GORDER: An Efficient Method for KNN Join Processing

Chenyi Xia[1] Hongjun Lu[2] Beng Chin Ooi[1] Jing Hu[1]

[1] Department of Computer Science,National University of Singapore,{xiacheny, ooibc, hujing}@comp.nus.edu.sg
[2]Department of Computer Science, Hong Kong University of Science and Technology, luhj@cs.ust.hk

Abstract

An important but very expensive primitive operation of high-dimensional databases is the K-Nearest Neighbor (KNN) similarity join. The operation combines each point of one dataset with its KNNs in the other dataset and it provides more meaningful query results than the range similarity join. Such an operation is useful for data mining and similarity search.

In this paper, we propose a novel KNN-join algorithm, called the *Gorder* (or the G-ordering KNN) join method. *Gorder* is a block nested loop join method that exploits sorting, join scheduling and distance computation filtering and reduction to reduce both I/O and CPU costs. It sorts input datasets into the *G-order* and applied the *scheduled block nested loop join* on the G-ordered data. The distance computation reduction is employed to further reduce CPU cost. It is simple and yet efficient, and handles high-dimensional data efficiently. Extensive experiments on both synthetic cluster and real life datasets were conducted, and the results illustrate that Gorder is an efficient KNN-join method and outperforms existing methods by a wide margin.

1 Introduction

K-nearest neighbor join (KNN-join) is a new operation proposed recently [5]. The operation combines each point of one dataset with its K-nearest neighbors in another dataset. With its set-a-time nature, KNN-join can be used to efficiently support various applications where multidimensional data is involved.

In particular, it is identified that many standard algorithms in almost all stages of knowledge discovery process can be accelerated by including KNN-join as a primitive operation. For examples,

- In each iteration of the well-known k-means clustering process, the nearest cluster centroid is computed for each data point. A data point is assigned to 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 first step of LOF [7] (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.
- In the hierarchical clustering method called Chameleon [18], 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[4].

In this paper, we study the efficient processing of the KNN-join. To the best of our knowledge, the MuX KNN-join [5, 4] is the only up-to-date algorithm specifically designed for KNN-join. MuX [6] is essentially an R-tree based method designed to satisfy the conflicting optimization requirements of CPU and I/O cost. It employs large-sized pages (the hosting page) to optimize I/O time and uses the secondary structure, the buckets which are MBRs (minimum bounding boxes) of much smaller size, to partition the data with finer granularity so that CPU cost can be reduced.

MuX iterates over the R pages, and for R page in the memory, potential KNN-joinable pages in S are retrieved through MuX index on S and searched for K-nearest neighbors. Since MuX makes use of an index to reduce the number of data pages retrieved, it suffers as an R-tree based 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

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

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.

In this paper, we propose a novel KNN-join algorithm, the *Gorder* (or the G-ordering KNN) join method. *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 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-tiers 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.

Our contributions can be summarized as follows.

- We developed a novel algorithm *Gorder* for an important operation KNN-join, that requires no index for the source data sets.
- A comprehensive performance study was conducted experimentally that indicates the efficiency, scalability and robustness of the proposed algorithm.

Note that 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. We therefore developed the join method based on the block nested loop join, however, enhanced it with sorting, data scheduling, and distance computation filtering and reduction to attain good KNN-join performance.

The remainder of the paper is organized as follows. Section 2 defines the KNN-join problem and investigates its properties and reviews some related work. Section 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 4 describes a performance study and presents the experimental results. Finally, Section 5 concludes the paper.

2 Preliminary

2.1 KNN Join

In this section, we define the KNN-join problem formally and identify its properties.

Definition 2.1 (KNN-join) *Given two data sets R and S, an integer K and the similarity metric $dist()$, the KNN-join of R and S, denoted as $R \ltimes_{KNN} S$, returns pairs of points (p_i, q_j) such that p_i is from the outer dataset R and q_j from the inner dataset S, and q_j is one of the K-nearest neighbors of p_i.*

Essentially, the KNN-join combines each point of the outer dataset R with its K-nearest neighbors from the inner dataset S. A data point in our study is a multi-dimensional feature vector corresponding to a complex object such as an image. The distance metric in our consideration is the L_ρ metric, where

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

For demonstration purposes, we shall use the most commonly used metric, the square of L_2 (the Euclidean distance). The proposed technique can be adapted to other L_ρ metrics such as the Manhattan distance (L_1) and the maximum distance (L_∞) straightforwardly. In the rest of the paper, we use R to symbolize the outer dataset and S the inner dataset.

