Nearest Window Cluster Queries

Chen-Che Huang, Jiun-Long Huang, Tsung-Ching Liang, Jun-Zhe Wang,
Wen-Yuah Shih and Wang-Chien Lee[†]
Department of Computer Science
National Chiao Tung University, Hsinchu, Taiwan, ROC
[†]Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802, USA
E-mail: cchuang.cs95g@nctu.edu.tw, jlhuang@cs.nctu.edu.tw, dy93_@hotmail.com,
jzwang@cs.nctu.edu.tw, wen21318@hotmail.com, wlee@cse.psu.edu

ABSTRACT

In this paper, we study a novel type of spatial queries, namely Nearest Window Cluster (NWC) queries. For a given query location q, NWC (q,l,w,n) retrieves n objects within a window of length l and width w, where the distance between the query location q to these n objects is the shortest. To facilitate efficient NWC query processing, we identify several properties and accordingly develop an NWC algorithm. Moreover, we propose several optimization techniques to further reduce the search cost. To validate our ideas, we conduct a comprehensive performance evaluation using both real and synthetic datasets. Experimental results show that the proposed NWC algorithm, along with the optimization techniques, is very efficient under various datasets and parameter settings.

Keywords: Nearest window cluster query, spatial query processing, location-based service, spatial database.

1. INTRODUCTION

Spatial queries have received tremendous attention from the research community in past decades. In the past several years, owing to the emerging location-based services, a number of new spatial queries have been proposed to meet various application needs [16] [11][7][13][22][9]. However, many interesting/important applications still are not well supported by existing spatial queries. The following is an example.

- Suppose Bob is attending a business meeting in a foreign city. He wishes to buy some souvenirs for his family. With only a rough idea of what to buy (e.g., some local-brand clothes), he would like to search for some, say n, nearby clothes shops which are close to each other in a small area so he can walk around these clothes shops to find the souvenirs.

In this example, Bob aims to find the nearest area with sufficient choices (i.e., n clothes shops) clustered in the area so he can go around to compare products and prices and even have fun doing some bargaining. Figure 1 illustrates the example above where each

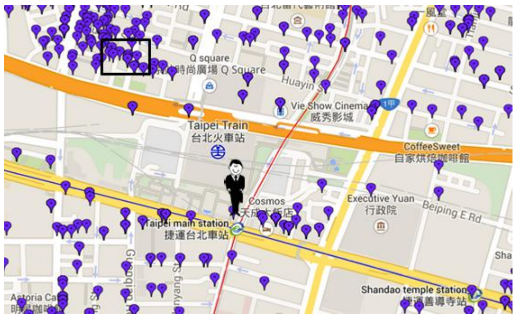


Figure 1: Example of a nearest window cluster query.

bubble indicates a clothes shop. Ideally, a location-based service would be able to suggest the clothes shops within the window back to Bob. Unfortunately, existing spatial queries such as kNN, trip-planning queries [15] and collective spatial keyword queries [2] do not meet Bob's need effectively and efficiently. To the best of our knowledge, no previous work on spatial queries address the query problem arising in the example scenario.

In this paper, we propose a novel type of spatial queries, namely *Nearest Window Cluster (NWC)* queries that finds a clustered of objects located in a spatial window nearest to a query point, e.g., the query issuer's location. Given a query point q, window length l and window width w, and the number of objects to find n, NWC (q,l,w,n) returns n objects located within a window of length l and width w, where the *distance* from these n objects to q is the shortest.[1]

Two main challenges arise in processing the NWC queries. First, while the locations of data objects are given, the locations of *qualified* windows are unknown in advance.[2] Second, the number of qualified windows may be huge. To address these challenges, we identify several properties that allow us to find qualified windows quickly. Accordingly, we develop an NWC algorithm that iteratively finds the nearest qualified window to return the objects within the qualified window. To facilitate efficient visits of data objects, we adopt R-tree to index the data objects.

Observing that the bottleneck of the NWC algorithm lies in finding the nearest qualified window, we propose four optimization techniques, namely *search region reduction (SRR)*, *distance-based pruning (DIP)*, *density-based pruning (DEP)* and *incremental window query processing (IWP)* to accelerate searching the nearest

[1]We will discuss distance measures in Section 2.
[2]A window is considered as qualified if it contains n objects.

qualified window, thereby improving the efficiency of the NWC algorithm. The SRR technique takes advantage of the distance of the best objects found so far to shrink the search regions of qualified windows and prune some objects from further processing. The DIP technique uses the distance of the best objects found so far to safely prune the index nodes distant from the query location q, thereby reducing the I/O cost. Inspired by the *clustering effect* of spatial objects [1] (e.g., certain objects such as clothes shops are usually clustered in some hot areas), the DEP technique maintains a density grid of the whole object space that records the number of objects in each grid cell. With the density grid, the DEP technique is able to prune the index nodes with insufficient objects and to avoid redundant window queries incurred during query processing. Finally, to alleviate the cost of window queries generated by the NWC algorithm, the IWP technique inserts *backward* and *overlapping pointers* into the leaf nodes and some intermediate nodes of the R-tree. With these pointers, fewer index nodes are involved for window queries, thereby achieving better performance.

The rest of this paper is organized as follows. In Section 2, we review related work and present the problem definition. In Section 3, we elaborate the properties of NWC queries and develop the NWC algorithm based on these properties. In addition, four optimization techniques are proposed to accelerate NWC query processing. The performance of the NWC algorithm is analyzed in Section 4, while the experimental results of the NWC algorithm are reported in Section 5. Finally, we conclude this paper in Section 6.

2. PRELIMINARIES

2.1 Problem Formulation and Transformation

Based on the application scenario discussed earlier, we consider a set of static data objects, denoted as P, in two-dimensional Euclidean space.[3] The nearest window cluster query is formally defined as below.

Definition 1 (Nearest Window Cluster (NWC) Query) Given a query point q, a spatial window area specified by length l and width w, and the number of data objects n, a nearest window cluster query $NWC(q,l,w,n)$ aims to retrieve n objects satisfying the following criteria:

1. these n objects are clustered within a spatial window of length l and width w, and
2. the *distance* from the window with these n objects reside to q is the shortest among all other windows with n objects satisfying (1).

To facilitate our discussion, we define the notion of qualified window with respect to an NWC query as follows.

Definition 2 (Qualified Window) Given an NWC query (q,l,w,n), a *qualified window*, denoted as *qwin*, is a window of length l and width w that contains data objects $S_{qwin} = \{p_1, p_2, \dots, p_{|S_{qwin}|}\} \subseteq P$, where $|S_{qwin}| \geq n$.

Let $MINDIST(q,qwin)$ be the distance from q to the closest point in the qualified window *qwin* covering $\{p_1, p_2, \dots, p_n\}$. As the distance measure mentioned above aims to measure the distance between q and these n objects, denoted as $\{p_1, p_2, \dots, p_n\}$, we consider the following in our algorithm.

[3]We focus on 2D data objects in accordance with real-world applications. The proposed algorithms could be easily adjusted to three dimensional space.

- Minimum distance:
$$dist_{min}(q,\{p_1,p_2,\dots,p_n\}) = \min_{i=1,2,\dots,n} dist(q,p_i) \quad (1)$$
- Maximum distance:
$$dist_{max}(q,\{p_1,p_2,\dots,p_n\}) = \max_{i=1,2,\dots,n} dist(q,p_i) \quad (2)$$
- Average distance:
$$dist_{avg}(q,\{p_1,p_2,\dots,p_n\}) = \frac{1}{n}\sum_{i=1}^{n} dist(q,p_i) \quad (3)$$
- Nearest window distance:
$$dist_{nearest}(q,\{p_1,p_2,\dots,p_n\}) = \min_{\forall qwin \in qwins} \big(MINDIST(q,qwin)\big), \quad (4)$$
where *qwins* is a set of all qualified windows containing $\{p_1, p_2, \dots, p_n\}$.

With the above definitions, we can transform the problem of NWC query processing as below.

Problem Transformation. When $MINDIST(q,qwin)$ is always smaller than or equal to $dist(q,\{p_1,p_2,\dots,p_n\})$, the processing of an NWC query can be performed by incrementally finding the next nearest qualified window to q and using the distance of the best objects found so far, denoted as $dist_{best}$, to prune the search space until no better qualified window is found.

Specifically, we can solve the NWC query by the following steps.

Step 1: Set $dist_{best}$ to ∞ and set $objs$ to $\emptyset$.

Step 2: Find the nearest qualified window *qwin*.

Step 3: If *qwin* is found and $MINDIST(q,qwin) < dist_{best}$, perform Steps 4-6. Otherwise, go to Step 7.

Step 4: Let $\{p_1,p_2,\dots,p_n\}$ be the n objects in *qwin* of the shortest distance to q.

Step 5: If $dist(q,\{p_1,p_2,\dots,p_n\}) < dist_{best}$, set $objs$ to $\{p_1,p_2,\dots,p_n\}$ and $dist_{best}$ to $dist(q,\{p_1,p_2,\dots,p_n\})$.

