6-2007

Continuous Nearest Neighbor Queries over Sliding Windows

Kyriakos MOURATIDIS
Singapore Management University, kyriakos@smu.edu.sg

Dimitris Papadias
Hong Kong University of Science and Technology

Citation

MOURATIDIS, Kyriakos and Papadias, Dimitris. Continuous Nearest Neighbor Queries over Sliding Windows. (2007). *IEEE Transactions on Knowledge and Data Engineering*. 19, (6), 789-803.
Available at: https://ink.library.smu.edu.sg/sis_research/204

Continuous Nearest Neighbor Queries over Sliding Windows

Kyriakos Mouratidis
School of Information Systems
Singapore Management University
80 Stamford Road, Singapore 178902
kyriakos@smu.edu.sg

Dimitris Papadias
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
dimitris@cse.ust.hk

Abstract

This paper studies continuous monitoring of nearest neighbor (NN) queries over *sliding window* streams. According to this model, data points continuously stream in the system and they are considered *valid* only while they belong to a sliding window that contains (i) the *W* most recent arrivals (*count-based*), or (ii) the arrivals within a fixed interval *W* covering the most recent timestamps (*time-based*). The task of the query processor is to constantly maintain the result of long running NN queries among the valid data. We present two processing techniques that apply to both count-based and time-based windows. The first one adapts *conceptual partitioning*, the best existing method for continuous NN monitoring over update streams, to the sliding window model. The second technique reduces the problem to *skyline* maintenance in the *distance-time* space and pre-computes the future changes in the NN set. We analyze the performance of both algorithms, and extend them to variations of NN search. Finally, we compare their efficiency through a comprehensive experimental evaluation. The skyline-based algorithm achieves lower CPU cost, at the expense of slightly larger space overhead.

Index Items: J.9.a Location-dependent and sensitive, H.2.4.k Spatial databases, H.2.4.h Query processing.
Additional Keywords: Nearest neighbors, Data Streams, Sliding Windows.

To appear in IEEE Transactions on Knowledge and Date Engineering (TKDE).

Contact Author:
Dimitris Papadias
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong

Office: ++852-23586971 http://www.cs.ust.hk/~dimitris/
Fax: ++852-23581477 E-mail: dimitris@cs.ust.hk

1. Introduction

Given a set of points *P* in a multi-dimensional space, the nearest neighbor (NN) of a query point *q* is the point in *P* that is closest to *q*. Similarly, the *k*NN set of *q* consists of the *k* points in *P* with the smallest distances from *q* (usually according to the Euclidean metric). The first techniques for NN retrieval considered static queries and data (e.g., [H84, RKV95, HS99]). Later work focused on moving NN queries in client-server architectures (e.g., [ZL01, ZZP+03, SR01, TP03]). In this setting, the goal is to provide, in addition to the current result, information about its validity in order to reduce the number of future re-computations (when the client/query moves). Other existing methods, return all the query results up to a future timestamp, assuming that the query and the data objects move linearly with known velocity [TP03, BJKS06].

The above techniques deal with the efficient processing of a single *snapshot* query since they report the NN set at the query time, possibly with some validity information, or generate future results based on predictive features (e.g., velocity vectors of queries or data objects). On the other hand, *continuous monitoring* assumes a central server that collects the current locations of data objects, and continuously updates the results of multiple long-running queries. Processing usually takes place in main-memory, in order to provide fast answers in an on-line fashion, and attempts to minimize factors such as the CPU or communication cost (as opposed to I/O overhead).

Continuous monitoring of spatial queries is becoming increasingly important due to the wide availability of inexpensive and compact positioning devices, the evolution of mobile communications and the need for improved location-based services. Consequently, several techniques (reviewed in Section 2.1) have been recently developed for continuous NN queries. These methods assume *update streams* where an object issues an update if and only if it moves to a new location. The server processes the stream of position updates and incrementally maintains the NNs of numerous queries. Objects that do not issue updates are assumed to be at the last reported positions.

This paper, on the other hand, studies *k*NN monitoring over *sliding windows*, assuming the *append-only* data stream model [BBD+02]. In this context, each data item is *valid* only while it belongs to a sliding window. We consider the two most common versions of windows: a *time-based window* contains all data that arrived within a fixed interval *W* covering the most recent timestamps, whereas a *count-based window* contains the *W* most recent data items (independently of when they arrived). Even though some existing methods for update streams can be extended to sliding windows (by treating new points as object insertions and points falling outside the window as deletions), we show that the first-in-first-out deletion order that is particular to this setting allows for faster NN monitoring.

In general, sliding windows are used to restrict the temporal scope of query processing in the absence of explicit deletions. As an application example, consider a set of sensors taking measurements of their surrounding environment and reporting their coordinates to a central server when they detect some particular event. Imposing a sliding window on the stream of reports excludes old events from consideration. Depending on the application domain, NN monitoring in this setting may be used for wild animal tracking, intrusion detection, etc. As an instance of *k*NN monitoring over a time-based sliding window, assume a set of sensors in a forest that report their location whenever they detect an animal passing by (using motion, temperature measurements, etc). In this scenario, a user may want to continuously monitor the *k* closest animals to his/her location. Old reports correspond to obsolete animal positions; only the ones received within the last *W* time units (e.g., 30 seconds) are taken into account.

Continuous *k*NN processing is not restricted to the spatial domain, but can be utilized in other problems with a multi-dimensional aspect. As an example of a count-based window application, assume a user that subscribes a query (i.e., a set of keywords) to a web-based news agency (e.g., CNN, Reuters). The agency reports to the user the *k* closest matches among the last *W* news articles. Typically, each article is represented as a point in some space, where its Euclidean distance from the query defines its similarity; i.e., the problem is essentially a continuous NN search in the mapped space[1]. An article ceases to be among the results (i) if it is replaced by a better (i.e., more similar to the query) and more recent one, or (ii) when *W* news articles arrive after its publication. A similar problem can be defined in terms of time-based windows, e.g., the server may continuously report the closest matches among the articles published within the last 24 hours. In this example, each article received at the server corresponds to a new distinct data item, for which there are no further updates.

This paper presents and compares two techniques for NN monitoring over sliding windows, covering both count-based and time-based windows, arbitrary *k*, and static or moving queries. The first one adapts *conceptual partitioning* [MHP05], the best existing method for NN monitoring over update streams, to the sliding window model. The second technique reduces the problem to *skyline* maintenance in the *distance-time* space and partially pre-computes future changes in the NN sets. The skyline-based algorithm achieves lower CPU cost, at the expense of slightly larger space overhead.

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 presents the index and book-keeping structures used by our algorithms. Section 4 extends conceptual partitioning to the sliding window model, while Section 5 describes the skyline-based

[1] Dimensionality reduction techniques are commonly applied to decrease the number of dimensions, so

method. Section 6 provides an analysis of the proposed techniques, and Section 7 extends them to other NN query types. Section 8 experimentally compares our algorithms and, finally, Section 9 concludes the paper with directions for future work.

2. Related Work

Section 2.1 reviews previous work on continuous monitoring of spatial queries, focusing mostly on conceptual partitioning due to its relevance to our work. Section 2.2 presents existing techniques for skyline computation in database systems and discusses the relation between skylines and NN queries.

2.1 Continuous Monitoring of Spatial Queries

Assuming static range queries over moving objects (i.e., in update streams), *Q-index* [PXK+02] uses an R-tree [G84, BKSS90] at the server to index the queries. When updates from moving objects arrive, the server probes the R-tree to retrieve the influenced queries. *Q-index* utilizes the concept of *safe* (i.e., validity) *regions* to reduce the number of updates. In particular, each object *p* is assigned a circular or rectangular region, such that *p* needs to issue an update only if it exits this area. Kalashnikov et al. [KPH04] show that a grid implementation of *Q-index* is more efficient (than R-trees) for main memory evaluation. MQM [CHC04] and *Mobieyes* [GL04] exploit the object computational capabilities in order to reduce the processing load of the server. In SINA [MXA04], the server continuously updates the reported results by performing a spatial join between moving objects and queries in three phases: (i) the hashing phase receives information about moving objects and queries and generates positive updates, (ii) the invalidation phase is performed every *T* timestamps or when the memory is full, and reports

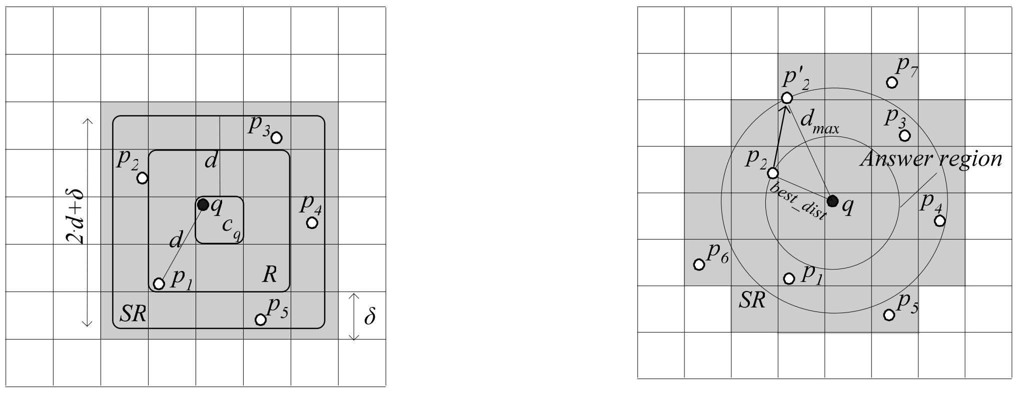
that the documents can be indexed effectively.

negative updates, (iii) the joining phase, triggered after the invalidation phase, joins the contents of the main-memory with those of the disk, generating both positive and negative updates.

