Description:

```
1: Initialize queue Q with root of T;
2: while Q is not empty do
3:    Dequeue a node N from Q;
4:    if N is an internal node then
5:       for each sub-node N' of N do
6:          if MinDist(N', q) ≤ Max_ED(N') then
7:             Insert N' to Q;
8:    else
9:       for each point p in N do
10:         if Dist(p, q) ≤ ednn_k(p) then
11:            Insert p into A;
```

N'; $MaxDnn_{\mathcal{K}}$ is the maximal $\mathcal{K}$NN-distance of all data points in the subtree rooted at N'.

$$MaxDnn_{\mathcal{K}} = Max_{i=1}^{m} dnn_{\mathcal{K}}(p_i) \tag{4.8}$$

where $p_1, ..., p_m$ are all points within N'. We use the algorithm described in [125] to build the Rdnn-tree with $\mathcal{K}$NN-distance, except that the NN queries are replaced by the $\mathcal{K}$NN queries. The tree can also be constructed using the bulk approach proposed in [84].

Filter Procedure

In the filtering step, ERkNN retrieves a set of points p whose estimated kNN-distance is equal to or greater than distance from p to the query point q. The estimated kNN-distance $ednn_k(p)$ is computed with either the PDE or the kDE method. This estimation-based filter has the advantage that its computation cost is much lower than the filtering strategies employed by space pruning methods [115, 116, 112].

During the tree traversal, we apply the following *pruning strategy*:

If $MinDist(N, q) \geq Max\_ED(N)$, tree node N can be pruned from traversal,

where $MinDist(N, q)$ is the minimum distance between the query point q and the MBR of N and $Max\_ED(N)$ is computed as follows:

- The PDE method is employed for kNN-distance estimation:

$$Max\_ED(N) = \quad MaxDnn_{\mathcal{K}} \cdot \sqrt[d]{\frac{k}{\mathcal{K}}} \tag{4.9}$$

- The kDE method is employed for kNN-distance estimation:

$$Max\_ED(N) = \begin{cases} MaxDnn_{\mathcal{K}} \cdot \prod_{i=\mathcal{K}}^{k-1}(1 + \frac{1}{i \cdot d}) & \text{if } k > \mathcal{K} \\ MaxDnn_{\mathcal{K}} & \text{if } k = \mathcal{K} \\ \frac{MaxDnn_{\mathcal{K})}}{\prod_{i=k}^{\mathcal{K}-1}(1+\frac{1}{i \cdot d})} & \text{if } k < \mathcal{K} \end{cases} \tag{4.10}$$

Since $MaxDnn_{\mathcal{K}} = Max_{i=1}^{m} dnn_{\mathcal{K}}(p_i)$, according to the computation of $Max\_ED(N)$, $Max\_ED(N) = Max_{i=1}^{m} ednn_k(p_i)$, where $ednn_k(p_i)$ is the estimated kNN-distance of p_i and p_i is a point in N. Therefore, a node N such that $MinDist(N, q) > Max\_ED(N)$ can be pruned from traversal.

Algorithm 6 presents the candidate retrieval procedure that traverses the Rdnn-tree in a breadth-first manner. It utilizes a queue Q to store tree nodes that shall be visited. Q contains the root of the Rdnn-tree initially. While Q is not empty, the algorithm dequeues a node N from Q and processes it according to the node type:

- If N is an *internal* node (line 4-7): for each sub-node N' represented by an entry e in N, it calculates $MinDist$ from the query point q to N', computes $Max\_ED(N')$ and inserts N' such that $MinDist(N', q) \leq Max\_ED(N')$ into Q to be visited later.

Algorithm 7 Refinement(T, q, k, $\mathcal{A}$)

Input:

T is the Rdnn-tree, q is the query point, k is an integer, $\mathcal{A}$ is the set of RkNN candidates.

Description:

1: Initiate range queries;
2: **Refine_in_memory**($\Re$, $\mathcal{A}$);
3: **Range_Queries**(T, k, $\Re$, R_a, $\mathcal{A}$);
4: Output points p_i in $\mathcal{A}$;

- If N is an *leaf* node (line 8-11): for each point p in N, it computes the distance between p and the query point q, estimates the kNN-distance of p and inserts points such that $Dist(p, q) \leq ednn_k(p))$ into the candidate set $\mathcal{A}$.

The algorithm stops when Q is empty, that is, all the tree nodes have been either visited or pruned. All the data points p such that $Dist(p, q) \leq ednn_k(p)$ are retrieved and stored in the candidate set $\mathcal{A}$.

Refinement Procedure

The candidate set $\mathcal{A}$ contains false hits due to the over-estimation of a point p's kNN-distance. A refinement step is needed to remove the false hits.

A point p is a reverse k-nearest neighbor of q if and only if there are *less than* k points p' such that $Dist(p', p) < Dist(p, q)$ [112, 116]. According to this property, the refinement procedure removes candidates with a set of range queries. These range queries have the candidate points as the query points and the distances between the candidate points and the query point of the RkNN query q as query ranges. Candidates that have at least k points within their corresponding query ranges shall be removed from $\mathcal{A}$.

Algorithm 7 shows the four steps in the refinement procedure.

Step 1: Initialization of queries: for each point p_i in $\mathcal{A}$, a range query $R_i(p_i, r_i)$ is initialized and inserted into the query set $\Re$, where p_i is the query point, r_i is the query range and $r_i = Dist(p_i, q)$.

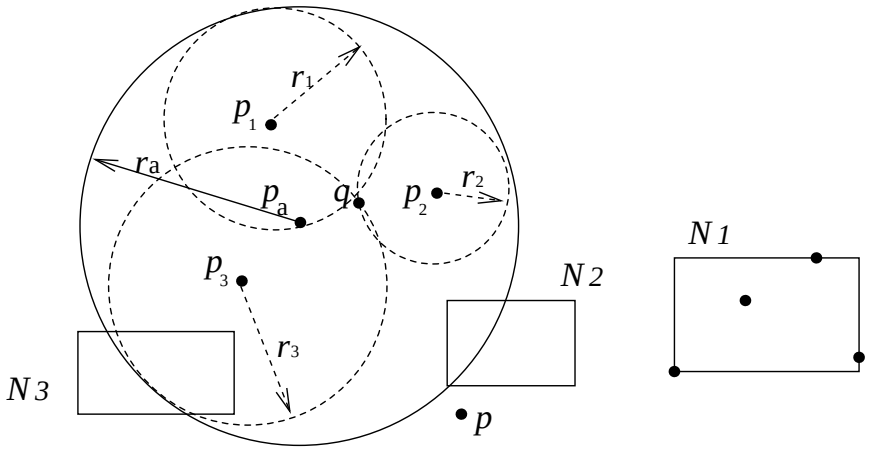


Figure 4.1: Query aggregation and illustration of pruning.

Step 2: A fast refinement in memory: the range queries are first evaluated among the candidates. That is, for each range query R_i, it checks how many candidate points are within R_i's query range. Candidate p_i that has at least k points within its query ranges is removed from $\mathcal{A}$.

Step 3: Range queries: it performs the range queries on the Rdnn-tree and removes data points that has at least k points in their query ranges from the candidate set.

Step 4: Points remains in $\mathcal{A}$ are output as RkNNs.

Steps 1-2 and 4 are straightforward. Step 3 dominates the cost incurred in the refinement procedure. In order to reduce both I/O and CPU cost, we apply the *aggregation strategy* in this step. The basic idea is to first compute an *aggregated range query* $R_a(p_a, r_a)$ before carrying out the individual range queries. The query range of R_a, which is centered at p_a and of radius r_a, covers the search ranges of all R_i in $\Re$. Figure 4.1 gives an example. There are three candidates p_1, p_2 and p_3. The dashed circles are their query ranges. The solid circle is the aggregated query R_a. The computation query sphere of R_a is corresponding to the *minimum enclosing ball* problems [38] whose complexity is lower bounded by $O(|\mathcal{A}|)$, where $|\cdot|$ is the cardinality of a set.

We design the following *pruning strategies* based on the aggregated query R_a:

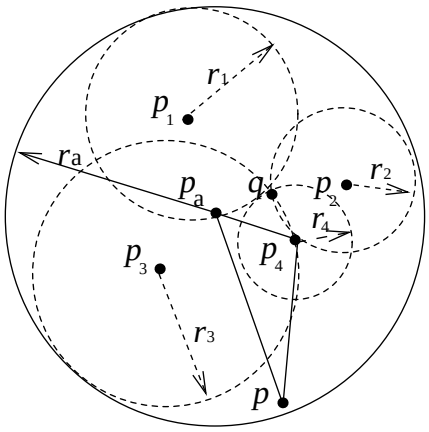


Figure 4.2: Illustration of using triangular inequality property to reduce distance computation.

- *Node pruning*: For a node N, if $MinDist(N, p_a) \geq r_a$, N is surely out of the query range of any R_i in $\Re$ and can be cut off safely (*e.g.* N_1 in Figure 4.1). If $MinDist(N, p_a) < r_a$, we then check whether N intersects with at least one range query R_i in $\Re$. If N intersects with none of them, N can also be pruned away (*e.g.* N_2 in Figure 4.1).

- *Point pruning*: For a data point p, if $Dist(p, p_a) \geq r_a$, p is surely out of the query range of any R_i in $\Re$. (*e.g.* p in Figure 4.1).

- *Query pruning*: When a node N is being visited, if $MinDist(N, p_i) \geq r_i$, all the entries in N are surely out of the query range of R_i. Thus, R_i is marked *ignored* while N is being visited (*e.g.*, when N_3 in Figure 4.1 is being visited, R_1 and R_2 are to be ignored).

- *Distance computation pruning*: The *triangular inequality property* can be used to prune distance computations When the similarity is measured by metric distance. For any point p within the aggregated search region (p_a, r_a), before computing the distance between p and each unpruned candidate points p_i, we check first whether $\mid Dist(p_a, p) - Dist(p_a, p_i) \mid \geq r_i$. If $\mid Dist(p_a, p) - Dist(p_a, p_i) \mid \geq r_i$, p is out-

Algorithm 8 Range_Queries(T, k, $\Re$, R_a, $\mathcal{A}$)

Input:

T is the Rdnn-tree, k is an integer, $\Re$ is a set of range query, R_a is the aggregated query of $\Re$.

Description:

```
1:  Initialize c_i=0 for each query R_i in ℜ;
2:  Initialize priority queue Q with root of T;
3:  while Q is not empty and ℜ is not empty do
4:     Dequeue a node N from Q;
5:     Apply query pruning;
6:     if N is an internal node then
7:        for each sub-node N' of N do
8:           Apply node pruning;
9:           Insert N' into Q if it cannot be pruned;
10:    else
11:       for each point p in N do
12:          if p cannot be pruned by point pruning then
13:             for each not ignored R_i in ℜ do
14:                if Dist(p, p_i) < r_i then
15:                   Increase c_i by 1;
16:                   if c_i = k then
17:                      Remove R_i from ℜ and p_i from A;
```

side of query region of R_i because of the *triangular inequality property*. Therefore, distance computation between p_i and p can be pruned [2]. Figure 4.2 illustrates an example. Distance computation between p_4 and p can be saved because $\mid Dist(p_a, p) - Dist(p_a, p_4) \mid \geq r_4$.

The above pruning strategies show that with the aggregated query R_a, a point p or a node N can be pruned away with a single distance computation of $Dist(p, p_a)$ or $MinDist(N, p_a)$ instead of checking its distance to each query point p_i. This saves a large amount of distance computation and reduces the CPU cost.

Algorithm 8 describes the procedure Range_Queries. c_i counts the number of points within query range of R_i. Q is a priority queue and sorts nodes in ascending order of their $MinDist$ to p_a. Initially, Q contains the tree root. When the first queue item

[2] Note that $Dist(p_a, p_i)$ and $Dist(p_a, p)$ are already known and need not to be calculated again.

N is dequeued, the query pruning strategy is applied to mark the queries such that $MinDist(N, p_i) \geq r_i$ as *ignored* (line 5). Node N is then processed according to its type:

- If N is an *internal* node (line 6-9): for each sub-node N' represented by an entry e in N, it applies the *node pruning* strategies. Node N' that cannot be pruned are inserted into the priority queue Q and shall be visited later.

- If N is an *leaf* node (line 10-17): for each point p in N, it first applies the *point pruning* strategy. If p is not pruned, it checks whether p is within the query range of each *not ignored* query R_i. If true, c_i is increased by 1. Whenever there are k points inside of query range of R_i, p_i is identified as a false hit and removed from $\mathcal{A}$ and range query R_i is also removed from $\Re$.

The procedure stops when either Q is empty or $\Re$ is empty, implying that all the tree nodes that intersect with at least one range query R_i in $\Re$ have been searched or the RkNN query has an empty answer set (*e.g.*, the R2NN of p_8 in Figure 1.1 is empty).

4.3.3 Accuracy Analysis

With the refinement procedure, the precision of the answer set produced by ERkNN is 100%. However, ERkNN may miss some correct answers due to the estimation error. We study the *recall* of the RkNN answer set retrieved by ERkNN in this section.

Lower Bound of the Recall

Definition 4.3.1 *Let $\mathcal{A}$ be the RkNN answer set retrieved by ERkNN and Ω be the complete answer set of the RkNN query, the* recall *of $\mathcal{A}$ is denoted as $\mathcal{R}_{\mathcal{A}}=\frac{|\mathcal{A}|}{|\Omega|}$, where $|\cdot|$ is the cardinality of a set.*

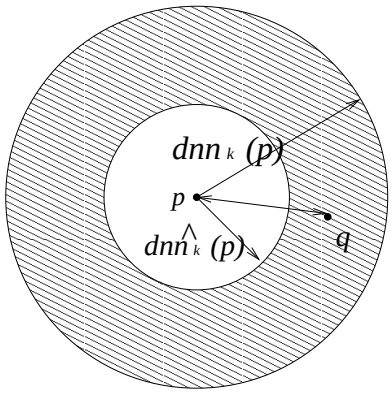


Figure 4.3: Points within the shade area are false misses.

Theorem 4.3.1 *The recall of the answer set retrieved by ERkNN is lower-bounded by* $\int_0^\infty f(x)dx$*, where* $f(x)$ *is the probability distribution of the estimation errors.*

Proof: For a point p in the complete answer set Ω, p is falsely missed when both the following conditions are true: (1) $ednn_k(p) < dnn_k(p)$; (2) $ednn_k(p) < Dist(p,q)$ (see Figure 4.3 as an illustration). Let $Pr\{\cdot\}$ be the probability of an event.

$$\mathcal{R}_{\mathcal{A}} = 1 - \frac{\mid \Omega \mid \cdot Pr\left\{ednn_k(p) < Dist(p,q) \cap ednn_k(p) < dnn_k(p)\right\}}{\mid \Omega \mid}$$

$$= 1 - Pr\left\{ednn_k(p) < Dist(p,q) \cap ednn_k(p) < dnn_k(p)\right\}$$

Since $Pr\left\{ednn_k(p) < Dist(p,q) \cap ednn_k(p) < dnn_k(p)\right\} < Pr\left\{ednn_k(p) < dnn_k(p)\right\}$,

$$\mathcal{R}_{\mathcal{A}} \geq 1 - Pr\left\{ednn_k(p) < dnn_k(p)\right\}$$

$$= 1 - Pr\left\{ednn_k(p) - dnn_k(p) < 0\right\}$$

Let $err(p)$ be the estimation error,

$$err(p) = ednn_k(p) - dnn_k(p).$$

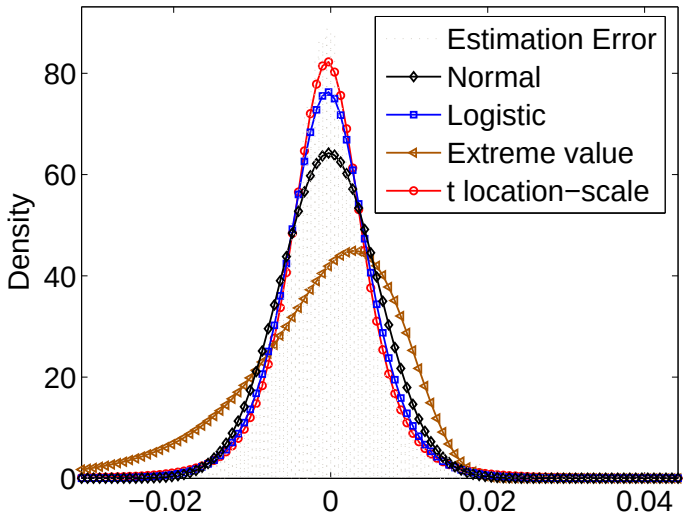


Figure 4.4: Density distribution of estimation errors of Zipf dataset (dim=8, $\mathcal{K}$=15, k=8)

Let $f(x)$ be the probability distribution of $err(p)$.

$$\mathcal{R}_{\mathcal{A}} \geq 1 - Pr\left\{err(p) < 0\right\}$$

$$= 1 - \int_{-\infty}^{0} f(x)dx = \int_{0}^{\infty} f(x)dx$$

Hence, we have Theorem 4.3.1. ■

The probability distribution function of the estimation error $f(x)$ can be modelled by sampling the estimation errors. Our study on various datasets shows that the estimation errors can be well fitted with the Student's t distribution. Figure 4.4 shows an example. We let $\mathcal{K} = 15$ and estimate 8NN-distance using the kDE method on an 8-dimensional Zipf dataset. We plot the frequency of estimation errors with gray dots in the graph and fit it with various distributions. It shows the t location-scale distribution fits the density distribution of estimation errors best. The probability distribution function of estimation errors $f(x)$ can be modelled as follows [2]:

$$f(x) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\,\Gamma\left(\frac{\nu}{2}\right)} \left[\frac{\nu + \left(\frac{x-\mu}{\sigma}\right)^2}{\nu}\right]^{-\left(\frac{\nu+1}{2}\right)} \tag{4.11}$$

μ is the location parameter (i.e., the mean of estimation error); σ is the scale parameter; ν is degrees of freedom. In this example μ=-0.000437092, σ=0.00455945, ν=4.07209.