KNN-join has following properties:

1. It is asymmetric, that is, ($R \ltimes_{KNN} S \nLeftrightarrow S \ltimes_{KNN} R$). The reason is that the K-nearest neighbor is asymmetric.
2. The cardinality of the answer set of a KNN-join is predictable, since a KNN-join returns K-nearest neighbors for each point of R.
3. The distance from each point in R to its nearest neighbors is unknown apriori.

Property 2 makes KNN-join more useful than another similarity join – the range-join in situations where a good range ε cannot be determined easily. The range-join returns pairs of points from two data sets with their similarity distance not exceeding a given value. One of the difficulties to use similarity range-join in real application is that the distribution of data points are often unknown and giving an appropriate similarity distance threshold between points is rather difficult, if not impossible. As such the results of similarity range-join are somehow unpredictable that requires applications run on trial-and-error basis.

Property 3 inherits the difficulty of the nearest neighbor query. In order to filter unnecessary distance computation, popular algorithms based on an index such as the R-tree [12] (the RKV [14, 23] and the HS [23]) compute the MinDist (minimum distance between the query point and a node of the R-tree) and choose to traverse the node with the minimum MinDist first. The MinDist is also compared with the pruning distance (the distance between the query point and its Kth nearest neighbor candidate). Nodes with MinDist greater than the pruning distance is pruned away.

Nearest neighbor search, which is I/O bound, has been well studied. KNN-join raises new challenges, just as join to selection in relational databases. We have two starting points as the devising of the KNN-join algorithm.

1) indexed-based multiple KNN query (index nested loop join)

2) 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[5]. In addition, as a well-known fact, the index often fails in high-dimensional space, where it performs worse off than sequential scan. On the contrary, the block sequential search is optimized for I/O time. However, without any distance computation pruning, the CPU cost is enormous and the number of distance computation is $|R| \cdot |S|$. Gorder optimizes the block nested loop join with efficient data scheduling and distance computation filtering.

For ease of discussion, in the following, we assume that the data space is a unit hypercube $[0..1]^d$.

2.2 Related Work

Apart from the MuX join method introduced in the introduction, we shall briefly review existing work on similarity join. Most existing techniques have been proposed to support the similarity *range-join* (also known as the distance join[13]). They can be broadly classified into three categories. In the first category, the join methods utilize indexing structures, and examples include the R-tree Spatial Join (RSJ) [8], the breadth first R-tree join [15], the incremental distance join [13] and the MuX range-join [6]. These methods traverse the indexes of R and S synchronously and form joining pairs according to the lower bounding property of the minimum bounding rectangle (MBR). The second category of techniques are hash-based. Examples include the Spatial Hash Join [20] and the Partition-based Spatial Merge Join [21] which partition the data space into buckets and perform the join on pairs of buckets in a recursive manner. The major drawback of such techniques is that the data replication rate grows quickly as dimensionality increases. The third category of techniques are sort-based. The Multi-dimensional Spatial Join (MSJ) [19], GESS [10], and the Epsilon Grid Order (EGO)[3] all belong to this category. [13] introduced the method to use the incremental distance join to support the distance semi-join (similar to the KNN-join) directly by discarding pairs reported by the distance join. However, due to the difficulty in pre-determining the search radius in the KNN-join, the direct application of range-join algorithms to the KNN-join or the implementation of KNN-join as iterative range join is inefficient and I/O expensive.

3 Gorder

We now introduce Gorder KNN-join, a simple yet efficient KNN-join algorithm based on ordering according to grid

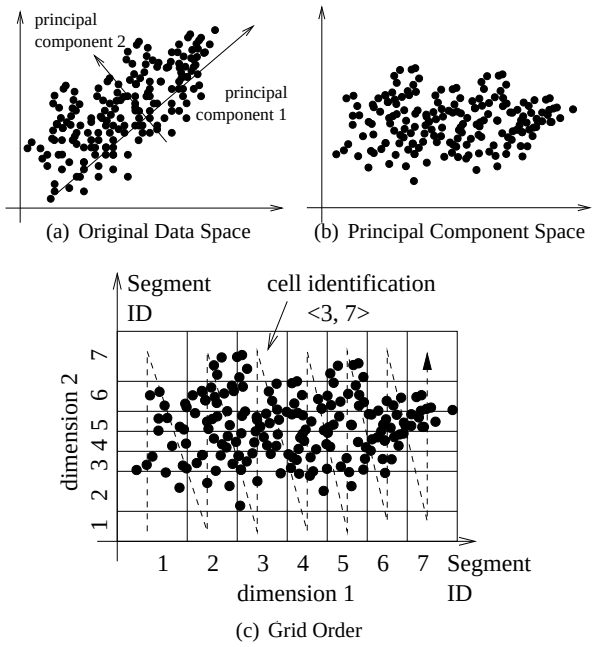


Figure 1: Illustration of G-ordering.