The aforementioned methods focus on range query monitoring, and their extension to NN search is either impossible or non-trivial. Henceforth, we discuss algorithms that target explicitly NN processing. Koudas et al. [KOTZ04] describe DISC, a technique for e-approximate kNN queries over streams of multi-dimensional points. The returned k^{th} NN lies at most e distance units farther from q than the actual k^{th} NN of q. DISC partitions the space with a regular grid of granularity such that the maximum distance between any pair of points in a cell is at most e. To avoid keeping all arriving data in the system, the server maintains only K points for each cell c. An exact kNN search in the retained points corresponds to an approximate ekNN answer over the original dataset provided that $k \leq K$. DISC indexes the data points with a B-tree that uses a space-filling curve mechanism to facilitate fast updates and query processing. The authors show how to adjust the index to: (i) use the minimum amount of memory in order to guarantee a given error bound e, or (ii) achieve the best possible accuracy, given a fixed amount of memory. DISC can process both snapshot and continuous ekNN queries.

Yu et al. [YPK05] propose a method, hereafter referred to as YPK-CNN, for continuous monitoring of exact kNN queries in update streams. All objects are assumed to fit in main memory and are indexed with a regular grid of cells with size $\delta \times \delta$. The server does not process updates as they arrive, but directly applies the changes to the grid. When a continuous query q is evaluated for the first time, a two step NN search technique retrieves its result. The initial step visits the cells inside an iteratively enlarged square R around the cell c_q covering q until k objects are found. Figure 2.1a, shows an example of a single NN query where the first candidate NN is p_1 with distance d from q; p_1 is not necessarily the actual NN since there may be objects (e.g., p_2)

in cells outside *R* with distance smaller than *d*. To retrieve such objects, the second step searches in the cells intersecting the square search region (*SR*) centered at c_q with side length $2 \cdot d+\delta$, and determines the actual *k*NN set of *q* therein. In Figure 2.1a, the server processes p_1 up to p_5 and returns p_2 as the actual NN. The accessed cells appear shaded. To maintain the result in subsequent timestamps, it computes the current distance d_{max} of the previous NN that moved furthest from *q*, and retrieves the new NN set by searching in all cells intersecting the square centered at c_q with side length $2 \cdot d_{max}+\delta$.



(a) NN search in YPK-CNN (b) Update handling in SEA-CNN

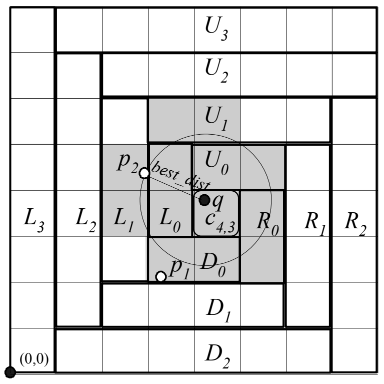
Figure 2.1: YPK- and SEA-CNN examples

SEA-CNN [XMA05] focuses exclusively on monitoring the NN changes, without including a module for the first-time evaluation of an arriving query *q* (i.e., it assumes that the initial result is available). The server indexes moving objects with a regular grid. The *answer region* of a query *q* is defined as the circle with center *q* and radius *best_dist*, where *best_dist* is the distance of the current k^{th} NN. Book-keeping information is stored in the cells that intersect the answer region of *q* to indicate this fact. When updates arrive at the system, the server determines a circular search region *SR* around *q* and computes the new *k*NN set of *q* therein.

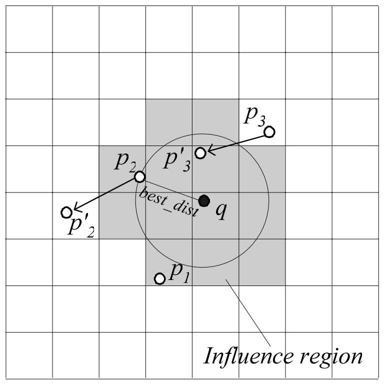
To determine the radius *r* of *SR*, SEA-CNN distinguishes the following cases: (i) If some of the current NNs move within the answer region or some outer objects enter the answer region, the server sets *r*=*best_dist* and processes all objects falling in the answer region in order to retrieve the new NN set. (ii) If any of the current NNs moves out of the answer region, then $r = d_{max}$ (where d_{max} is the distance of the furthest previous NN), and the NN set is computed among the objects lying in *SR*. Assume that in Figure 2.1b the current NN p_2 issues an update reporting its new location p'_2. The server sets $r=d_{max}=dist(p'_2,q)$, determines the cells intersecting *SR* (these cells appear shaded), collects the corresponding objects (p_1 up to p_7), and retrieves the new NN p_1. (iii) Finally, if the query *q* moves to a new location *q'*, the server sets $r = best\_dist+dist(q,q')$, and computes the new *k*NN set of *q* by processing all the objects that lie in the circle centered at *q'* with radius *r*.

CPM [MHP05] is the state of the art NN monitoring method for update streams. It assumes the same system architecture and indexing and book-keeping structures as YPK-CNN and SEA-CNN. When a query *q* arrives at the system, the server computes its initial result by organizing the cells into conceptual (hyper) rectangles based on their proximity to *q*. Each rectangle *rect* is defined by a *direction* and a *level number*. The direction is U, D, L, or R (for up, down, left and right), and the level number indicates how many rectangles are between *rect* and *q*. Figure 2.2a illustrates the conceptual partitioning of the space around the cell $c_{4,3}$ of *q* in our running example. If *mindist*(*c*,*q*) is the minimum possible distance between any object in cell *c* and *q*, the NN search considers the cells in ascending *mindist*(*c*,*q*) order.

In particular, CPM initializes an empty heap *H* and inserts (i) the cell of *q* with key equal to 0, and (ii) the level zero rectangles for each direction *DIR*, with key $mindist(DIR_0,q)$. Then, it starts de-heaping entries iteratively. If the de-heaped entry is a cell, it examines the objects inside and updates accordingly the list *best_NN* of the closest NNs found so far. If the de-heaped entry is a rectangle DIR_{lvl}, it inserts into *H* (i) each cell $c \in DIR_{lvl}$ with key *mindist*(*c*,*q*) and (ii) the next level rectangle DIR_{lvl+1} with key $mindist(DIR_{lvl+1},q)$. The algorithm terminates when the next entry in H (corresponding either to a cell or a rectangle) has key greater than the distance *best_dist* of the kth NN found. It can be easily verified that the server processes only the cells that intersect the circle with center at q and radius equal to *best_dist*. This is the minimal set of cells to visit in order to guarantee correctness. In Figure 2.2a, the search processes the shaded cells, and returns p_2 as the result.



(a) NN search



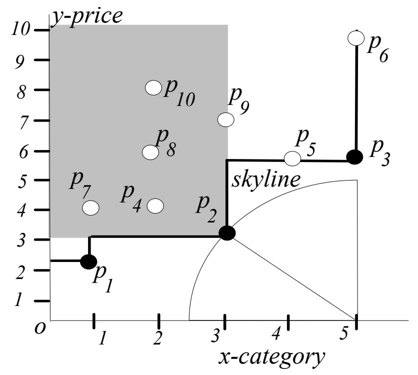
(b) Update handling

Figure 2.2: CPM examples

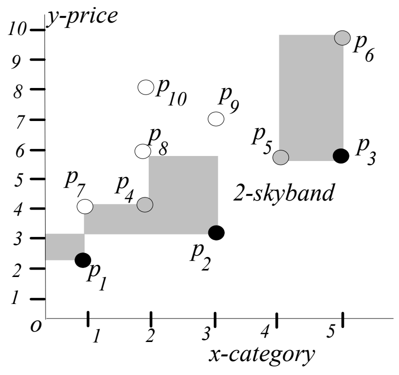
The encountered cells constitute the *influence region* of q, and only updates therein can affect the current result. When updates arrive for these cells, then CPM monitors how many objects enter or leave the circle centered at q with radius *best_dist*. If the outgoing objects are more than the incoming ones, then the result has to be computed from scratch. Otherwise, the new NN set of q can be inferred by the previous result and the update information, without accessing the grid at all. Consider the example of Figure 2.2b, where p_2 and p_3 move to positions p'_2 and p'_3, respectively. Object p_3 moves closer to q than the previous *best_dist* and, therefore, CPM replaces the outgoing NN p_2 with the incoming p_3. The evaluation of [MHP05] confirms that CPM is significantly faster than YPK- and SEA-CNN for all tested problem settings. Section 4 discusses its adaptation to sliding windows.

2.2 Skyline Queries

Assume that in Figure 2.3a we have a set of hotels, and for each hotel we store its price (*y* axis) and category (*x* axis, 1 means one star, etc). The skyline contains the most *interesting* hotels p_1, p_2 and p_3, i.e., the ones that are not *dominated* by another hotel on both dimensions. For example, p_2 dominates p_4, p_7, p_8, p_9, and p_{10} because it is cheaper and at the same time it belongs to a higher (or at least the same) category. In other words, p_2 is preferable (to p_4, p_7, p_8, p_9, p_{10}) according to any *preference* function which is increasingly monotone on the *x* and decreasingly monotone on the *y* axis. Similar examples can be given for skylines that minimize/maximize any combination of dimensions. Skyline computation has received considerable attention in relational databases [BKS01, TEO01] and web information systems [BGZ04]. Lin et al. [LYWL05], Tao and Papadias [TP06] propose methods for skyline monitoring over sliding windows. The skyline maintenance is performed by an in-memory incremental algorithm, which discards records that cannot participate in the skyline until their expiration.



