Finding Data Broadness Via Generalized Nearest Neighbors

Jayendra Venkateswaran[1], Tamer Kahveci[1] and Orhan Camoglu[2]

[1] CISE Department University of Florida Gainesville, FL 32611
{jgvenkat, tamer}@cise.ufl.edu
[2] University of California, Santa Barbara, CA 93106
{orhan}@cs.ucsb.edu

Abstract. A data object is broad if it is one of the k-Nearest Neighbors (k-NN) of many data objects. We introduce a new database primitive called Generalized Nearest Neighbor (GNN) to express data broadness. We also develop three strategies to answer GNN queries efficiently for large datasets of multidimensional objects. The R*-Tree based search algorithm generates candidate pages and ranks them based on their distances. Our first algorithm, Fetch All (FA), fetches as many candidate pages as possible. Our second algorithm, Fetch One (FO), fetches one candidate page at a time. Our third algorithm, Fetch Dynamic (FD), dynamically decides on the number of pages that needs to be fetched. We also propose three optimizations, Column Filter, Row Filter and Adaptive Filter, to eliminate pages from each dataset. Column Filter prunes the pages that are guaranteed to be non-broad. Row Filter prunes the pages whose removal do not change the broadness of any data point. Adaptive Filter prunes the search space dynamically along each dimension to eliminate unpromising objects. Our experiments show that FA is the fastest when the buffer size is large and FO is the fastest when the buffer size is small. FD is always either fastest or very close to the faster of FA and FO. FD is significantly faster than the existing methods adapted to the GNN problem.

1 Motivation

Given two datasets R and S, an object in S is called *broad* if it is one of the k Nearest Neighbors (k-NN) of many objects in R. A k-NN query seeks the k closest objects in a dataset S to a given query object q with respect to a predefined distance function, where k is a given positive integer. Finding the broadness of data is needed in many applications such as life sciences (e.g., detecting repeat regions in biological sequences [10] or protein classification [6]), distributed systems (e.g., resource allocation), Spatial databases (e.g.,Decision Support Systems or Continuous Referral Systems [12]), profile-based marketing, etc.

In this paper, we define a new database primitive, called the *Generalized Nearest Neighbor (GNN)* which naturally detects data broadness. Given two datasets R and S, the GNN query finds all the objects in $S' \subseteq S$ that appear in the k-NN set of at least t objects of R, where t is a cutoff threshold. The objects in the result set of a GNN query are broad. Here, S' is the set of objects that the user focuses on for broadness property. If $R = S$, then it is called *mono-chromatic* query. Otherwise, it is called *bi-chromatic* query. Following examples present GNN queries in two different problem domains.

Example 1. (*Bioinformatics*) Functional relations among families of genes are usually found through similarity of their features (e.g., sequences, structures). If a gene from family S is one of the top closest genes to many of the genes from family R, then that gene is considered too broad (or useless) for classification. Such genes are filtered to obtain better efficiency and classification quality since they mask the results from other genes [8]. In this example, R = first family of genes, $S = S'$ = second family of genes. □

Example 2. (*Spatial databases*) Suppose that people usually dine at one of the three closest restaurants to their houses. An entrepreneur who wants to invest in Mexican restaurant business would want to know the Mexican restaurants that potentially have many customers. In this example R = set of houses, S = set of restaurants, and S' = set of Mexican restaurants. □

The trivial solution to a GNN query is to run a k-NN query for each object in R one by one, and accumulate the results for each object in S. Currently, biologists are using this approach for the queries in Example 1. However, this approach suffers from both excessive amount of disk I/Os and CPU computations. When the datasets do not fit into the available buffer, a page that will be needed again might be removed from buffer while processing a single k-NN query. CPU cost also accounts for a significant portion of the total cost.

In this paper, we assume datasets to be larger than the available buffer. We propose three solutions. Our methods arrange the data objects into pages. Each page contains a set of objects and is represented by their minimum bounding rectangle (MBR). Two R*-trees [2] are built on objects in R and S. The candidate pages from S that may contain k-NNs for the MBRs of R are predicted. Each candidate page $s \in S$ is assigned a priority based on its proximity to a MBR $r \in R$ and is stored in a Priority Table (PT). Our first algorithm, pessimistic approach, fetches as many candidate pages as possible from S for each r. Our second algorithm, optimistic approach, fetches one s at a time for each r. Our third algorithm dynamically decides the number of pages that needs to be fetched for each r by analyzing query history. We reduce the CPU and I/O cost significantly through three optimizations by dynamically pruning 1) pages of S that are not in the k-NN set of sufficiently many objects in R, and 2) pages of R whose nearest neighbors do not contribute to the result 3) objects in candidate MBRs of S that are too far from the MBRs of R. We further reduce these costs by pre-processing the input datasets using a packing technique called Sort-Tile-Recursive (STR) [16].

Experiments show that our optimistic strategy works best when the buffer size is small and pessimistic strategy when the buffer size is large. On the other hand, dynamic strategy is always either the best of the three or very close to the better of the two other strategies. Our methods are significantly faster than sequential scan, R-tree–based branch-and-bound method [15], GORDER [21], MuX index [5] and RkNN [20] .

Data broadness is a new problem that requires new approaches. To our best knowledge, all of the following ideas are introduced first time in this paper and has not been discussed elsewhere before:

- The GNN problem.
- Use of PT to obtain global-coarse-grain view.
- Three search strategies: FA, FO and FD.
- Column, Row and Adaptive Filters.

The rest of the paper is organized as follows. Section 2 presents background on well known NN query types. Section 3 introduces the problem. Section 4 explains how the candidate pages are determined. Section 5 discusses our pessimistic and optimistic strategies. Section 6 discusses our dynamic strategy. Section 7 presents an optimization strategy for reducing the CPU and I/O costs. Section 8 presents our experimental results. We end with a brief discussion in Section 9.

2 Related work

A number of index-based methods have been developed for k-NN queries. Hjaltson and Samet [9] used PMR quadtree to index the search space. They search this tree in a depth-first manner until the k nearest neighbors are found. Roussopoulos et. al., [15] employed a branch-and-bound R-Tree traversal algorithm. The two-phase method [13] determines k closest objects based on feature distance. It then runs a range query using the actual distance to the kth closest object found in the first phase as the query range. Seidl and Kriegel [18] proposed a method that runs in multiple phases iteratively updating the upper and lower bounds of kth NN. It stops when these bounds coincide. Berchtold et. al., [3] divide the search space using Voronoi cells. Beyer et. al., [4] show that for a broad set of data distributions most of the known k-NN algorithms run slower than sequential scan. Thus, despite its simplicity, sequential scan still remains a formidable competitor to index-based k-NN methods.

Korn et. al., [12] introduced the *Reverse Nearest Neighbor (RNN)* problem. They precompute the NN of all the objects in the dataset. Yang and Lin introduced the Rdnn-tree for RNN queries [22]. Stanoi et. al., proposed to compute a region of influence with the help of a Voronoi Diagram [19]. It then performs a range search with radius equal to the radius of the influence region. Tao et. al., [20] generalize the RNN problem to arbitrary number of NNs using a filter-and-refine approach.