– 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. We describe the algorithm 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);

3.1 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, we design an ordering based on grid called the *G-ordering* to group nearby data points together, so that in the *scheduled block nested loop join* phase we can identify the partition of a block of G-ordered data and schedule it for join.

As illustrated in Figure 1, the G-ordering has two steps – the PCA (principal component analysis) transformation and the *Grid Order* sorting.

The first step of G-ordering performs the principal component analysis [17] on the input datasets R and S together

and transform the original data into the principal component space. 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 condenses into the first few dimensions along which the variances in the data distribution are the largest. The first principal component (or dimension) accounts for as much of the variability in the data as possible, and each succeeding component accounts for as much of the remaining variability as possible.

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 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, we assume the cells are 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.

We define the *identification vector* of cell 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 1.

The *Grid Order* is defined as below.

Definition 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.

The G-ordered data exhibit two interesting properties:

1. Suppose we have 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 [9].

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

To compute the *bounding box*, we first calculate the *active dimension* [3] of the G-ordered data.

Definition 3.2 (Active Dimension of the G-order Data) *Assume ν_1 (ν_m) is 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 $\nu_1.s_j < \nu_m.s_j$ ($1 \leq j \leq d$).

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.

3.2 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.

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, 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 R data in buffer. 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 R data in buffer early.

The large block size reduces disk seek time, however, as a side effect, it may introduce additional CPU cost due to redundant pair-wise checking of tuples for KNN-join. To overcome such a problem, we introduce the second-tier partitioning in memory. 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 20–50 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.

We measure the similarity of two blocks of G-ordered data by the distance between their *bounding boxes*. As presented in Section 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 (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.*

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

$$d_k = max(b_k - u_k, 0) \quad (1)$$

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

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

Definition 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.*

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

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

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 3.1 *For point p_r in block B_r and point p_s in block B_s, $MinDist(B_r, B_s)$ is 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 3.1, we have 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 Kth 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.

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 $B_r \in R$ **do**
2: ReadBlock(B_r);
3: SortBlocks(S, B_r);
4: **for each** $B_s \in$ NotPruned(S, B_r) **do**
5: ReadBlock(B_s);
6: MemoryJoin(B_r, B_s);
7: OutputKNN(B_r);

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 blocks are sorted in the increasing order of their distance to B_r (line 3).[1] 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).

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 $B'_r \in B_r$ **do**
3: SortBlocks(B_s, B'_r);
4: **for** each sub-block $B'_s \in$ NotPruned(B_s, B'_r) **do**
5: **for** each point $p_r \in B'_r$ **do**
6: **if** MinDist(B'_r, B'_s) $\leq$ PrunDist(p_r) **then**
7: **for** each point $p_s \in B'_s$ **do**
8: ComputeDist(p_s, p_r, d^2_α);

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

[1] Note that after the G-ordering, the bounding box for each block of S is kept in in memory, so the sorting doesn't require any disk accesses. The memory for recording the bounding boxes is very limited as there are only a small number of blocks.

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,[2] where $\alpha = min(B'_r.\alpha, B'_s.\alpha)$.

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 we can utilize for distance computation reduction.

Property 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:

Property 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 3.2 *For any point p and q from the G-ordered blocks B_r and B_s respectively, if $MinDist(B_{r,\alpha-1}, B_{s,\alpha-1}) + dist(p_{\{\alpha,k\}}, q_{\{\alpha,k\}})(\alpha \le k \le d)$ 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 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_α^2;
2:  for k := α 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_α^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;
```

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 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.

3.4 Analysis of Gorder

The Gorder algorithm produces 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 3.1). 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 3.2). Hence,

[2]Refer to Equation 1 in Definition 3.3.

for all blocks of R data, Gorder finds the correct K-nearest neighbors.

Now we analyse the I/O and CPU cost of Gorder. 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 [22].

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:

$$\begin{array}{l} 2(N_r + N_s) + +N_r + \frac{N_r}{n_r} \cdot N_s \cdot \gamma_1 \\ +2N_r\left(\left\lceil log_{B-1}\frac{N_r}{B}\right\rceil + 1\right) + 2N_s\left(\left\lceil log_{B-1}\frac{N_s}{B}\right\rceil + 1\right) \end{array}$$

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 $(N_r + N_s) \cdot d^2$ multiply [11]. 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 [2] and [6] which has been shown to be effective in high-dimensional data.

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

$$V_{sphere}^{d-k}(\varepsilon) = \frac{\sqrt{\pi^{d-k}}}{\Gamma\left(\frac{d-k}{2}+1\right)} \cdot \varepsilon^{d-k} \quad (3)$$

$$\varepsilon = \sqrt[d]{\frac{K \cdot \Gamma(d/2+1)}{N_S}} \cdot \frac{1}{\sqrt{\pi}} \quad (4)$$

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

Following the analysis in [2], we simplify Equation 2 by approximating the *bounding boxes* with the hypercube. Therefore,

Parameter	Default Setting
number of nearest neighbors (K)	10
buffer size	around 10% of total size of R and S
size of R data in buffer	around 20% of buffer
number of segments per dimension	32
buffer page size	8192

Table 1: Default parameter values.

$$\gamma = \sum_{k=0}^{d}\left(\begin{array}{c} d \\ k \end{array}\right)\left(\sqrt[d]{\frac{M_r}{P_r}} + \sqrt[d]{\frac{M_s}{P_s}}\right)^k \cdot V_{sphere}^{d-k}(\varepsilon) \quad (5)$$

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 (or sub-block) of data R and S, we get γ_1 (or γ_2).

4 Performance Evaluation

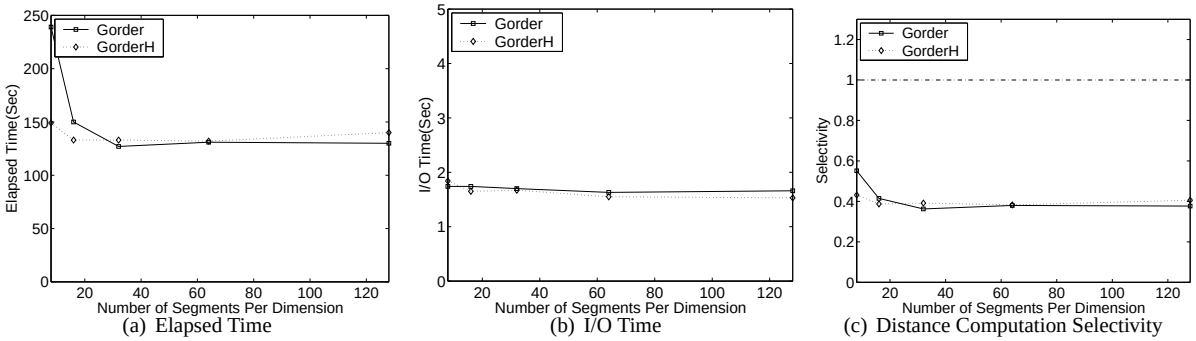