Improvement of the Recall

The above study shows that the false misses are introduced by the under-estimation of the kNN-distance. Therefore, we can improve the recall by simply introducing a positive adjustment to the estimated kNN-distance. Here, we present two approaches, the *local* adjustment and the *global* adjustment. Let ${ednn_k}'(p)$ be p's estimated kNN-distance after adjustment.

- *local* adjustment: Let ξ be a real number and $\xi > 1$.

$${ednn_k}'(p) = ednn_k(p) \cdot \xi \tag{4.12}$$

  ξ is called the local adjustment factor.

- *global* adjustment: Let λ be a real number and $\lambda > 0$.

$${ednn_k}'(p) = ednn_k(p) + \lambda \tag{4.13}$$

  λ is called the global adjustment factor.

$ednn_k(p)$ in both Equation 4.12 and 4.13 shall be substituted with either Equation 4.6 or 4.7. ERkNN then retrieves a set of points p such that $Dist(p,q) \leq {ednn_k}'(p)$ in the filtering phase and then apply the same refinement algorithm to verify the candidates.

These adjustments reduces $Pr\{err(p) < 0\}$ and hence increases the recall. At the same time, the adjustment makes the filtering step of ERkNN retrieve more data points

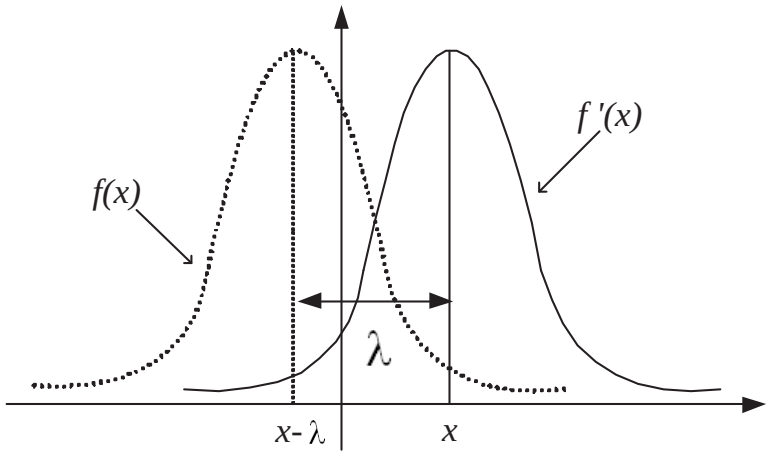


Figure 4.5: Illustration of estimation error distribution after global adjustment.

as candidates which may increase the refinement cost. However, the in-memory refinement procedure filters a number of candidates effectively, so its impact on the overall performance is not significant.

The adjustment of the local method is proportional to the local density of each point. Therefore, local adjustment is more effective than the global adjustment especially for the skewed datasets and real life datasets. The global adjustment has its own advantage as well. Since the distribution of the estimation error after global adjustment only moves along the x-axis (See Figure 4.5 for illustration). The lower bound of the recall after the global adjustment can be predicted immediately according to the distribution of estimation error before the global adjustment.

Theorem 4.3.2 *The recall of $\mathcal{A}$ after global adjustment is lower bounded by $\int_{-\lambda}^{\infty} f(x)dx$, where λ is the global adjustment factor, $\mathcal{A}$ is a set of data points retrieved by ERkNN according to the estimated kNN-distance with λ global adjustment. $f(x)$ is the probability distribution function of estimation error before the global adjustment.*

Proof: Let $err'(p) = ednn_k(p, \lambda) - dnn_k(p)$. As $err'(p) = err(p) + \lambda$, probability distribution function of $err'(p)$, $f'(x) = f(x - \lambda)$ (see Figure 4.5). According to

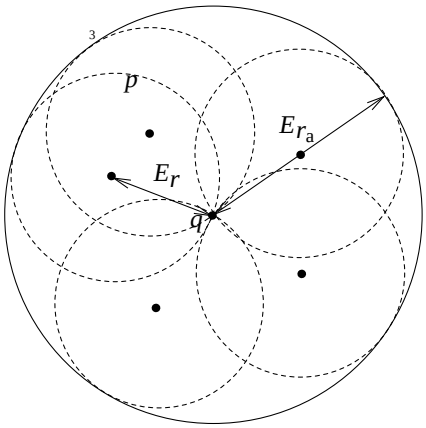


Figure 4.6: Expected aggregated range.

Theorem 4.3.1, $\mathcal{R}_{\mathcal{A}} \geq \int_0^{\infty} f'(x)dx = \int_{-\lambda}^{\infty} f(x)dx$. Therefore, we prove Theorem 4.3.2. ■

Moreover, given a required lower bound of recall r, we can calculate the global adjustment factor λ by solving equation $\int_{-\lambda}^{\infty} f(x)dx = r$. Using the example in Figure 4.4, assume that the required recall is 90%. By checking the table of upper critical values of Student's t distribution, we find the upper critical point of 10% accumulated probability is 1.53 when degrees of freedom (ν) of Student's t distribution is 4 [2]. Then, $\lambda = 1.53\sigma - \mu$=0.0074130505.

4.3.4 Cost Analysis

The filter step of ERkNN executes the point enclosure query and accesses the R-tree nodes containing data point p such that $Dist(p, q) \leq ednn_k(p)$. Let E_r be the expected estimated distance of data points in a dataset. The filter step of the ERkNN is expected to access the R-tree nodes nr such that $MinDist(nr, q) \leq E_r$ and its query cost is equal to the query cost of the range query of query radius E_r.

The refinement step of ERkNN executes the aggregated range query and accesses the R-tree nodes nr such that $MinDist(nr, q)$ is smaller than the aggregated query radius

r_a. So the cost of the refinement step of the ERkNN is equal to the expected cost of the range query of query radius E_{r_a}. As illustrated in Figure 4.6, the expected aggregated range

$$E_{r_a} = 2 \cdot E_r.$$

According to the Minkowski Sum model proposed in [15] and [21], the number of node accesses of a range query of query radius r,

$$A(r) = \frac{N}{C_{eff}} \sum_{l=0}^{d} \binom{d}{l} \cdot \left(\left(1 - \frac{1}{C_{eff}}\right) \cdot \sqrt[d]{\frac{C_{eff}}{N}} \right)^k \cdot \frac{\sqrt{\pi^{d-l}}}{\Gamma\left(\frac{d-l}{2}+1\right)} \cdot r^{d-l} \quad (4.14)$$

where C_{eff} is effective data page capacity, that is, the average number of entries per node.

Therefore the total number of node accesses of ERkNN

$$A_{all} = A_{E_r} + A_{E_{r_a}} \quad (4.15)$$

$$A_{E_r} = \frac{N}{C_{eff}} \sum_{l=0}^{d} \binom{d}{l} \cdot \left(\left(1 - \frac{1}{C_{eff}}\right) \cdot \sqrt[d]{\frac{C_{eff}}{N}} \right)^k \cdot \frac{\sqrt{\pi^{d-l}}}{\Gamma\left(\frac{d-l}{2}+1\right)} \cdot E_r^{d-l} \quad (4.16)$$

$$A_{E_{r_a}} = \frac{N}{C_{eff}} \sum_{l=0}^{d} \binom{d}{l} \cdot \left(\left(1 - \frac{1}{C_{eff}}\right) \cdot \sqrt[d]{\frac{C_{eff}}{N}} \right)^k \cdot \frac{\sqrt{\pi^{d-l}}}{\Gamma\left(\frac{d-l}{2}+1\right)} \cdot (2 \cdot E_r)^{d-l} \quad (4.17)$$

The expected estimated distance E_r is close to the average kNN-distance of the dataset.

$$E_r \approx \sqrt[d]{\frac{k \cdot \Gamma(d/2+1)}{N}} \cdot \frac{1}{\sqrt{\pi}}$$

where, $\Gamma(x+1) = x\Gamma(x), \;\; \Gamma(1) = 1, \;\; \Gamma(1/2) = \sqrt{\pi}$.

The number of distance computation of the filter step is:

$$A_{E_r} \cdot C_{eff}.$$

The number of distance computation of the refinement step is:

$$A_{E_{ra}} \cdot C_{eff} \cdot E_{c_{num}}.$$

Where $E_{c_{num}}$ is the expected number of reverse k-nearest neighbor candidates.

$E_{c_{num}}$ is equal to the expected number of points within the hyper-sphere of radius E_r.

$$E_{c_{num}} = \frac{N \cdot \sqrt{\pi^d} \cdot E_r^d}{\Gamma(d/2+1)}$$

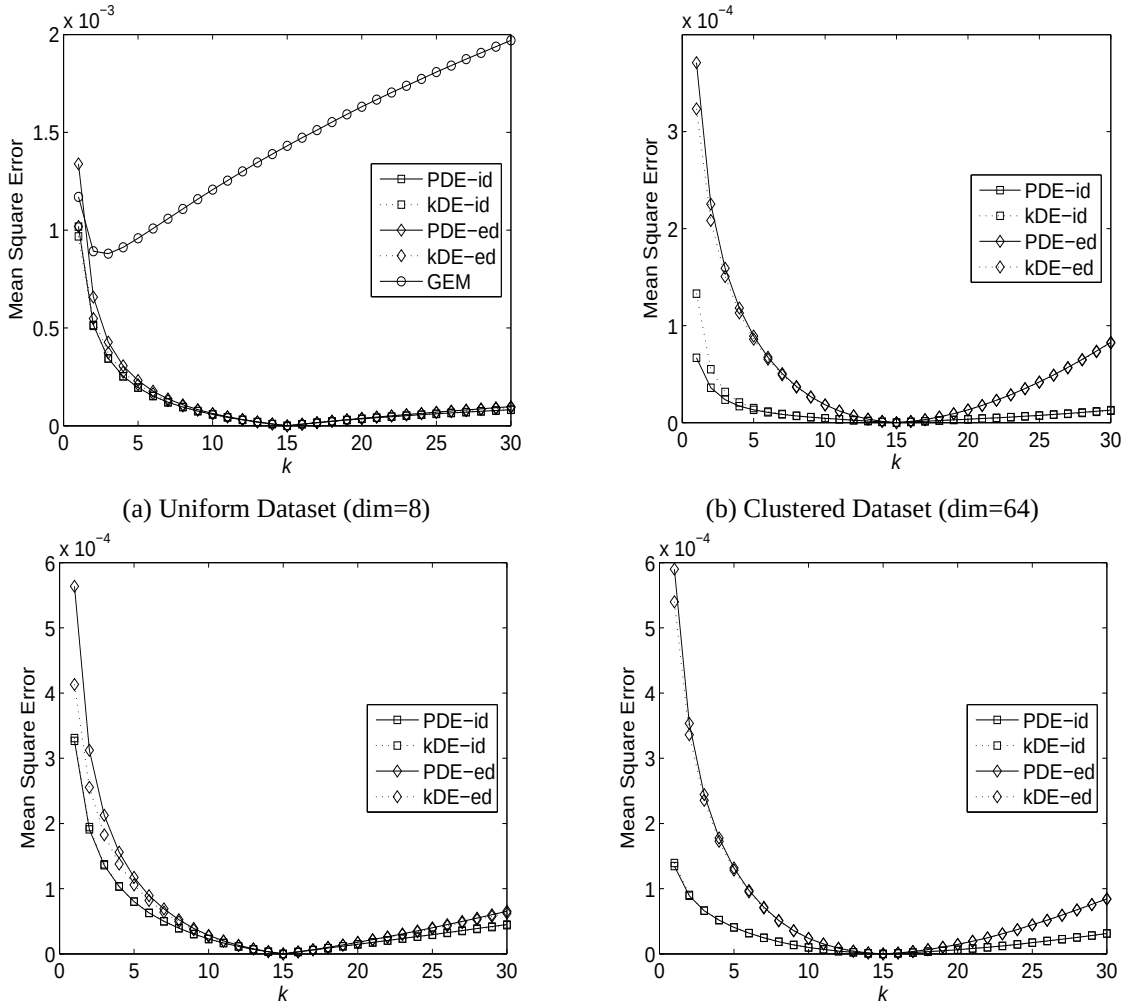
Therefore, the expected total number of distance computation is:

$$A_{E_r} \cdot C_{eff} + A_{E_{ra}} \cdot C_{eff} \cdot \frac{N \cdot \sqrt{\pi^d} \cdot E_r^d}{\Gamma(d/2+1)}.$$

4.4 Performance Study

In this section, we present the results of our experiments to evaluate ERkNN. We use both synthetic and real life datasets. The synthetic datasets are of different distributions - uniform distributed, Zipf distributed and clustered. The synthetic cluster datasets were generated using the method described in [68]. The real life datasets are the Corel dataset from UCI KDD data repository [3] which contains 32 dimensional feature vectors of around 60K images.

We compare ERkNN with TPL [116] and SFT [112]. We exclude SAA because its performance is significantly worse than SFT and TPL [116]. For both the R-tree (used by SFT and TPL) and the Rdnn-tree (used by ERkNN), the node size is 8192 bytes.



(a) Uniform Dataset (dim=8)

(b) Clustered Dataset (dim=64)

(c) Zipf Dataset (dim=8)

(d) Corel Dataset (dim=32)

Figure 4.7: Comparison of kNN-distance Estimation Methods

By default, SFT retrieves $5 \cdot k$ nearest neighbors as candidates in its filtering phase and $\mathcal{K}$NN-distance used by ERkNN is 15NN-distance. The experiments are conducted on a Pentium 4 2.6GHz PC running WinXP. We measure the performance in terms of CPU time, number of node accesses and the total cost which includes both CPU time and I/O overhead by charging each node access 20ms [28]. The results are the average of 200 RkNN queries. The query points are randomly picked from the datasets.

Dataset	Uniform	Zipf	Corel	Clustered
Embedded	8	8	32	64
Intrinsic	7.2	5.74	6.48	14.451

Table 4.2: Dimensionality of datasets.

4.4.1 Study of kNN-Distance Estimation

The first set of experiments study the proposed local kNN-distance estimation methods - the PDE method and the kDE method. We estimate the kNN-distance of k=1,2,..., 30 and evaluate the estimation accuracy by the mean square error (MSE).

$$MSE = \frac{\sum_{i=1}^{N}(ednn_k(p_i) - dnn_k(p_i))^2}{N} \tag{4.18}$$

where N is the number of data points in the dataset.

Figure 4.7 shows the results on the uniform, Zipf, clustered and real life Corel datasets. PDE-id (*or* kDE-id) indicates PDE (*or* kDE) method using the *intrinsic dimensionality*. PDE-ed (*or* kDE-ed) is the PDE (*or* kDE) method using the *embedded dimensionality*. GEM is a global estimation method that calculates the average kNN-distance using the method proposed in [15].

$$ednn_{kGEM} = \sqrt[d]{\frac{k \cdot \Gamma(d/2+1)}{N}} \cdot \frac{1}{\sqrt{\pi}} \tag{4.19}$$

where N is the cardinality of the dataset.

$$\Gamma(x+1) = x\Gamma(x), \;\; \Gamma(1) = 1, \;\; \Gamma(1/2) = \sqrt{\pi}.$$

We observe that the PDE method and the kDE method have similar accuracies on all the datasets. Estimations using the intrinsic dimensionality are better than the estimations using the embedded dimensionality. The superiority of PDE-id and kDE-id over PDE-ed and kDE-ed is very clear on the Zipf dataset, the Corel dataset and the

clustered dataset where the intrinsic dimensionality is much lower than the embedded dimensionality (see Table 4.2). As for the uniform dataset, its intrinsic dimensionality and embedded dimensionality are similar, so there is not much difference between the estimations using the intrinsic dimensionality and the embedded dimensionality.