(a) Skyline



(b) 2-skyband

Figure 2.3: Skyline and skyband examples

Skylines are closely related to NN search. In particular, it can be easily shown that the first NN (i.e., p_2 in Figure 2.3a) of point (5,0) always belongs to the skyline. Based on this observation, the method of [KRR02] applies a NN algorithm on point (5,0) to retrieve p_2. Then, it prunes all the points in the shaded area of Figure 2.3a since they are dominated by p_2 (and, therefore, they are not part of the skyline). The remaining space is split into two partitions based on the coordinates of p_2 and the process is repeated recursively. Papadias et al. [PTFS05] propose an improved algorithm based on incremental NN computation, which is optimal in terms of I/O accesses.

Motivated by the fact that the NN always belongs to the skyline, we follow the opposite direction, i.e., we use skyline maintenance to monitor NN results. Since the skyline corresponds to single NN retrieval (whereas we are interested in *k*NNs), we adopt the concept of *k-skyband* [PTFS05]. Specifically, the *k*-skyband contains the points that are dominated by at most k-1 other ones. According to this definition, the skyline is a special instance of the skyband, where k=1. In Figure 2.3b, the 2-skyband consists of all points (p_1, ..., p_6) in the shaded region. The next section illustrates how to exploit *k*-skybands (in a transformed space) for efficiently maintaining *k*NNs over sliding windows.

3. PRELIMINARIES

Assuming a two-dimensional space, each tuple *p* of the input stream has the form <*p.id*, *p.x*, *p.y*, *p.t*>, where *p.id* is a unique identifier for *p*, *p.x* and *p.y* are its *x* and *y* coordinates, and *p.t* is its arrival time. Stream records are treated as points and, thus, in the rest of the paper the terms tuple, point, and record are used interchangeably. Since in real-world systems processing takes place at discrete timestamps, multiple points may arrive/expire in the same processing cycle. Our discussion focuses on this general scenario[2], but the proposed algorithms apply without modification to the case where points stream in/expire one by one.

Similar to existing monitoring approaches (e.g., [KOTZ04, YPK05, XMA05, MHP05]), we

[2] The same assumption underlies the previous NN monitoring techniques (e.g., [YPK05, XMA05]), as well as most data stream management systems (e.g., [CCC+02, LMT+05]).

use a regular grid to index the valid data because a more complicated access method (e.g., a main memory R-tree) is very expensive to maintain dynamically. The extent of each cell on every dimension is δ, so that cell $c_{i,j}$ at column i and row j (starting from the low-left corner of the workspace) contains all valid points with x coordinate in the range $[i\cdot\delta, (i+1)\cdot\delta)$ and y coordinate in the range $[j\cdot\delta, (j+1)\cdot\delta)$. Conversely, given a point p with coordinates ($p.x$, $p.y$), its covering cell can be determined (in constant time) as $c_{i,j}$, where $i=\lfloor p.x/\delta \rfloor$ and $j=\lfloor p.y/\delta \rfloor$.

Furthermore, it is important to provide an efficient mechanism for evicting expiring data. In both versions of the sliding window (i.e., count-based and time-based), the points are evicted in a first-in-first-out manner, since W contains the most recent ones. Therefore, all the valid point positions are stored in a single list. The new arrivals are placed at the end of the list, and the points that fall out of the window are discarded from the head of the list. Each cell contains a list of pointers to the corresponding (valid) points, as shown in Figure 3.1. Since insertions and deletions to a cell also occur in a fist-in-first-out fashion, each operation on the content list takes $O(1)$ time.

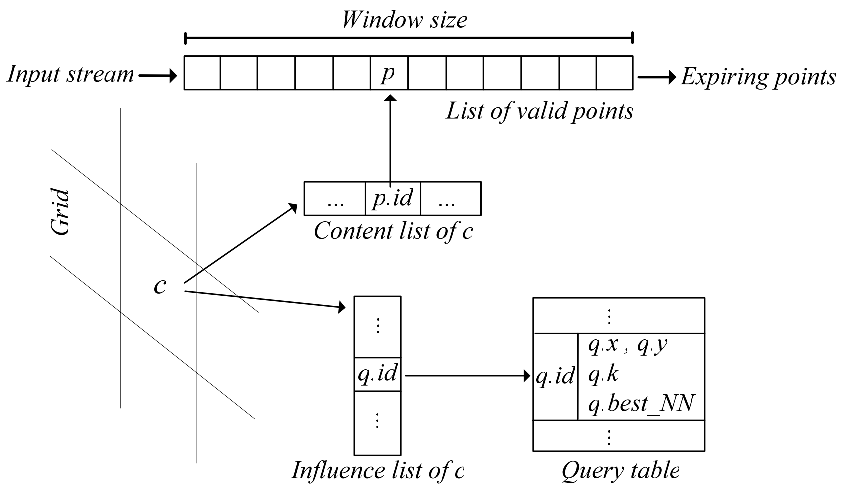
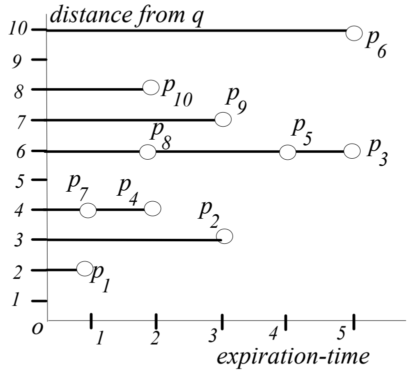


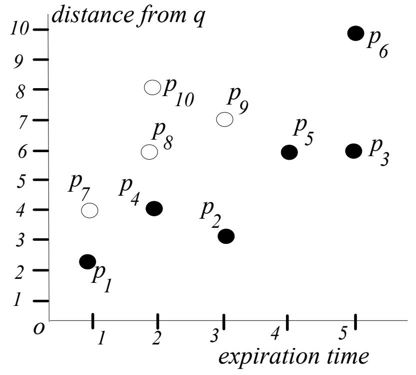
Figure 3.1: Index and book-keeping data structures

The running queries q are stored in a query table QT. QT maintains for each q a unique identifier $q.id$, its coordinates $q.x$ and $q.y$, the number of NNs required $q.k$, and its current result *q.best_NN*. The distance of the *k*th point in *q.best_NN* (referred to as *q.best_dist*) implicitly defines the influence region of *q*. To restrict the scope of the *k*NN maintenance algorithms, each cell *c* is associated with an *influence list* IL_c that contains an entry for each query *q* whose influence region intersects *c*. Since the query influence regions change dynamically, IL_c is organized as a hash-table on the query identifiers for supporting fast search, insertion and deletion operations.

We propose two monitoring algorithms: the first one adapts CPM to sliding windows, whereas the second utilizes the concept of *k*-skyband. In order to demonstrate the relation between *k*-skybands and *k*NN in the context of append-only streams, we use the example of Figure 3.2a where the server monitors a 2NN query *q* over ten valid points. The horizontal axis denotes the expiration time of points and the vertical dimension indicates their distance from *q*. Assuming that there are no further arrivals, we can predict all future results. The 2NN set at time 0 is $\{p_1,p_2\}$. When p_1 expires at time 1, it is replaced by p_4. At time 2, p_4 expires and the result becomes $\{p_2,p_3\}$. Similarly, at time 3, the 2NN set is $\{p_5,p_3\}$. Finally, at time 4, p_6 replaces p_5. The important observation is that the points that appear in the result at some time are the ones that belong to the 2-skyband in the *distance-time* space. The skyband records in our example are shown solid in Figure 3.2b (which is similar to Figure 2.3b, except for the meaning of the axes).



(a) Point lifespans



(b) 2-skyband

Figure 3.2: Transformation of a 2NN query into a 2-skyband in the *distance-time* space

Lemma: Given the expiration time of all valid points in the system, and assuming that there are no further arrivals, the points that will appear in the result of a *k*NN query *q* in the future are exactly the ones that belong to the *k*-skyband in the *distance* (from *q*) - *expiration time* space.

Proof: Consider a point *p* that belongs to some (future) *k*NN result. Then, there exists some time instance when *p* has a larger distance than (is dominated by) at most *k*-1 other valid points. Therefore, *p* is part of the *k*-skyband. Conversely, consider that *p* belongs to the *k*-skyband in the *distance-time* space. This implies that there are at most *k*-1 other points with distance lower than that of *p* and expire after *p*. Thus, there exists some time instance when *p* is one of the *k*NNs of query *q*.

The validity of the above lemma is independent of the dimensionality; i.e., the skyband is always computed in the two-dimensional *distance-time* space even if the data dimensionality is higher than 2. The lemma, however, assumes that there are no point arrivals. In Section 5, we present an algorithm that extends it to the sliding window context; it maintains the *k*-skyband dynamically and utilizes it to continuously report NN results as old points expire and new ones enter the system. The reduction from *k*NN to *k*-skyband monitoring applies to both kinds of sliding windows (i.e., count-based and time-based ones) because in both cases the expiration order is the same as the arrival order. Moreover, it extends to general data indexes, even though we focus on regular grids (for the reasons explained in the beginning of the section). Before introducing the skyband-based algorithm, we discuss the adaptation of CPM to sliding windows in the next section.