We conducted extensive experimental study to evaluate the performance of Gorder and present the results in this section. In the study, we used both synthetic cluster datasets and real life datasets. The synthetic cluster datasets were generated using the method described in [16], and the real life datasets are from UCI KDD data repository [1]. We used the Corel dataset which contains 64 dimensional feature vectors of 30K images, and the Forest FCoverType dataset which contains 580K records. The original Forest FCoverType dataset has 54 attributes(10 real, 44 binary) and we used the 10 attributes of real value in the experiments.

We compared Gorder with MuX and simple block nested loop join (NLJ). The MuX join [6, 5] 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 Sunfire 4800 server with 750MHz Ultra Sparc III CPU and connected with 2 Sun T3 Disk Array. The buffer allocated for all methods is around 10% 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 1.

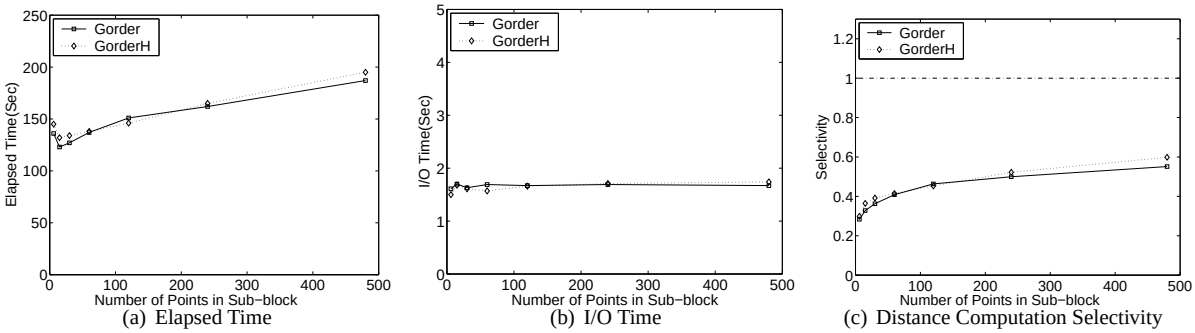
Performance is presented in terms of the elapsed time (which includes I/O and CPU time), the I/O time and the distance computation selectivity. The elapsed time and I/O time of Gorder includes the time for both G-ordering and joining phases. Time of MuX does not include the index building time. Distance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

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



(a) Elapsed Time (b) I/O Time (c) Distance Computation Selectivity

Figure 2: Effect of the number of segments per dimension (Corel dataset)



(a) Elapsed Time (b) I/O Time (c) Distance Computation Selectivity

Figure 3: Effect of sub-block size (Corel dataset)

4.1 Evaluation Using Real Datasets

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

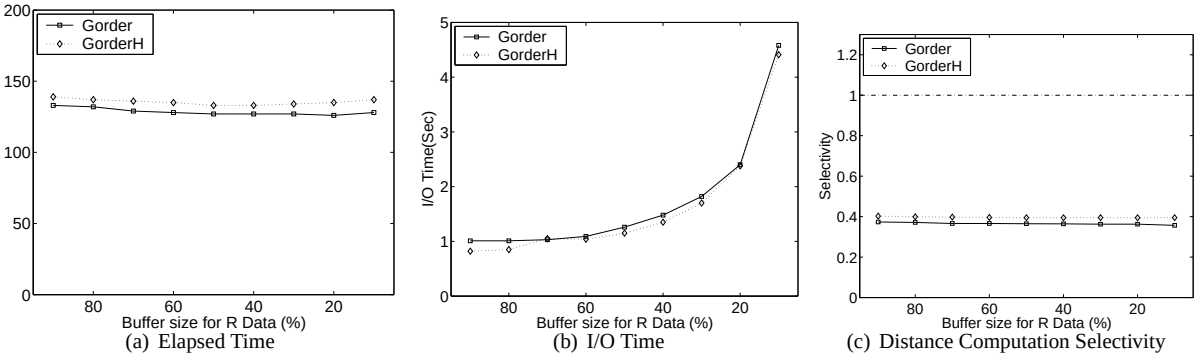
4.1.1 Study of Parameters of Gorder

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 a 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 KNN-join on the datasets. The presented 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 128. Figure 2 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 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.

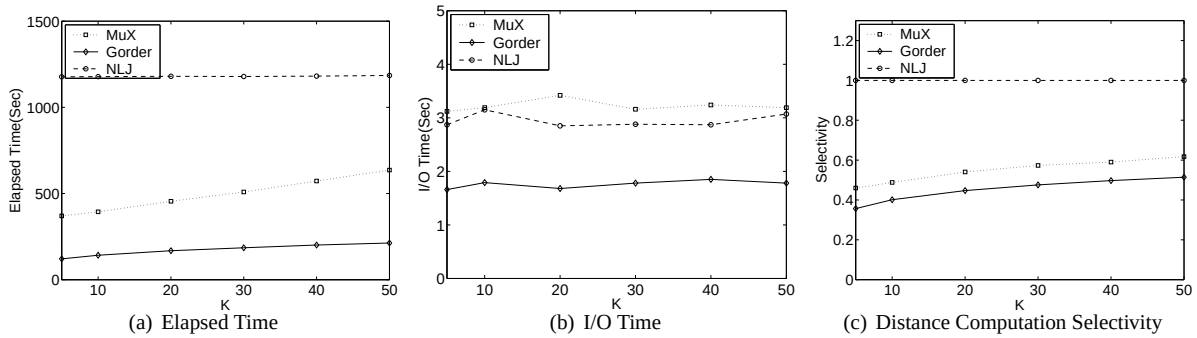
Comparing the I/O time with the total elapsed time, we notice that the I/O time is much less significant than the CPU time (only around 1% of the total response time), which confirms the benefit of using the block accessing and that the KNN-join is CPU critical due to the large number and the complexity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

Effect of sub-block size Figure 3 summarizes the effect of the size of the sub-block on KNN-join processing. In this experiment, the size of the sub-block is varied from 6 to 480 and we conducted the experiment on the Coral 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 6 to 15, 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 decrease of sub-block size increases the number of sub-blocks



(a) Elapsed Time (b) I/O Time (c) Distance Computation Selectivity

Figure 4: Effect of buffer size for R data (Corel dataset)