Despite its wide use in many areas, *All Nearest Neighbor (ANN)* is the least studied NN query type. MuX uses a two-level index structure called MuX index [5]. At the top level, MuX index contains large pages (or MBRs). At the next level, these pages contain much smaller buckets. For each bucket from R, it computes a pruning distance as it scans the candidate points from S. It prunes the pages, buckets, and points of S beyond this distance for each bucket of R. GORDER [21] is a block nested loop join method. It first reduces the dimensionality of R and S by using Principal Component Analysis (PCA). It then places a grid on the space defined by PCA and hashes data objects into grid cells. Later, it reads blocks of data objects from grid cells by traversing the cells in grid order and compares all the objects in pairs of grid cells whose MINDIST is less than the pruning distance defined by the kth NN.

3 Problem definition

Let R and S be two datasets. The GNN query is defined by a 5-tuple GNN(R, S, S', k, t), where $S' \subseteq S$, and k and t are positive integers. This query returns the set of tuples (s, R_s), where $s \in S'$, $R_s \subseteq R$ is the set of objects that have s as one of their k-NN, and $|R_s| \geq t$. We use the Euclidean distance as the distance measure in this paper unless otherwise stated.

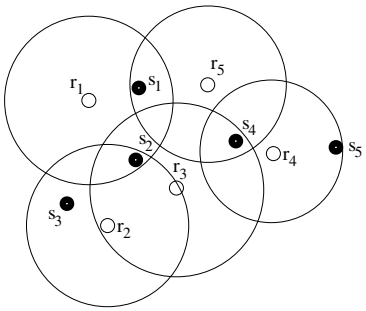


Fig. 1. The white and black points are the locations of the 2-D objects in datasets $R = \{r_1, \cdots, r_5\}$ and $S = \{s_1, \cdots, s_5\}$ respectively. The circles show the 2-NNs of the objects of R in S.

Assume that the white and black points in Figure 1 show the layout of 2-D datasets $R = \{r_1, \cdots, r_5\}$ and $S = \{s_1, \cdots, s_5\}$ respectively. Consider the following query:

GNN(R, S, $S' = \{s_1, s_2, s_5\}$, 2, 3).

This translates as: "Find the objects in S' that are in the 2-NN set of at least three objects in R". In Figure 1, the circles centered at each $r_i \in R$ covers the 2-NN of r_i, $\forall i$. Only s_2 and s_4 are covered by at least three circles. We ignore s_4 since $s_4 \notin S'$. The set $\{r_1, r_2, r_3\}$ has s_2 in their 2-NN. Therefore, the output of this query is $\{(s_2, \{r_1, r_2, r_3\})\}$. Note that we cannot ignore the data points in $S - S'$ prior to GNN query. In other words GNN(R, S, S', k, t) $\neq$ GNN(R, S', S', k, t). For example, in Figure 1, removal of s_3 and s_4 prior to GNN query changes the 2-NNs of r_2, r_3 and r_4. As a result s_1 becomes one of the 2NNs of r_2 and r_3. Hence s_1 is incorrectly classified as broad.

A nice property of the GNN query is that both mono-chromatic and bi-chromatic versions of the standard k-NN, ANN and RNN queries are its special cases. Following observations state these cases. One can prove these from the definition of the GNN query. Note that the goal of this paper is not to find different solutions to each of these special cases. Our goal is to solve a broader problem which can not be solved trivially using these special cases.

Observation 1 *GNN($\{r\}$, S, S, k, 1) returns the k-NN of the object r in S. If $r \in S$, then it corresponds to the mono-chromatic k-NN query. Otherwise, it corresponds to the bi-chromatic k-NN query.*

Observation 2 *GNN(R, S, S, 1, 1) returns the ANN of R in S. If $R = S$, then it is the mono-chromatic case, otherwise bi-chromatic case.*

Observation 3 *GNN(R, S, $\{s\}$, 1, 1), where $s \in S$, returns the RNN of the object s in R. If $R = S$, then it is the mono-chromatic case, otherwise bi-chromatic case.*

4 Predicting the solution: Priority Table Construction

Let R and S be two given datasets, and $\mathcal{A}$ and $\mathcal{B}$ be the sets of MBRs that contain the objects in these datasets. We discuss the computation of the candidate set of MBR pairs one from $\mathcal{A}$ and the other from $\mathcal{B}$ that needs to be inspected to calculate a given GNN query. We assume that the datasets are packed and indexed prior to the GNN query. We discuss packing of the dataset in more detail in Section 7. This is a one time cost per dataset; the same index will be used for all the queries. We use STR [16] for a total ordering of the data. Throughout this paper R*-Tree is used to index the datasets.

Other index structures can be used to replace the R*-tree. For simplicity, we choose the capacity of each MBR of the R*-tree as one disk page and use leaf level MBRs to prune the solution space.

Given two MBRs B_1 and B_2, we define MAXDIST(B_1, B_2) and MINDIST(B_1, B_2) as the maximum and minimum distance between B_1 and B_2. The following lemma establishes an upper bound to the k-NN distance to the objects in a set of MBRs.

Lemma 1. *Let A be the MBR of a set of objects and $a \in A$ be an object. Let $\mathcal{B} = \{B_1, \cdots, B_{|\mathcal{B}|}\}$ be the set of leaf level MBRs of an index structure built on a dataset. Assume that the MBRs in $\mathcal{B}'$,where $\mathcal{B}' \subseteq \mathcal{B}$, contain at least K objects. Let ε denote the distance of object a to its kth NN in $\mathcal{B}$, then*

$$\varepsilon \leq \max_{B \in \mathcal{B}'} \{\text{MAXDIST}(A, B)\}, \forall k, 1 \leq k \leq K.$$

Proof follows from the observation that all objects in B' appear in B too.For a given positive integer k, let m be the integer, $1 \leq m < |\mathcal{B}|$, for which

$$\sum_{i=1}^{m-1} \mid B_i \mid < k \leq \sum_{i=1}^{m} \mid B_i \mid,$$

where $|B_i|$ is the number of objects in B_i. Let MAXDIST(A, B_i) $\leq$ MAXDIST(A, B_{i+1}), $\forall i$ $1 \leq i <|\mathcal{B}|$, where $|\mathcal{B}|$ is the cardinality of $\mathcal{B}$.

From Lemma 1, we know that the k-NN distance of the objects in A to the objects in $\mathcal{B}$ is at most MAXDIST(A, B_m). Hence, if MAXDIST(A, B_m) $<$ MINDIST(A, B), $B \in \mathcal{B}$, then B does not contain any object from the k-NN set of any object in A. Therefore, B can be pruned away from $\mathcal{B}$ without any false dismissals during the computation of the k-NNs of the objects in A. From these observations, given a GNN query, GNN(R, S, $S' \subseteq S$, k, t), for each $A \in \mathcal{A}$, we compute a priority list of the candidate boxes in $\mathcal{B}$ as follows:

- For each $A \in \mathcal{A}$,
  Step 1: Compute MAXDIST(A, B_m) for the given value of k as discussed above.
  Step 2: Find the MBRs, $B \in \mathcal{B}$ for which MINDIST(A, B) $\leq$ MAXDIST(A, B_m).
  Step 3: Assign priorities to these MBRs in increasing MINDIST(A, B) order.