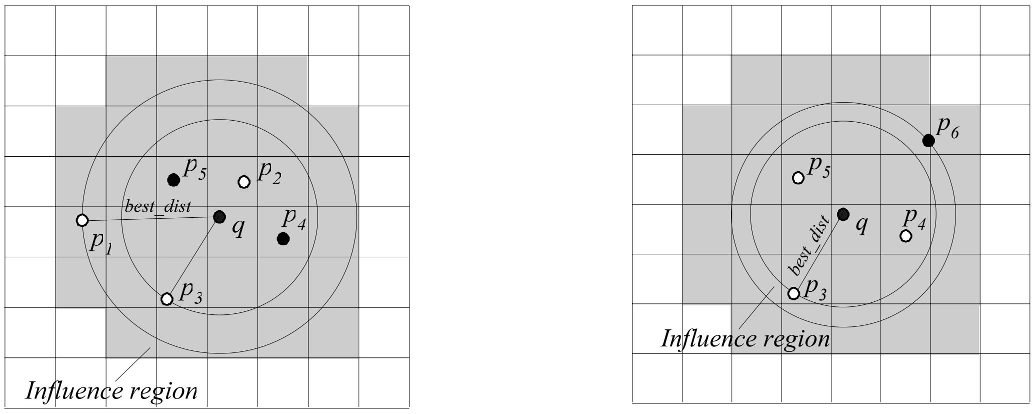
4. CPM ON SLIDING WINDOWS

CPM applies to the sliding window model by considering that the expiring points move infinitely far away from any query. However, several improvements of the update handling module are possible. The first-time result of a query q is produced with the NN computation algorithm of CPM, in the way discussed in Section 2.1. The processed cells constitute the influence region, and receive an entry for q in their influence lists. If *best_dist* is the distance of the k^{th} NN, the current result can change only due to point arrivals and expirations in the circle centered at q and radius equal to *best_dist*. Assume that, in the current processing cycle, a set P_{ins} of points arrive at the system, while a set P_{del} of existing ones expire. Before processing the updates, we initialize for each query q (i) a list *q.in_list* with maximum capacity of k entries to store the best incoming records[3], and (ii) a counter of outgoing NNs *q.out_count*=0. For each point $p \in P_{ins}$, we insert (a pointer to) p into the content list of the corresponding cell c. Next, we traverse the influence list IL_c. For every query $q \in IL_c$, we compare $dist(p,q)$ with *q.best_dist*. If $dist(p,q) \leq$ *q.best_dist*, then p is treated as an incoming point and is inserted into *q.in_list*. Note that *q.in_list* maintains only the k best incomers, since we do not need more than that in any case. Concerning the expirations, for each record $p \in P_{del}$, we delete it from its cell c, and traverse the influence list IL_c. For every $q \in IL_c$, we check whether p belongs to the current result *q.best_NN*. If $p \in$ *q.best_NN* we delete p from *q.best_NN* and increase *q.out_count* by one. The next step of the algorithm is to determine the new results. For each query q, if *q.in_list* contains at least as many points as *q.out_count*, then the result consists of the k best points in *q.best_NN* $\cup$ *q.in_list*. Otherwise (if *q.in_list* has fewer entries than *q.out_count*), the result of q is computed from scratch with the CPM NN search algorithm described in Section 2.1.

Consider the example of Figure 4.1a, where the result of a 3NN query q consists of records p_1, p_2 and p_3 (the existing points appear hollow). Assume that p_4 and p_5 arrive at the system (new points appear solid), while p_1 and p_2 expire. Current NNs p_1 and p_2 are expunged from the

[3] The list *q.in_list* is a temporary data structure, which is discarded after update handling terminates.

system, yielding *q.out_count*=2. On the other hand, arriving records p_4 and p_5 have distance less than *best_dist*, and are inserted into *q.in_list*. Since the number of incoming points is equal to *q.out_count*, we merge *q.in_list* with the remaining NNs (i.e., p_3) and form the new result *best_NN*=$\{p_5,p_4,p_3\}$. Even though *best_dist* changes, we do not update the influence lists of the cells that no longer influence *q* (i.e., the shaded cells that do not intersect the inner circle). The influence lists are updated only after a NN computation from scratch, as discussed next. This lazy approach does not affect the correctness of the algorithm because potential insertions (or deletions) in these cells are simply ignored (upon comparison with the new *best_dist*).



(a) $P_{ins}=\{p_4,p_5\}$, $P_{del}=\{p_1,p_2\}$ (b) $P_{ins}=\{p_6\}$, $P_{del}=\{p_3\}$

Figure 4.1: Update handling examples

Assume that in the next processing cycle $P_{ins}=\{p_6\}$ and $P_{del}=\{p_3\}$, as shown in Figure 4.1b. Point p_6 has larger distance than *best_dist* and, thus, it is simply inserted into its cell. The expiring NN p_3 yields *q.out_count*=1. Since there are no incoming points, the result of *q* has to be computed from scratch. The new NN set contains p_4, p_5 and p_6. Its influence region contains the cells intersecting the circle centered at *q* with radius equal to the new *best_dist*=*dist*(p_6,*q*). The final step of the algorithm is to remove *q* from the influence list of all cells (i.e., the shaded cells outside the outer circle in Figure 4.1b) that no longer influence *q* (recall from Figure 4.1a that the lists of these cells were not updated during the previous update handling). The updating procedure starts with the entries that remain[4] in H after the termination of the NN computation, and continues in a way similar to the NN search, but instead of processing the contents of the encountered cells, we simply delete q from their influence lists. The update terminates when de-heaping the first cell c whose IL_c does not contain q; the remaining cells do not contain q in their lists since their *mindist* is guaranteed to be higher than or equal to *mindist*(c,q). The complete CPM algorithm for the sliding window model is illustrated in Figure 4.2. The influence list updating procedure is performed in lines 19-26.

```
CPM
1.  In every processing cycle do
2.    $P_{ins}$ = set of arriving points; $P_{del}$ = set of expiring points
3.    For each query $q$ in $QT$
4.      Set $q.out\_count$=0, and initialize an empty list $q.in\_list$ of size $k$
5.    For each point $p$ in $P_{ins}$
6.      Insert $p$ into the content list of the corresponding cell $c$
7.      For each $q$ in $IL_c$
8.        If $dist(p,q) \leq q.best\_dist$, insert $p$ into $q.in\_list$
9.    For each point $p$ in $P_{del}$
10.     Delete $p$ from the content list of the corresponding cell $c$
11.     For each $q$ in $IL_c$
12.       If $p \in q.best\_NN$
13.         Delete $p$ from $q.best\_NN$, and set $q.out\_count = q.out\_count + 1$
14.   For each query $q$
15.     If the number of points in $q.in\_list$ is greater than or equal to $q.out\_count$
16.       Set $q.best\_NN$ = the $k$ best points in $q.best\_NN \cup q.in\_list$, and $q.best\_dist$ = the distance of the $k^{th}$ NN
17.     Else // the NN set of $q$ has to be re-computed
18.       Perform $k$NN computation from scratch, and set $H$ = the search heap after the NN search termination
19.       Repeat
20.         Get the next entry of $H$
21.         If it is a cell entry <$c$, $mindist(c,q)$>
22.           If there is an entry for $q$ in the influence list of $c$, remove it
23.           Else, go to line 14 and continue with the next query //i.e., the influence list updating is complete
24.         Else // it is a rectangle entry <$DIR_{lvl}$, $mindist(DIR_{lvl},q)$>
25.           For each cell $c$ in $DIR_{lvl}$, insert <$c$, $mindist(c,q)$> into $H$
26.           Insert <$DIR_{lvl+1}$, $mindist(DIR_{lvl+1},q)$> into $H$
27.   Report changes to the client
```

Figure 4.2: The sliding window version of CPM algorithm

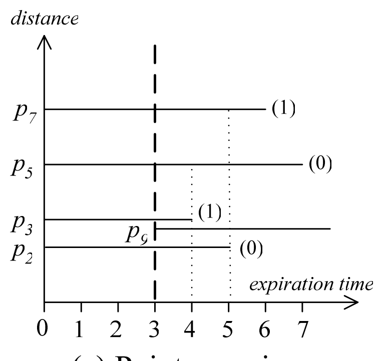
When a query q is terminated, we delete it from the query table and remove it from all the influence lists in the grid. The latter task is performed in a way similar to lines 19-26. Query

[4] Some cells and rectangles are en-heaped, even though their distance exceeds *best_dist.* These entries are not de-heaped during the result computation, and reside in the search heap H after the NN retrieval.

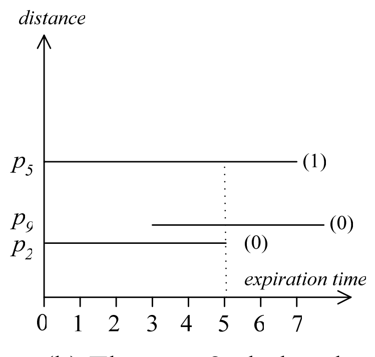
movement is handled as a deletion followed by a new query insertion.