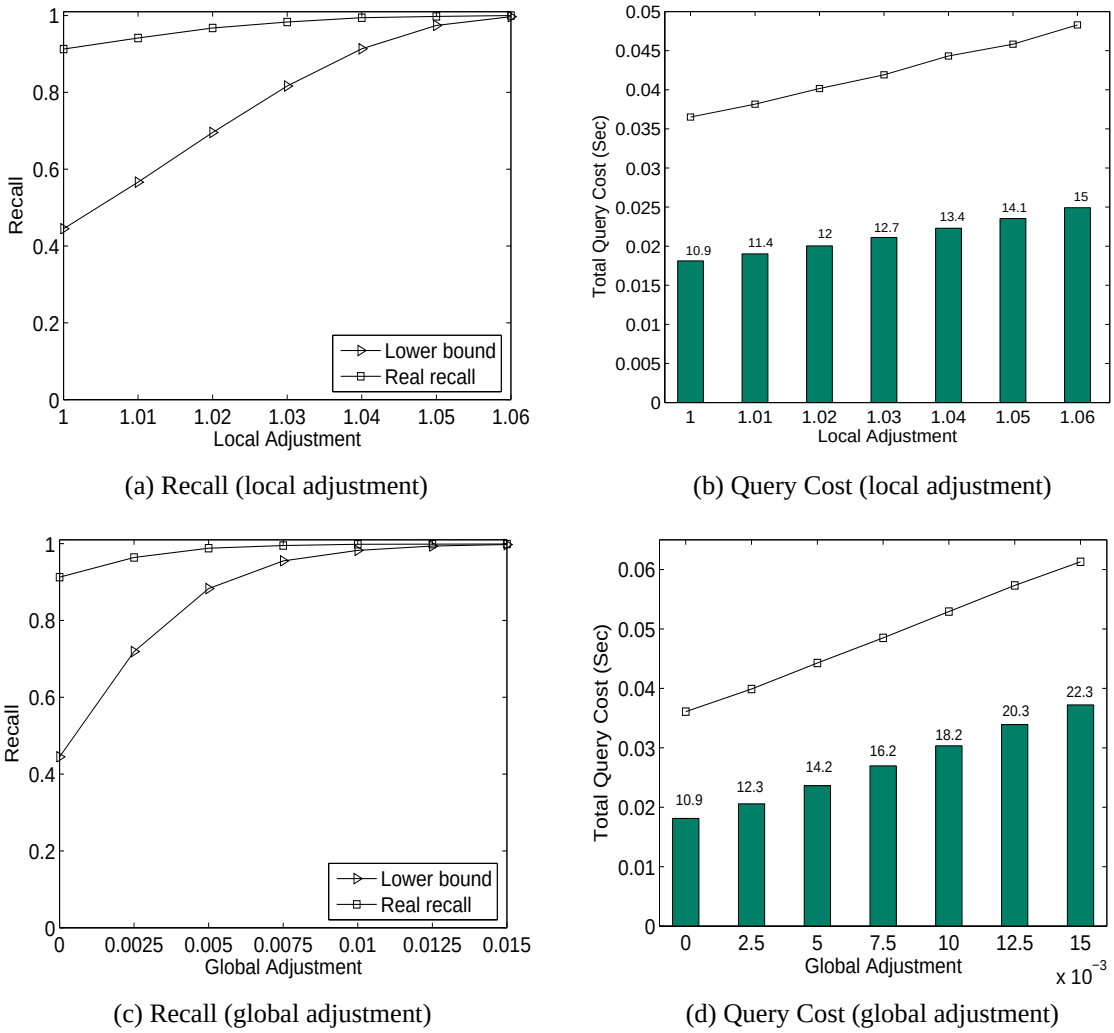
The local estimation methods are much more accurate than the global estimation method. Graphs (a) and (b) in Figure 4.7 demonstrate that the MSE of GEM is much greater than the MSE of PDE and kDE on the Uniform and the clustered datasets. On the uniform dataset, the local estimations are on average 37 times better than GEM. On the Corel and the Zipf datasets, the local estimations outperform GEM even more significantly.[3] On the Zipf dataset, the MSE of GEM using the intrinsic dimensionality is 351 times of the MSE of the local methods averagely. On the Corel dataset, the MSE of GEM with the intrinsic dimensionality is 5310 times of the MSE of local methods averagely. On the 64-dimensional clustered dataset, the MSE of GEM with the intrinsic dimensionality is around 40,000 times of the MSE of local methods averagely. GEM using the embedded dimensionality is even worse. Its MSE is 1530 and 1,210,000 higher than the MSE of the local methods on the Zipf and the Corel datasets respectively.

The study demonstrates that local estimations outperform the global approach significantly and yield more accurate approximation of the kNN-distance of each point on both uniformly distributed datasets and real and skewed datasets. The study also confirms that the intrinsic dimensionality captures the effective data dimensionality and leads to better estimations.

4.4.2 Study of the Recall

Next, we evaluate the recall of the answer set retrieved by ERkNN. We query RkNN $k = 10$ and use the PDE method to estimate local kNN-distance. We first evaluate the

[3] We do not plot the MSE of GEM on the clustered, Zipf and Corel datasets in the graphs because they are too big.



(a) Recall (local adjustment)

(b) Query Cost (local adjustment)

(c) Recall (global adjustment)

(d) Query Cost (global adjustment)

Figure 4.8: Study of recall of ERkNN

local adjustment. Figure 4.8 (a) and (b) present the results on the 100K 8-dimensional Zipf datasets. Figure 4.8(a) exhibits the average recall when we vary the local adjustment factor ξ from 1 to 1.06. As expected, the actual recall is always higher than the lower bound. As ξ increases, the recall approaches 1 and the lower bound becomes tighter. Figure 4.8(b) shows the influence of the local adjustment on the performance of ERkNN in terms of the total query cost. The real number on top of the bars indicate the number of RkNN candidates retrieved. Both the cost of ERkNN and the number of RkNN

candidates increase moderately with the increase of ξ.

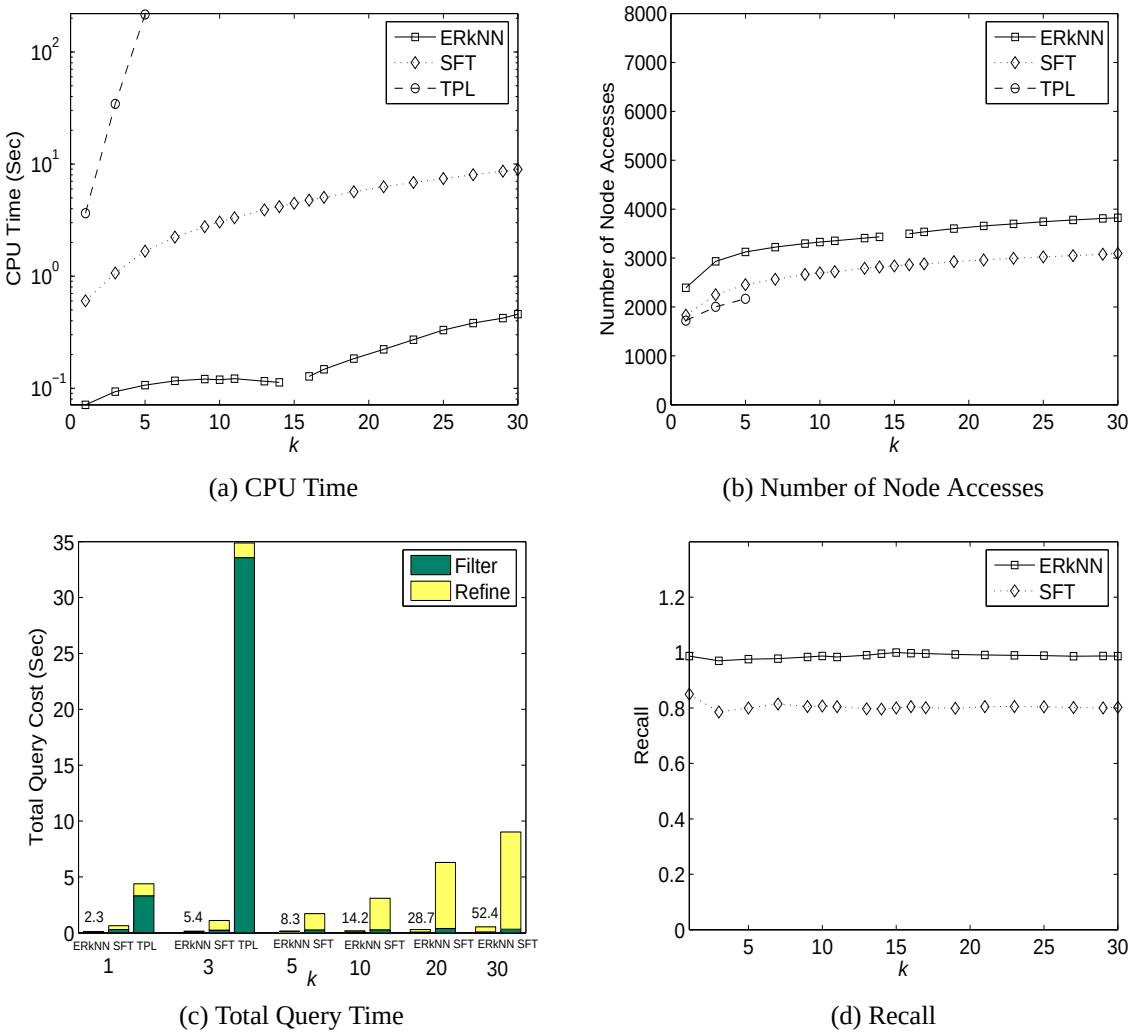
We evaluate the effect of global adjustment on various datasets. Figure 4.8 (c) and (d) show the results of the study on the 8-dimensional Zipf dataset. The global adjustment factor λ is varied from 0 to 0.015. Our study finds that the global adjustment has the similar influence on the recall and performance of ERkNN as the local adjustment on the uniform data but works more effectively on the skewed Zipf dataset. On the Zipf dataset, when ERkNN with local adjustment reaches 100% recall, only 15 RkNN candidates are retrieved. While ERkNN with global adjustment needs to retrieve 22 candidates. The reason is the local adjustment is more adaptive to the data density distribution.

The experiment shows that the recall of ERkNN can be adjusted effectively and the adjustments affect the performance of ERkNN moderately.

4.4.3 Study on Real Dataset

We now compare the performance of ERkNN with SFT and TPL on real datasets with varying values of k. Figure 4.9 presents the results on the Corel datasets when we vary k from 1 to 30. Note that the lines of ERkNN have a break at k=15 because when $k = \mathcal{K}$ the RkNN queries are answered using the point enclosure query directly. The reasons that SFT is more efficient than TPL in our experiments, which is contrary to the experiment results in [116], are two-fold. First, SFT retrieves only $5 \cdot k$ points as candidates in our experiments, while SFT retrieves $10 \cdot d \cdot k$ points as candidates in the experiments in [116]. Second, we use an optimized SFT with batch execution of the boolean range queries [112]. The batch execution of boolean range queries reduces I/O cost and speeds up the query performance considerably [112].

This study shows that ERkNN outperforms both TPL and SFT significantly. The speed-up factor in terms of the total query time is 50.5 when k is 1 and 2024.45 when k is 3. The average speed-up of ERkNN over SFT is more than 20. TPL is expensive when



(a) CPU Time

(b) Number of Node Accesses

(c) Total Query Time

(d) Recall

Figure 4.9: Effect of k (Corel dataset)

k is large mainly because the *k-trim* algorithm used by TPL to prune an R-tree node requires to do $\binom{n_c}{k}$ times *clippings* [116], where n_c is the number of RkNN candidates.

ERkNN is more efficient than SFT because its low CPU cost for candidates retrieval and refinement. During the filtering phase, ERkNN performs the point enclosure query, while SFT performs the kNN query. kNN query is more expensive due to the additional CPU cost to sort and insert the kNN candidates. It is considerable especially when k is large. The refinement procedure of ERkNN is also more efficient because

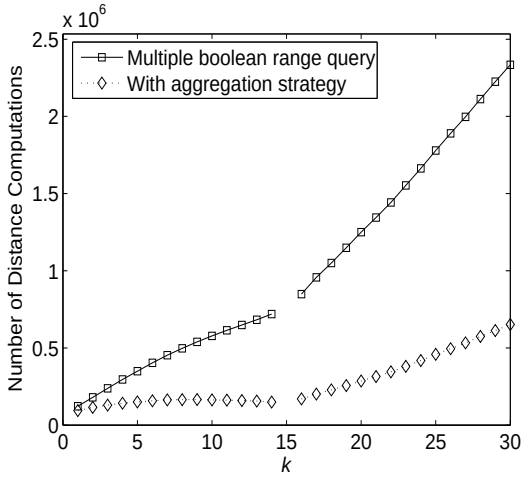
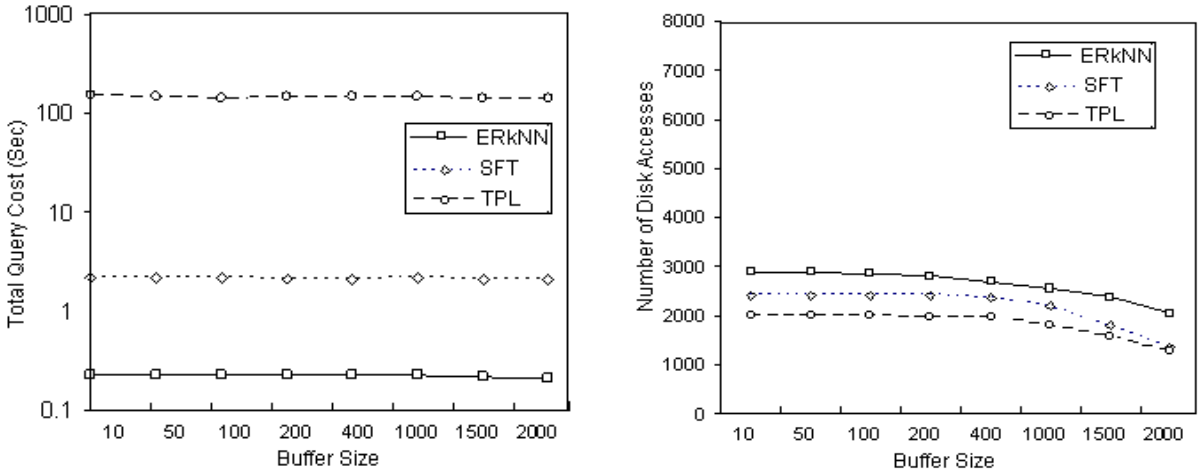


Figure 4.10: Number of distance computation on Corel dataset

ERkNN retrieves much fewer candidates than SFT and the *aggregation strategy* employed by ERkNN prunes a large number of distance computations, thus reducing the CPU cost greatly. Experiment results on the Corel dataset show that the *aggregation strategy* prunes around 75% distance computations on average (see Figure 4.10).

We observe that ERkNN incurs more node accesses than SFT. This is because ERkNN accesses more nodes during the filtering phase (ERkNN accesses 1863.95 nodes for the Corel dataset while SFT accesses 1319.65 nodes on average). The reason is that ERkNN may access more tree nodes to retrieve some potential RkNNs which are far from the query point according to the estimated kNN-distance. Further, the lowest recall of ERkNN is 97.12% on the Corel dataset. The lowest recall of SFT is 79.6%.