The algorithm for Step 1 takes an MBR A, the root node of an R*-tree, and an integer k as input. The root node is stored using a min-heap. The node with the smallest MAXDIST to A is extracted from this heap. If the MINDIST of this node to A is more than the threshold, then it is omitted. Otherwise it is inspected. If it is an internal node, then its children are inserted into the min-heap. Otherwise, it is inserted into the candidate set, which is maintained using a max-heap. If the candidate set contains more objects than necessary, then the MBR with the largest MINDIST value is removed from the candidate set. Although the worst case time complexity of this step is $O(\mid \mathcal{B} \mid)$ (i.e., the entire index is traversed) with an amortized complexity of $O(\log(\mid \mathcal{B} \mid)$. Step 2 is computed using the classic range search algorithm on R-trees having a complexity of $O(\log(\mid \mathcal{B} \mid)$. This step eliminates all the leaf level MBRs that only contain irrelevant points. Naturally, if an MBR contains at least one relevant point, it will be processed by the strategies proposed in Section 5. Step 3 takes $O(C \log C)$ time where C is number of candidate MBRs found at Step 2.

The candidates for all MBRs in $\mathcal{A}$ are stored in a *Priority Table (PT)*. Figure 2 depicts the PT constructed for the GNN(R, S, S', k, t) query. Here, r_i and s_i correspond

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
r_1	3		1				2	
r_2			2	1			3	
r_3						1		2
r_4		1			2			3
r_5	3	2		4				1
r_6	1					2		
r_7	1							
r_8		1				2		

Fig. 2. A sample Priority Table for two datasets R and S. Each row/column corresponds to a page of R/S on disk. Numbers in cells show the priority of pages in S for that row.

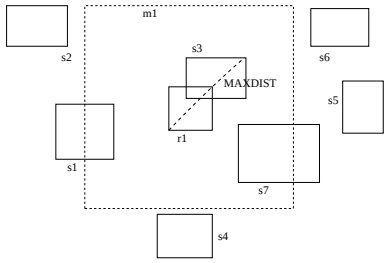


Fig. 3. First row of the Priority Table. Here $r_1 \in R$ and $S = \{s_1, \cdots, s_7\}$. Objects in m_1 are within MAXDIST distance from r_1.

to MBRs for R and S. We assume that $S' = \{s_1, s_3, s_4, s_5, s_7\}$ in this example. In this figure, each row and column corresponds to r_i and s_i respectively. For simplicity, we make two assumptions without affecting the generality: 1) The objects in R and S are located sequentially on disk. 2) Each row and column of the PT (i.e., each MBR) corresponds to one disk page. The numbers at each row show the priority of the candidate MBRs in S for the corresponding MBR in R. For example in row 1, the MBRs s_1, s_3 and s_7 are in the candidate set of r_1, such that s_3 has the highest priority and s_1 has the lowest priority. This is depicted in Figure 3. If an MBR of S is not in the candidate set of an MBR in R, then the corresponding cell is unnumbered.

Given a query, GNN(R, S, $S' \subseteq S$, k, t), our search methods reduce the solution space by pruning the PT. Following two optimizations can be made to reduce search space by inspecting the PT:

Optimization 1 (Column Filter) *Let s_i correspond to an MBR in S'. If the total number of objects in the MBRs in R which have s_i in their candidate set is less than t, then that column can be removed from S'.*

For example, in Figure 2, s_5 is in the candidate set of only r_4. If the total number of objects in r_4 is less than t, then s_5 can be removed from S' safely. The correctness of *Column Filter* can be proven from the fact that an object in a column, s_i can be in the k-NN set of the objects in the rows only that have s_i in the candidate set.

Optimization 2 (Row Filter) *If a row does not contain any candidate MBRs in S', then it can be removed from PT.*

For example, in Figure 2, rows r_3 and r_8 do not have any candidates in S'. Therefore, these rows can be omitted safely. If s_5 is pruned from S' due to Column Filter, then the row, r_4, can also be ignored.

5 Static search strategies

PT defines the MBR pairs that (potentially) need to be compared to answer a given GNN query. In this section, we develop two methods to compute a GNN query, GNN(R, S, $S' \subseteq S$, k, t), given the PT of the datasets R and S. We name these methods *Fetch All (FA)* and *Fetch One (FO)*. We assume a limited buffer space B throughout this section. That is, the sizes of both R and S are larger than B.

Procedure ProcessBuffer(k) `/* Let k be a positive integer. */`
For each row r_i in the buffer **do**

- **while** row r_i has more uninspected candidate MBRs in the buffer **do**
  1. Let s_{r_i} be the next unprocessed candidate MBR with the highest priority.
  2. Find the k-NN of each object in r_i in s_{r_i}. Store the maximum of these k-NN distances in d_{max}.
  3. Remove all candidates, s, of r in PT for which MINDIST(r, s) $> d_{max}$.

Fig. 4. The procedure to process a Buffer.

5.1 Fetch All

Our first method uses a pessimistic strategy: to process each page (i.e., MBR) of R, (i.e., one row in PT), it reads as many candidate pages from S as possible into buffer at once starting from the one with the highest priority. For example, for r_1, FA reads s_1, s_3, and s_7. FA runs in 3 phases: (1) find maximal clusters that fit into buffer, (2) reorder the clusters to maximize the overlap and (3) read the pages for each cluster and process the contents. Next, we elaborate on each phase.

Phase 1: We create clusters by iteratively adding rows into the current cluster, starting from the first row, until its size becomes B. When the cluster becomes large enough, we start a new cluster. For example, if B = 6 pages, then the clusters for the PT in Figure 2 are $C_1 = \{r_1, r_2, s_1, s_3, s_4, s_7\}$, $C_2 = \{r_3, r_4, s_2, s_5, s_6, s_8\}$, $C_3 = \{r_5, s_1, s_2, s_4, s_8\}$, and $C_4 = \{r_6, r_7, r_8, s_1, s_2, s_6\}$. The total cost of this step is linear in the number of candidate pages since each candidate page is visited only once.

Phase 2: The order for reading the clusters affects the total amount of disk I/O. This is because, if consecutive clusters have common pages, these pages will be reused and they do not need to be read again. For example, if C_3 is read after C_1, then s_1 and s_4 will be reused, saving two disk reads. Given a read schedule of clusters, the total amount of disk reads saved by reusing buffer is equal to the sum of the common pages between consecutive clusters. For example, if the clusters are read in the order of C_1, C_3, C_2, C_4, then total savings adds up to 6 pages (i.e., $|C_1 \cap C_3| + |C_3 \cap C_2| + |C_2 \cap C_4| = 6$).