5. Skyband *k*NN Monitoring

The Skyband NN (SNN) algorithm exploits the skyband in order to avoid computation from scratch when some NNs expire. Consider, for instance, a 2NN query *q* and the records of Figure 5.1a, shown as intervals in the two-dimensional *distance-time* space. The number in the parenthesis corresponds to the *dominance counter* (*DC*) of each point *p*, i.e., the number of points with smaller distance (to *q*) that arrive after[5] *p*. At time 0, the result of the 2NN query contains p_2 and p_3, whereas the 2-skyband contains p_2, p_3, p_5, p_7. At time 3, p_9 arrives, and expires after all other points in the system. It follows that (i) p_9 is not dominated by any point (i.e., $p_9.DC = 0$), and (ii) all the points *p* with $dist(p,q) \geq dist(p_9,q)$ are dominated by p_9. Therefore, the dominance counters of p_5, p_3, p_7 increase by one, i.e., $p_5.DC$=1 and $p_3.DC = p_7.DC = 2$. Consequently, p_3 and p_7 are removed from the 2-skyband at time 3. The updated 2-skyband, shown in Figure 5.1b, contains p_2, p_9 and p_5. The new 2NN set consists of the two elements in the skyband with the smallest distances (i.e., p_2 and p_9). After the expiration of p_2 (at time 5) the result changes to $\{p_9, p_5\}$.



(a) Point p_9 arrives



(b) The new 2-skyband

Figure 5.1: Skyband maintenance

[5] In both count-based and time-based windows the arrival order is the same as the expiration order.

In general, the monitoring of future results is reduced to a *k*-skyband maintenance task. SNN restricts the skyband maintenance for a query *q* to points falling inside its influence region. Specifically, the initial *k*NN set of *q* is retrieved by the NN computation module of CPM. The retrieved points are inserted into *q.skyband*, which contains entries of the form <*p.id*, *dist*(*p*,*q*), *p.DC*> in ascending order of *dist*(*p*,*q*). Then, SNN scans *q.skyband*, and for each point *p* it computes *p.DC*. To speed up the dominance counter computation, the arrival time of every processed element of *q.skyband* is stored into a balanced tree *BT* sorted in descending order. Thus, *p.DC* is simply the number of points that precede *p* in *BT* (since the NNs are processed in ascending distance order, these points are preferable to *p* in terms of both distance and expiration time). Each internal node in *BT* contains the cardinality of the sub-tree rooted at that node so that the computation of each dominance counter takes $O(\log k)$ time. After the dominance counter computation, *BT* is discarded and the *q.skyband* contains exactly *k* elements; *q.best_dist* is the distance of the *k*th element. The above procedure takes in total $O(k \cdot \log k)$ time.

The skyband maintenance algorithm handles only points *p* with *dist*(*p*,*q*) less than or equal to the *q.best_dist* after the previous NN computation from scratch. When such a point arrives at the system, it is inserted into *q.skyband* increasing its cardinality. The first *k* points of the skyband constitute the *q.best_NN* (in accordance with the CPM terminology), which is not stored explicitly. The dominance counter of all points with distance higher than *dist*(*p*,*q*) is increased by 1 and the ones whose counter reaches *k* are evicted. Regarding deletions, the element *p* of *q.skyband* with the earliest arrival time (i.e., the one expiring first) belongs to the current result, as can be shown by contradiction. Specifically, if the expiring point *p* was not in the current result, then all the *k* NNs would dominate *p* since they have smaller distance and expire later. Thus, *p* could not belong to the *k*-skyband. Returning to the maintenance procedure, when a point expires, it is removed and the first *k* elements of the updated *q.skyband* are reported as the new *q.best_NN*. Note that the deleted *p* does not dominate any other point, and therefore the dominance counters of the remaining elements in *q.skyband* are not affected.

The SNN algorithm is illustrated in Figure 5.2. An important remark concerns the situation where the skyband contains fewer than *k* points. This happens when some NNs expire and the recent arrivals were not inserted into the skyband (because their distance was larger than *q.best_dist*). In such cases, we have to compute the result from scratch and form a new skyband. The pseudo-code of Figure 5.2 handles this case in lines 18-20.

```
SNN
1.  In every processing cycle do
2.    $P_{ins}$ = set of arriving points; $P_{del}$ = set of expiring points
3.    For every point p in $P_{ins}$
4.      Insert p into the content list of the corresponding cell c
5.      For each q in $IL_c$
6.        If $dist(p,q) \le q.best\_dist$  // the distance of the $k^{th}$ NN after the last computation from scratch
7.          Insert p into q.skyband and set p.DC=0
8.          For each point p' in q.skyband with $dist(p',q) \ge dist(p,q)$
9.            p'.DC=p'.DC+1
10.           If p'.DC=k evict p' from q.skyband
11.   For every point p in $P_{del}$
12.     Delete p from the content list of the corresponding cell c
13.     For each q in $IL_c$
14.       If $p \in q.best\_NN$, delete p from q.skyband
15.   For each query q whose skyband has changed
16.     If q.skyband has at least k elements
17.       q.best_NN = the first k elements of q.skyband
18.     Else // q.skyband has fewer than k elements
19.       Perform kNN computation from scratch
20.       Form q.skyband and compute dominance counters therein
21.  Report changes to the client
```

Figure 5.2: The SNN algorithm

SNN is expected to be faster than CPM, since it involves less frequent calls to the NN search algorithm. For instance, consider the example of Figure 4.1b, where p_6 arrives and p_3 expires at the same processing cycle. As discussed in Section 4, in this scenario CPM re-computes the query from scratch. SNN, on the contrary, avoids the NN search overhead. Since *dist*(p_6,*q*) is less than *dist*(p_1,*q*) (i.e., the *best_dist* after the last re-computation from scratch - see Figure 4.1a), SNN inserts p_6 into the skyband, and directly reports it as the third NN when p_3 is deleted. On the other hand, the space requirements of SNN are higher than CPM, since it maintains the skyband (which is a superset of the current NN set) of each query. In the next section we analytically compare the performance and space requirements of the proposed algorithms.

6. PERFORMANCE ANALYSIS

Similar to previous approaches in the literature [KPH04, MHP05, XMA05, YPK05], we assume that (i) the average data cardinality at each timestamp is N, (ii) the valid positions are uniformly distributed in a unit two-dimensional workspace, and (iii) the stream rate is, on the average, r points per processing cycle. If δ is the cell extent per axis, the total number of cells is $(1/\delta)^2$ and each cell contains on the average $N\cdot\delta^2$ points. According to [MHP05], the running time of the kNN computation module (involved in both CPM and SNN) is $T_{comp}=O(C\cdot\log C + C\cdot N\cdot\delta^2\cdot\log k)$. The quantity C corresponds to the number of cells intersecting the influence region of a query and it holds that $C = O(\lceil k/(N\cdot\delta^2)\rceil)$. The term $O(C\cdot\log C)$ is due to heap operations (en-heaping/de-heaping cells and conceptual rectangles), and the term $O(C\cdot N\cdot\delta^2\cdot\log k)$ is due to updates of *q.best_NN* with encountered points, assuming that *q.best_NN* is implemented as a red-black tree.

Concerning the maintenance cost of CPM, in every processing cycle, r new points arrive at the system, while r old ones expire. Hence, the grid update time is $O(r)$. Each cell receives $r\cdot\delta^2$ insertions and $r\cdot\delta^2$ deletions. Therefore, the influence region of a query q is affected by $2\cdot C\cdot r\cdot\delta^2$ events. The time required to check whether the corresponding points belong to the current result is $O(C\cdot r\cdot\delta^2)$ (by comparing with *q.best_dist*). Among them, $k\cdot r/N$ new points are considered for insertion into *q.best_NN*, and $k\cdot r/N$ old ones are deleted from it; the total cost for updating *q.best_NN* is $O(k \cdot r \cdot \log k/N)$. Note that for uniform data distribution, the number of insertions in the influence region of q equals the number of deletions therein. Therefore, the number of incoming points equals the number of outgoing ones, and CPM does not invoke the *k*NN computation from scratch. In this case, the time complexity of CPM for a processing cycle is $T_{CPM} = O(r + Q \cdot (C \cdot r \cdot \delta^2 + k \cdot r \cdot \log k/N))$, where Q is the number of running queries.

For SNN, the index update cost is the same as for CPM (i.e., $O(r)$). Also, the number of the arriving (expiring) points in the cells intersecting the influence region of a query q is $O(C \cdot r \cdot \delta^2)$. Initially (after the application of the *k*NN computation module), the skyband contains k elements. Among the inserted (deleted) points, $O(k \cdot r/N)$ have distance less than *q.best_dist* and have to be included in (excluded from) the skyband. An insertion to *q.skyband* requires $O(k)$ time, because we have to retain the order (according to distance), and at the same time update the dominance counters of the entries with distance higher than that of the new point. Each deletion also has $O(k)$ cost. Similar to CPM, according to the uniformity assumption, the *k*-skyband contains exactly k elements, and SNN does not resort to computations from scratch. In summary, the total running time is $T_{SNN}=O(r + Q \cdot (C \cdot r \cdot \delta^2 + k^2 \cdot r/N))$ for each processing cycle.

Finally, we analyze the memory requirements of the proposed methods. The index has $O(N + N + Q \cdot C)$ size, where $O(N)$, $O(N)$ and $O(Q \cdot C)$ are the amounts of storage required for the N valid points, for N pointers (in the content lists of the cells), and for the influence lists of the Q queries. Each query table entry for CPM has size $O(2 + 2 \cdot k)$, for storing the query coordinates and the tuple <*p.id*, *dist*(*p*,*q*)> for every point p in the result. For SNN, each entry of *QT* takes up $O(2 + 3 \cdot k)$, since in addition to the identifier and the distance, *q.skyband* also contains the dominance counters of the points. Recall that SNN does not need to explicitly store *q.best_NN*, because the result set consists of the first k entries of *q.skyband*. Summarizing, the space requirements of

CPM and SNN are $S_{CPM}=O(N + Q\cdot(C + d + 2\cdot k))$ and $S_{SNN}=O(N + Q\cdot(C + d + 3\cdot k))$, respectively.