The study also shows that for the RkNN query, I/O cost is no longer the dominant cost. For SFT and ERkNN, both methods execute a set of range queries simultaneously and traverse the index tree only once in the refinement procedure. CPU cost is more expensive because there are multiple candidates to be verified. On Corel data, the average I/O overhead of ERkNN and SFT is 0.069 sec (3342.26 node accesses) and 0.055 sec (2763.46 nodes accesses) respectively, while the CPU cost is 0.19 sec and 4.68 sec. CPU



(a) Total Query Time

(b) Number of Disk Accesses

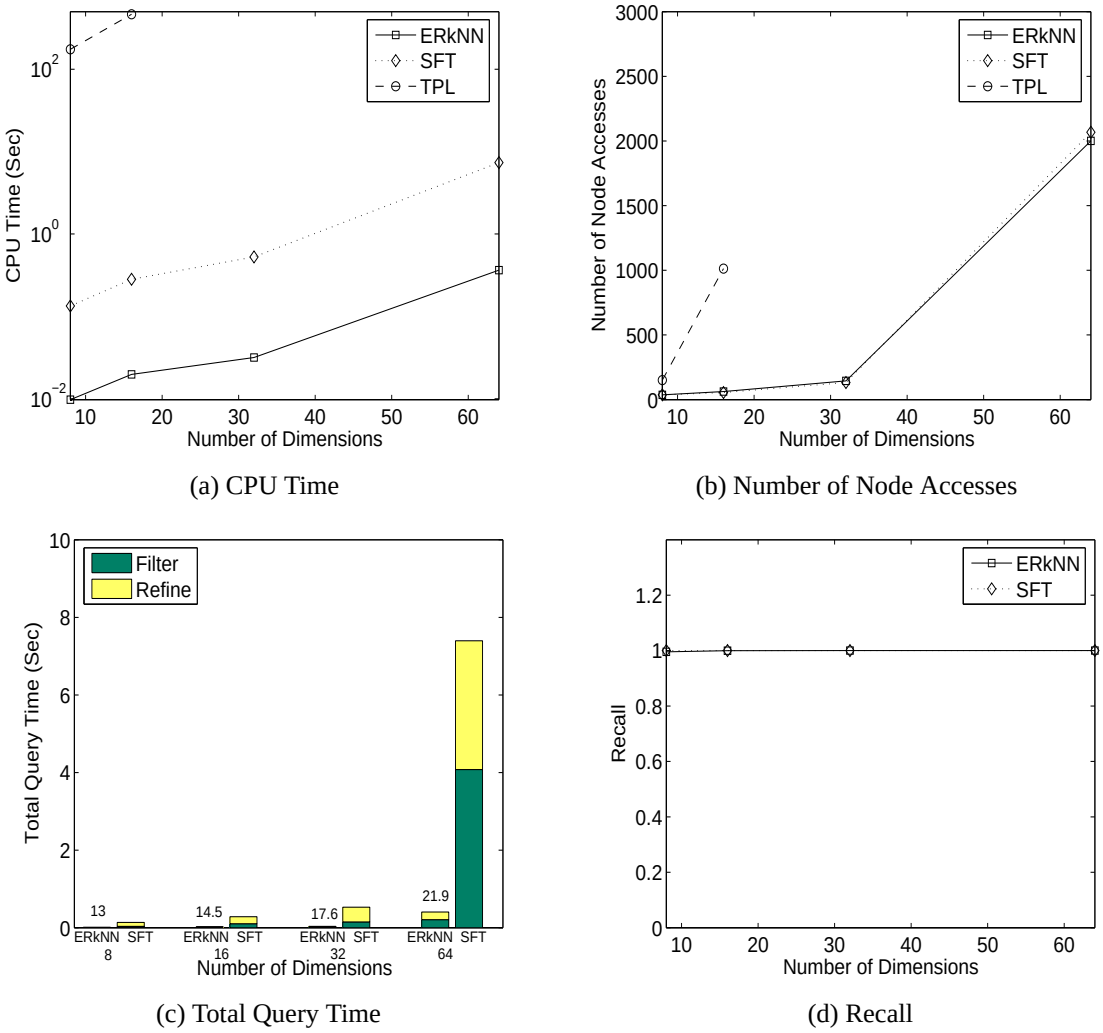
Figure 4.11: Effect of buffer size on Corel dataset

cost of TPL is expensive mainly because of its expensive pruning method in the filter procedure.

Figure 4.11 presents the study of the effect of buffer size on the RkNN query. The buffer size is improved from 10 pages to 2000 pages. The query k is 5. As we expected, the number of page accesses decreases when the buffer size is increased. However, the total query time does not change much since the CPU cost is the major cost of the RkNN query.

4.4.4 Study on Synthetic Datasets

In this section, we study ERkNN, SFT and TPL on synthetic clustered datasets of various sizes and dimensions. On all these datasets, we adjust ERkNN with a local adjustment factor 1.06 so that the recall of ERkNN is almost 1.



(a) CPU Time

(b) Number of Node Accesses

(c) Total Query Time

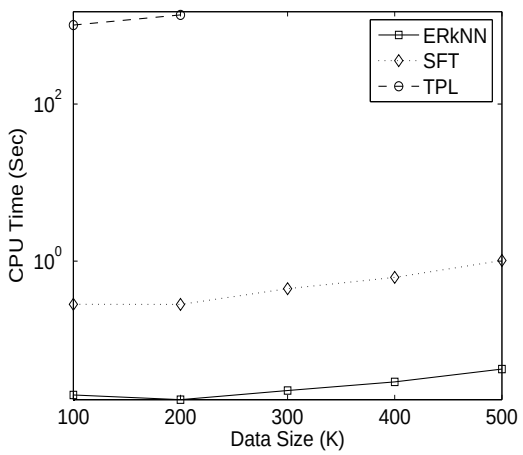
(d) Recall

Figure 4.12: Effect of Data Dimensionality (Clustered Dataset, 100K)

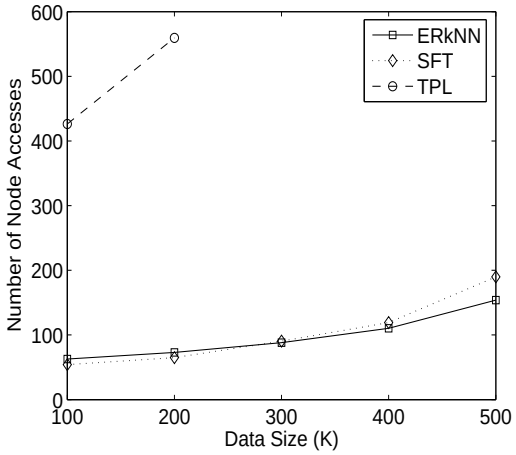
Effect of Dimensionality

First, we evaluate the effect of data dimensionality on the RkNN query by varying the number of dimensions from 8 to 64. k is equal to 10. We conduct the study on the clustered datasets. Figure 4.12 presents the results.

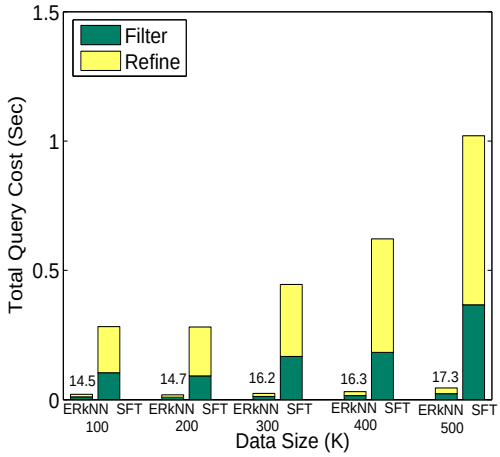
We observe that ERkNN keeps being the most efficient method. The speed-up factor of ERkNN over SFT increases from 11 to 17 when the number of dimensions increases from 8 to 64. ERkNN outperforms TPL even more significantly. In 16-dimensional



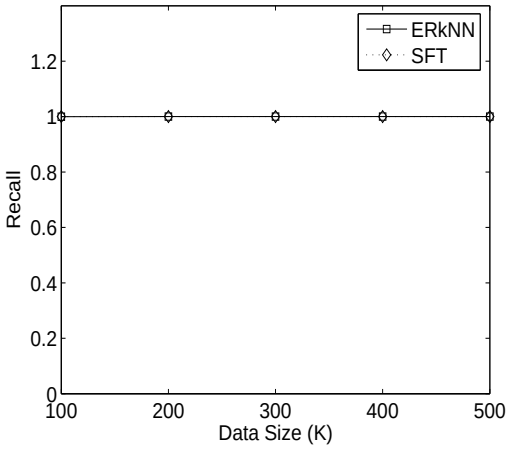
(a) CPU Time



(b) Number of Node Accesses



(c) Total Query Time



(d) Recall

Figure 4.13: Effect of Data Size (Clustered Dataset, Dim=16)

spaces, TPL and SFT take 464.36 sec and 0.28 sec to answer an RkNN query respectively. ERkNN takes only 0.021 sec. The study demonstrates that ERkNN is more scalable to RkNN queries in high-dimensional spaces than TPL and SFT.

Effect of Data Size

We examine the RkNN query performance on datasets of varying sizes. We query RkNN $k = 10$ on the clustered datasets and vary the dataset size from 100K to 500K objects.

Figure 4.13 shows the results. We observe that ERkNN outperforms SFT and TPL significantly. In terms of elapse time, the average speed-up factor of ERkNN over SFT and TPL is 16.9 and 60565.67 respectively. When we increase data size from 100K to 500K, the speed-up factor of ERkNN over SFT increases from 12.5 to 21.6. So ERkNN is more scalable to data size.

4.5 Summary

RkNN queries have important applications in many database systems. However, existing methods are expensive and not scalable to RkNN queries in high-dimensional spaces or of large values of k. In this chapter, an innovative *estimation-based* approach -ERkNN (the estimation-based RkNN search) which can efficiently handle RkNN queries in high-dimensional data spaces and for large values of k is proposed. ERkNN retrieves RkNN candidates based on the *local estimated kNN-distance* and verifies the candidates using an efficient *aggregated range query*. Two local kNN-distance estimation methods, the PDE method and the kDE method, are provided, which are proved to work effectively on both uniform and skewed datasets. Employing the effective estimation-based filtering strategy and the efficient refinement procedure, ERkNN outperforms previous methods by a significant margin. Extensive experiments demonstrate that ERkNN is efficient, scalable and outperforms pervious methods significantly.

Chapter 5

BORDER: A Data Mining Tool for Efficient Boundary Point Detection

5.1 Introduction

Advancements in information technologies have led to the continual collection and rapid accumulation of data in repositories. Knowledge discovery in databases is a non-trivial process of identifying valid, interesting and potentially valuable patterns in data [37]. Given the urgent need for efficient and effective analysis tools to discover information from these data, many techniques have been developed for knowledge discovery in databases to identify valid, interesting and potentially valuable patterns from the data. Such techniques include data classification and mining association rule, cluster and outlier analysis [52] as well as data cleaning and data preparation techniques to enhance the validity of the data by removing anomalies and artifacts.

In this chapter, we present a novel data mining tool - BORDER for effective boundary point detection. Boundary points are data points that are located at the margin of densely distributed data such as a cluster. Boundary points are useful in data mining applications because they represent a subset of population that possibly straddles two or more classes. For example, this set of points may denote a subset of population that should have developed certain diseases, but somehow they do not. Special attention is

certainly warranted for this set of people since they may reveal some interesting characteristics of the disease. The knowledge of these points is also useful for data mining tasks such as classification [67] since these points can be potentially mis-classified.

Intuitively, boundary points can be defined as follows:

Definition 5.1.1 *A boundary point p is an object that satisfies the following conditions*
i) It is within a dense region $\mathbb{R}$;
ii) $\exists$ region $\mathbb{R}'$ near p, $Density(\mathbb{R}') \gg Density(\mathbb{R})$ or $Density(\mathbb{R}') \ll Density(\mathbb{R})$.

Note that *boundary points* are different from outliers [4, 23, 6] or its statistical counterpart - the change-point [90, 25, 7]. While outliers are located in the sparsely-populated areas, *boundary points* occurs at the margin of dense regions.

We develop a method called BORDER (a BOundaRy points DEtectoR) that utilizes the special property of the reverse k-nearest neighbor (RkNN) [78], and employs the state-of-the-art database technique - the Gorder kNN join [124] to find boundary points in a dataset.

As illustrated in Figure 1.3 in Chapter 1, the points whose reverse 50-nearest neighbors are less than 30 clearly define the the boundaries of the clusters in the dataset. Utilizing this property of the reverse k-nearest neighbor in data mining tasks will require the execution of a RkNN query for each point in the dataset (the set-oriented RkNN query). However, this is very expensive and the complexity will be $O(N^3)$ since the complexity of a single RkNN query is $O(N^2)$ time using sequential scan for non-indexed data [116], where N is the cardinality of the dataset. In the case where the data is indexed by some hierarchical index structure [16], the complexity can be reduced to $O(N^2 \cdot logN)$. However, the performance of these index structures is often worse than sequential scan in high-dimensional spaces.

Instead of running multiple RkNN queries, the proposed approach utilizes Gorder kNN join [124] (or the G-ordering kNN join method) to find the reverse k-nearest neigh-

bors of a set of data points. BORDER 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 the boundary points whose RkNN number is smaller than a user predefined threshold can be output incrementally. Experimental studies show that the proposed BORDER method is able to detect boundary points effectively and efficiently. Moreover, it helps the density-based clustering method DBScan [33] to find out the correct clusters and improves the classification accuracy for various classifiers. Note that BORDER 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, this assumption may not hold for datasets which are not well-clustered and the boundary is not so clear, in which case, BORDER may fail to find the correct boundary points.

The remainder of the chapter is organized as follows.

- Section 5.2 presents the preliminary study of the relationship between the location of a point and the number of its reverse k-nearest neighbors.
- Section 5.3 describes BORDER algorithm in detail and analyzes the cost of BORDER.
- Section 5.4 presents the results of our performance study.
- Finally, Section 5.5 concludes this chapter with a summary.

5.2 Preliminary Study

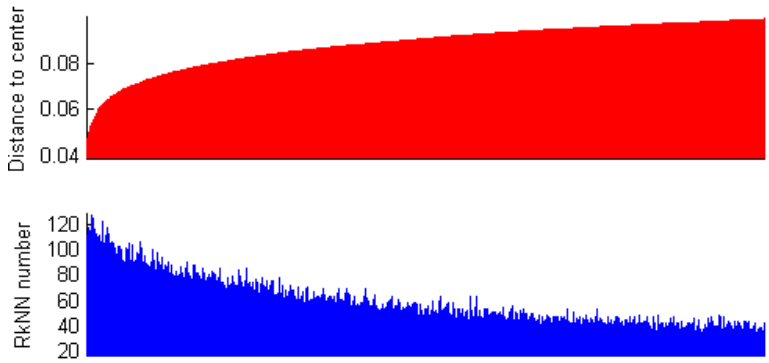
The reverse k-nearest neighbors (RkNN) of an object p are points that look upon p as one of their k-nearest neighbors. A property of reverse k-nearest neighbor is that it examines the neighborhood of an object with the view of the entire dataset instead of the object itself. Hence, it can capture the distribution property of the underlying data and allow the identification of boundary points that lie between two or more distributions.

Figure 1.3 in Chapter 1 shows the results of one of our preliminary studies. Given a 2-dimensional dataset as shown in Figure 1.3(a), we plot the points whose reverse 50-nearest neighbors answer set contain less than 30 points. Figure 1.3(b) shows that the boundaries of the clusters are clearly defined by those points having fewer number of RkNN.

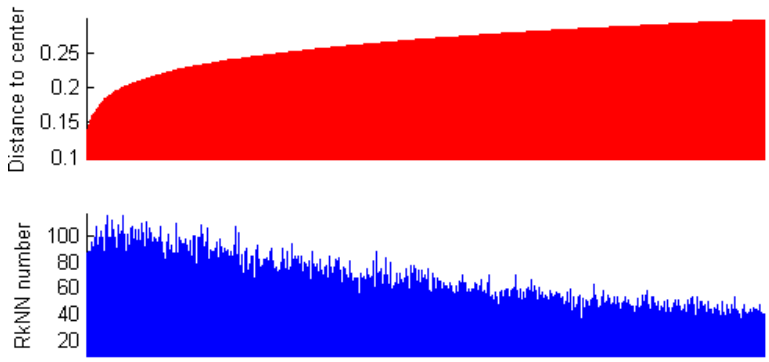
We also carry out another preliminary study to find out the relationship between the location of a point p and the number of its RkNN in high-dimensional spaces. In order to determine the boundary of a densely distributed region, we use hyper-sphere datasets [1] which contain the dense regions of the shape of the high-dimensional spheres. The boundary points of spherical regions are always located at the area farthest from the center of the sphere and so can be easily determined by calculating the distances between the data points and the centers of the hyper-spheres they belong to.