One can show that the Traveling Salesman Problem (TSP) can be reduced to the problem of finding the best schedule for reading clusters. Intuitively, the proof is as follows. Each vertex of TSP maps to a cluster. Each edge weight $w_{i,j}$ between clusters C_i and C_j is computed as the number of overlapping pages between C_i and C_j. The best schedule on this graph is the Hamiltonian Path that maximizes the sum of edge weights. Since TSP minimizes the sum of edge weights, we update the weight of each edge $w_{i,j}$ as $w'_{i,j} = w_{\max} - w_{i,j}$, where $w_{\max}$ is the largest edge weight. This guarantees that the new edge weights are non-negative. Then we create a new node v and is connected to all nodes by zero-weight edges. The optimal schedule is the path with the smallest sum of edge weights which begins at vertex v and visits all nodes once. We use a greedy heuristic to find a good schedule as follows: We start with an empty path. While there are unvisited vertices, we insert the next edge with the smallest weight into the path if it does not destroy the path. Finally, we attach the disconnected paths randomly if there are any.

/*Let k be a integer.*/

Procedure FO(k)
While there are unprocessed rows **do**

1. Fill half of buffer with the MBRs,r_i, from R and one page from S (s_{r_i}) for each r_i.
2. ProcessBuffer(k).
3. Remove all the rows, r_i, from the buffer that has no other uninspected candidate MBRs.
4. Apply Optimizations 1 and 2 on PT.

Fig. 5. The Fetch One procedure.

/*Let k be an integer.*/
/*Let B be buffer size.*/

Procedure FD(k)

1. Initialize f.
2. **while** there are unprocessed rows **do**
   (a) Fill buffer with $\lfloor \frac{B}{f+1} \rfloor$ pages (r_i) from R and f pages from S (s_{r_i}) for each r_i.
   (b) ProcessBuffer(k).
   (c) Remove all rows r_i, from buffer that has no other uninspected candidate MBRs.
   (d) Apply Optimization 1 and 2 on PT.
   (e) Update value of f.

Fig. 6. The Fetch Dynamic procedure.

Phase 3: Once the cluster schedule is determined, the contents of each cluster are iteratively read into buffer using optimal disk scheduling [17]. Figure 4 shows the procedure used to process each cluster after it is fetched into buffer. For each row in the cluster, the algorithm searches the k-NN of each object starting from the box with the highest priority (Steps 1 and 2). The results obtained at this step are used to prune the candidate set (Step 3). After the candidate set is pruned, Optimizations 1 and 2 are applied to PT in order to further reduce the solution space.

5.2 Fetch One

FA reads many redundant pages if only a small percentage of the candidate pages contain actual k-NNs. FO uses optimistic approach to avoid this problem. FO iteratively reads one page per row as long as there are more candidates.

Figure 5 presents the pseudocode for FO. The algorithm splits buffer equally for each of the datasets. This is because, one candidate page is read per row starting from the highest priority (Step 1). Therefore, the number of pages from each dataset in the buffer will be equal at all times if all the candidate pages are distinct. After searching each candidate page (Step 2), PT is further pruned by eliminating the pages that are farther than the kth NN found so fa (Step 3)r, and using Optimizations 1 and 2 (Step 4).

For example, for the PT in Figure 2, let buffer size be 6 pages, then FO reads $\{r_1, r_2, r_3\}$ and $\{s_3, s_4, s_6\}$ into buffer. Assume that the third candidate of r_1 is pruned at the end of this step. Next, $\{s_7, s_8\}$ are read to replace $\{s_4, s_6\}$. Although it is the second candidate of r_2, we do not read s_3 at this step since it is already in buffer. Assume that the third candidate of r_2 is pruned at the end of this step. Since none of the rows $\{r_1, r_2, r_3\}$ have any remaining candidates FO does not need to read any more pages for these rows. Therefore, $\{r_1, r_2, r_3\}$ is replaced with $\{r_4, r_5, r_6\}$, and the search continues recursively.

6 Dynamic strategy

FO reads only the necessary pages (i.e., MBRs) to compute a given GNN(R, S, S', k, t) query since it reads one page at a time starting from the highest priority and stops when the distance to the next MBR is more than the distance to the kth NN found so far. However, this does not guarantee that the total I/O cost is minimized. This is because FO incurs a random seek cost every time a new page is fetched from disk. Since a random seek is significantly more costly than a page transfer, reading a few redundant pages sequentially at once may be faster than FO. Thus, neither FO nor FA ensures the optimal I/O cost. The number of pages read at each iteration, f, that minimizes the I/O cost depends on the query parameters and the distribution of the database. A good approximation to this number can be obtained by sampling the MBRs of R.

Our third method, *Fetch Dynamic (FD)* adaptively determines the value of f as follows. It starts by guessing the value of f. It then reads the first cluster using this value. As it finds the k-NNs of all the objects in the first cluster, it computes the optimal value of f for that cluster. It then uses this value of f to choose the next cluster. After processing each cluster, it iteratively updates f as the median of the number of pages needed for all of the rows processed so far. Note that, the choice of the initial value of f has no impact in the performance after the first step, since f is updated immediately after every iteration. As more rows are processed in each iteration, f adapts to the query parameters and data distribution.

Figure 6 presents the pseudocode of FD. The algorithm first assigns an initial value for f ($1 \leq f \leq$ candidate size). We use 20 % of the average number of candidates of R as our initial guess. Let B denote the buffer size. While there are unprocessed rows, FD reads $\lfloor \frac{B}{f+1} \rfloor$ pages (r_i) from R and f pages (s_{r_i}) from S with the highest priority for each R page in buffer (Step 2.a). Thus, if all the candidates are distinct, buffer is filled with pages from R and S. Steps 2.b processes each candidate page s_{r_i}. The processed pages (r_is) are removed from buffer at Step 2.c. The algorithm continues with Steps 2.a to 2.c until all the rows in buffer are exhausted. Then Optimizations 1 and 2 are applied (Step 2.d). The value of f is updated at Step 2.e as the median of the number of candidates of the processed pages in R.

7 Further improvements for GNN queries

So far we have discussed two optimizations, row filter and column filter to trim both I/O and CPU costs. In this section, we will discuss further optimizations to cut down both CPU and I/O costs of FA, FO, and FD.

Adaptive Filter: Our third optimization follows from the following observation. For a given MBR r we expand it by $d_{\max}$ in all dimensions. If a candidate MBR s overlaps with this expanded MBR, then we compute the distance between all pairs of points from r and s. (Steps 2 and 3 of Figure 4) This incurs $O(t^2)$ comparisons if each MBR contain $O(t)$ points. We reduce this cost in two ways. First, instead of expanding by $d_{\max}$, we can adaptively expand by different amounts along different dimensions. Second, we avoid t^2 comparisons by pruning unpromising points from S in a single pass. More formally, we first find all points in a candidate MBR s that are contained in the expanded MBR of r. Next, we compute the distances between all those points and all points of r.

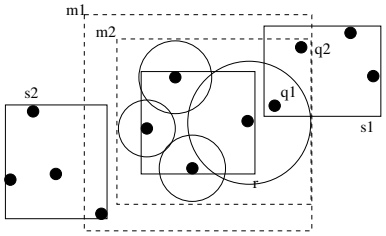


Fig. 7. m_1, m_2 are the expanded MBRs for the r without using and using the *Adaptive Filter* respectively. s_1, $s_2 \subseteq S$, s_1 is the MBR of the points $\{q_1, q_2, q_3, q_4\}$.