Step 6: Find the next nearest qualified window *qwin* and go to Step 3.

Step 7: Return $objs$.

Clearly, $MINDIST(q,qwin)$ is always smaller than or equal to minimum, maximum, average and nearest window distances between q and $\{p_1,p_2,\dots,p_n\}$ (see Equations (1), (2), (3) and (4), respectively). As the bottleneck of the above procedure is in finding the nearest qualified window, we will focus on nearest qualified window search for the rest of this paper. The advantages of using nearest qualified window search to process an NWC query are twofold. First, the above procedure can be used for other distance measures as long as $MINDIST(q,qwin)$ can be used as the lower bound of the employed distance measures. Second, the properties of nearest qualified window search can be used to efficiently process NWC queries.

Table 1: List of used symbols

Symbol	Description
P	Data object set
T_P	R-tree on P
q	query location
n	the desired number of data objects
SR_p	search region of object p
$qwin_p$	best qualified window of p
S_{qwin_p}	the set of the data objects inside $qwin_p$
$\lvert S_{qwin_p} \rvert$	the cardinality of S_{qwin_p}
$dist(q,\{p_1,p_2,\ldots,p_n\})$	the distance between q and $\{p_1,p_2,\ldots,p_n\}$
$objs$	the best objects found so far
$dist_{best}$	the distance of the best objects found so far

2.2 Related Work

In the past two decades, a large number of spatial queries have been proposed and studied by researchers in the database community. Here we focus on the variants of NN queries due to their relevance to our work. Constrained NN [8] queries find the nearest neighbor(s) constrained to a specific region instead of the entire data space. Nearest surround (NS) queries [13] consider the object orientation and retrieve the nearest neighbors at different angles with respect to the query location q. A reverse NN (RNN) query [12][21] finds all the data objects with q as their nearest neighbor. RkNN queries [19][3] search for all the data objects that have q as one of their k nearest neighbors. The ranked RNN (RRNN) query [14] allows to identify and rank the t data objects most influenced by q. Different from typical RNN queries, RFN [22] queries find the objects that have q as their *furthest* neighbor. With RFN queries, a plant producing hazardous gas could be constructed at a point with fewest residents being affected.

Group NN (GNN) queries [16] (also known as aggregate NN [17] queries) retrieve the data object(s) with the smallest sum of distance to Q where Q is the set of query points. GNN queries are useful when a group of friends intend to find a meeting restaurant with the minimum distances to them. Group nearest group (GNS) queries [6], a generic version of GNN queries, return more objects for gathering to reduce the traveling costs of users. With data object set P and target object set Q, optimal-location-selection (OLS) queries [9] find $q \in Q$ outside a specific region R with maximal optimality where the optimality metric is determined by to the number of data objects in R and the accumulated distances to q. OLS query is useful for applications like optimal lifeguard station selection. Range NN (RangeNN) queries [11][5] search for the NNs for every point in a rectangle. It could be used to offer location privacy and computation saving. Given a specified window size, the maximizing range sum (MaxRS) problem [4] is to find the window *win* with the largest sum of weights of all objects within *win* among all candidate windows of the specified window size. Although bearing similarity to the proposed NWC queries, the MaxRS problem does not consider any query location and thus is naturally different from the proposed NWC query.

3. PROCESSING NEAREST WINDOW CLUSTER QUERIES

In this section, we elaborate the NWC query processing based on the procedure of nearest qualified window search mentioned in Section 2.1. First, we identify some unique properties of the nearest qualified windows in Section 3.1. Based on these properties, we develop an NWC algorithm to find the nearest qualified window of the NWC query in Section 3.2. To improve the efficiency of NWC search, in Section 3.3, we further propose four optimization techniques, including (i) *search region reduction*, (ii) *distance-based pruning*, (iii) *density-based pruning* and (iv) *incremental window query processing* to reduce the search cost. To facilitate better readability, the symbols used throughout this paper are listed in Table 1.

3.1 Properties of the Nearest Qualified Windows

In this section, we introduce the following properties regarding the nearest qualified window to facilitate efficient NWC search.

Lemma 1 The nearest qualified window of an NWC query, or one of its equivalent qualified windows, has at least one object on one vertical edge and at least one object on one horizontal edge.

PROOF. The proof is omitted for the interest of space. □

A qualified window *qwin* is said to be *generated by* a data object p when p is on at least one edge of *qwin*. Therefore, we use data objects as the basis to *generate* qualified windows and consider only those qualified windows generated by some data objects for NWC query processing. In other words, we consider only those qualified windows generated by data objects on one vertical or horizontal edge based on the relative position of q and object p. Furthermore, based on Lemma 1, we can utilize the lying quadrant of p (with q as the origin) to determine that we merely need to evaluate the qualified windows with p on the right or left edge (top or bottom edge) by the following two observations.

1. Consider a qualified window, say *qwin*, generated by data object p on one of the vertical edges (denoted by e_R or e_L). When p is in the first or fourth quadrant with respect to the origin q, p must be on the right edge e_R of *qwin*; when p is in the second or third quadrant, p must be on the left edge e_L of *qwin*.

2. Consider a qualified window, say *qwin*, generated by data object p on one of the horizontal edges (denoted by e_T or e_B). When p is in the first or second quadrant with respect to the origin q, p must be on the top edge e_T of *qwin*; when p is in the third or fourth quadrant, p must be on the bottom edge e_B of *qwin*.

3.2 NWC Algorithm

With the above properties and the steps discussed in Section 2.1, we present the NWC algorithm which incrementally finds the next qualified window to q and uses the distance of the best object found so far ($dist_{best}$) to prune the search space until no better qualified window is found. Since the bottleneck of the NWC algorithm lies in finding the nearest qualified window, we focus on nearest qualified window search. The idea of the NWC algorithm is as follows. The NWC algorithm visits all data objects based on their distance to the query location q in ascending order. To facilitate efficient visits of data objects, we adopt R-tree to index the data objects. When visiting an object p, the NWC algorithm creates the *search region* for object p (denoted as SR_p) to cover all qualified windows generated by p, and then find all qualified windows generated by p (i.e., qualified windows within SR_p). When a qualified window, say $qwin_p$, is discovered and $MINDIST(q, qwin_p) < dist_{best}$, the NWC algorithm retrieves the n objects, say $\{p_1, p_2, \cdots, p_n\}$, in $qwin_p$ of the shortest distance to q and checks whether $dist(q, \{p_1, p_2, \cdots, p_n\}) < dist_{best}$. If so, the NWC algorithm sets $objs$ and $dist_{best}$ to $\{p_1, p_2, \cdots, p_n\}$ and $dist(q, \{p_1, p_2, \cdots, p_n\})$, respectively. The

NWC algorithm repeats the above steps until no better qualified window is found.

We now discuss how to build SR_p covering all qualified windows generated by p. According to the observations in Section 3.1, p has to be on the vertical and horizontal edges in order to guarantee all potential qualified windows generated by p are contained in SR_p. In fact, we consider p only on either the vertical or the horizontal edge for building SR_p because the other case is handled naturally while processing other objects. Specifically, if SR_p is built on the condition that p is on the vertical edge, the other potential qualified windows on the condition that p is on the horizontal edge will be contained within the search region of another object p' on the vertical edge (i.e., $SR_{p'}$). To avoid redundant computation, when visiting data object p, we consider p only on the vertical edge for SR_p construction. Due to space limitation, we mainly explain the case where p is in the first quadrant with respect to the origin q (i.e., p is on the right edge) since the other cases are able to be addressed similarly.

With p on the right edge, the four vertexes v_1, v_2, v_3 and v_4 of SR_p are defined below, where x_p and y_p are the x and y coordinates of p, respectively.

$$x_{v_1} = x_p - l \quad y_{v_1} = y_p - w \quad x_{v_2} = x_p \quad y_{v_2} = y_p - w$$
$$x_{v_3} = x_p \quad y_{v_3} = y_p + w \quad x_{v_4} = x_p - l \quad y_{v_4} = y_p + w$$

It is obvious that all windows generated by p are inside SR_p. Therefore, the NWC algorithm is able to evaluate only data objects inside SR_p for identifying the qualified windows generated by p. To do so, the NWC algorithm retrieves all the objects within SR_p by issuing a window query with SR_p as the query window. To efficiently obtain the qualified windows generated by p, the NWC algorithm first reorders the objects in S_{SR_p} based on their y coordinates in ascending order and then skips the objects with y coordinates lower than p. The data objects below p are skipped because their associated qualified windows would not contain p. Finally, the NWC algorithm sequentially visits each object remaining in S_{SR_p}, say p', to consider the window, say $qwin_p$, with p on the right edge and p' on the top edge for each p'.