Figure 5.1 summarizes the results of the experiments on the hyper-sphere datasets of different distributions. We compute the number of reverse k-nearest neighbors of each point in the dataset and the distance of each point to the center of the cluster that the point belongs to. Then we sort the data points according to the distance of each point to the center of the cluster that the point belongs to and plot the distance to cluster center and the number of reverse k-nearest neighbors of each point as in Figure 5.1. Each vertical line in the graphs in Figure 5.1 corresponds to one data point. The height of the lines in

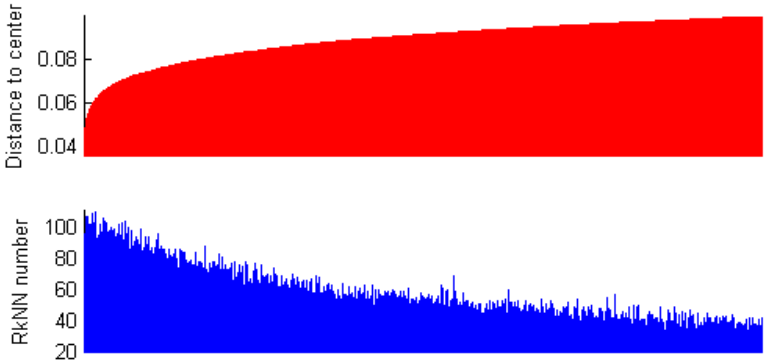
[1] The generation of hyper-sphere data is given in the experiment section.



(a) Uniform Distribution (Dimension = 8, Data Size = 6000)



(b) Normal Distribution (Dimension = 8, Data Size = 6000)



(c) Zipf Distribution (Dimension = 8, Data Size = 6000)

Figure 5.1: Preliminary Studies.

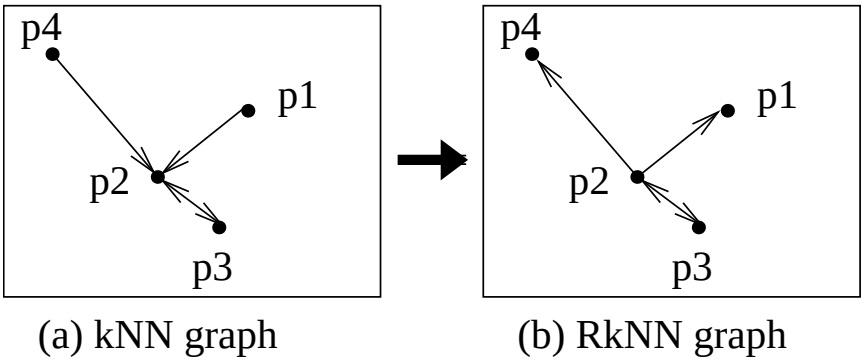


Figure 5.2: kNN graph vs. RkNN graph

the upper sub-graphs represents the distance to cluster center and the height of the lines in the lower sub-graphs is corresponding to the number of reverse k-nearest neighbors of each point. This study indicates that the number of RkNN decreases as the distance of a point from the center increases. The result confirms that for well-clustered datasets in high-dimensional spaces, the boundary points which lie at the margin of the clusters tend to have fewer reverse k-nearest neighbors.

Utilizing this property of RkNN to detect boundary points will require the execution of a RkNN query for each point in the dataset (the set-oriented RkNN query). However, this is very expensive and the complexity will be $O(N^3)$ using sequential scan for non-indexed data or $O(N^2 \cdot logN)$ for indexed data, where N is the cardinality of the dataset. BORDER overcomes this difficult by transforming the set-oriented RkNN query into the set-oriented kNN query (i.e., the kNN join) by utilizing the *reversal-ship* between the k-nearest neighbor and the reverse k-nearest neighbor, that is, if p_i is one of p_j's k-nearest neighbors, then p_j is one of p_i's reverse k-nearest neighbors.

Lemma 5.2.1 *The reverse k-nearest neighbors of all points in dataset R can be derived from the k-nearest neighbors of all points in R. By reversing all pairs (p_i, p_j) produced by the self-kNN join of R, we obtain the complete set of pairs (p_j, p_i) that p_i is p_j's reverse k-nearest neighbor.*

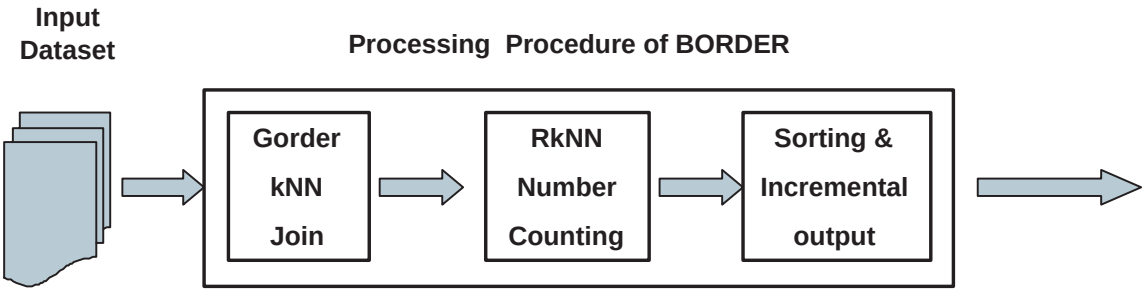


Figure 5.3: Overview of BORDER

Figure 5.2 illustrates the kNN and RkNN relationship with an edge $\overrightarrow{p_ip_j}$. Figure 5.2(a) is the kNN graph and each edge $\overrightarrow{p_ip_j}$ denotes a kNN pair (p_i, p_j) such that p_j is p_i's kNN. Figure 5.2(b) is the RkNN graph and each edge $\overrightarrow{p_ip_j}$ denotes a RkNN pair (p_i, p_j) such that p_j is p_i's RkNN. Given the kNN of all points in a dataset, we can derive the RkNN of each point by simply reversing the direction of the edges in the kNN graph. Hence, we have the lemma.

5.3 BORDER

Figure 5.3 gives an overview of BORDER. It comprises of three main steps:

1. A kNN-join operation with Gorder to find the k-nearest neighbors for each point in the dataset.

2. An RkNN counter to obtain each point's RkNN number (the cardinality of each point's RkNN answer set).

3. Points are sorted according to their RkNN number. Points that its RkNN number is smaller than a user defined threshold are output incrementally as boundary points.

In the following sections, we will give the details of each step.

Algorithm 9 Gorder_Self_kNN(R)

Input:

R is input dataset.

Description:

1: G_Ordering R;
2: Join_Grid_Ordered_Data(R, R);
3: Output kNN pairs into the kNN-file;

Algorithm 10 RkNN_Counter(R, kNN-file)

Input:

R: the input dataset; kNN-file: a file records k-nearest neighbors of each points in R.

Description:

1: **for each** point $p \in R$ **do**
2: Read its k-nearest neighbors $kNN(p, R)$ from kNN-file;
3: **for each** point $p_i \in kNN(p, R)$ **do**
4: increase $rnum_{p_i}$ by 1;

5.3.1 kNN Join

Based on above discussion, the first step of BORDER performs a self kNN join of the input dataset R to compute all the k-nearest neighbors pairs of R. It makes use of the up-to-date kNN join algorithm - Gorder, which is an optimized block nested loop join with efficient data scheduling and distance computation filtering and outperforms previous works significantly.

Algorithm 9 presents Gorder self kNN join algorithm which regards the input dataset R as both the query dataset and the point dataset.

In Line 1 of algorithm 9, dataset R is sorted into the G-order as we introduced in Chapter 3. Line 2 calls the scheduled block nested loop join (see Section 3.3.2 for the detail of the algorithm). It takes dataset R as both the query dataset and the point dataset. At the end, the k-nearest neighbors of all points in R are found and saved in the kNN-file (Line 3).

Algorithm 11 Sort_and_Output(R, $k_{threshold}$)

Input:

R: the input dataset;

Description:

```
1: Sort points in R in ascending order according to their RkNN number;
2: for Points p_i in R do
3:     if rnum_{p_i} < k_threshold then
4:         Output p_i;
```

5.3.2 RkNN Counter

In this step, BORDER counts the number of reverse k-nearest neighbors (RkNN number) for each point p (denoted as $rnum_p$) utilizing the kNN information saved in the kNN-file. According to the reversal-ship between kNN and RkNN which we have discussed in Lemma 5.2.1, the number of each point's k-nearest neighbor can be obtained by a scanning of the kNN-file and for each point p_i in the kNN set of a point p, increasing $rnum_{p_i}$ by 1.

Algorithm 10 depicts the count procedure.

5.3.3 Sorting and Output

Data points then can be sorted according to their RkNN number so that they can be output incrementally. We let $k_{threshold}$ be a user defined threshold which is tunable. For all point p, if its RkNN number $rnum_p < k_{threshold}$, they are output as detected boundary points. Algorithm 11 shows the details.

5.3.4 Cost Analysis

Next, we analyze the I/O and CPU cost of BORDER.

The major cost of BORDER lies in the kNN join procedure. The number of I/O

reading incurred during the kNN join procedure in terms of the number of page is [124]:

$$3N_r + 2N_r \left(\left\lceil log_{B-1} \frac{N_r}{B} \right\rceil + 1 \right) + \frac{N_r}{n_r} \cdot N_r \cdot \gamma_1$$

where N_r is the total number of R data pages, n_r is the allocated buffer pages for query data, and B is the total buffer pages available in memory.

Since the kNN-file is written on the disk and scanned during the step of RkNN counter. There are additional N_{knn} pages I/O writing and N_{knn} I/O reading, where N_{knn} is the size of the kNN-file.

Hence, the total number of pages of I/O reading is:

$$\left(3N_r + 2N_r \left(\left\lceil log_{B-1} \frac{N_r}{B} \right\rceil + 1 \right) + \frac{N_r}{n_r} \cdot N_r \cdot \gamma_1 + N_{knn} \right)$$

And the total number of pages of I/O writing is N_{knn}.

The major CPU cost of BORDER is the distance computation in the kNN join phase. The number of distance computations is:

$$P_r^2 \cdot \gamma_2$$

where P_r is the number of objects in the dataset, γ_2 is the selectivity of distance computation.

The selectivity ratio γ_1 and γ_2 are estimated as following [124]:

$$\gamma_1 = \sum_{l=0}^{d} \begin{pmatrix} d \\ l \end{pmatrix} \left(\sqrt[d]{\frac{p_q}{P_r}} + \sqrt[d]{\frac{p_p}{P_r}} \right)^l \cdot V_{sphere}^{d-l}(\varepsilon) \tag{5.1}$$

where, p_q and p_p are the numbers of points in the query buffer of size n_r buffer pages and in the point buffer of size n_s buffer pages.

$$p_q = \frac{n_r \cdot \text{page size}}{\text{size of data vector}} \text{ and } p_p = \frac{n_s \cdot \text{page size}}{\text{size of data vector}}$$

$$\gamma_2 = \sum_{k=0}^{d} \begin{pmatrix} d \\ l \end{pmatrix} \left(\sqrt[d]{\frac{p'_q}{P_r}} + \sqrt[d]{\frac{p'_p}{P_r}} \right)^l \cdot V_{sphere}^{d-l}(\varepsilon) \tag{5.2}$$

Where p'_q and p'_p are the numbers of points in the sub-block of the query buffer and the point buffer.

$$V_{sphere}^{d-l}(\varepsilon) = \frac{\sqrt{\pi^{d-l}}}{\Gamma\left(\frac{d-l}{2}+1\right)} \cdot \varepsilon^{d-l} \tag{5.3}$$

$$\varepsilon = \sqrt[d]{\frac{k \cdot \Gamma(d/2+1)}{P_r}} \cdot \frac{1}{\sqrt{\pi}} \tag{5.4}$$

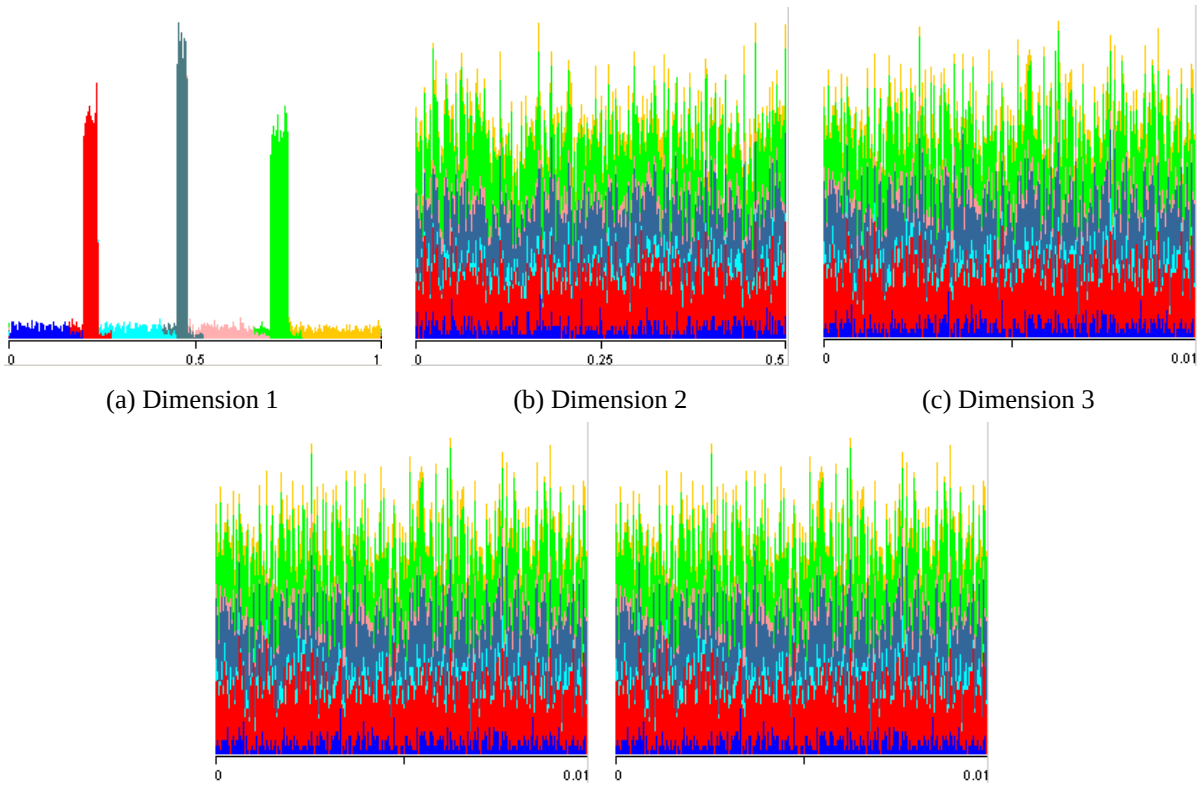
$$\Gamma(x+1) = x\Gamma(x), \;\; \Gamma(1) = 1, \;\; \Gamma(1/2) = \sqrt{\pi}$$

5.4 Performance Study

We conducted extensive experimental study to evaluate the performance of BORDER and present the results in this section. We implemented BORDER in C++ and studied its effectiveness as follows:

1. Dataset I. A set of high-dimensional hyper-sphere datasets with various data distributions and sizes.

   These hyper-sphere datasets are used to demonstrate the ability of BORDER in detecting boundary points in high-dimensional spaces. The hyper-sphere datasets are generated as follows: Given the distribution, the number and the centers and



(a) Dimension 1 (b) Dimension 2 (c) Dimension 3

(d) Dimension 4 (e) Dimension 5

Figure 5.4: Data distribution of Dataset IV on each dimension.

the radii of the hyper-spheres, data points are generated according to the specified distribution. Points that are within the defined hyper-spheres are inserted into the dataset, whereas points that are outside of the hyper-spheres are discarded. To show the location of the found boundary points, we capture and present the distribution of the $\ell(p, c)$, where p is a detected point, c is the center of the hyper-sphere which p belongs to, and $\ell(p, c)$ is the distance between p and c.

2. Dataset II. A set of 2-dimensional clustered dataset of arbitrary cluster shapes.

   This set of datasets aims to exhibit the ability of BORDER to find out boundary points located at the border of *arbitrary-shaped* clusters *visually*. We use the 2-dimensional datasets so that we can plot the detected boundary points in a plane to

show the effectiveness of BORDER.

3. Dataset III: A clustered dataset with mixed clusters.

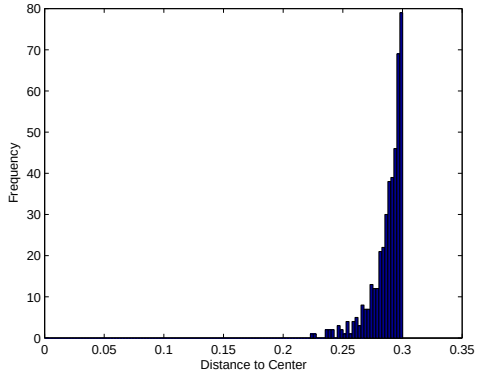
   In this dataset, the dense clusters mix with some less dense clusters. Traditional density-based clustering method such as DBScan cannot identify the clusters properly in this type of datasets. We show that removing the boundary points will help DBScan to find the clusters correctly. The removed boundary points can be inserted back into the identified clusters with a post-processing procedure by checking their connectivity and density.