(a) Elapsed Time (b) I/O Time (c) Distance Computation Selectivity

Figure 5: Effect of K (Corel dataset)

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 between 20–50.

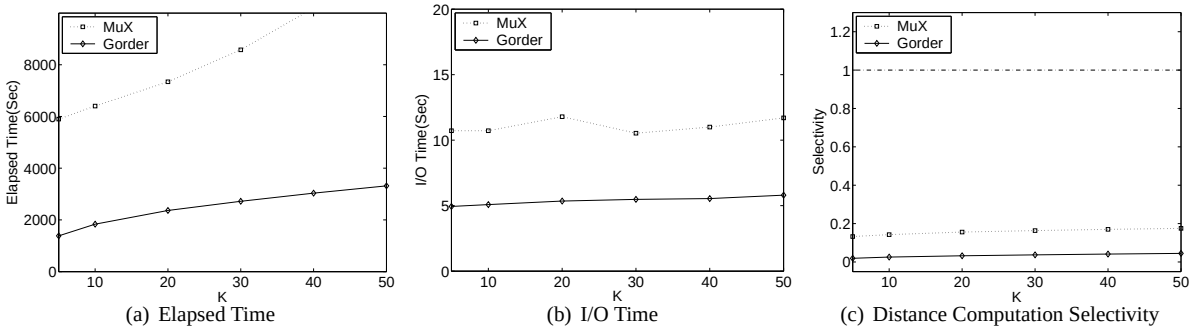
Effect of buffer size for R data Next we study the effect of buffer size allocated to R data and present our study in Figure 4. We fixed the buffer size at around 10% of input data set and decreased the number of buffer pages for R from 90% of buffer to 10% of buffer. Size of sub-block is 30. Figure 4 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 hasn't 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 and the selectivity is improved. Therefore, the increase of the I/O time is absorbed by the decrease of CPU time.

4.1.2 Effect of K

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

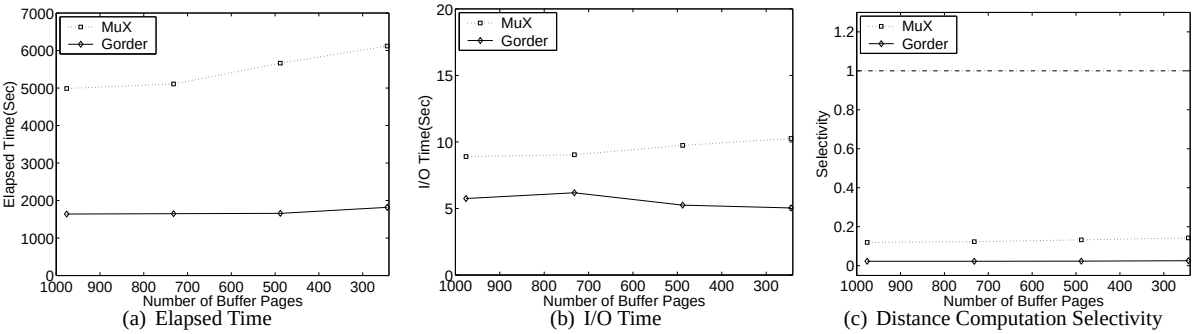
From the results, we observe that with the increase of number of nearest neighbors, the elapsed time of Gorder increases moderately, while MuX is more affected by K. The gap of the elapsed time between MuX and Gorder widens with the increasing K. On average, Gorder outperforms MuX with the speed-up factor of around 2 with regard to the elapsed time. In terms of distance computation selectivity, Gorder is better than MuX by the average factor of 1.22. 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 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 5(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. Gorder is also more efficient than NLJ because with the pruning strategy it filters out S blocks that will not yield KNNs.

Figure 6 presents the results on the Forest dataset. Costs of NLJ are not shown on the graphs because its elapsed time is more than 10,000 seconds. Again, Gorder outperforms MuX significantly with the speed-up factor of 2.45 in terms of elapsed time.



(a) Elapsed Time (b) I/O Time (c) Distance Computation Selectivity

Figure 6: Effect of K (Forest dataset)



(a) Elapsed Time (b) I/O Time (c) Distance Computation Selectivity

Figure 7: Effect of buffer size (Forest dataset)

4.1.3 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 Forest dataset and we reduced the buffer size from around 1000 pages (40% of the dataset size) to around 250 pages (10% of the dataset size). The buffer size for R was kept at 25 pages. In Figure 7, we compare the performance of Gorder and MuX. The result shows that MuX is more sensitive to the decrease of buffer size and its elapsed time increases by 23% when the buffer size decreases from 1000 pages to 250 pages. In comparison, performance of Gorder is more stable and degenerates by only 10% for the same amount of reduction. Gorder is therefore more efficient with respect to buffer space.