When a window $qwin_p$ is considered, the NWC algorithm evaluates whether $qwin_p$ is qualified. If the number of objects within $qwin_p$ is smaller than n (i.e., $|S_{qwin_p}| < n$), $qwin_p$ is not qualified and the NWC algorithm skips $qwin_p$. When $qwin_p$ is qualified, (i.e., $|S_{qwin_p}| \geq n$), the NWC algorithm retrieves the n objects, say $\{p_1, p_2, \cdots, p_n\}$, in $qwin_p$ of the shortest distance to q. If $dist(q, \{p_1, p_2, \cdots, p_n\}) < dist_{best}$, the NWC algorithm sets $dist_{best}$ and $objs$ to $dist(q, \{p_1, p_2, \cdots, p_n\})$ and $\{p_1, p_2, \cdots, p_n\}$, respectively. Otherwise, the NWC algorithm skips $qwin_p$ and considers another window generated by p.

We use the example in Figure 2 to illustrate the process of identifying qualified windows in SR_p. Let's consider an intermediate step, where p_5 is the candidate data object under examination. To find $qwin_{p_5}$, the NWC algorithm first builds SR_{p_5} based on the residing quadrant of p_5 to q and retrieves all data objects within SR_{p_5}. Since p_5 is in the first quadrant, the NWC algorithm sorts all data objects within SR_{p_5} based on their y coordinates in ascending order. As shown, the y coordinate of p_4 is smaller than that of p_5, so the NWC algorithm skips p_4 and sequentially considers p_5, p_6, and p_7 on the top edge. Suppose that the desired number of objects in a qualified window is three (i.e., $n = 3$). When p_5 is considered, the window with p_5 on the right and top edges is not qualified since this window contains only two objects. When p_6 is evaluated, the window with p_5 on the right edge and p_6 on the top edge is set to $qwin_{p_5}$ since this window contains three objects. The NWC algorithm retrieves the three objects of the shortest distance to q from $qwin_{p_5}$ (i.e., $\{p_4, p_5, p_6\}$) and checks whether $dist(q, \{p_4, p_5, p_6\}) < dist_{best}$. If so, $dist_{best}$ and $objs$ are set to $dist(q, \{p_4, p_5, p_6\})$ and $\{p_4, p_5, p_6\}$, respectively. Then the NWC algorithm continues to find the next window in SR_p until all windows in SR_p have been evaluated.

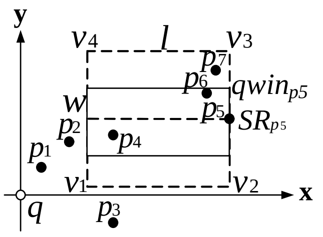


Figure 2: Discover $qwin_p$ from SR_p.

3.3 Optimization Techniques

While being able to answer NWC queries, the NWC algorithm suffers from costly and redundant evaluations of objects and index nodes. In light of this, we design the following optimization techniques to mitigate the cost of NWC search.

- *Search region reduction (SRR)*: The search region reduction technique exploits the distance between q and the best objects found so far (i.e., $dist_{best}$) to reduce the search regions of the remaining data objects, thereby saving the cost of qualified window discovery. Besides, with $dist_{best}$, SRR tries to exclude those objects that are unlikely to create closer qualified windows to q, eliminating the redundant evaluation of those objects.

- *Distance-based pruning (DIP)*: The distance-based pruning technique takes advantage of $dist_{best}$ to save the access to the index nodes that are too distant to create closer qualified windows, thereby achieving reductions in I/O cost.

- *Density-based pruning (DEP)*: We propose to build a density grid that maintains the number of objects residing in each grid cell. With the density grid, we propose a density-based pruning technique to prune the index nodes with insufficient objects (compared with the numbers of objects requested by NWC queries) to eliminate unnecessary I/O cost. In addition, DEP is able to prune some redundant window queries which do not produce any qualified window.

- *Incremental window query processing (IWP)*: To reduce the I/O cost to process the window queries generated by the NWC algorithm, we propose to enhance the R-tree by inserting *backward pointers* and *overlapping points* into the leaf nodes and some intermediate nodes, respectively. We design the incremental window query processing technique to use these backward pointers and overlapping pointers to efficiently process these window queries with less I/O costs.

The details of these optimization techniques are described in the following subsections.

3.3.1 Search Region Reduction

To identify each qualified window $qwin_p$ generated by object p, the NWC algorithm issues a window query with the search region

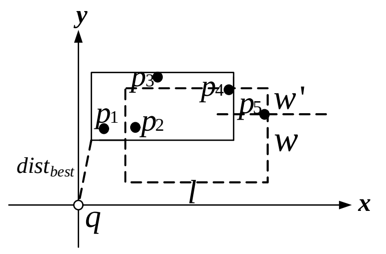


Figure 3: Reduced SR_{p_5} with $dist_{best}$.

SR_p of p as the query window. Obviously, the smaller SR_p is, the lower the I/O cost is. Thus, we propose the search region reduction technique (abbreviated as SRR) to 1) avoid evaluations of unnecessary search regions or 2) reduce the sizes of the search regions by exploiting the following two observations.

- Object p may be so distant to q that all qualified windows generated by p having distances greater than $dist_{best}$. For such object p, building SR_p is unnecessary and thus no window query is issued.
- The qualified windows in the specific portions of SR_p may never have a distance smaller than $dist_{best}$. Thus, SR_p can be shrunk, leading to smaller query windows.

With $dist_{best}$, when processing object p, SRR first checks whether the creation of SR_p is necessary based on x_p or y_p. Specifically, let the coordinates of the bottom-left vertex of SR_p, say v_1, be (x_{v_1}, y_{v_1}). When the distance from q to v_1 is greater than $dist_{best}$, there is no need to build SR_p since no qualified window generated by p will be closer to q than $dist_{best}$. Thus, the reduced search region of p, denoted as SR'_p, is set to be empty. Otherwise, the building of SR_p is necessary. Then, SRR tries to utilize $dist_{best}$ to shrink SR_p into a smaller search region SR'_p so that the distance of each qualified window in SR'_p is shorter than $dist_{best}$. The coordinates of the four vertices of SR'_p are calculated below.

$$\begin{array}{llll} x'_{v_1} = x_p - l & y'_{v_1} = y_p - w & x'_{v_2} = x_p & y'_{v_2} = y_p - w \\ x'_{v_3} = x_p & y'_{v_3} = y_p + w' & x'_{v_4} = x_p - l & y'_{v_4} = y_p + w' \end{array}$$

It is clear that w' is the maximal value making the following two equations satisfied.

$$0 \le w' \le w \tag{5}$$

$$(x_p - l - x_q)^2 + (y_p + w' - w - y_q)^2 \le dist(q, qwin_{best})^2 \tag{6}$$

Equation (5) indicates that p should be within SR'_p, while Equation (6) indicates that the minimum distance from q to each window of length l and width w in SR'_p should be shorter than $dist_{best}$. According to Equations (5) and (6), the value of w' is

$$w' = \min\left(w, \sqrt{dist_{best}^2 - (x_p - l - x_q)^2} - (y_p - w - y_q)\right).$$

In Figure 3, SSR helps to reduce SR_{p_5} with w' being only

$$\sqrt{dist_{best}^2 - (x_{p_5} - l - x_q)^2} - (y_{p_5} - w - y_q).$$

3.3.2 Distance-Based Pruning

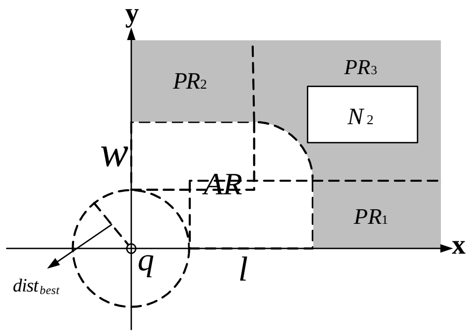


Figure 4: Node N_2 can be safely pruned based on $dist_{best}$.

In addition to reducing search regions of objects, $dist_{best}$ is able to be used to safely prune some index nodes as long as all qualified windows created by any object inside these pruned index nodes are guaranteed to be of distances to q greater than $dist_{best}$. For example, index node N_2 in Figure 4 can be safely pruned because any qualified window created by any object inside N_2 is unlikely to have the distance to q smaller than $dist_{best}$. Based on this observation, we propose the distance-based pruning technique (abbreviated as DIP) to facilitate the pruning of unvisited index nodes. DIP defines a *pruning region* (denoted as PR) based on $dist_{best}$, l, and w. If an unvisited index node N is completely inside PR, N is able to be safely pruned without being visited. With $dist_{best}$, PR is defined as below.

$$\begin{array}{rcl} PR_1 & = & \{(x,y) | x \ge x_q + dist_{best} + l, y_q \le y \le y_q + w\} \\ PR_2 & = & \{(x,y) | x_q \le x \le x_q + l, y \ge y_q + dist_{best} + w\} \\ PR_3 & = & \{(x,y) | x \ge x_q + l, y \ge y_q + w \\ & & \setminus (x - (x_q + l))^2 + (y - (y_q + w))^2 \le dist_{best}^2\} \\ PR & = & PR_1 \cup PR_2 \cup PR_3 \end{array} \tag{7}$$

As defined in Equation (7), PR is composed of three subregions: PR_1, PR_2, and PR_3. PR_1 guarantees that each qualified window generated by each object in PR_1 has the vertical distance to q greater than $dist_{best}$, while PR_2 ensures that the horizontal distance between q and each qualified window generated by each object in PR_2 is greater than or equal to $dist_{best}$. PR_3 excludes the region where an object can be used to generate a qualified window with the distance to q smaller than $dist_{best}$. Hence, if index node N is totally contained within PR, no closer qualified window can be generated by each object in N and thus N can be safely pruned to reduce I/O cost.