4. Dataset IV. Labelled datasets for classification.

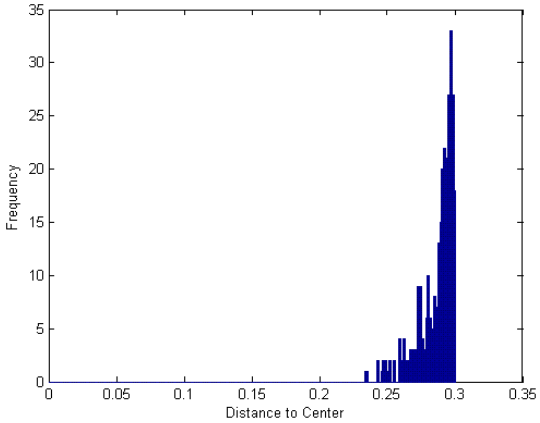
   This synthetic dataset contains 7 classes with 5 attributes. The dataset is generated as follows: We divide the first dimension into 7 segments. Each segment corresponds to one class. We assign points within each segment to its corresponding class and those points lying at the adjacent region of each segment to different classes. Thus, the 7 classes do not have a distinct separable boundaries. Data points are distributed randomly in the rest of the dimensions. Figure 5.4 shows the data distribution of the dataset on each dimension. Note that in dimension 1, points that are located at the boundary region of two adjacent classes belong to different classes. We use this dataset to show that by removing the boundary points which straddle two classes of difference density can improve the accuracy of classifiers.

5.4.1 On Hyper-sphere Datasets

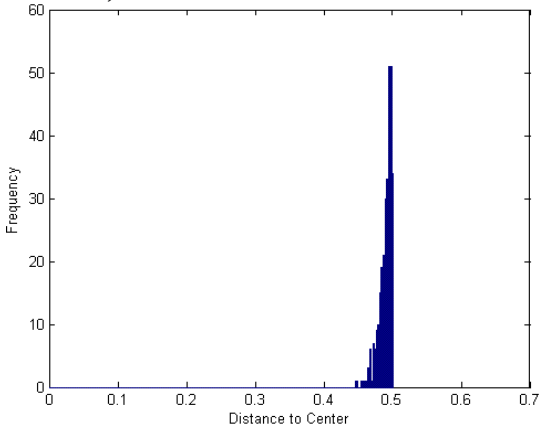
We first study the effectiveness of BORDER on the hyper-sphere datasets of various distributions, different numbers of dimension and containing different number of clusters. Figure 5.5 summarizes the experiment results. We incrementally output 300 points with



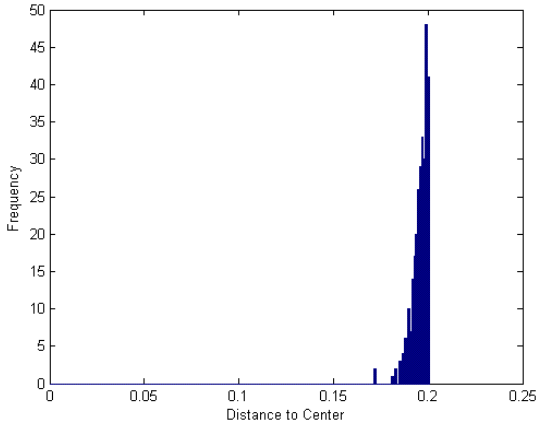
(a) Normal distribution, dimension=8, number of clusters=1, radius=0.3



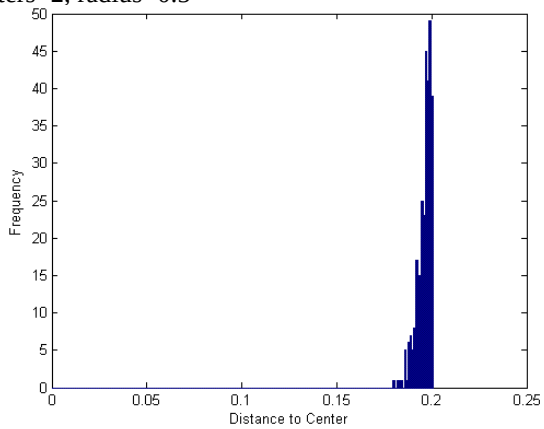
(b) Normal distribution, dimension=10, number of clusters=5, radius=0.3



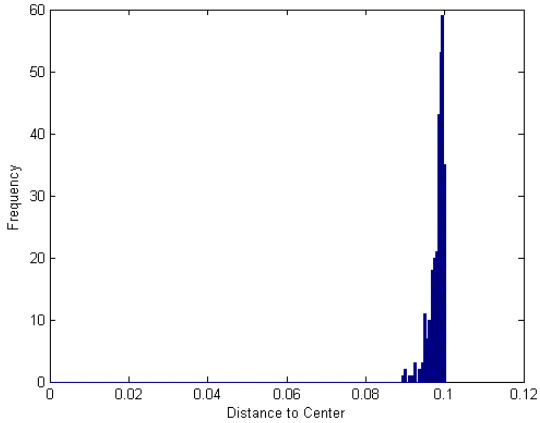
(c) Zipf distribution, dimension=4, number of clusters=2, radius=0.5



(d) Zipf distribution, dimension=6, number of clusters=3, radius=0.2



(e) Uniform distribution, dimension=8, number of clusters=2, radius=0.3



(f) Uniform distribution, dimension=12, number of clusters=2, radius=0.1

Figure 5.5: Study on hyper-sphere datasets.

Distribution	Cluster Number	Size	Radius	Dimension	k	Processing Time (Sec)
Normal	1	6000	0.3	8	50	6.547
Normal	5	6000	0.3	10	50	4.313
Zipf	3	6000	0.2	6	50	4.625
Zipf	2	6000	0.5	4	50	2.89
Uniform	2	6000	0.2	8	50	3.938
Uniform	2	6000	0.1	12	50	5.875

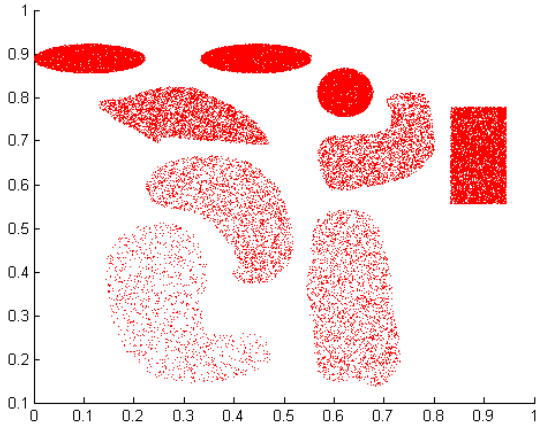
Table 5.1: Hyper-sphere datasets and processing time.

Distribution	Dimension	Radius	Mean	Standard Deviation
Normal	8	0.3	0.2868	0.0141
Normal	10	0.3	0.2868	0.0133
Zipf	6	0.2	0.1955	0.0043
Zipf	4	0.5	0.49	0.0092
Uniform	8	0.2	0.1959	0.0037
Uniform	12	0.1	0.0981	0.0019

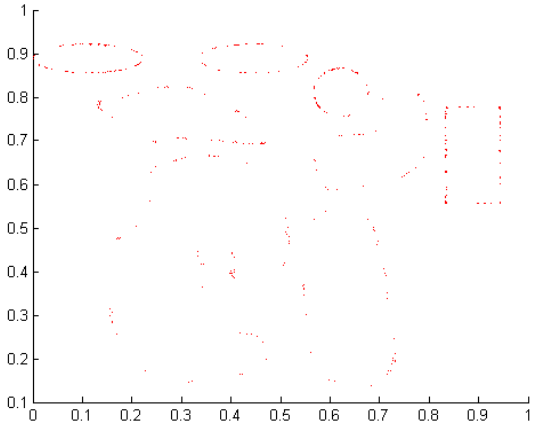
Table 5.2: Mean and standard deviation of the distance of detected boundary points to the hyper-sphere center.

lowest RkNN number of as boundary points. For each graph, the x-axis is the point p's distance to the center of the hyper-sphere that p belongs to ($\ell(p, c)$) and the y-axis is the frequency. We observe that for all datasets, $\ell(p, c)$ of the output points by BORDER is equal or close to the radii of the hyper-spheres.

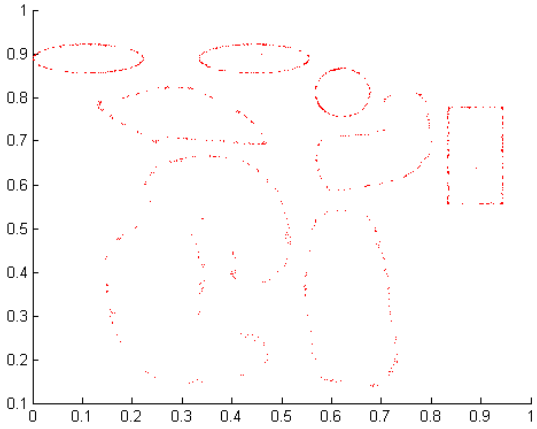
Table 5.2 summarizes the mean and standard deviation of the distance of detected boundary points to the hyper-sphere center. Both the plotting and statistical information of the output of BORDER indicate that these points are indeed the boundary points of the hyper-spherical-shaped clusters. The properties and the processing time of BORDER on the hyper-sphere datasets are presented in Table 5.1.



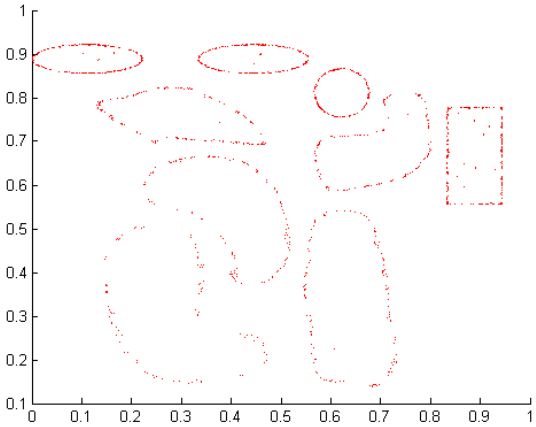
(a) Original Dataset



(b) Detected Boundary Points ($k_{threshold}$=30)

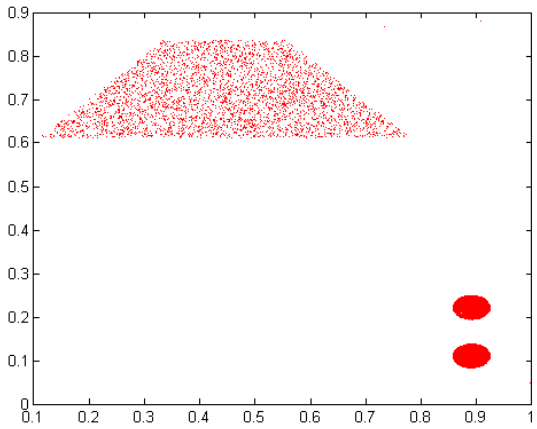


(c) Detected Boundary Points ($k_{threshold}$=33)

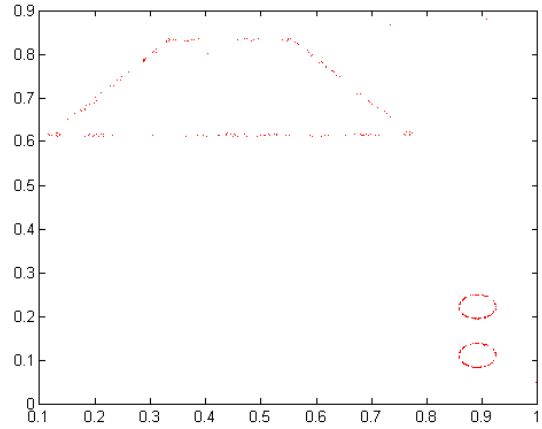


(d) Detected Boundary Points ($k_{threshold}$=35)

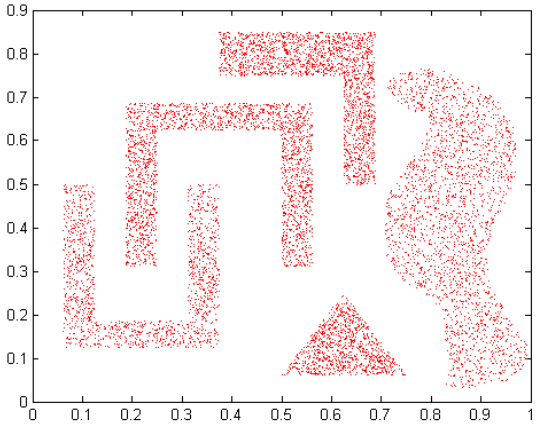
Figure 5.6: Incremental output of detected boundary points of dataset 1.



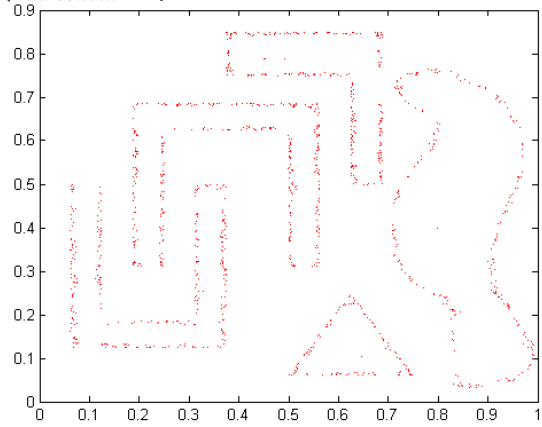
(a) Dataset 2



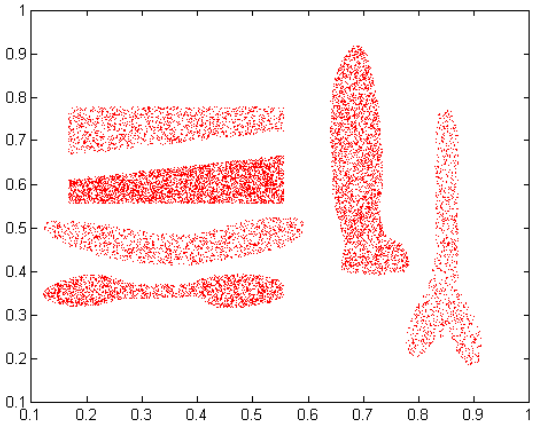
(b) Detected Boundary Points of Dataset 2 ($k_{threshold}$=35)



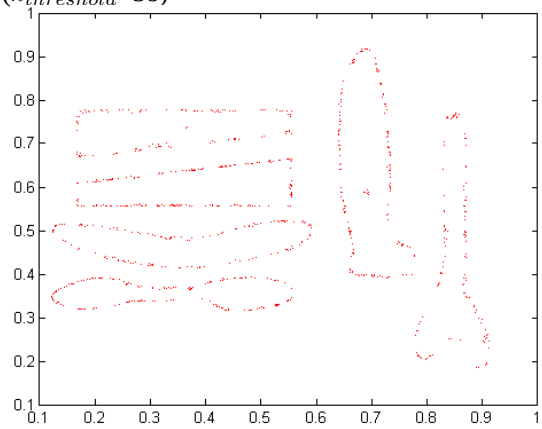
(c) Dataset 3



(d) Detected Boundary Points of Dataset 3 ($k_{threshold}$=38)



(e) Dataset 4



(f) Detected Boundary Points of Dataset 4 ($k_{threshold}$=35)

Figure 5.7: Study on other datasets.

	Size	Dimension	k	Processing Time (Sec)
dataset 1	40000	2	50	9.985
dataset 2	10680	2	50	2.781
dataset 3	9050	2	50	2.562
dataset 4	12950	2	50	3.469

Table 5.3: Datasets (with clusters) and processing time.