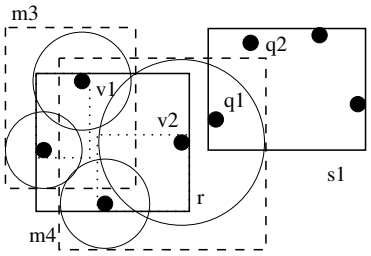


Fig. 8. v_1 and v_2 are two partitions of MBR r and m_3 and m_4 are their extended MBRs. $s_1 \subseteq S$ is the MBR of the points $\{q_1, q_2, q_3, q_4\}$.

Let t', $t' \leq t$, be the number of points in s that are contained in the expanded MBR of r. The CPU cost for the comparison of MBR pairs drops from $O(t^2)$ to $O(t + t \cdot t')$. This is summarized as our third optimization, *Adaptive Filter*.

Optimization 3 (Adaptive Filter) *Let p be a point in MBR r. Let d be the kNN distance of p to the points in MBR s. Let kNN-sphere of p denote the sphere with radius d, centered at p. Let M denote the MBR that tightly covers the kNN-spheres of all the points in r. A point can not be a kNN of a point in r if it is not contained in M.*

In Figure 7, the expanded MBR of r in the worst case is given by m_1. When adaptive bounds are used, the expanded MBR m_2 is obtained. In the former case, two MBRs s_1 and s_2 intersect with m_1. Thus, 3 disk I/Os (r, s_1, s_2) and 32 comparisons are made. However, only s_1 intersects with m_2 in the latter case. Hence MBR s_2, which do not have any point inside m_2, can be pruned. This reduces the I/O cost to 2 page reads (r, s_1) and the CPU cost to 16 comparisons. However, Optimization 3 states that a point in S is considered only if it is inside m_2. Therefore, we scan each point in s_1 once to find such points. These points are then compared to the points in r to update k-NNs. Thus the CPU cost reduces to 12 comparisons (4 for scanning s and 8 for comparison of the points in r with q_1 and q_2).

Partitioning: Optimization 3 is improved further by partitioning the MBR r along selected dimensions. Dimensions with high variances are selected for the partitioning. We start from the dimension with the highest variance. We split the MBR along this dimension into two MBRs, such that each resulting MBR contains the same number of objects. Each of these MBRs are then recursively partitioned along the dimension with the next highest variance recursively.

Partitioning improves the performance in two ways. First, since each of the partitions is smaller than the original MBR, the pruning distance ($d_{\max}$) along each dimension is reduced. This reduces the I/O cost. Second, without partitioning, an object in MBR s is compared to all the objects in r if the extended MBR of r contains it. However, with partitioning, an object in s is not compared to the objects in partitions whose extended MBR does not contain it. Thus, CPU cost is reduced by avoiding unnecessary comparisons. Note that as the number of partitions increases, the number of point-MBR comparisons increases. When the number of partitions becomes $O(t)$ (i.e., the number of objects per MBR), the number of such comparisons becomes $O(t^2)$. Thus, partitioning becomes useless. In our experiments, we partitioned along at most eight dimensions for the best performance.

In Figure 8, horizontal dimension is used to partition the MBR r into two partitions. v_1 and v_2 are the MBRs of these partitions having m_3 and m_4 as extended MBRs. MBR s_2, which do not have any points inside m_2, can be pruned. We scan each point in s_1 once to find the candidate points for the partitions v_1 and v_2. Only q_1 is present in the extended MBR of v_2, reducing the CPU cost to 10 (8 for comparing points in s_1 with v_1 and v_2 and 2 for comparing the points in v_2 with q_1).

Packing: The performance of R-Tree based methods can be improved by using $packing$ algorithms which group similar objects (objects within a close neighborhood) together. we employ the Sort-Tile-Recursive (STR) method [16] for packing the R*-Tree, built on the datasets. Let N be the number of d-dimensional objects in a dataset, B be the capacity of a node in R-Tree and let $P = \lceil \frac{N}{B} \rceil$. STR sorts objects according to the first dimension. Then the data is divided into $S = \lceil P^{\frac{1}{d}} \rceil$ slabs, where a slab consists of a run of $n.\lceil P^{\frac{d-1}{d}} \rceil$ consecutive objects from the sorted list. Now each slab is processed recursively using the remaining $d-1$ coordinates. It has been shown in [16] that for most types of data distributions STR-Ordering performs better than space-filling-curve based Hilbert-Ordering [11].

8 Experimental evaluation

We use two classes of datasets in our experiments.

Image datasets: Each of *Image1* and *Image2* contains 60-dimensional feature vectors of 34,433 satellite images. We have created two datasets from Image1 and Image2 by splitting each 60-dimensional vector into 30 two-dimensional vectors. Each of the resulting datasets contains 1,032,990 data points.

Protein structure datasets: Each of *Protein1* and *Protein2* contains 288,156 three-dimensional feature vectors for secondary structures of proteins from Protein Data Bank (`ftp://ftp.rcsb.org/pub/pdb`) as discussed in [6].

In addition to FA, FO, and FD, we have implemented three existing methods: sequential search (SS) and the R-tree-based NN method of Roussopoulos et. al., (RT) [15] and Mux-Join [5]. To implement the buffer restrictions into RT, we use half of the available buffer for R and the other half for S. In order to adapt these methods GNN(R, S, S', k, t), we performed a k-NN search for each object in R. We included SS in our experiments, as SS is better than many complicated NN methods for a broad set of data distributions [4]. We also obtained the source codes of GORDER [21] and RkNN [20], from their authors. However, at its current state, we found it impossible to restrict memory usage of GORDER to a desired amount. Therefore, we used GORDER in only one of our experiments where it was possible.

In all our experiments, we use $S' = S$ unless otherwise stated. We use 4 kB as the page size in all our experiments. We ran our experiments on an Intel Pentium 4 processor with 2.8 GHz clock speed.

8.1 Evaluation of Optimizations

In this section, we inspect the performance gain due to Optimizations 1, 2 and 3 and the improvements in Section 7. We perform GNN query by varying the size of S' from 0.5 % to 8 % of S, by selecting pages of S randomly. In this experiment, we use FD for

$k = 10$, $t =$ 3,000, and buffer size $= 10\,\%$ of the dataset size. The queries are run on the two dimensional image dataset.

Figure 9 displays the I/O and the running times for Optimizations 1, 2 and 3 on an unpacked dataset. According to our results, the main performance gain is obtained from Optimizations 2 and 3, yet there is a slight performance gain from Optimization 1. The reason that the Optimization 1 has a smaller impact can be explained as follows. t is only 0.3 % of the total number of objects in R. Thus, Optimization 1 can eliminate a page of S only if it is in the candidate set of less than 3 pages of R. The impact of Optimization 1 is larger when the ratio of the average number of candidate pages to t is lower. This happens when t is large or k is small. Optimization 2 has a high impact when S' is smaller. This is because fewer rows in PT have candidates in S' for small S'. Another way to obtain high filtering rate from Optimization 2 is to reduce the average number of candidate pages per row by choosing a small value for k. Optimization 3 effectively reduces the CPU and I/O cost for different sizes of S'. We can also see that for higher percentages of S, the impact of this optimization remains constant and is independent of the size of S'. This can be understood from the fact that for a fixed value of k and at higher percentages of S, every MBR $r \in R$ has same number of candidate MBRs from S. This results in constant reduction in CPU and I/O costs.