3.3.3 Density-Based Pruning

The spatial clustering effect in practice leads objects like clothes shops to be clustered in certain areas. Thus, an index node may be so sparse that all windows generated by each object in the index node are not qualified. With this observation, we devise the density-based pruning technique (abbreviated as DEP) to save I/O cost by avoiding visits to sparse index nodes. To facilitate DEP, the whole object space is divided into a $g_d \times g_d$ density grid and each grid cell is associated with the number of objects within the cell. When processing an index node, DEP first extends the MBR of the index node and checks whether the summation of the objects in the grid

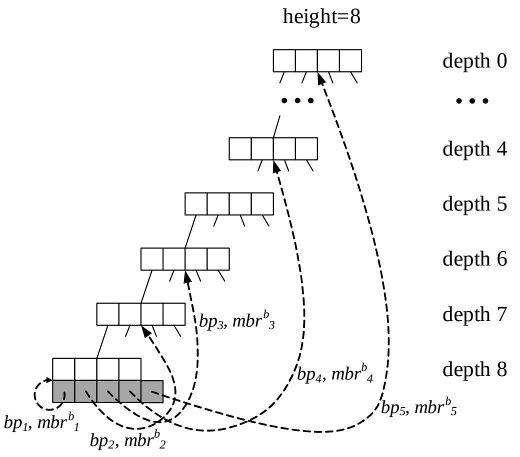


Figure 5: An illustrative example of backward pointers

cells intersecting the extended MBR is smaller than n. If so, this index node will be pruned. We now describe the method to extend an MBR in the first quadrant with respect to the origin q, and the other cases can be handled in a similar manner. Suppose that the counterclockwise order of the four vertices of the MBR is v_1, v_2, v_3 and v_4, and v_1 is the bottom-left vertex of the MBR. The MBR can be *extended* as follows to ensure that all windows generated by each object within the MBR must be within the extended MBR.

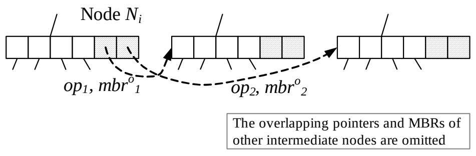
$$\begin{array}{llll} x'_{v_1} = x_{v_1} - l & y'_{v_1} = y_{v_1} - w & x'_{v_2} = x_{v_2} & y'_{v_2} = y_{v_2} - w \\ x'_{v_3} = x_{v_3} & y'_{v_3} = y_{v_3} + w' & x'_{v_4} = x_{v_4} - l & y'_{v_4} = y_{v_4} + w' \end{array}$$

Besides, it is possible that the search region of an object does not contain enough objects to generate any qualified window. DEP also utilizes this observation to eliminate the window queries of such objects with the aid of the density grid. Specifically, before issuing a window query for the currently processing object p, DEP checks whether the summation of the objects in the grid cells intersecting the search region SR_p is smaller than n. If so, DEP cancels the window query since SR_p never contains any qualified window.

3.3.4 Incremental Window Query Processing

As mentioned in Section 3.2, the NWC algorithm issues a window query with query window SR_p (SR'_p when SRR is used) to identify the qualified windows generated by object p. With traditional window query processing, solving a window query requires to access the R-tree from root node to some leaf nodes. However, we observe that some index nodes in the R-tree, especially the index nodes close to the root node, are usually unnecessary to visit when processing the window queries issued by the NWC algorithm. In view of this, we propose to add some *backward pointers* into each leaf node of the R-tree and some *overlapping pointers* into the nodes pointed by backward pointers. Based on backward and overlapping pointers, we design an incremental window query processing technique (abbreviated as IWP) to allow window queries to be processed from intermediate nodes instead of root node, thereby reducing the I/O cost.

Consider an R-tree with height h. Each leaf node is with depth h. Suppose that there are r backward pointers (denoted as (bp_1, mbr^b_1), $(bp_2, mbr^b_2), \ldots, (bp_r, mbr^b_r)$) for each leaf node. It is obvious that SR'_p is very likely to be totally covered by the intermediate nodes



Node N_i
op_1, mbr^o_1
op_2, mbr^o_2
The overlapping pointers and MBRs of other intermediate nodes are omitted

Figure 6: An illustrative example of overlapping pointers

close to the leaf node containing p. Thus, inspired by Exponential Index [20], for a leaf node s, the backward pointers are set by the following rules.

1. The first backward pointer bp_1 points to s.
2. bp_i, where $1 < i < r$, points to the ancestor of s with depth $h - 2^{i-2}$.
3. The last backward pointer bp_r points to the root node.
4. mbr^b_i, where $1 \leq i \leq r$, is the MBR of the node pointed by bp_i.

According to the third rule, r is the smallest integer making the following equation true.

$$h - 2^{r-2} \leq 0$$

Thus, we can obtain that $r = \lceil \log_2 h + 2 \rceil$.

Figure 5 shows an illustrative example of backward pointers. Only part of an R-tree with height eight is shown for better readability. Since $h = 8$, each leaf node is of $r = \lceil \log_2 8 + 2 \rceil = 5$ backward pointers (i.e., gray squares in Figure 5). bp_1 points to the leaf node, while bp_2, bp_3, bp_4 and bp_5 point to the ancestors of the leaf node with depth 7, 6, 4 and 0, respectively.

Since most variants of R-tree do not guarantee the MBRs of intermediate nodes in the same depth level to be non-overlapped, using only backward pointers to incrementally process window queries may lead to wrong query results. Therefore, some overlapping pointers are needed for the nodes pointed by backward pointers (except the root node) to facilitate correct incremental window query processing. We use the example in Figure 6 to illustrate the overlapping pointers. Since overlapping with two other nodes with the same depth, the node N_i is of two overlapping pointers ((op_1, mbr^o_1), (op_2, mbr^o_2)) where op_j is the pointer pointing to the j-th intermediate node with the same depth as N_i and overlapping with N_i, and mbr^o_j is the MBR of the node pointed by op_j.

With backward pointers and overlapping pointers, a window query can be incrementally processed as follows. When an object p is inserted into the priority queue PQ, the backward pointers of the leaf node where p is stored are also inserted into PQ along with p. When processing the window query with SR'_p as the query window, instead of searching from the root node of the R-tree, IWP retrieves the corresponding backward pointers, finds the smallest value of i so that SR'_p is totally covered by mbr^b_i, and searches from the node pointed by bp_i. In addition, for each overlapping pointer op_j of the node pointed by bp_i, IWP also executes the window query from the node pointed by op_j when mbr^o_j overlaps with the query window.

Finally, the NWC algorithm enhanced with optimizations as well as some companion functions are given in Algorithms 1, 2 and 3.

3.4 *k* Nearest Window Cluster Query Processing

An NWC query is to provide the user with an area with n objects (choices). In practice, it is possible that the user even would like

Algorithm 1: NWC algorithm enhanced with optimizations

```
Data: NWC query $(q,l,w,n)$, priority queue $PQ$
Result: a set of $n$ objects
$dist_{best} \leftarrow \infty$, $objs \leftarrow \emptyset$, $PR \leftarrow \emptyset$ ;
$PQ$.enqueue($Root$ of $T_P$) ;
while $|PQ| > 0$ do
    $p \leftarrow PQ$.dequeue() ;
    if $p$ is an index node then
        $MBR_p \leftarrow$ the MBR of $p$ ;
        Extend $MBR_p$ as $MBR'_p$ based on DEP ;
        if $MBR_p - PR \neq \emptyset$ and isPrunedByDEP($MBR'_p$) is FALSE then
            for each child $c$ of $p$ do
                $PQ$.enqueue($c$) ;
    else
        // p is an object
        determine the lying quadrant of $p$ with respect to $q$ ;
        determine whether $p$ is on the left or right edge ;
        build $SR_p$ and reduce $SR_p$ as $SR'_p$ by SRR ;
        if $SR'_p \neq \emptyset$ and isPrunedByDEP($SR'_p$) is FALSE then
            $S_{SR'_p} \leftarrow IWP(SR'_p)$ ;
            sort $S_{SR'_p}$ in ascending/descending order of y coordinates ;
            remove $p_i$ from $S_{SR'_p}$ when the y coordinate of $p_i$ is
              smaller/larger than the y coordinate of $p$ ;
            for $i = 1$ to $|S_{SR'_p}|$ do
                build $qwin_p$ by setting $p$ on the right/left edge and $p_i$ on the
                  top/bottom edge ;
                if $|S_{qwin_p}| \geq n$ and $MINDIST(q, qwin_p) < dist_{best}$ then
                    let $\{p_1, p_2, \ldots, p_n\}$ be the $n$ objects in $qwin_p$ of the
                      shortest distance to $q$ ;
                    if $dist(q, \{p_1, p_2, \ldots, p_n\}) < dist_{best}$ then
                        $dist_{best} \leftarrow dist(q, \{p_1, p_2, \ldots, p_n\})$ ;
                        $objs \leftarrow \{p_1, p_2, \ldots, p_n\}$ ;
                        update $PR$ accordingly ;
return $objs$ ;
```