We observe that the reduction in buffer space does not affected the I/O performance much. The reduction in buffer size reduces the volume of the bounding box and consequently, leads to the improved effectiveness of the filtering of S blocks. However, the smaller block size of S introduces more disk seeking time. As a balance, the I/O time of Gorder is not much affected by the buffer size.

4.2 Evaluation Using Synthetic Datasets

We study the scalability of Gorder on the synthetic datasets of various sizes and dimensionalities. Since real life data set are often clustered and correlated, we utilized method in [16] to generate clustered datasets containing 10 clusters.

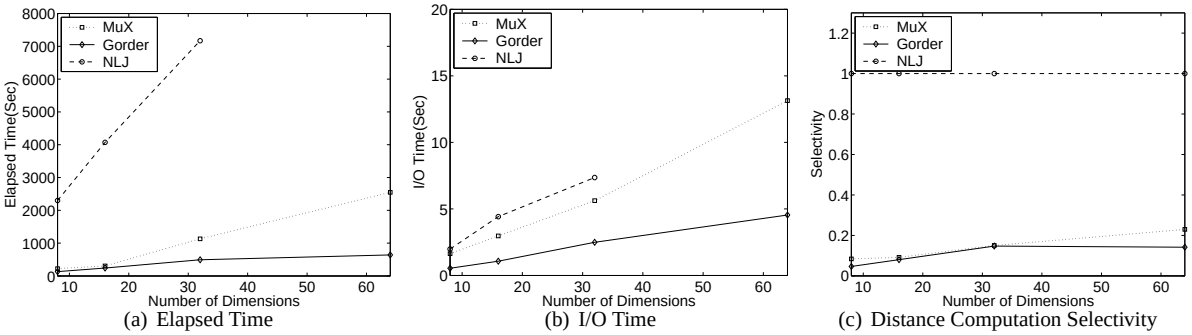
4.2.1 Effect of Dimensionality

In this experiment, we shall evaluate the effect of data dimensionality on the join performance by varying the number of dimensions from 8 to 64. Figure 8 presents the results on the 100K clustered datasets. We observe that the efficiency of MuX deteriorates with the increasing dimensionality. The reason is that MuX, like the R-tree, its performance degenerates with the increase of data dimensionality. Figure 8(c) shows the degeneration of distance computation selectivity of MuX with the increase of the number of dimensions. In addition, the cost of similarity computation of MuX also goes up linearly with the data dimensionality.

The deterioration of distance computation selectivity with the increasing dimensionality is not obvious for Gorder. In addition, Gorder employs the distance computation reduction technique to alleviate the distance computation cost for high dimensional data. Therefore, Gorder is more scalable to high-dimensional data and its performance gain over MuX widens as the dimensionality grows. The speed-up factor of Gorder over MuX increases from 0.68 at dimensionality of 8 to 2.9 at dimensionality of 64.

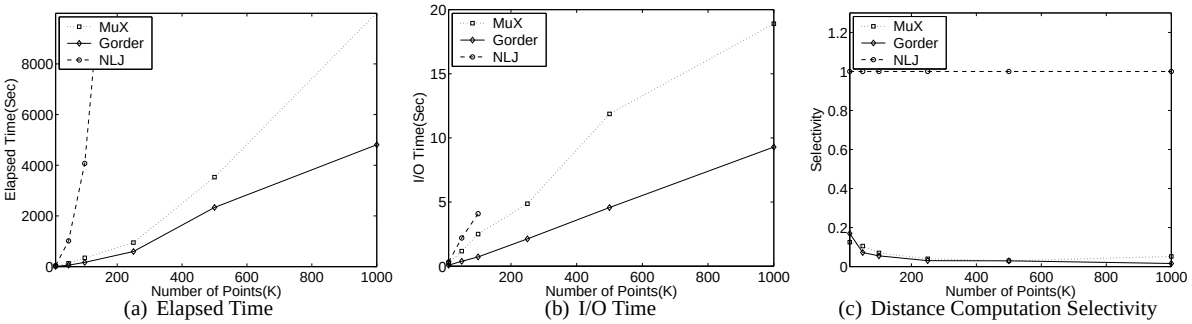
4.2.2 Effect of Size of Dataset

In the second experiment, we study the performance behavior with varying size of datasets. We performed the self



(a) Elapsed Time (b) I/O Time (c) Distance Computation Selectivity

Figure 8: Effect of dimensionality (100k clustered dataset)



(a) Elapsed Time (b) I/O Time (c) Distance Computation Selectivity

Figure 9: Effect of data size (16-dimensional clustered datasets)

KNN-join of the clustered data in the 16-dimensional space and varied the dataset size from 10,000 to 1,000,000 objects. The results are summarized in Figure 9. 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 0.51 to 2.6. Note that even for small datasets where the distance computation selectivity of Gorder is higher than MuX, the elapsed time of Gorder is still lower than MuX due to the use of distance computation reduction technique.

From Figure 9 (c), we observe that the distance computation selectivity improves when the number of data points grows. The reason is that the increase of the number of data points densifies the clusters 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.