Figure 10 compares the performance gains on top of Optimizations 1, 2, and 3 (Unpartitioned algorithm) by partitioning the MBRs and by using the STR-method based packing algorithm. Here, packing is applied along with partitioning. Partitioning reduces the I/O cost up to factor of 3 and CPU costs by orders of magnitude. The tighter bounds of the extended MBRs of the partitions resulted in a reduction of the pruning distance. This explains the I/O and CPU performance gains from the partitioned algorithm. Packing utilizes the distribution of data and groups similar objects in MBRs that have common parent and a better organization of the R*-Tree index structure. This results in a lower value for the parameter f in FD and hence has better performance gains. Packing reduces the I/O cost up to 10 times and CPU cost by orders of magnitude faster than an unpartitioned algorithm. It outperforms partitioned algorithm by up to a factor of 2 and 6 in I/O and CPU costs respectively. From here on we will use all the optimizations in all of our methods.

Scheduling pages is known as paging problem [14]. Chan [7] proposed heuristic based $O((R_p + S_p)^2)$ algorithms (R_p and S_p are the number of pages in two datasets) for Index-based Joins. For large datasets, however, these heuristics are not efficient. An online scheduling algorithm is evaluated using *competitive analysis* [1]. In competitive analysis, an online algorithm is compared with an optimal off-line algorithm which knows all candidate pages in advance. An algorithm is *c-Competitive* if for all sequences of page requests, $C_A \leq c.\bar{C} + b$, where C_A is the cost of the given algorithm, $\bar{C}$ is the cost of the off-line algorithm, b is a constant, and c is the *competitive ratio*.

We compared the performance of our online methods with its off-line version, named $Oracle$. For each MBR $r \in R$, we provide $Oracle$, the set of MBRs from S such that every MBR in this set contains at least one k-NN of at least one object in r. We then optimize the number of I/Os of $Oracle$ using the heuristic discussed in Section 5. We also compute a lower bound to the optimal number of I/Os as the total number of pages in R and S. The purpose of this experiment is to observe how the I/O cost of our online methods compare to that of an off-line method and the minimum possible I/O cost. Table 1 compares the performance of Oracle with our methods.

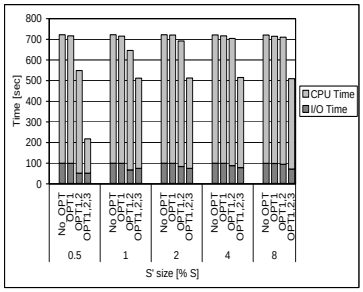


Fig. 9. The CPU and I/O time of FD with four different settings of Optimizations 1 and 2 (obtained by turning these optimizations on and off) on two-dimensional image datasets for different sizes of S'. Buffer size is 10 % of $R + S$, $k = 10$, and $t = 3{,}000$.

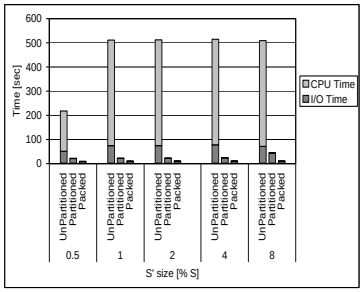


Fig. 10. The CPU and I/O time of FD with three different settings Unpartitioned, Partitioned, and Packed (along with partition) on two-dimensional image datasets for different sizes of S'. Buffer size is 10 % of $R+S$, $k = 10$, and $t = 3{,}000$.

Table 1. Number of Page Reads from FA, FD and FO and Optimal Page Reads on two-dimensional image datasets. $k = 10$, and $t = 100$.

Buffer Size (%)	5	10	20	40
Oracle	11245	10980	10244	10252
FO	14601	13425	12710	12393
FD	16706	13825	11702	10789
FA	155727	73238	23885	10328

Since each dataset has 5064 pages, we compute the lower bound to the number of disk I/Os as 10128 (5064+5064). The *competitive ratio* of FA is smallest for large buffer sizes (1.008 for 40% buffer) and for FO it is smallest for small buffer sizes (1.3 for 5% buffer). FD has a smaller *competitive ratio* (1.5 for 5% buffer to 1.05 for 40% buffer). We conclude that our methods perform very close to the off-line method.

8.2 Comparison of our methods

In this section, we compare FA, FO, and FD to each other for different parameter settings.

Evaluation of buffer size: Here, we compare the performance of FA, FO, and FD when the buffer size varies from 5 to 40 % of the total size of R and S. We use two-dimensional image dataset with $k = 10$ and $t = 500$.

Figure 11 shows the I/O time and the running time of our methods. For lower buffer sizes, FA retrieves all the candidate MBRs for every row and hence I/O cost takes up most of the total time. We can observe this from the performance of FA at buffer size 5 % and is dominated by the I/O cost. As buffer size increases, the cost of all three strategies drop since more pages can be kept in buffer at a time. For small buffer sizes FO has the lowest cost since it does not load unnecessary candidates. As buffer size increases, FA has the lowest cost since it keeps almost entire S in buffer. However, in all these experiments, the cost of FD is either the lowest or very close to the lower of FA and FO. This means that FD can adapt to the available buffer size.

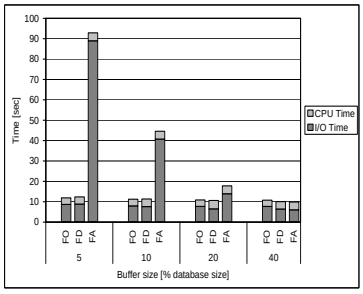


Fig. 11. The CPU and I/O time of FA, FO, and FD on two-dimensional image for different buffer sizes with $k = 10$ and $t = 500$.

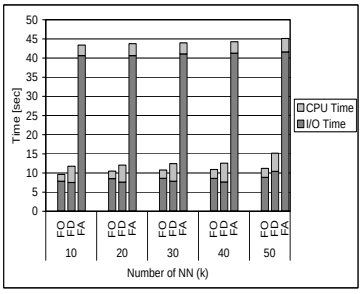


Fig. 12. The CPU and I/O time of FA, FO, and FD on the two-dimensional image, dataset for different values of k.

Table 2. Memory Usage and Running times (seconds) of GORDER on image dataset with varying grid sizes. FD runs in only 11.03 seconds for the same dataset using 20% buffer.