Algorithm 2: Function *isPrunedByDEP*(*rect*, *n*)

```
Data: a rectangle $rect$
Result: whether $rect$ is pruned by DEP
$ub \leftarrow 0$ ;
for each cell $cell$ in the density grid do
    if $cell$ intersects $rect$ then
        $ub \leftarrow ub +$ the number of objects in $cell$;
if $ub < n$ then
    return TRUE;
else
    return FALSE;
```

Algorithm 3: Function *IWP*(*rect*)

```
Data: a rectangle $rect$
Result: the set of objects within $rect$
$result \leftarrow \emptyset$, $nodes \leftarrow \emptyset$ ;
fetch the backward pointers associated with $p$ ;
for $i$=1 to $r$ do
    if $rect \subseteq mbr^b_i$ then
        $N_i \leftarrow$ the index node pointed by $bp_i$ ;
        insert $N_i$ into $nodes$ ;
        break ;
for each overlapping pointer $op_j$ of $N_i$ do
    if $mbr^o_j \cap rect \neq \emptyset$ then
        insert the index node pointed by $op_j$ into $nodes$ ;
for each node $N$ in $nodes$ do
    perform traditional window query processing with query window $rect$
      starting from $N$ ;
    insert the resultant objects into $result$ ;
return $result$ ;
```

to retrieve multiple areas so that the user is able to pick a proper area from them. To satisfy such needs, we extend NWC queries to kNWC queries that enable users to retrieve k object groups where each group consists of n objects located within a window of length l and width w. It is obvious that the user is not willing to get k object groups and each pair of groups consist of almost the same objects. Thus, we introduce a new parameter m which allows a user to specify the maximal number of identical objects allowed in each pair of groups. The formal definition of a kNWC query is given below.

Definition 3 (kNWC Query) Given a query location q, a spatial window area specified by length l and width w, the number of data objects n, and the maximal number of identical objects, say m, in any two object groups, a k nearest window cluster (kNWC) query (k,q,l,w,n,m) retrieves k object groups, $objs_1, objs_2, \ldots, objs_k$, satisfying the following criteria.

- Each object group consists of n objects within a window of length l and width w.
- For each pair of object groups, $objs_1$ and $objs_2$, there are at most m objects in both $objs_1$ and $objs_2$ (i.e., $|objs_1 \cap objs_2| \leq m$).
- The k object groups are ordered by their distances to q in ascending order. That is, $dist(q, objs_i) \leq dist(q, objs_j)$ $\forall i, j$ where $i < j$.
- For each group of n objects, say obj', within a window of length l and width w and $obj' \neq obj_i$ where $i = 1, 2, \ldots, k$, at least one of the following conditions should be satisfied.
    1. $dist(q, objs_k) \leq dist(q, objs')$.
    2. There exists an integer i ($1 \leq i \leq k$) so that $dist(q, obj_i) \leq dist(q, obj')$ and $|obj_i \cap obj'| > m$.

It is obvious that Lemma 1 also holds for kNWC queries. To answer kNWC queries, similar to NWC queries, we are allowed to consider only those qualified windows with objects on their vertical and horizontal edges. Based on Lemma 1, we now design the kNWC algorithm, an extension of the NWC algorithm, to support kNWC queries as below. The kNWC algorithm maintains the k object groups $\{objs_1, objs_2, \ldots, objs_k\}$ found so far in *groups* and sorts the k object groups by their distances to query location q in ascending order. The optimization techniques proposed in Section 3.3 can also be used to mitigate the I/O cost of identifying the nearest qualified windows of a given kNWC query. Specifically, when k object groups are obtained, the distance between q and the k-th object group (i.e., $dist(q, objs_k)$) is employed in SRR to reduce the search regions of remaining objects and in DIP to prune remaining index nodes. When finding a qualified window $qwin_p$, the kNWC algorithm performs the following steps to handle $qwin_p$.

- Step 1: Let $objs_p = \{p_1, p_2, \ldots, p_n\}$ be the n objects in $qwin_p$ of the shortest distance to q.
- Step 2: Scan *groups* in reverse order to find the first object group, say $objs_i$, which is of distance shorter than $objs_p$. In case that no such i exists, set i to 0 and go to Step 4. If $i = k$, drop $objs_p$ and stop this procedure.
- Step 3: Check whether $|objs_p \cap objs_j| \leq m$ for each $objs_j$, $j = 1, 2, \ldots, i$. If not, drop $objs_p$ and stop this procedure.
- Step 4: Remove $objs_k$ from *groups* and insert $objs_p$ into *groups* at position $i+1$ (i.e., as $objs_{i+1} = objs_p$).

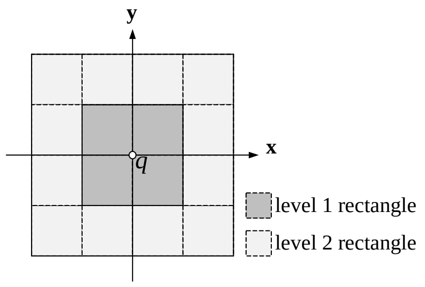


Figure 7: Illustration of analysis

- Step 5: Check whether $|objs_p \cap objs_j| \leq m$ for each $objs_j$, $j = i+2, i+3, \ldots, k-1$. Remove $objs_j$ from *groups* when $|objs_p \cap objs_j| > m$.

4. THEORETICAL ANALYSIS

4.1 Time Complexity Analysis of NWC Algorithm

In this section, we develop a cost model to analyze the I/O cost of the NWC algorithm. To facilitate the following discussion, as shown in Figure 7, the space is divided into multiple disjoint rectangles of height l and width w. Since the NWC algorithm visits the objects according to their distances to q in ascending order [10], it is very likely that the NWC algorithm visits the objects in all level-i rectangles and then visits the objects in all level-$i+1$ rectangles until the best objects are found.

We assume that the objects in an area are Poisson distributed with mean λ. For simplicity, we also assume that the probability that a window is not qualified is independent of the probability of any other window. Thus, the average number of objects in a window of length l and width w is $\lambda \times l \times w$ and the probability that a window is not qualified is

$$\mathbf{P} = P\{X \leq n-1\} = e^{-\lambda \times l \times w} \sum_{i=0}^{n-1} \frac{(\lambda \times l \times w)^i}{i!}. \quad (8)$$

An object in a level-i rectangle is called a level-i object, while a qualified window generated by a level-i object is called a level-i qualified window. Due to the effect of DIP, we also assume that the objects within a level-i qualified window can be verified as the best objects only when 1) there is no level-j qualified window, where $j = 1, 2, \cdots, i-1$, and 2) all level-i qualified windows have been checked.

Consider an object p. The NWC algorithm issues a window query to retrieve all objects within the search region of p (i.e., SR_p) and the average number of objects in the upper-half of SR_p is $\lambda \times l \times w$. For each object p' on the upper-half of SR_p, the NWC algorithm then generates one window with p on the right edge and p' on the top edge and checks whether the window is qualified or not. Thus, the average number of windows generated by an object is $\lambda \times l \times w$, and the probability that an object cannot generate a qualified window is $\mathbf{P}^{\lambda \times l \times w}$.

Let $\mathbf{N}(i)$ be the number of level-i rectangles and it is clear that

$$\mathbf{N}(i) = (2i)^2 - (2(i-1))^2 = 8i - 4. \quad (9)$$

We can obtain that the average number of level-i objects is $\mathbf{N}(i) \times \lambda \times l \times w$. Denote the probability that there is no level-i qualified window (that is, all windows generated by all level-i objects are not qualified) to be $\mathbf{Q}(i)$. Since there is no level-0 rectangle, $\mathbf{Q}(0) = 1$.

For each positive integer i, we can derive that

$$\mathbf{Q}(i) = \prod_{j=1}^{\mathbf{N}(i) \times \lambda \times l \times w} \mathbf{P}^{\lambda \times l \times w} = \mathbf{P}^{\mathbf{N}(i) \times (\lambda \times l \times w)^2}.$$

The probability that the qualified window consisting of the best objects is a level-i qualified window is $(1 - \mathbf{Q}(i)) \times \prod_{j=0}^{i-1} \mathbf{Q}(j)$.