In summary, for uniform data the result updating of CPM is more efficient than the skyband maintenance of SNN (with time complexities $O(k\cdot r\cdot \log k/N)$ and $O(k^2\cdot r/N)$ per query, respectively). For non-uniform data, however, SNN is expected to be faster than CPM because the latter one resorts more frequently to *k*NN computation from scratch. This is also verified by our experiments. Concerning the space overhead, SNN uses more memory than CPM because (i) the *q.skyband* stores additional information about the dominance counters, and (ii) in practice, the *k*-skyband may contain more than *k* entries. The performance of both algorithms depends on the cell side-length δ. Large cells minimize the time spent on heap operations, but lead to unnecessary processing of points that are outside the influence region (but fall in cells that intersect the influence region). Large δ also implies lower space consumption, because queries are affected by fewer cells, and the cell influence lists take up less memory. The running time of the proposed techniques increases with *k*, *Q*, *N*, and *r*. The same holds for the space consumption, with the exception of *r*.

7. OTHER NN QUERY TYPES

In this section we extend the proposed algorithms to variations of NN search. In particular, we describe the monitoring of *constrained* and *aggregate nearest neighbor* queries. A constrained NN query *q* specifies a region of interest and requests the NNs of *q* therein [FSAA01]. Consider for instance the example of Figure 7.1, where the user requests the NN of *q* among the points that have higher *x* and *y* coordinates than *q* (i.e., the region of interest is the striped area). CPM and SNN can be easily adapted to monitor constrained NNs over sliding windows. The difference is that during the initial NN set computation, we en-heap only cells and conceptual rectangles that intersect the region of interest, and process only points that fall inside it.

In Figure 7.1, the algorithm en-heaps rectangles U_0, U_1, U_2, R_0, R_1, and R_2, it processes the shaded cells, and it returns p_3 as the result. Note that the un-constrained NN of q is point p_2, but it is not encountered because its cell is not visited. On the other hand, point p_4 is processed, but ignored because it falls outside the (constrained) region of interest. Concerning the monitoring of result changes, neither CPM nor SNN require modifications. The de-heaped cells (appearing shaded in Figure 7.1) receive an entry for q in their influence lists, and only updates therein are monitored.

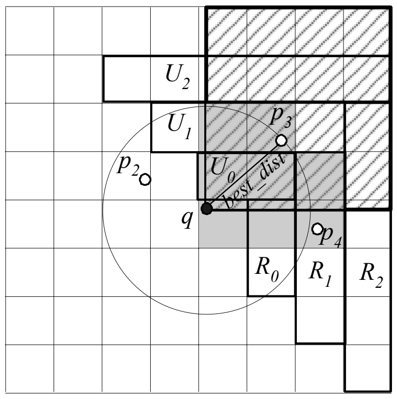


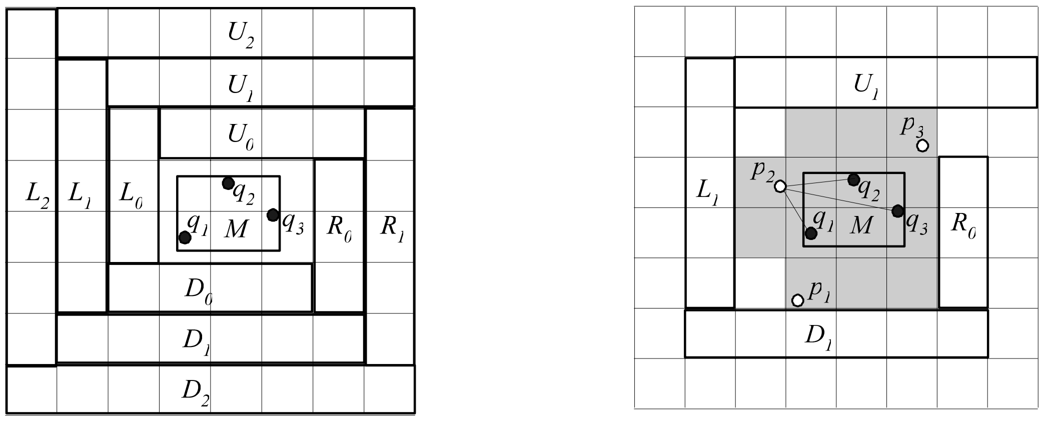
Figure 7.1: A constrained NN search example

Another interesting variant is the aggregate nearest neighbor query (ANN). Given a set of query points $Q = \{q_1,q_2, ...,q_n\}$ and a data point p, the aggregate distance *adist*(p,Q) is defined as a function f over the individual distances *dist*(p,q_i) between p and each point $q_i \in Q$. The result of the ANN query is the point p that minimizes *adist*(p,Q). Papadias et al. [PTMH05] propose algorithms for snapshot ANN queries on static datasets when f is a monotonically increasing[6] function over the individual distances *dist*(p,q_i). Under the same assumption (i.e., monotonicity of f), both CPM and SNN extend to ANN monitoring over sliding windows. In the following, we focus on the *sum*, *max* and *min* aggregate functions, as they are the most commonly used ones.

[6] A function f is monotonically increasing iff: $x_i \geq x'_i \ \forall i$ implies that $f(x_1, ...,x_n) \geq f(x'_1,...,x'_n)$.

Consider n users at locations q_1, q_2, ..., q_n and f=*sum*. The aggregate NN minimizes $adist(p,Q) = \sum_{q_i \in Q} dist(p,q_i)$, i.e., the *sum* of distances that the users have to travel in order to meet at the position of p. Similarly, if f=*max*, the ANN query reports the point p that minimizes the maximum distance that any user has to travel to reach p. In turn, this leads to the earliest time that all users will arrive at the location of p (assuming that they move with the same speed). Finally, if f=*min*, the result is the point p which is closest to any user, i.e., p has the smallest $adist(p,Q) = min_{q_i \in Q}\, dist(p,q_i)$.

To extend our algorithms to continuous ANN monitoring, we have to use a different partitioning of the space (than that of simple NN queries). Consider the example of Figure 7.2a, where $Q = \{q_1,q_2,q_3\}$. The partitioning applies to the space around the minimum bounding rectangle M of Q, as shown in the figure. Given a rectangle *rect*, the function $amindist(rect,Q) = f_{q_i \in Q}\, mindist(rect,q_i)$ is a lower bound of the distance $adist(p,Q)$ for any point p in *rect*. Due to the monotonicity of f, the *amindist* of the conceptual rectangles in a direction is increasing with their level number. This property allows for the application of the conceptual partitioning methodology to compute the first-time result.



(a) Conceptual partitioning (b) Processing when f = *sum*

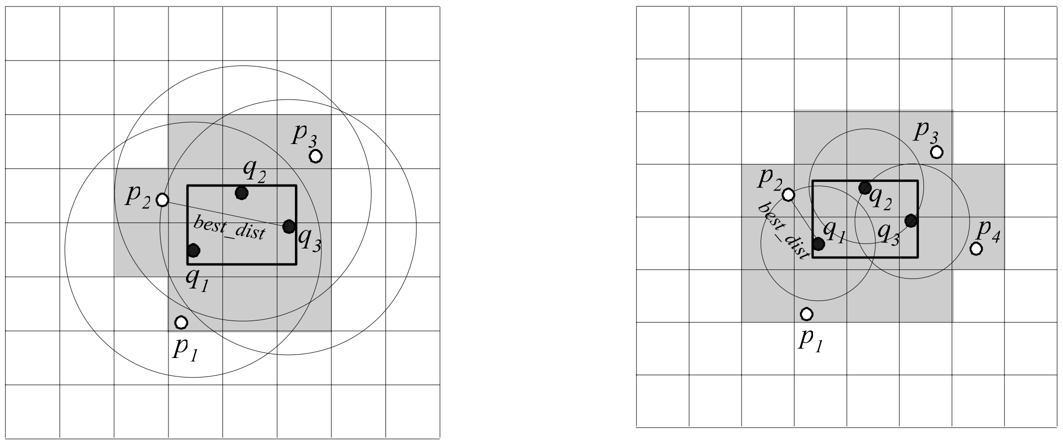
Figure 7.2: An ANN search example

The ANN search initially en-heaps the cells c intersecting M with key *amindist*(c,Q), and the level zero rectangles for each direction DIR with key *amindist*(DIR_0,Q). Then, it starts de-heaping entries iteratively. If the de-heaped entry is a cell c, then the algorithm computes *adist*(p,Q) for every p inside c, and updates the list *best_NN* of the best points found so far. It also inserts q into the influence list IL_c. If the entry is a conceptual rectangle, then it en-heaps the corresponding cells and the next level rectangle in the same direction, with keys equal to their *amindist*. The procedure terminates when the next entry in the heap has key equal to or greater than the distance *best_dist* of the k^{th} ANN found.

Returning to our running example and letting $f = sum$, the ANN search en-heaps all the cells falling in M, U_0, D_0, and L_0, and de-heaps the ones appearing shaded in Figure 7.2b. It processes points p_1, p_2 and p_3, and returns p_2 as the result. The monitoring of the ANN set upon point arrivals and expirations is the same as in Sections 4 and 5 for CPM and SNN, respectively. The only difference is that now the measure of interest is the aggregate distance of the points. In the case of SNN, this implies that the k-skyband is computed and maintained in the *aggregate distance-time* space.