Grid Size	1000	500	200	100
Buffer Size (%)	175	108	88	85
Time (seconds)	305	535	1519	4259

Table 3. Running times (seconds) of FD and RkNn on image dataset with k = 10, 20, 30 , 40 and 50. using 10% buffer for 100 queries.

k	10	20	30	40
RkNN	2620	11750	84145	175495
FD	101	101.76	101.3	101.83

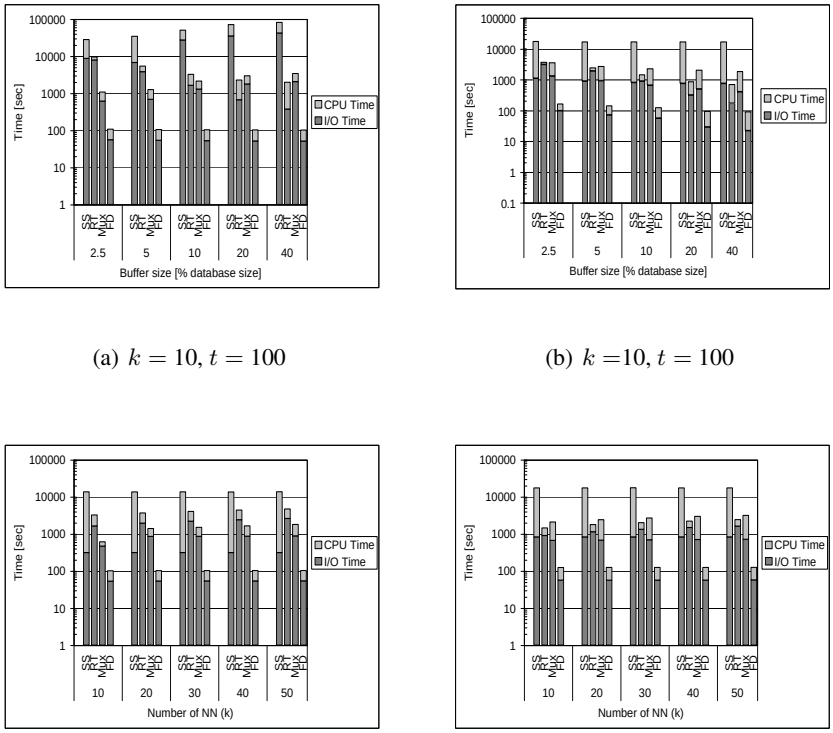
Evaluation of the number of NN: Our next experiment compares the performance of FA, FO, and FD for different values of k. We use 10 % buffer size and $t = 500$ for the two-dimensional image dataset.

Figure 12 presents the I/O and the running times. The costs of all these methods increase as k increases. For different values of k FO has the lowest cost and FA has the highest cost, since we use a small buffer size (10%). Even when it does not have the lowest cost, FD is very close to FO. This means that FD can adapt to the parameter k.

8.3 Comparison to existing methods

In this section, we compared FD to five existing methods SS, RT, Mux-Index, RkNN, and GORDER for different parameter settings. We used two-dimensional image and protein datasets in our experiments. Due to space limitations, we do not include theoretical comparison of FD to existing k-NN, RNN and ANN methods as the main intent of this paper is not to solve these problems. We present experimental results comparing FD with well known methods for these special cases.

Evaluation of buffer size: In this experiment set, we fixed the values of k and t, and vary the buffer size. We used the two-dimensional image dataset and k = 10. The running times of GORDER with different amounts of memory usage and that of FD with 1.6 MB memory were computed. We measured the actual memory usage of the methods using the *top* command of Linux. Although we set the buffer size (an input parameter to GORDER) to 20 % of the total dataset size, we observed that GORDER uses significant amount of memory (up to 175 % of the dataset size) for additional book keeping. In order to reduce the actual memory usage we ran GORDER with grid numbers 1000, 500, 200, and 100. However the actual memory usage of GORDER was always much larger than 20 % of the total dataset size (i.e., 8 MB). For different memory settings, the



(a) $k = 10, t = 100$

(b) k =10, $t = 100$

(c) Buffer size = 10 %,t =100

(d) Buffer size = 10 %, $t = 100$

Fig. 13. The CPU and I/O time of SS, RT, Mux and FD on (a) two-dimensional image, and (b) protein datasets for different buffer sizes. The CPU and I/O time of SS, RT, and Mux FD on (c) two-dimensional image and (d) protein datasets for different values of k.

running time of GORDER varied from 300 to 4000 seconds while for the same query, FD running times varied from 10 to 13 seconds (see Table 2. According to these experiments, FD runs an order of magnitude faster than GORDER even when it uses much smaller buffer. We found it impossible to reduce the actual memory usage of GORDER to 20 % at its current implementation. Therefore, in order to be fair, we do not include it in our remaining experiments.

Figures 13(a) and 13(b) show the I/O and the running times of SS, RT, Mux-index [5] and FD for different buffer sizes on two-dimensional image and protein datasets. We use $k = 10$ and t =100. FD is the fastest of the three methods in all settings. We can see that for small buffer sizes RT is dominated by I/O cost. As buffer size increases, CPU cost of RT dominates. Sequential scan is dominated by the CPU cost in all the experiments. The I/O cost of FD is a fraction of that of RT. FD also reduces CPU cost aggressively through Optimizations 1 to 3 and partitioning. In all the experiments, the total time of FD is less than the I/O time of RT or SS alone. Mux-Index is dominated by I/O costs in all experiments. This is because for each block in R it fills the buffer with blocks from S. Because of the nature of GNN queries, one needs to load pages multiple times while working with limited amount of memory, independent of the method used, naive (sequential scan) or more sophisticated (RT and Mux-Index). FD performs only the necessary leaf comparisons and uses the near optimal buffering

schedule, thus reduces both the CPU and I/O cost effectively.

Evaluation of the number NN: Here, we compare the performance of FD, SS, Mux, RkNN and RT for different values of k. We use 10 % buffer size and $t = 500$ for two-dimensional image and protein datasets. We evaluated the RkNN by querying for 100 random query points for different values of k.

Figures 13(c) and 13(d) present the I/O and the running times. The cost of SS is almost the same for all values of k. It increases slightly as k increases due to maintaining cost of the top k closest objects. The costs of RT, Mux and FD increase as k increases since their pruning power drops for large values of k. The running times of RT, Mux and FD do not exceed SS as k increases. FD runs significantly faster than others. Depending on the value of k, FD runs orders of magnitude faster than RT, SS and Mux. The I/O cost increases much slower for FD. This is because FD adapts to different parameter settings quickly to minimize the amount of disk reads. Table 3 present the running times of FD and RkNN for 100 query points. While the running time of RkNN increases at faster rate and is not scalable for higher values of k, the running times of FD, including the time taken for the creation of priority table for each k, for the same query set is almost constant and is order of magnitude faster than RkNN.

Evaluation of dataset size: In this experiment, we observe the performance of FD, SS, Mux, and RT for increasing dataset sizes. We create smaller datasets from the original two-dimensional image datasets by randomly choosing 50, 25, and 12.5 % of all the vectors. We fix the buffer size to 10 % of the original image dataset, $k = 10$, and $t = 500$.

Figure 14 shows the I/O and the running times. As R and S grows, the running time of FD increases almost linearly. This is because when both datasets are doubled, the average number of candidate pages per row in the PT stays almost the same. On the other hand, the total running time of SS increases quadratically since it has to compare all pairs of data points. The running time of RT is dominated by I/O cost and increases faster than that of FD and slower than that of SS. Like SS, the running time Mux increases quadratically since it fills the buffer with blocks from S and is dominated by I/O costs. Thus, the speedup of FD over SS, Mux and RT increases as dataset size increases. This means that our method scales better with increasing dataset size.