Consider the case that the qualified window consisting of the best objects is a level-i qualified window. All level-j objects, where $j = 1, 2, \ldots i$, should be retrieved. Let the average number of objects to be retrieved in this case be $\mathbf{O}(i)$. We can obtain

$$\mathbf{O}(i) = \sum_{j=1}^{i} \mathbf{N}(i) \times \lambda \times l \times w = 2 \times i^2 \times \lambda \times l \times w. \quad (10)$$

Since these objects are retrieved in accordance with their distances to q, the average I/O cost to retrieve them is close to the average I/O cost of using K nearest neighbor query to retrieve $\mathbf{O}(i)$ objects. For each retrieved object p, the NWC algorithm issues a window query to get the objects within SR_p, and thus, the average number of issued window queries is $\mathbf{O}(i)$. Let the average I/O cost to use K nearest neighbor query to retrieve K objects be $\mathbf{KNN}(K)$, and let the average I/O cost of a window query of length l and width w be $\mathbf{WIN}(l, w)$. The average I/O cost when the qualified window consisting of the best objects is a level-i qualified window is $\mathbf{O}(i) \times \mathbf{WIN}(l, w) + \mathbf{KNN}(\mathbf{O}(i))$.

Suppose that the whole space contains at most level-*MaxLV* rectangles. The average I/O cost of the NWC algorithm is

$$\sum_{i=1}^{MaxLV} \left\{ \left[(1 - \mathbf{Q}(i)) \times \prod_{j=0}^{i-1} \mathbf{Q}(j) \right] \times \left[\mathbf{O}(i) \times \mathbf{WIN}(l, w) + \mathbf{KNN}(\mathbf{O}(i)) \right] \right\},$$

where $\mathbf{WIN}(l, w)$ can be obtained from [18] and $\mathbf{KNN}(K)$ can be obtained from [10].

4.2 Time Complexity Analysis of *k*NWC Algorithm

Let the probability that a window is not qualified be $\mathbf{P}$ where $\mathbf{P}$ can be obtained by Equation (8). Let the probability that a qualified window consists of at most m identical objects with each object group in *groups* be $Pr(m, k)$. Thus, the probability that the objects within a window cannot be inserted into *groups* be $\mathbf{P}' = 1 - [(1 - \mathbf{P}) \times Pr(m, k)]$. Suppose that the object group within a level-i qualified window will not be removed from *groups* due to the insertion of any object group within a level-j qualified window where $j > i$. Therefore, when the object group within a level-i qualified window becomes the k-th nearest object group, the kNWC algorithm will terminate when all object groups within level-i qualified windows have been checked.

We now derive the probability that the k-th nearest object group is within a level-i qualified window. Due to the effect of SRR and DIP, we assume that the k-th nearest object group is within a level-i qualified window only when 1) the number of object groups within level-j, where $j = 1, 2, \ldots, i-1$, qualified windows inserted into $qwins_{best}$ is smaller than k and 2) the number of object groups within level-j, where $j = 1, 2, \ldots, i$, qualified windows inserted into $qwins_{best}$ is larger than or equal to k.

As shown in Equation (10), the average number of all level-j objects, where $j = 1, 2, \ldots, i$, is $\mathbf{O}(i)$. Since the average number of windows generated by an object is $\lambda \times l \times w$, the probability that there are a object group within level-j, where $j = 1, 2, \ldots, i$, qualified windows inserted into *groups* is

$$\mathbf{R}(i, a) = C_a^{\mathbf{O}(i) \times \lambda \times l \times w} \times (1 - \mathbf{P}')^a \times \mathbf{P}'^{\mathbf{O}(i) \times \lambda \times l \times w - a}.$$

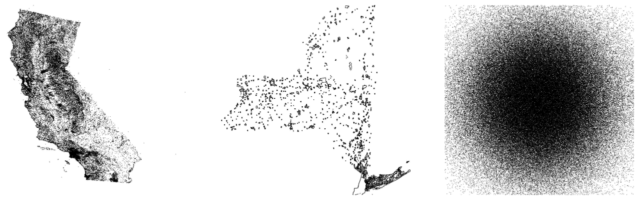

(a) CA Dataset (b) NY Dataset (c) Gaussian Dataset

Figure 8: Distributions of the used datasets

Table 2: Description of datasets

Dataset	Cardinality	Description
CA	62,556	Real places in California
NY	255,259	Real places in New York
Gaussian	250,000	Generated by Gaussian distribution

According to Equation (9), there are $\mathbf{N}(i)$ level-i rectangles and the average number of level-i objects is $\mathbf{N}(i) \times \lambda \times l \times w$. Therefore, the probability that there are at least b object groups within level-i qualified windows inserted into *groups* is

$$\mathbf{S}(i,b) = 1 - \sum_{d=1}^{b-1} \left[C_d^{\mathbf{N}(i)\times(\lambda\times l\times w)^2} \times (1-\mathbf{P}')^d \times \mathbf{P}'^{\mathbf{N}(i)\times(\lambda\times l\times w)^2-d} \right].$$

Therefore, the probability that the k-th nearest object group within a level-i qualified window is

$$\sum_{j=0}^{k-1} \left[\mathbf{R}(i-1,j) \times \mathbf{S}(i,k-j) \right].$$

When the k-th nearest object group is within a level-i qualified window, the kNWC algorithm visits all objects in all level-j rectangles, where $j = 1,2,\ldots,i$. Similar to the derivations in the previous subsection, the average I/O cost when the k-th nearest object group is a level-i qualified window is $\mathbf{O}(i) \times \mathbf{WIN}(l,w) + \mathbf{KNN}(\mathbf{O}(i))$. Suppose that space contains at most level-*MaxLV* rectangles. The average I/O cost of the kNWC algorithm is

$$\sum_{i=1}^{MaxLV} \left\{ \left[\sum_{j=0}^{k-1} \left[\mathbf{R}(i-1,j) \times \mathbf{S}(i,k-j) \right] \right] \times \left[\mathbf{O}(i) \times \mathbf{WIN}(l,w) + \mathbf{KNN}(\mathbf{O}(i)) \right] \right\}.$$

5. PERFORMANCE EVALUATION

In this section, we conduct experiments to evaluate the performance of the proposed NWC algorithm and the proposed optimization techniques. All algorithms are implemented in Java. Three datasets, two real and one synthetic, are used in the evaluation. The CA dataset contains 62,556 places in California[4], while the NY[5] dataset contains 255,259 places in New York. The data space for these two real datasets are normalized to a square of width 10,000. The synthetic dataset is created based on Gaussian distribution with mean 5000 and standard deviation 2000. The cardinality of the Gaussian dataset is default at 250,000. These datasets are summarized in Table 2, while Figure 8 depicts the object distributions of these datasets.

[4]`http://www.chorochronos.org/`

[5]`http://www.census.gov/geo/www/tiger`

Table 3: Description of schemes

	Optimization Technique(s) Used			
Scheme	SRR	DIP	DEP	IWP
NWC	-	-	-	-
SRR	✓	-	-	-
DIP	-	✓	-	-
DEP	-	-	✓	-
IWP	-	-	-	✓
NWC+	✓	✓	-	-
NWC*	✓	✓	✓	✓

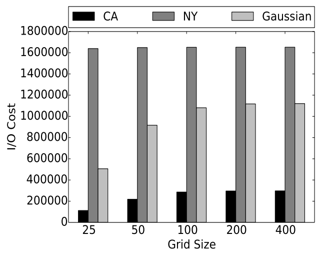


Figure 9: Effect of grid size

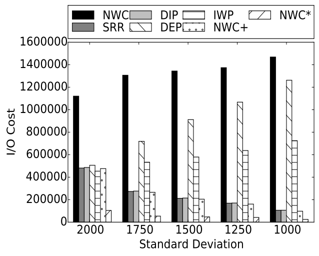


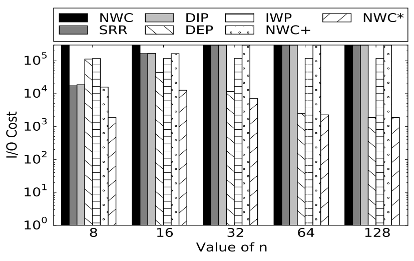
Figure 10: Effect of object distribution

All datasets are indexed by R*-trees with the page size set to 4096 bytes. The maximum number of entries in a node is 50. The default value of n is 8 and the window length and width are both 8. The grid cell size is set to 25. Similar to [18][10], we consider I/O cost as the performance metric, which is the number of R*-tree nodes visited, since I/O cost dominates the total execution time of the NWC algorithm. We run 25 queries for each experiment and report the average as the experimental result. To measure the benefit of each optimization technique, we separately run the experiments of the NWC algorithm augmented with each optimization technique (labelled as SRR, DIP, DEP and IWP, respectively). In addition, we devise a scheme NWC+ by enabling only SRR and DIP (which do not incur extra storage overhead). Finally, we devise a scheme NWC* which enables all optimization techniques proposed in this work. The schemes compared in the experiments are summarized in Table 3. In the following, we show our experimental results by varying various parameter settings, including grid size, object distribution, number of objects and window size.