The algorithms also apply to *max* and *min* ANN query monitoring, by defining *amindist*(*rect*,Q) and *adist*(p,Q) accordingly. Consider Figures 7.3a and 7.3b, where $f = max$ and $f = min$, respectively. The ANN search processes the shaded cells and returns p_2 as the result in both cases. Note that for $f = max$ it visits the cells that overlap with the intersection of all circles with centers at q_i and radii equal to *best_dist*, because these cells have *amindist*(c,Q) < *best_dist* and could potentially contain points with lower aggregate distance than *best_dist*. For the same reason, when $f = min$, it processes the cells that overlap with at least one of the circles with centers at q_i and radii equal to *best_dist*.

The number n of query points in Q may be large, and computing the aggregate distance of points (cells) may be very expensive because it requires calculation of n Euclidean distances (*mindist* functions). Depending on the definition of f, some points (cells) can be pruned without computing all these n distances. For example, assume that $f = sum$. If while computing *adist*(p,Q) (*amindist*(c,Q)) the sum of distances calculated so far exceeds the current *best_NN*, then point p (cell c) can be immediately pruned (without considering the remaining points in Q). Similarly, when $f = max$, if the distance of point p (*mindist* of cell c) from one of the query points is already larger than *best_NN*, then p (c) can be safely excluded from consideration without wasting further computations for the exact value of *amindist*. On the other hand, in the case of *min*, such an optimization is not possible.



(a) $f = max$ (b) $f = min$

Figure 7.3: ANN search examples for $f = max$ and $f = min$

8. EXPERIMENTAL EVALUATION

In this section we experimentally evaluate CPM and SNN. Sections 8.1 and 8.2 study their performance for NN and ANN monitoring, respectively. In both cases, the input stream is created by randomly choosing points from a real dataset of 1,314,620 two-dimensional points, corresponding to endpoints of streets in Los Angeles (available at *www.rtreeportal.org*). The dataset is normalized to cover a unit workspace (i.e., [0,1]×[0,1]). We assume count-based windows with size *N* between 100K and 1M records. During each timestamp, *r* new points arrive at the system. In our NN monitoring experiments (Section 8.1), we use two sets of queries; in UNI, queries are uniformly distributed in the workspace, while in SKW, they are randomly drawn from our real dataset (i.e., they follow the same distribution as the stream points). In Section 8.2, each ANN query consists of *n* points uniformly distributed in a square. The square covers area A_q, and its location is randomly chosen in the workspace. The simulation length is 100 timestamps. Table 8.1 summarizes the parameters under investigation, along with their ranges and default values. In each experiment we vary a single parameter, while setting the remaining ones to their default values. The asterisk next to a description indicates that it is used only in the ANN experiments. For all simulations we use a Pentium 3.2 GHz CPU with 1 GByte memory.

Parameter	**Default**	**Range**
Number of valid points (*N*)	100K	100, 300, 500, 700, 1000 (K)
Arrival rate (*r*)	1K	0, 10, 30, 50, 70, 100 (K)
Number of queries (*Q*)	1K	1, 2, 3, 4, 5 (K)
Number of NNs (*k*)	16	1, 4, 16, 64, 256
Area of ANN query (A_q) *	4%	1%, 2%, 4%, 8%, 16%
Number of points in ANN query (*n*) *	16	4, 8, 16, 32, 64

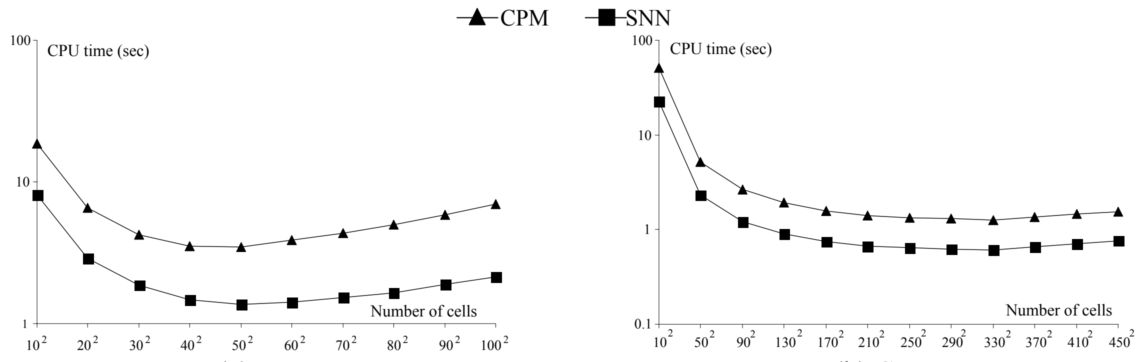
Table 8.1: System parameters

8.1 Nearest Neighbor Monitoring

In this section we focus on monitoring of conventional NN queries. First, we study the effect of the grid granularity on CPM and SNN for the default settings (i.e., *N*=100K, *r*=1K, *Q*=1K, *k*=16). For UNI queries (Figure 8.1a) we experiment on grids with 10^2 up to 100^2 cells, while for SKW (Figure 8.1b) we reach up to 450^2 because the optimal granularity[7] is much higher than

[7] Optimal here refers to the granularity that leads to the lowest CPU cost for the default parameters.

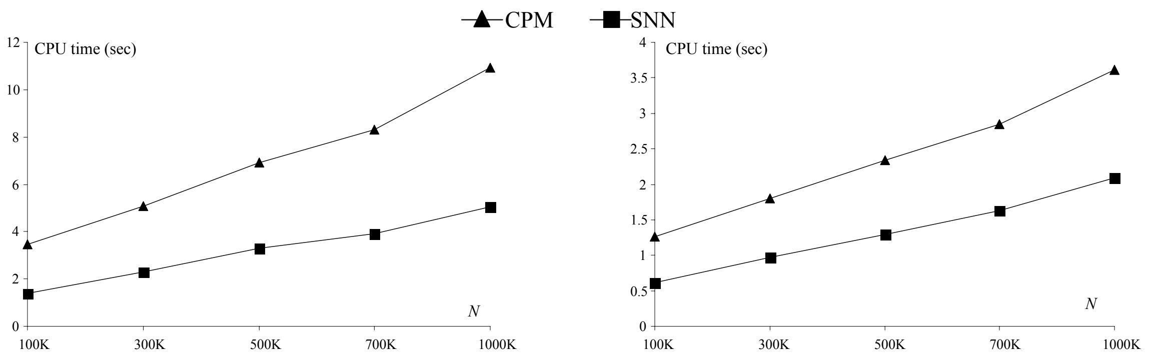
UNI. We plot the overall running time in seconds, in logarithmic scale. The best performance for both algorithms is achieved with a 50×50 grid for UNI, and a 330×330 one for SKW. The optimal grid granularity for SKW is much finer, because SKW queries follow the data distribution and the cells around them contain many points. In both cases, a very fine grid is expensive because of the heap operations on the cells, whereas a sparse one leads to unnecessary processing of points outside the query influence regions. For the remaining experiments, we use the respective optimal granularities for UNI and SKW.



(a) UNI (b) SKW

Figure 8.1: CPU time versus grid granularity

In Figure 8.2 we vary *N* from 100K to 1M, and set the arrival rate *r* to *N*/100 tuples per timestamp (i.e., during each timestamp, 1% of the data points are replaced by new ones). As shown in Figures 8.2a and 8.2b, the running time increases with *N*. SNN is more than two times faster than CPM for both UNI and SKW. Over the 100 timestamps of the simulation, for UNI (SKW), CPM computes a query from scratch 12.9 (13.6) times on the average, versus only 4.4 (4.9) for SNN. An interesting observation, which is apparent in all experiments, is that both algorithms are slower for UNI. This happens because in UNI the queries are more likely to lie far away from their NNs (as they follow different distribution from the data), and NN search en-heaps/de-heaps many cells before retrieving the results.



(a) UNI (b) SKW

Figure 8.2: CPU time versus number of valid points *N* ($r = N/100$)

Next, we set *N*=100K and vary *r* between 0 and 100K, i.e., 0% up to 100% of the valid points are replaced per timestamp. Figures 8.3a and 8.3b show the running time versus *r* for UNI and SKW, respectively. The number next to each measurement is the average number of NN computations from scratch (during the 100 timestamps of the simulation) per query. The performance degrades with *r*, because a larger arrival rate causes more frequent re-computations and higher index update cost. SNN is better than CPM, except for $r = N$ (i.e., 100K) and SKW queries. In this case, even though SNN performs fewer re-computations, it is slower than CPM, because (i) NN search is relatively cheap for SKW (as the NNs are found close to the queries), and (ii) the cost of updating the skybands and the dominance counters is high (in every timestamp, *k* insertions and *k* deletions take place in each of them). Note that for $r = 0$ the algorithms have the same cost, since they both retrieve the initial result of each query, and do not perform any further computation (there are no data insertions/deletions in the subsequent timestamps).

In order to study the effect of the query cardinality, we vary *Q* between 1K and 5K, and plot the running time for UNI and SKW in figures 8.4a and 8.4b, respectively. The CPU cost of both methods scales linearly with *Q*, and SNN is the best algorithm. Similar to the data cardinality (Figure 8.2), the performance gap increases with *Q*, verifying the better scalability of SNN to large problems.