5.4.2 On Arbitrary-shaped Clustered Datasets

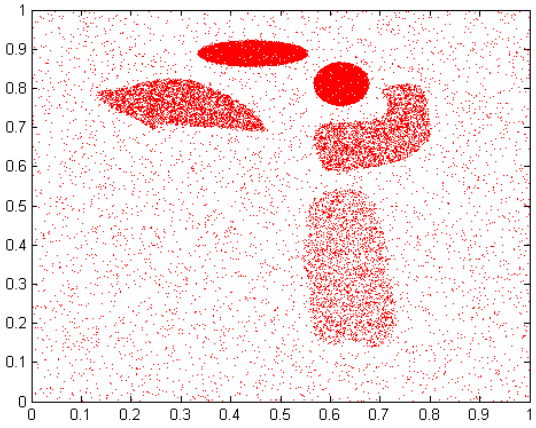
Next, we study the effectiveness of BORDER on datasets containing arbitrary-shaped dense regions. Although BORDER is also applicable to high-dimensional spaces, the experiments are carried out on 2-dimensional cluster datasets in order to visualize the results.

Figure 5.6 demonstrates the incremental output of BORDER executed on dataset 1. The thresholds are set as 30, 33 and 35. The graphs in Figure 5.6 shows that the plotted points outline the boundaries of the clusters in dataset clearly. Note that with the incremental output, we can stop whenever we are satisfied with the quality of detected boundary points. Figure 5.7 shows the results of boundary point detection on datasets 2, 3, 4. It is clear that BORDER can find the boundary points effectively. The processing time of BORDER is summarized in Table 5.3.

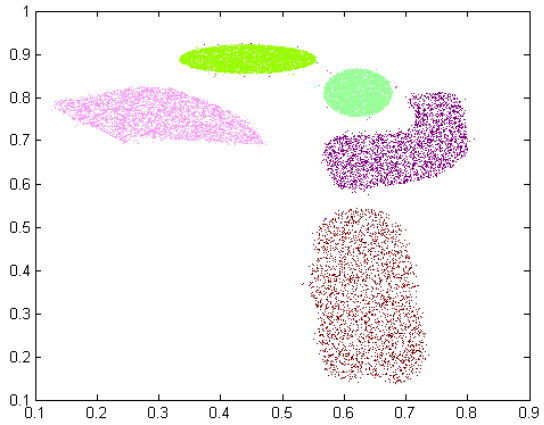
5.4.3 On Mixed Clustered Dataset

In this set of experiments, we mix the dense clusters with less dense clusters and study the ability of using BORDER for preprocessing data for clustering.

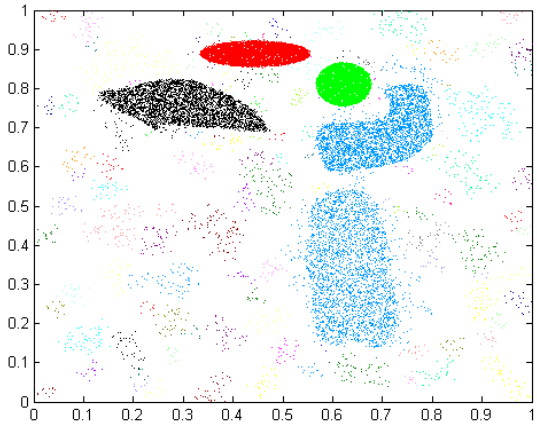
Figure 5.8 shows that it is difficult for DBScan [33] to identify the correct clusters in this type of datasets. Figure 5.8 (b), (c) and (d) plot the clusters detected by DBScan with different colors. We observe that if we set the density requirement of DBScan high, that is, Eps=0.00967926, MinPts=10, points in the sparse cluster are all regarded as outliers



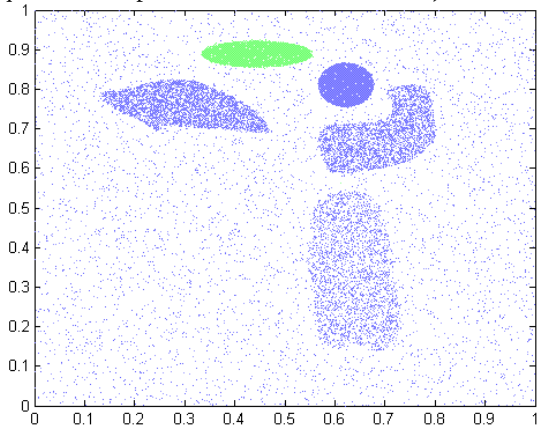
(a) Original Dataset



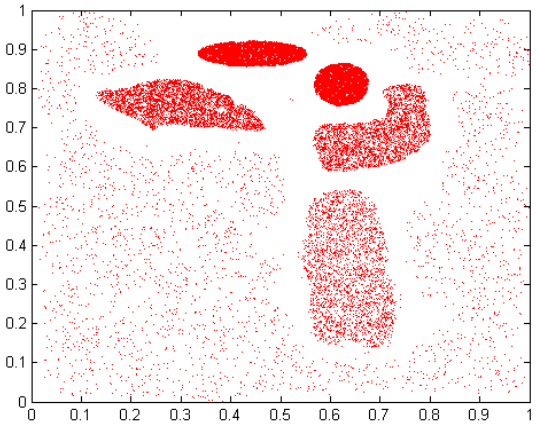
(b) Clusters detected by DBScan (high density requirement, Eps=0.00967926, MinPts=10)



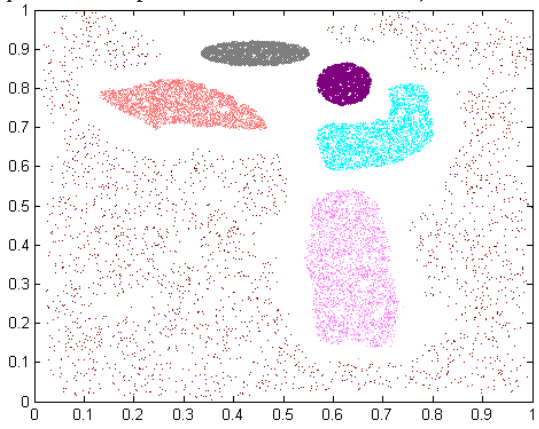
(c) Clusters detected by DBScan (low density requirement, Eps=0.0223533, MinPts=10)



(d) Clusters detected by DBScan (low density requirement, Eps=0.0469042, MinPts=10)



(e) After boundary points are removed



(f) Clusters detected by DBScan after boundary points are removed(Eps=0.0469042, MinPts=10)

Figure 5.8: Study on mixed clustered datasets.

(Figure 5.8(b)). If we set the density requirement of DBScan low, that is, Eps=0.0223533 and MinPts=10 (Figure 5.8(c)) or Eps=0.0469042, MinPts=10 (Figure 5.8(d)), DBScan returns clusters that mix dense and sparse regions.

Figure 5.8 (e) shows the dataset after we remove the boundary points ($k_{threshold} = 40$). Figure 5.8(f) shows the result of DBScan working on the dataset after the boundary points are removed. DBScan with parameters Eps=0.0469042 and MinPts=10 can easily identify the dense clusters as well as the sparse clusters correctly because they are now well separated. The removed boundary points can be inserted into the clusters with a post-processing procedure which examines the density of the points and their connectivity with the clusters.

5.4.4 On the Labelled Dataset for Classification

Finally, we conduct experiments on the labelled dataset for classification. We test various classification methods provided by Weka [65] and compare the classification accuracy before and after we remove the detected boundary points. The test accuracy is evaluated by 10-fold cross validation. The results show that removing the boundary points reduces the ratio of misclassified data points and improves the classification accuracy effectively.

Table 5.4 and Table 5.5 summarizes the results when we define different thresholds for the RkNN number. When we set the $k_{threshold}$ 25 or 30, the average improvement ratios in terms of incorrectly classified ratio are 20.03% and 43.51% respectively and the average improvement ratios in terms of incorrectly classified instance are 22.07% and 46.60% respectively.

	Before		After ($k_{threshold}$=25)		improvement	
Classification Method	Incorrectly Classified Ratio	Incorrectly Classified Instances	Incorrectly Classified Ratio	Incorrectly Classified Instances	Incorrectly Classified Ratio	Incorrectly Classified Instances
Decision Table	3.20%	391	2.57%	306	19.69%	21.74%
OneR	3.26%	398	2.21%	263	32.21%	33.92%
Nnge	3.25%	397	2.12%	252	34.77%	36.52%
Jrip	3.43%	418	2.36%	280	31.20%	33.01%
AdaBoost M1	19.14%	2335	18.13%	2156	5.28%	7.67%
MultiBoost AB	19.14%	2335	18.13%	2156	5.28%	7.67%
Raced Incremental Logit Boost	3.20%	391	2.53%	301	20.94%	23.02%
IB1	15.48%	1888	14.02%	1667	9.43%	11.71%
Naive Bayes Simple	3.48%	425	2.57%	306	26.15%	28.00%
SMO	5.93%	723	5.02%	597	15.35%	17.43%
Average	7.95%	970.1	6.97%	828.4	20.03%	22.07%

Table 5.4: Comparison of classification accuracy ($k_{threshold}$=25).

5.5 Conclusion

In this chapter, we introduce a novel data mining tool - BORDER for effective boundary point detection. The knowledge of boundary points can help in data mining tasks such as data preparation for clustering and classification. BORDER detects boundary points according to the finding that data points lying at the margin of densely distributed data tend to have much fewer reverse k-nearest neighbors. It transforms the expensive set-oriented RkNN query into the kNN join by utilizing the *reversal-ship* between the k-nearest neighbor relationship and the reverse k-nearest neighbor relationship and employs the state-of-the-art kNN join technique - Gorder. Experimental study demonstrates that BORDER is able to detect boundary points effectively and efficiently. Moreover, by

	Before		After ($k_{threshold}$=30)		improvement	
Classification Method	Incorrectly Classified Ratio	Incorrectly Classified Instances	Incorrectly Classified Ratio	Incorrectly Classified Instances	Incorrectly Classified Ratio	Incorrectly Classified Instances
Decision Table	3.20%	391	1.37%	158	57.19%	59.59%
OneR	3.26%	398	1.34%	155	58.90%	61.06%
Nnge	3.25%	397	1.26%	145	61.23%	63.48%
Jrip	3.43%	418	1.40%	162	59.18%	61.24%
AdaBoost M1	19.14%	2335	16.90%	1950	11.70%	16.49%
MultiBoost AB	19.14%	2335	16.90%	1950	11.70%	16.49%
Raced Incremental Logit Boost	3.20%	391	1.39%	160	56.56%	59.08%
IB1	15.48%	1888	12.44%	1435	19.64%	23.99%
Naive Bayes Simple	3.48%	425	1.36%	157	60.92%	63.06%
SMO	5.93%	723	3.67%	423	38.11%	41.49%
Average	7.95%	970.1	5.80%	669.5	43.51%	46.60%

Table 5.5: Comparison of classification accuracy ($k_{threshold}$=30).

applying it as part of data pre-processing step, we observe an improvement in the clustering quality of DBScan [33], as well as an overall increase in classification accuracies of various classifiers.

Chapter 6

Conclusion

In this chapter, we summarize the contributions of this thesis and discuss future work on the advanced similarity queries and their application in data mining.

6.1 Thesis Contributions

This thesis studied advanced similarity queries and their application in knowledge discovering and data mining. Two advanced similarity queries - the k-Nearest Neighbor join (kNN join) and the Reverse k-Nearest Neighbor query (RkNN query) have been studied and efficient algorithms for their processing are proposed. Furthermore, we investigated how to utilize these queries in data mining. A novel data mining tool - BORDER which is built upon the kNN join and utilizes a property of the reverse k-nearest neighbor was proposed. The contributions are detailed as follow:

- We designed an efficient algorithm Gorder (the *G-ordering* kNN join method) for the kNN join. Gorder is a block nested loop join method which achieves its efficiency by sorting data into the *G-order* that enables effective join pruning, data blocks scheduling and distance computation filtering and reduction. It utilizes a *two-tier partitioning strategy* to optimize I/O and CPU time separately and reduces

distance computational cost by pruning redundant computation based the distance of fewer dimensions. It does not require an index for the source datasets and is efficient and scalable with regard to both the dimensionality and the size of the input datasets. Experimental study shows that Gorder outperforms previous solutions with great margin.

- For the RkNN query, we proposed an innovative solution - ERkNN (the estimation-based RkNN search). ERkNN retrieves RkNN candidates based on the *local kNN-distance estimation* methods and verifies the candidates using the efficient *aggregated range query*. Two local kNN-distance estimation methods, the PDE method and the kDE method, are provided and both work effectively on uniform as well as skewed datasets. By employing the effective estimation-based filtering strategy and the efficient refinement procedure, ERkNN outperforms previous methods significantly and answers RkNN queries in high-dimensional data spaces and of large values of k efficiently and effectively.

- At last, we designed BORDER (a BOundaRy points DEtectoR) a novel data mining tool for effective boundary point detection. BORDER detects boundary points according to the finding that data points that are located at the margin of densely distributed data tend to have much fewer reverse k-nearest neighbors. It transforms the expensive set-oriented RkNN query into the kNN join by utilizing the *reversalship* between the k-nearest neighbor relationship and the reverse k-nearest neighbor relationship and employs the state-of-the-art kNN join technique - Gorder. Experimental study demonstrates BORDER detects boundary points effectively and can be used to improve the performance of clustering and classification analysis considerately.

6.2 Future Works

In this section, we suggest possible future research directions based on the work reported in this thesis.

6.2.1 Microarray Data

Microarray data are gene expressions of thousands of genes produced by DNA microarray analysis. Biologists have found that genes of similar function yield similar expression patterns in microarray data. Therefore, the computational analysis of microarray data provides accurate means for extracting biological significance and using the data to assign functions to genes. Currently microarray data are usually analyzed with the basic similarity queries or data mining functions such as classification or clustering. A property of microarray data is that they are of extremely high dimension (usually are of more than thousands features), which creates much challenge to the query processing.

Our future works include: 1) Apply the RkNN query and BORDER to the microarray data and study their outputs with biologists to find whether the RkNN query and BORDER can be addition analysis tools. 2) Design efficient algorithms for advanced similarity queries in extremely high-dimensional spaces. 3) Integrated kNN join into currently-used data mining tools which involve set-oriented kNN search.

6.2.2 Sequential Data

Sequential data include genome or protein sequences, text and times series data. There are many studies on the similarity search and data mining of sequential data. Examples include frequent pattern discovery of genome sequences, classification and clustering of protein sequences, pattern discovery and rule extraction in time series. The similarity of sequential data is usually measured by edit distance, a distance metric which is much

more expensive than common L_p distance metric.

Apart from applying the RkNN query and BORDER to the sequential data (particularly the genome and protein sequences) and analyze the results, we are interested in designing efficient advanced similarity query algorithms involving expensive distance metrics such as edit distance.

6.2.3 Stream Data

In applications such as network monitoring, telecommunications data management, web personalization, manufacturing, sensor networks, data come in continuously in multiple, rapid, time-varying, and unpredictable *streams*. Queries on stream data are usually time sensitive and allow high-quality approximate answers. In the recent years, many proposals have been made to improve the traditional data management and query processing technologies so that they can handle the infinite and continuous stream data efficiently. Our future work is to design algorithms for the kNN join and the RkNN query which could produce high-quality approximate answers efficiently for data streams.

In addition, we are also interested in being able to integrate the advance similarity query algorithms into existing data management systems cost effectively.

Bibliography

[1] www.georgetown.edu/uis/ia/dw/GLOSSARY0816.html.

[2] *Engineering Statistics Handbook*. http://www.itl.nist.gov/div898/handbook/eda/section3/eda3664.htm.

[3] http://kdd.ics.uci.edu/.

[4] C. C. Aggarwal and P. S. Yu. Outlier detection for high dimensional data. In *Proc. of SIGMOD*, 2001.

[5] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *Proc. 4th Int. Conf. of Foundations of Data Organization and Algo-rithms*, pages 69–84" YEAR=1993, ADDRESS=San Diego, CA, USA.

[6] V. Barnett and T. Lewis. *Outliers in Statistical Data*. John Wiley and Sons, 1994.

[7] M. Basseville and I.V. Nikiforov. *Detection of abrupt changes*. P T R Prenstice Hall, 1993.

[8] D. A. Beckley, M. W. Evens, and V. K. Raman. An experiment with balanced and unbalanced k-d trees for associative retrieval. In *Proc. 9th International Conference on Computer Software and Applications*, pages 256–262. 1985.

[9] D. A. Beckley, M. W. Evens, and V. K. Raman. Multikey retrieval from k-d trees and quad trees. In *Proc. 1985 ACM SIGMOD International Conference on Management of Data*, pages 291–301. 1985.

[10] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331. 1990.

[11] S. Berchtold, C. Bőhm, and H-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *Proc. 1998 ACM SIGMOD International Conference on Management of Data*, pages 142–153. 1998.