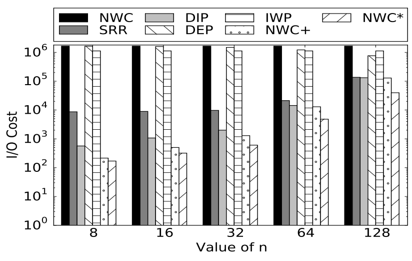
5.1 Effect of Grid Size

In this experiment, we investigate the effect of the grid size by varying the grid size from 25 to 400. Since only scheme DEP uses the density grid, we present only the experimental results of scheme DEP here. Figure 9 shows that for the CA and Gaussian datasets, the I/O cost increases along with the grid size. With the smaller grid size, the granularity of the density grid gets finer. Thus scheme DEP is able to obtain tighter upper bounds during query processing, thus achieving better pruning effect. For the NY dataset, it is interesting to see that the I/O cost of scheme DEP stays nearly constant regardless of the grid size, as depicted in Figure 9. The reason is that the objects in the NY dataset are highly clustered in certain areas, resulting in less effective pruning even when the grid size is small. This result indicates that scheme DEP could not benefit from the density grid for extremely clustered data distributions.

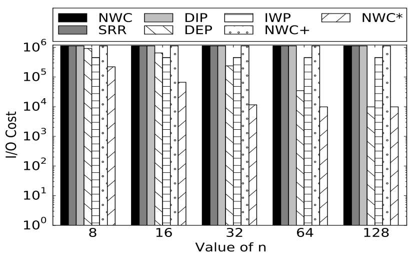
5.2 Effect of Object Distribution



(a) CA Dataset

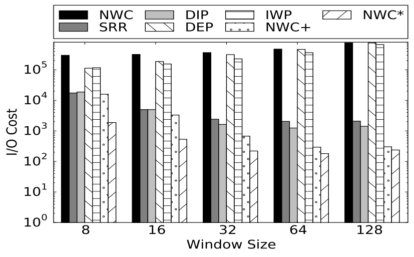


(b) NY Dataset

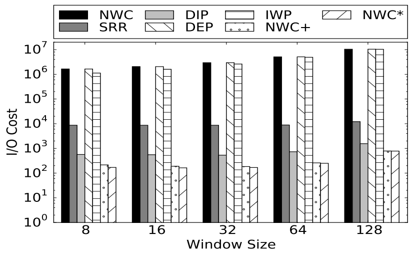


(c) Gaussian Dataset

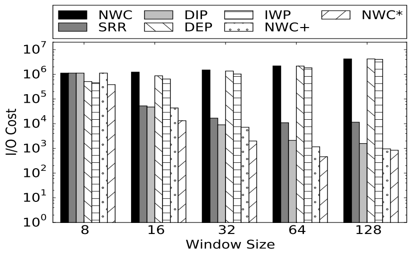
Figure 11: Effect of the number of search objects



(a) CA Dataset



(b) NY Dataset



(c) Gaussian Dataset

Figure 12: Effect of window size

We study the effect of the object distribution on the performance of all the schemes in this experiment by generating five Gaussian datasets. We fix the same mean 5,000 but varying standard deviations from 2,000 to 1,000. As shown in Figure 10, the I/O cost of scheme NWC increases as the standard deviation decreases. When the standard deviation gets smaller, the data objects are more clustered and more objects are within search regions. Thus, scheme NWC has to access more nodes to process these window queries issued for the search regions. On the contrary, for schemes SRR, DIP, and NWC+, their I/O costs decrease as the standard deviation gets smaller. In our experiment, the I/O cost reduction rates of schemes SRR, DIP and NWC+ over scheme NWC increase from 57% to 93% as the standard deviation decreases from 2000 to 1000. This is because the more clustered object distribution leads these schemes to be able to find *locally best qualified windows* (a qualified window *qwin* is called locally best if $MINDIST(q,qwin) < dist_{best}$ when *qwin* is discovered) more easily, achieving better pruning effect. It is not surprising that scheme NWC+ outperforms schemes SRR and DIP by using SRR and DIP together.

On the other hand, the I/O costs of schemes DEP and IWP increase as the standard deviation decreases. As discussed above, scheme DEP performs well in nearly uniformly distributed datasets, but achieves relatively poor performance when the object distribution is highly clustered. In our experiment, the I/O cost reduction rate of scheme DEP over scheme NWC decreases from 54.8% to 14.1% along with the decrease of the standard deviation. Regarding scheme IWP, the performance downgrades owing to that the more highly clustered data objects cause more index nodes to overlap together and thus increases the number of overlapping pointers. The more overlapping pointers are, the more index nodes scheme IWP accesses. In our experiment, the I/O cost reduction rate of scheme IWP over scheme NWC decreases from 59.5% to 55.6% as the standard deviation decreases from 2000 to 1000. From the experimental result, we can observe that the proposed optimization techniques are complementary with each other. SRR and DIP perform well on highly clustered datasets (i.e., with small standard deviations) while DEP and IWP outperform SRR and DIP on nearly uniformly distributed datasets (i.e., with large standard deviations). By combining the advantages of all the optimization techniques, scheme NWC* performs the best in terms of I/O cost and significantly reduces 98.3% I/O cost compared with scheme NWC. Moreover, the I/O cost reduction of scheme NWC* over scheme NWC+ is ranging from 73.8% to 79.7%, showing that the small storage overhead produced by DEP and IWP really pays off especially on the cases not suitable for SRR and DIP.

We now evaluate the storage overheads of scheme DEP and scheme IWP. When the grid size is set to 25, the density grid is of 160000 grids. As we use short integer to store the number of object in each cell, the storage overhead of DEP (the size of the density grid) is about 312KB. On the other hand, it is obvious that the numbers of backward and overlapping pointers are proportional of object numbers. The numbers of backward and overlapping pointers for the CA, NY and Gaussian datasets are 26473, 6236 and 29037, respectively. Suppose that the size of one pointer is 4 bytes. The storage overheads of IWP (the size of these pointers) in the CA, NY and Gaussian datasets are about 103KB, 24KB and 113KB, respectively. These storage overheads are acceptable.

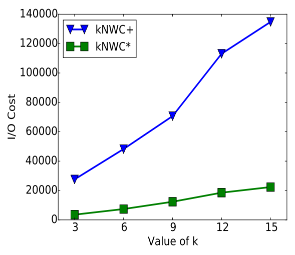
5.3 Effect of the Number of Searched Objects

This experiment evaluates the effect of the number of searched objects *n*. In the experiment, we vary the value of *n* from 8 to 128. Note that we set the y axis in logarithmic scale in this and the following experiments due to the varied scale of I/O cost. Figure 11 shows that the I/O cost of scheme NWC almost stays constant because scheme NWC accesses all the objects in R*-tree regardless of the value of *n*. The other schemes suffer from higher I/O costs with the value of *n* increasing, because a larger value of *n* causes more index nodes to be accessed in order to find the locally best qualified windows. When the value of n is too large to find any qualified window, schemes SRR, DIP and NWC+ would degenerate to scheme NWC since no pruning is achieved. As shown in Figure 11a, the I/O costs of scheme SRR, DIP, and NWC+ are equal to the I/O cost of scheme NWC when the value of n is larger than or equal to 32. Similarly, Figure 11c shows that schemes SRR, DIR, and NWC+ incur the same I/O cost as scheme NWC when the value of n is 8 or larger. These three schemes degenerate faster in the Guassian dataset because the data objects are nearly uniformly distributed in the Gaussian dataset. Different from the CA and Gaussian datasets, schemes SRR, DIP, and NWC+ still outperform scheme NWC in the NY dataset even when the value of n is 128. The reason is that the highly clustered objects in the NY dataset allow these schemes to find locally best qualified windows quickly even for large values of n.

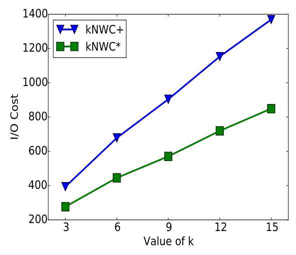
On the other hand, schemes DEP and IWP are more resilient to the increase of the value of n. For scheme DEP, it prunes more index nodes and search regions when the value of n gets larger. Thus, scheme DEP remains to perform well in the Gaussian dataset as long as the value of n is large to a certain extent. As shown in Figure 11c, when n increases from 8 to 128, the I/O cost reduction rate of scheme DEP over scheme NWC in the Gaussian dataset increases from 18% to 99.1%, showing the benefit of scheme DEP. For scheme IWP, a larger value of n only affects the performance slightly because IWP mainly focuses on saving the I/O cost of window query processing. As such, scheme IWP is able to reduce the I/O cost in the case that schemes SRR, DIP and DEP performs poorly (e.g., $n = 8$ in the Gaussian dataset). We can observe from Figure 11c that the I/O cost reduction rate of scheme IWP over scheme NWC keeps in 59.6% when n increases from 8 to 128. Due to the complementary advantages of the optimization techniques, scheme NWC* performs the best in all the cases, and compared with scheme NWC, is able to reduce 95.7%$\sim$99.4%, 97.6%$\sim$99.9% and 88.2%$\sim$99.1% I/O costs in the CA, NY and Gaussian datasets, respectively.