Evaluation of the number of dimensions: In this experiment, we observe the performance of FD, SS, Mux, and RT for increasing number of dimensions. We create datasets of $d = 2$, 4, 8, 16 dimensions by choosing the first d values of the feature vectors from the original 60-dimensional image datasets. We fix the buffer size to 10 % of the total size of R and S, $k = 10$, and $t = 500$. Figure 15 shows the I/O and the running times. As the number of dimensions increases, the running time of SS increases linearly. On the other hand, the running times of RT and Mux increases faster. This is also known as the *dimensionality curse*. For all the methods CPU time increases with the increase in dimension and is significantly larger for 16 dimensions. However even at 16 dimensions FD is 1.3 times faster than the sequential scan, up to 3.5 times faster than RT and up to 1.2 times faster than Mux-Index.

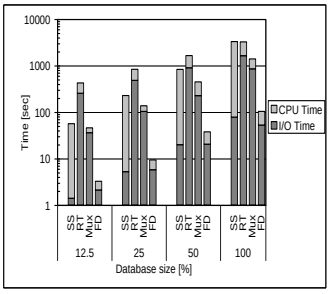


Fig. 14. The CPU and I/O time of SS, RT, Mux and FD on two-dimensional image datasets for varying database sizes. Buffer size is 10 % of the original image database, $k = 10$, and $t = 500$.

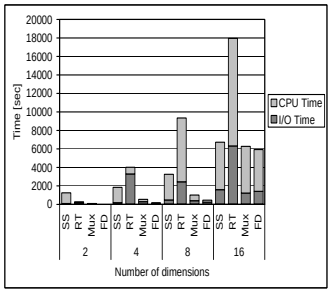


Fig. 15. The CPU and I/O time of SS, RT, and FD on two-dimensional image datasets for varying dimensionalities. Buffer size is 10 % of the database, $k = 10$, and $t = 500$.

9 Discussion

We considered the problem of detecting data broadness. We introduced a new database primitive called Generalized Nearest Neighbor (GNN) that expresses data broadness. We showed that the GNN queries can answer a much broader range of problems than the k-Nearest Neighbor query and its variants Reverse Nearest Neighbor query and All Nearest Neighbor query. Based on the available memory and the number of nearest-neighbors, either CPU or I/O time can dominate the computations. Thus, one has to optimize both I/O and CPU cost for this problem.

We proposed three methods to solve GNN queries. Our methods arrange two datasets into pages and compute a Priority Table for each page. Priority Table ranks the candidate pages based on their distance. Our first algorithm, FA, uses pessimistic approach. It fetches as many candidate pages as possible into available buffer. Our second algorithm, FO, uses optimistic approach. It fetches one candidate page at a time. Our third algorithm, FD, dynamically computes the number of pages that needs to be fetched by analyzing past experience. We also proposed three optimizations, Column Filter, Row Filter and Adaptive Filter to reduce the solution space of the priority table. We used packing and partitioning strategies which provided significant performance gains. These optimizations reduce the CPU cost of the k-NN searches and eliminates additional I/O costs by pruning the MBRs which do not have a k-NN.

According to our experiments, FA is best when the buffer size is large and FO is best when the buffer size is small. FD is the fastest method in most of the parameter settings. Even when it is not the fastest, the running time of FD is very close to that of the faster of FA and FO. FD is significantly faster compared to sequential scan and the standard R-tree based branch-and-bound k-NN solution to the GNN problem.

References

1. Susanne Albers. Competitive Online Algorithms. Technical Report LS-96-2, brics, September 1996.
2. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *International Conference on Management of Data (SIGMOD)*, pages 322–331, 1990.

3. S. Berchtold, B. Ertl, D.A. Keim, H.-P. Kriegel, and T. Seidl. Fast Nearest Neighbor Search in High-dimensional Space. In *International Conference on Data Engineering (ICDE)*, pages 209–218, 1998.
4. K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When Is "Nearest Neighbor" Meaningful? In *International Conference on Database Theory (ICDT)*, pages 217–235, 1999.
5. C. Böhm and F. Krebs. The k-Nearest Neighbour Join: Turbo Charging the KDD Process. *Knowledge and Information Systems (KAIS)*, 6(6), 2004.
6. O. Çamoğlu, T. Kahveci, and A. K. Singh. Towards Index-based Similarity Search for Protein Structure Databases. *Journal of Bioinformatics and Computational Biology (JBCB)*, 2(1):99–126, 2004.
7. Chee Yong Chan and Beng Chin Ooi. Efficient Scheduling of Page Access in Index-Based Join Processing. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 9(6):1005–1011, November/December 1997.
8. C. Ding and H. Peng. Minimum redundancy feature selection from microarray gene expression data. In *Computational Systems Bioinformatics Conference (CSB)*, pages 523–528, 2003.
9. G.R. Hjaltason and H. Samet. Ranking in Spatial Databases. In *Symposium on Spatial Databases*, pages 83–95, Portland, Maine, August 1995.
10. X. Huang and A. Madan. CAP3: A DNA Sequence Assembly Program. *Genome Research*, 9(9):868–877, 1999.
11. Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In *International Conference on Very Large Databases (VLDB)*, pages 500–509, 1994.
12. F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *International Conference on Management of Data (SIGMOD)*, pages 201–212, 2000.
13. F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast Nearest Neighbor Search in Medical Databases. In *International Conference on Very Large Databases (VLDB)*, pages 215–226, India, 1996.
14. T. H. Merrett, Yahiko Kambayashi, and H. Yasuura. Scheduling of Page-Fetches in Join Operations. In *International Conference on Very Large Databases (VLDB)*, pages 488–498, 1981.
15. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *International Conference on Management of Data (SIGMOD)*, San Jose, CA, 1995.
16. Mario Lopez Scott Leutenegger and Jeffrey Edgington. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *International Conference on Data Engineering (ICDE)*, pages 497–506, 1997.
17. B. Seeger. An analysis of schedules for performing multi-page requests. *Information Systems*, 21(5):387–407, 1996.
18. T. Seidl and H.P. Kriegel. Optimal Multi-Step k-Nearest Neighbor Search. In *International Conference on Management of Data (SIGMOD)*, 1998.
19. I. Stanoi, M. Riedewald, D. Agrawal, and A.E. Abbadi. Discovery of Influence Sets in Frequently Updated Databases. In *International Conference on Very Large Databases (VLDB)*, pages 99–108, 2001.
20. Y. Tao, D. Papadias, and X. Lian. Reverse kNN Search in Arbitrary Dimensionality. In *International Conference on Very Large Databases (VLDB)*, 2004.
21. C. Xia, H. Lu, B.C. Ooi, and J. Hu. GORDER: An Efficient Method for KNN Join Processing. In *International Conference on Very Large Databases (VLDB)*, 2004.
22. C. Yang and K.-I. Lin. An Index Structure for Efficient Reverse Nearest Neighbor Queries. In *International Conference on Data Engineering (ICDE)*, pages 485–492, 2001.