[12] S. Berchtold, D.A. Keim, and H.P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. 22nd International Conference on Very Large Data Bases*, pages 28–37. 1996.

[13] E. Bertino. A survey of indexing techniques for object-oriented databases. In *Proc. Dagsthul Seminar on Query Processing in Object-Oriented, Complex-Object and Nested Relational Databases*, pages 383–418. 1993.

[14] H. Blanken, A. Ijbema, P. Meek, and B. Akker. The generalized grid file: Description and performance aspects. In *Proc. 6th International Conference on Data Engineering*, pages 380–388. 1990.

[15] C. Böhm. A cost model for query processing in high dimensional data spaces. *ACM TODS*, 25(2):129–178, 2000.

[16] C. Böhm, S. Berchtold, and D.A. Keim. Searching in high dimensional spaces: index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[17] C. Böhm, B. Braunmueller, F. Krebs, and H.-P. Kriegel. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In *Proc. of ACM SIGMOD*, pages 379 – 388, 2001.

[18] C. Böhm and F. Krebs. High performance data mining using the nearest neighbor join. In *ICDM*, pages 43–50, 2002.

[19] C. Bohm and F. Krebs. Supporting kdd applications by the k-nearest neighbor join. In *Proc. of DEXA*, pages 504–516, 2003.

[20] C. Böhm and F. Krebs. The *k*-nearest neighbour join: Turbo charging the kdd process. *Knowledge and Information Systems*, 6(6):728–749, 2004.

[21] C. Böhm and H.-P. Kriegel. A cost model and index architecture for the similarity join. In *Proc. of ICDE*, pages 411–420, 2001.

[22] M. M. Breunig, H.-P. Kriegel, P. Kröger, and J. Sander. Data bubbles: quality preserving performance boosting for hierarchical clustering. *SIGMOD Record.*, 30(2):79–90, 2001.

[23] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. LOF: identifying density-based local outliers. In *Proc. of SIGMOD*, pages 93–104, 2000.

[24] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *Proc. of ACM SIGMOD*, pages 237–246, 1993.

[25] B.E. Brodsky and B.S. Darkhovsky. *Nonparametric methods in change-point problems*. Kluwer Academic Publishers, 1993.

[26] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: a new approach to indexing high dimensional spaces. In *Proc. 26th International Conference on Very Large Databases*, pages 89–100, 2000.

[27] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: a new approach to indexing high dimensional spaces. In *Proc. of VLDB*, pages 89–100, 2000.

[28] L. Chung, J. Gray, B. Worthington, and R. Horst. *Windows 2000 Disk IO Performance*. http://research.microsoft.com/ research/pubs/.

[29] P. Ciaccia, M. Patella, and P. Zezula. M-trees: An efficient access method for similarity search in metric space. In *Proc. 23rd International Conference on Very Large Data Bases*, pages 426–435. 1997.

[30] B. Cui, B. C. Ooi, J. Su, and K.-L. Tan. Contorting high dimensional data for efficient main memory knn processing. In *Proc. of ACM SIGMOD*, pages 479–490, 2003.

[31] J.P. Dirtrich and B. Seeger. Gess: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In *Proc. of ACM SIGKDD*, pages 47–56, 2001.

[32] J.-P. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *Proceedings of the 16th International Conference on Data Engineering*, pages 535–546, Washington, DC, USA, 2000. IEEE Computer Society.

[33] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *SIGKDD*, pages 226–231, 1996.

[34] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing — A fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[35] C. Faloutsos. Gray-codes for partial match and range queries. *IEEE Transactions on Software Engineering*, 14(10):1381–1393, 1988.

[36] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3(3):231–262, 1994.

[37] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. *Advances in Knowledge Discovery and Data Mining*. AAAI Press, 1996.

[38] K. Fischer. *Smallest enclosing ball of balls. Diploma thesis, Institute of Theoretical Computer Science*. ETH Zurish, 2001.

[39] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: The QBIC system. *IEEE Computer*, 28(9):23–32, 1995.

[40] M. Freeston. The BANG file: A new kind of grid file. In *Proc. 1987 ACM SIGMOD International Conference on Management of Data*, pages 260–269. 1987.

[41] K. Fukunaga. *Introduction to Statistical Pattern Recognition (2nd edition)*. Academic Press, 1990.

[42] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[43] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. 25th International Conference on Very Large Databases*, pages 518–529, 1999.

[44] J. Goldstein, R. Ramakrishnan, U. Shaft, and J. B. Yu. Processing queries by linear constraints. In *Proc. of ACM SIGACT-SIGMOD-SIGART*, pages 257–267, 1997.

[45] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1989.

[46] Y. Gong, H. C. Chua, and X. Guo. Image indexing and retrieval based on color histograms. In *Proc. 2nd Multimedia Modeling Conference*, pages 115–126. 1995.

[47] V. Gudivada and R. Raghavan. Design and evaluation of algorithms for image retrieval by spatial similarity. *ACM Transactions on Information Systems*, 13(1):115–144, 1995.

[48] O. Gunther. The design of the cell tree: An object-oriented index structure for geometric databases. In *Proc. 5th International Conference on Data Engineering*, pages 598–605. 1989.

[49] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57. 1984.

[50] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

[51] P.A.V. Hall and G.R. Dowling. Approximate string matching. *Computing Surveys*, 12(4):381–402, 1980.

[52] J. Han and M. Kamber. *Data Mining Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[53] A. Hanjalic, R.L. Lagendijk, and J.Biemond. Improving text retrieval for routing problem using laten semantic indexing. In *Proc. of the 17th Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 282–291, 1994.

[54] J. Hartigan and M. Wong. A k-means clustering algorithm. In *Applied Statistics, 28*, pages 100–108, 1979.

[55] K. Hattori and Y. Torii. Effective algorithms for the nearest neighbor method in the clustering problem. *Pattern Recognition*, 26(5), 1993.

[56] K. Hinrichs. Implementation of the grid file: Design concepts and experience. *BIT*, 25:569–592, 1985.

[57] K. Hinrichs and J. Nievergelt. The grid file: A data structure designed to support proximity queries on spatial objects. In *Proc. International Workshop on Graph-theoretic Concepts in Computer Science*, pages 100–113. 1983.

[58] G. Hjaltason and H. Samet. Ranking in spatial databases. In *Symposium on Large Spatial Databases*, pages 83–95, 1995.

[59] G. Hjaltason and H. Samet. Incremental distance join algorithm for spatial databases. In *Proc. of ACM SIGMOD*, pages 237–258, 1998.

[60] G. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM TODS*, 24(2):265–318, 1999.

[61] W. Hsu, M.-L. Lee, B. C. Ooi, P. K. Mohanty, K. L. Teo, and C. Xia. Advanced database technologies in a diabetic healthcare system. In *Proc. of VLDB*, pages 1059–1062, 2002.

[62] Y. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using r-trees: Breadth-first traversal with global optimizations. In *Proc. of VLDB*, pages 396–405, 1997.

[63] E. Hunt, M. P. Atkinson, and R. W. Irving. A database index to large biological sequences. In *VLDB*, pages 139–148, 2001.

[64] E. Hunt, M. P. Atkinson, and R. W. Irving. Database indexing for large dna and protein sequence collections. *Journal of VLDB*, 11(3):256 – 271, 2002.

[65] Data Mining Software in Java. *http://www.cs.waikato.ac.nz/ml/weka/*.

[66] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 332–342, May 1990.

[67] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[68] H. Jin, B. C. Ooi, H. T. Shen, C. Yu, and A. Y. Zhou. An adaptive and efficient dimensionality reduction algorithm for high-dimensional indexing. In *Proc. of ICDE*, pages 87–98, 2003.

[69] I. T. Jolliffe. *Principal Component Analysis*. Springer-Verlag, 1986.

[70] R. Agrawal K. Shim, R. Srikant. High-dimensional similarity joins. In *Proc. of ICDE*, 1997.

[71] K. V. Ravi Kanth, D. Agrawal, and A. Singh. Dimensionality reduction for similarity searching in dynamic databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 166–176, 1998.

[72] G. Karypis, E.-H. Han, and V. Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.

[73] N. Katamaya and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. 1997 ACM SIGMOD International Conference on Management of Data*. 1997.

[74] N. Katayama and S. Satoh. Distinctiveness-sensitive nearest-neighbor search for efficient similarity retrieval of multimedia information. In *Proc. of ICDE*, pages 493–502, 2001.

[75] E. M. Knorr and R. T. Ng. Algorithms for mining distance-based outliers in large datasets. In *Proc. of VLDB*, 1998.

[76] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In *Spatio-Temporal Database Management*, pages 119–134, 1999.

[77] F. Korn, H. Jagadish, and C. Faloutsos. Effciently supporting ad hoc queries in large datasets of time sequences. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 289–300, 1997.

[78] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proc. of ACM SIGMOD*, pages 201–212, 2000.

[79] F. Korn, S. Muthukrishnan, and D. Srivastava. Reverse nearest neighbor aggregates over data streams. In *Proc. of VLDB*, 2002.

[80] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest neighbor search in medical image databases. In *Proc. 22nd International Conference on Very Large Data Bases*, pages 215–226. 1996.

[81] N. Koudas and K.C. Sevcik. High dimensional similarity joins: algorithms and performance evaluation. *IEEE TKDE*, 12(1):3–8, 2000.

[82] K.P.Chan and A.W-C Fu. Efficient time series matching by wavelets. In *Proc. 15th Int. Conf. on Data Engineering*, pages 126–133, 1999.

[83] P. Larson. Dynamic hashing. *BIT*, 13:184–201, 1978.

[84] K.-I. Lin, M. Nolen, and C. Yang. Applying bulk insertion techniques for dynamic reverse nearest neighbor problems. In *IDEAS*, pages 290–297, 2003.

[85] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. 6th International Conference on Very Large Data Bases*, pages 212–223. 1980.

[86] W. Litwin, N. A. Neimat, and D. A. Schneider. LH* — Linear hashing for distributed files. In *Proc. 1993 ACM SIGMOD International Conference on Management of Data*, pages 327–336. 1993.

[87] M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proc. of ACM SIGMOD*, 1994.

[88] M.-L. Lo and C.V. Ravishankar. Spatial hash-joins. In *Proc. of ACM SIGMOD*, pages 247–258, 1996.

[89] D. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, 1990.

[90] L. Horváth M. Csörgö. *Limit Theorems in Change-Point Analysis*. Wiley, 1997.

[91] Y. Manopopoulos, Y. Theodoridis, and V.J. Tsotra. *Advanced Database Indexing*. Kluwer Academic, 2000.

[92] T. Matsuyama, L.V. Hao, and M. Nagao. A file organization for geographic information systems based on spatial proximity. *International Journal on Computer Vision, Graphics, and Image Processing*, 26(3):303–318, 1984.

[93] B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. Analysis of the clustering properties of hilbert space-filling curve. *Technical Report, Maryland*, 1996.

[94] W. Niblack, R. Barber W. Equitz, M. Flicker, E. Glasman, D. Petkovic, P. Yanker, and C. Faloutsos. The QBIC project: Query images by content using color, texture and shape. In *Storage and Retrieval for Image and Video Databases, Volume 1908*, pages 173–187. 1993.

[95] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[96] V. E. Ogle and M. Stonebraker. Chabot: Retrieval from a relational database of images. *IEEE Computer*, 28(9):40–48, 1995.

[97] B. C. Ooi. *Efficient Query Processing in Geographical Information Systems*. Springer-Verlag, 1990.

[98] B. C. Ooi, R. Sacks-Davis, and K. J. McDonell. Extending a dbms for geographic applications. In *Proc. 5th International Conference on Data Engineering*, pages 590–597. 1989.

[99] B. C. Ooi and K. L. Tan. B-trees: Bearing fruits of all kinds. In *Proc. Australasian Database Conference*. 2002.

[100] B. C. Ooi, K. L. Tan, C. Yu, and S. Bressan. Indexing the edge: a simple and yet efficient approach to high-dimensional indexing. In *Proc. 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 166–174. 2000.

[101] B. C. Ooi, K. L. Tan, C. Yu, and S. Bressan. Transformation-based method for indexing high dimensional data. *patent pending #200002639-3*, 2000.

[102] J. A. Orenstein. Spatial query processing in an object–oriented database system. In *Proc. 1986 ACM SIGMOD International Conference on Management of Data*, pages 326–336. 1986.

[103] J. A. Orenstein. An algorithm for computing the overlay of k-dimensional spaces. In *Proceedings of the Second International Symposium on Advances in Spatial Databases*, pages 381–400, London, UK, 1991. Springer-Verlag.

[104] O. Owolabi and D.R. McGregor. Fast approximate string matching. *Software — Practice and Experience*, 18:387–393, 1988.

[105] J.M. Patel and D.J. DeWitt. Partition based spatial-merge join. In *Proc. of ACM SIGMOD*, pages 259–270, 1996.

[106] I. Popivanov and R. J. Miller. Similarity search over time series data using wavelets. In *Proc. 17th Int. Conf. on Data Engineering*, pages 212–121, 2001.

[107] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2000.

[108] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. of ACM SIGMOD*, pages 71–79, 1995.

[109] Y. Sakurai, M. Yoshikawa, and S. Uemura. The a-tree: An index structure for high-dimensional spaces using relative approximation. In *Proc. 26th International Conference on Very Large Data Bases*, pages 516–526. 2000.

[110] B. Salzberg and V. J. Tsotras. A comparison of access methods for time evolving data. In *Technical Report NU-CCS-94-21*. Northeastern University, 1994.

[111] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi–dimensional objects. In *Proc. 13th International Conference on Very Large Data Bases*, pages 507–518. 1987.

[112] A. Singh, H. Ferhatosmanoglu, and A. Ş. Tosun. High dimensional reverse nearest neighbor queries. In *Proc. of CIKM*, pages 91–98, 2003.

[113] M. Smid. Closest point problems in computational geometry. In *Handbook on Computational Geometry*. Elsevier Science Publishing, 1997.

[114] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *Proc. of ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53, 2000.

[115] I. Stanoi, M. Riedewald, D. Agrawal, and A. El Abbadi. Discovery of influence sets in frequently updated databases. In *Proc. of VLDB*, pages 99–108, 2001.

[116] Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *Proc. of VLDB*, pages 744–755, 2004.

[117] A. K. H. Tung, J. Han, L. V.S. Lakshmanan, and R. T. Ng. Constraint-based clustering in large databases. In *Proc. of ICDE*, pages 405–419, 2001.

[118] R. Weber and S. Blott. An approximation-based data structure for similarity search. In *Technical Report 24, ESPRIT project HERMES (no. 91941)*, pages 194–205. 1997.

[119] R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. 24th International Conference on Very Large Data Bases*, pages 194–205. 1998.

[120] K. Whang and R. Krishnamurthy. Multilevel grid files. Technical Report RC-11516, IBM Thomas J. Watson Research Center, 1985.

[121] D.A. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. 12th International Conference on Data Engineering*, pages 516–523. 1996.

[122] Y.-L. Wu, D. Agrawal, and A. E. Abbadi. A comparison of dft and dwt based sim-ilarity search in time-series databases. In *Proc. 9th Int. Conf. on Information and Knowledge Management*, pages 488–495, 2000.

[123] N. Wyse, R. Dubes, and A.K. Jain. A critical evaluation of intrinsic dimensionality algorithms. *Pattern Recognition in Practice*, pages 415–425, 1980.

[124] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: An efficient method for knn join processing. In *Proc. of VLDB*, 2004.

[125] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *Proc. of ICDE*, pages 485–492, 2001.

[126] C. Yu. *High-Dimensional Indexing*. PhD thesis, Department of Computer Science, National University of Singapore, 2001.

[127] C. Yu, S. Bressan, B. C. Ooi, and K. L. Tan. Query high-dimensional data in single dimensional space. *VLDB Journal*, 13(2):105–119, 2004.

[128] C. Yu, B. C. Ooi, and K. L. Tan. Transformation-based method for similarity search. *patent filed*, 2000.

[129] C. Yu, B. C. Ooi, K. L. Tan, and H. Jagadish. Indexing the distance: an efficient method to knn processing. In *Proc. 27th International Conference on Very Large Data Bases*. 2001.

[130] M. Zait and H. Messatfa. A comparative study of clustering methods. In *FGCS Journal, Special Issue on Data Mining*, pages 149–159. 1997.

[131] P. Zezula, P. Savino, G. Amato, and F. Rabitti. Approximate similarity retrieval with m-trees. *VLDB Journal*, 7(4):275–293, 1998.

[132] J. Zobel and P. Dart. Phonetic string matching: Lessons from information retrieval. In *Proc. 19th ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 166–173, 1996.