5.4 Effect of the Window Size

In this experiment, we explore the effect of the window size on I/O costs for all schemes by increasing the window length and width from 8 to 128. We can see in Figure 12 that the I/O cost of scheme NWC gets larger as the window size increases. This is because the larger window sizes result in larger search regions and more data objects involved in the discovery of qualified windows. On the contrary, as the window size increases, it is easier to find locally best qualified windows. With more locally best qualified windows, schemes SRR and DIP achieve better performance. We can see in Figure 12c that schemes SRR and DIP degenerate to scheme NWC in the Gaussian dataset when window size is set to 8. Setting window size to 8 is too small to find any qualified window in the Gaussian dataset since distribution of the objects in the Gaussian dataset is close to uniform distribution. Except for such an extreme case, the I/O cost reduction rates of schemes SRR and DIP over scheme NWC increase from 93.7% to 99.8% and from 95.5% to 99.9% in the CA and Gaussian datasets, respectively, as the window size gets from 16 to 128. As depicted in Figure 12b, the I/O cost reduction rates of schemes SRR and DIP over scheme NWC in the NY dataset keep in 99.5%$\sim$99.9%. The reason is that in the NY dataset, there are a large number of data objects and these data objects are highly clustered. Schemes SRR and DIP are still able to get enough locally best qualified windows even window size is set to 8. Thus, increasing the window size does not significantly increase I/O costs of scheme SRR and DIP. Similar to previous experiments, scheme NWC+ outperforms schemes SRR and DIP in all cases.



(a) CA Dataset



(b) NY Dataset

Figure 13: Effect of k

On the contrary, schemes DEP and IWP do not benefit from larger window sizes. When the window size gets larger, the number of qualified windows increases, thereby reducing the pruning effect of DEP. As the window size is large to a certain extent, scheme DEP could not achieve any pruning and thus will degenerate to scheme NWC, referring to Figure 12a and Figure 12b. For scheme IWP, the large window size results in more overlapping index nodes when processing window queries, reducing the benefit of scheme IWP. As such, scheme IWP is less effective for large window sizes. Fortunately, DEP and IWP are still able to reduce some I/O costs when SRR and DIP are used. Therefore, scheme NWC* achieves the best performance and the I/O cost reduction rates of scheme NWC* over scheme NWC+ are from 88.1% to 21.4%, 21% to 0.6% and 65.8% to 11.9% in the CA, NY and Gaussian datasets, respectively, when window size gets from 8 to 128.

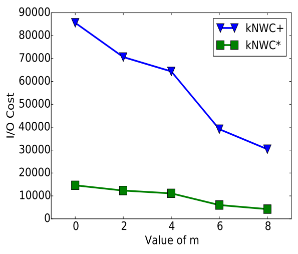
5.5 Effect of k

We now evaluate the performance of these schemes for kNWC queries. We can observe from the above experiments that scheme NWC+ is of the best performance when extra storage except R-tree is not available. On the other hand, scheme NWC* performs the best when extra storage is available. Thus, we only compare the performance of scheme kNWC+ and scheme kNWC* (extensions of scheme NWC+ and scheme NWC*, respectively, for kNWC queries) in this and the following experiments.

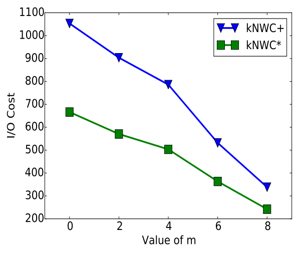
Figure 13 shows the effect of k on the CA and NY datasets. It is obvious that when k gets larger, both schemes need to spend more time in exploring more qualified windows. As shown in Figure 13, the I/O costs for both schemes almost linearly increase. As mentioned in Section 5.4, since the NY dataset is of many highly clustered objects, the I/O costs of both scheme in the CA dataset are higher than the I/O costs in the NY dataset. In addition, due to the effect of DEP and IWP, scheme kNWC* outperforms scheme kNWC+ in both datasets. Since both schemes perform well in the NY dataset, the I/O cost reduction rate of scheme kNWC* over scheme kNWC+ in the CA dataset is higher than that in the NY dataset. In our experiment, the average I/O cost reduction rates of scheme kNWC* over scheme kNWC+ in the CA and NY datasets are around 84.3% and 35.3%, respectively.

5.6 Effect of m

Figure 14 shows the effect of m on the CA and NY datasets. Setting m to a larger value means that users accept more identical objects in the nearest qualified windows. When a nearest qualified window is found, the qualified windows nearby are of high likelihood to be the qualified windows. Thus, it is easier



(a) CA Dataset



(b) NY Dataset

Figure 14: Effect of *m*

for both schemes to find k nearest qualified windows when m gets larger. Similarly, both schemes are of higher I/O costs in the CA dataset than in the NY dataset. Fortunately, with the aid of DEP and IWP, scheme kNWC* outperforms scheme kNWC+ in both datasets. In our experiment, the average I/O cost reduction rates of scheme kNWC* over scheme kNWC+ in the CA and NY datasets are around 83.8% and 33.9%, respectively.

6. CONCLUSIONS

In this paper, we propose a novel type of spatial queries, namely nearest window cluster (NWC) queries. To process NWC queries, we identify several properties to find qualified windows, leading to the development of the NWC algorithm. To further accelerate NWC search, we present four optimization techniques to reduce I/O cost. We conduct several experiments to evaluate the performance of the NWC algorithm and the proposed optimization techniques. Experimental results show that these optimization techniques are complementary with each other, and the NWC algorithm with these optimization techniques performs the best in terms of I/O cost.

Acknowledgement

This work was supported in part by Ministry of Science and Technology, Taiwan, under contracts 102-2221-E-009-147 and 103-2221-E-009-126-MY2.

7. REFERENCES

[1] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *Proc. of the VLDB Endowment (PVLDB)*, 3(1), 2010.

[2] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *Proc. of the ACM International Conference on Management of Data (SIGMOD)*, June 2011.

[3] M. A. Cheema, X. Lin, W. Zhang, and Y. Zhang. Influence zone: Efficiently processing reverse k nearest neighbors queries. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, pages 577–588, 2011.

[4] D.-W. Choi, C.-W. Chung, and Y. Tao. A scalable algorithm for maximizing range sum in spatial databases. *Proc. of the VLDB Endowment (PVLDB)*, 5(11), 2012.

[5] C.-Y. Chow, M. F. Mokbel, J. Naps, and S. Nath. Approximate evaluation of range nearest neighbor queries with quality guarantee. In *Proc. of the International Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 283–301, 2009.

[6] K. Deng, S. Sadiq, X. Zhou, H. Xu, G. P. C. Fung, and Y. Lu. On group nearest group query processing. *IEEE Transactions on Knowledge and Data Engineering*, 24(2):295–308, February 2012.

[7] Y. Du, D. Zhang, and T. Xia. The optimal-location query. In *Proc. of theinternational Conference on Advances in Spatial and Temporal Databases (SSTD)*, pages 163–180, 2005.

[8] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A. E. Abbadi. Constrained nearest neighbor queries. In *Proc. of the International Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 257–278, 2001.

[9] Y. Gao, B. Zheng, G. Chen, and Q. Li. Optimal-location-selection query processing in spatial databases. *IEEE Transactions on Knowledge and Data Engineering*, 21(8):1162–1177, Aug. 2009.

[10] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999.

[11] H. Hu and D. L. Lee. Range nearest-neighbor query. *IEEE Transactions on Knowledge and Data Engineering*, 18(1):78–91, January 2006.

[12] F. Korn and S. M. ukrishnan. Influence sets based on reverse nearest neighbor queries. In *Proc. of the ACM International Conference on Management of Data (SIGMOD)*, pages 201–212, 2000.

[13] K. C. Lee, W.-C. Lee, and H. V. Leong. Nearest surrounder queries. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, page 85, 2006.

[14] K. C. K. Lee, B. Zheng, and W.-C. Lee. Ranked reverse nearest neighbor search. *IEEE Transactions on Knowledge and Data Engineering*, 20(8):894–910, July 2008.

[15] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. In *Proc. of the International Conference on Advances in Spatial and Temporal Databases*, August 2005.

[16] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, pages 301–312, 2004.

[17] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Transactions on Database Systems*, 30(2):529–576, June 2005.

[18] G. Proietti and C. Faloutsos. I/O complexity for range queries on region data stored using an R-tree. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, March 1999.

[19] Y. Tao, D. Papadias, X. Lian, and X. Xiao. Multidimensional reverse knn search. *The VLDB Journal*, 16(3):293–316, Jul. 2007.

[20] J. Xu, W.-C. Lee, and X. Tang. Exponential index: A parameterized distributed indexing scheme for data on air. In *Proc. of the ACM Internation Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2004.

[21] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, pages 485–492, 2001.

[22] B. Yao, F. Li, and P. Kumar. Reverse furthest neighbors in spatial databases. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, pages 664–675, 2009.