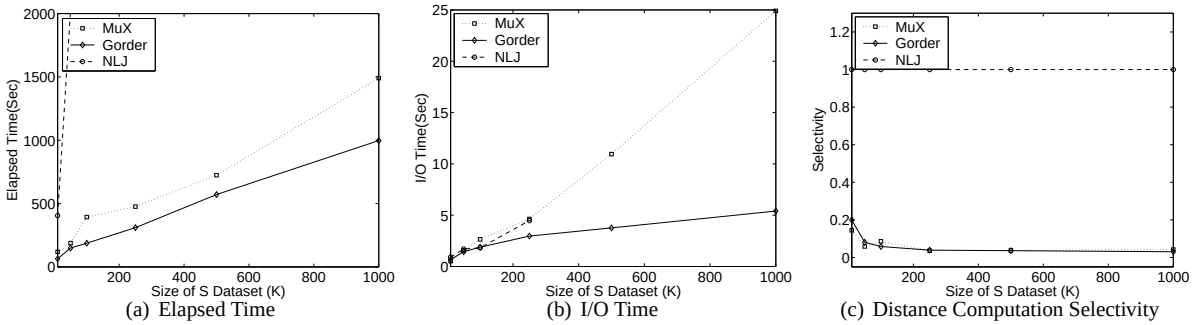
4.2.3 Effect of Relative Size of Dataset

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 1:10. Figure 10 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 0.59, which confirms the scalability of Gorder with respect to the data size again.

5 Conclusion

In this paper, we have investigated 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 data set, 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 proposed *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 presented our performance study on both synthetic cluster and real life datasets and the results confirm that Gorder is efficient and scalable with regard to both data dimensionality and size, and that it outperforms existing methods by a significant margin. Our future work is to design the KNN-join algorithm based on [24].



(a) Elapsed Time (b) I/O Time (c) Distance Computation Selectivity

Figure 10: Effect of relative size of datasets (16-dimensional clustered datasets).

Acknowledgment: The authors would like to thank Christian Böhm for providing the code of MuX similarity join.

References

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

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

[3] 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.

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

[5] C. Böhm and F. Krebs. The k-nearest neighbor join: Turbo charging the kdd process. *Knowledge and Information Systems (KAIS)*, 2004.

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

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

[8] 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.

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

[10] J. 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.

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

[12] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD*, pages 47–57. 1984.

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

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

[15] 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.

[16] 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.

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

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

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

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

[21] J. Patel and D. DeWitt. Partition based spatial-merge join. In *Proc. of ACM SIGMOD*, pages 259–270, 1996.

[22] R. Ramakrishnan and J. Gehrke. *Database Management Systems (2nd Edition)*. McGraw-Hill, 1999.

[23] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. of ACM SIGMOD*, pages 71–79, 1995.

[24] C. Yu, B. C. Ooi, K. L. Tan, and H. Jagadish. Indexing the distance: an efficient method to knn processing. In *Proc. of VLDB*, 2001.