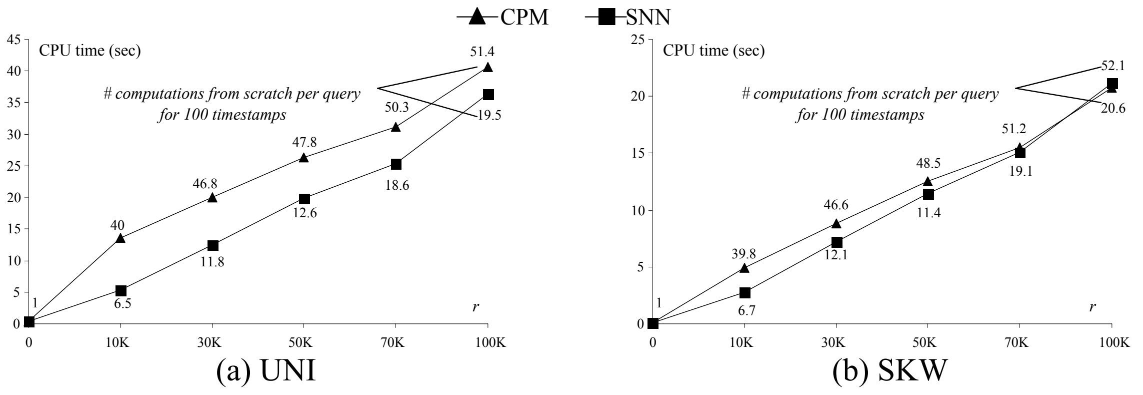


Figure 8.3: CPU time versus arrival rate *r*

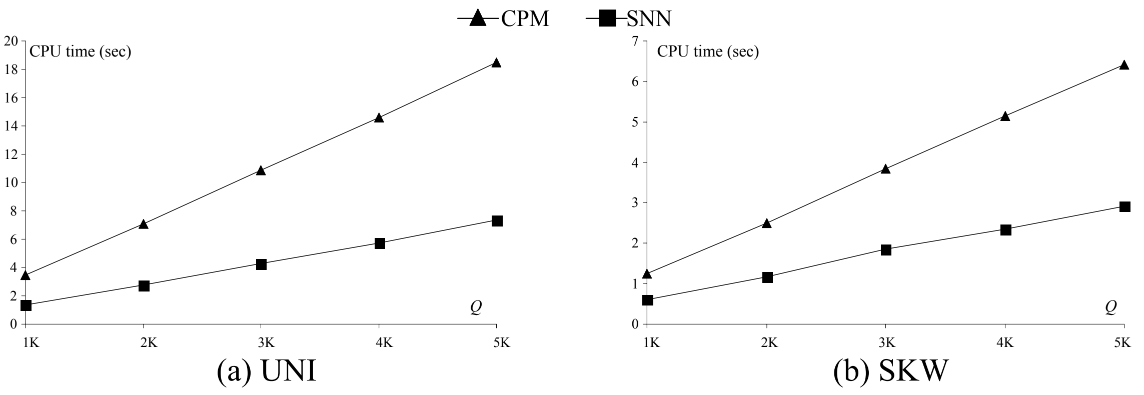


Figure 8.4: CPU time versus query cardinality *Q*

Figures 8.5a and 8.5c show the processing time versus the number *k* of NNs for UNI and SKW. The influence region of the queries and, consequently, the number of processed cells/points increases with *k*, implying higher NN retrieval and maintenance overhead for both methods. SNN is faster in all cases and, since it performs fewer re-computations than CPM, its degradation with *k* is smaller. Figures 8.5b and 8.5d illustrate the corresponding space requirements. SNN consumes only a few KBytes more space than CPM. A larger *k* implies longer influence lists and, thus, higher memory consumption for both methods. The numbers appearing above the measurements for SNN correspond to the average cardinality of the skybands in the system. Interestingly, SNN maintains very few extra points.

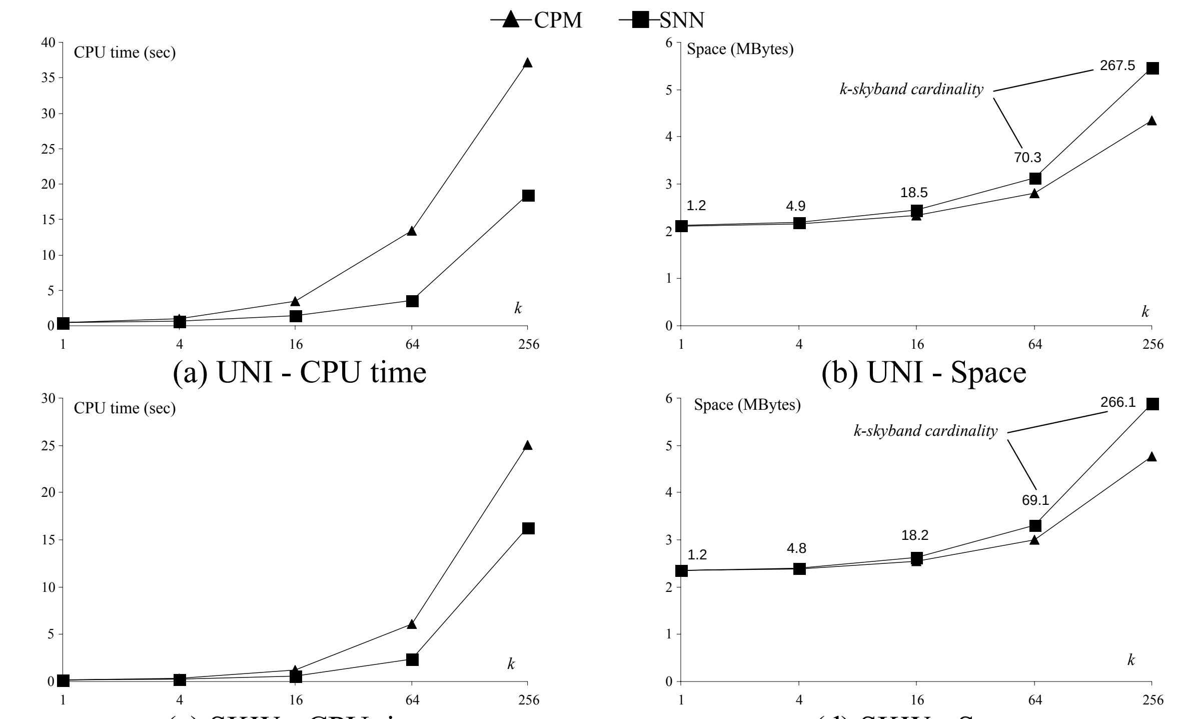
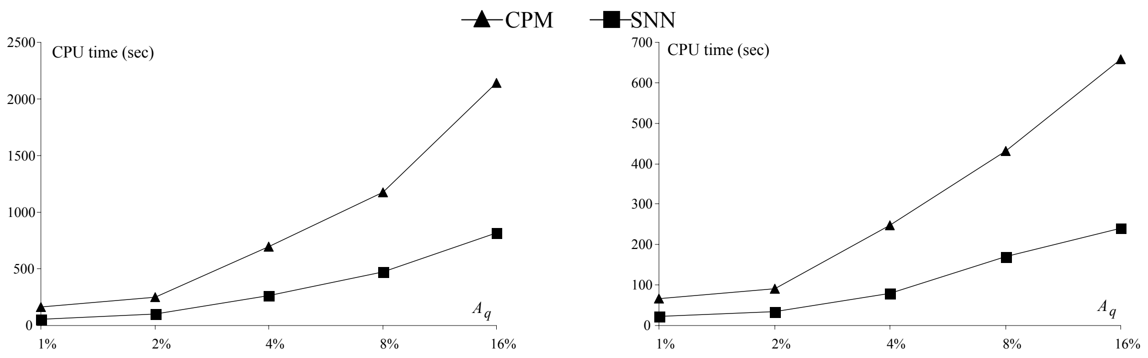


Figure 8.5: Performance versus number *k* of NNs

8.2 Aggregate Nearest Neighbor Monitoring

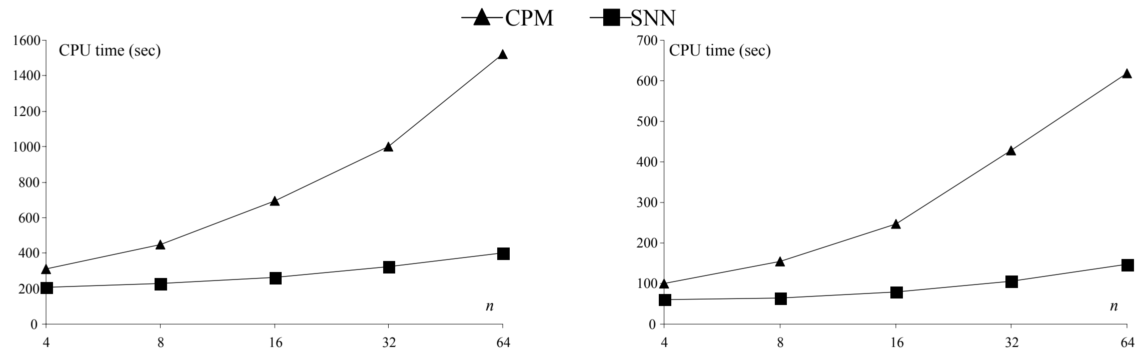
In this section, we evaluate our methods for ANN monitoring. We focus on *sum* and *max* aggregate functions. The results for *min* were very similar to *sum* and, thus, omitted. First, we fine-tune the grid granularity, similar to Figure 8.1. The best grid size is 140×140 for *sum* and 80×80 for *max*; we use this granularity for all following experiments. Figure 8.6 shows the CPU time versus the area A_q of the minimum bounding rectangle of the queries (using the default settings for the remaining parameters, i.e., N=100K, r=1K, Q=1K, k=16). We vary A_q from 1% to 16% of the total workspace area. For small A_q, the points of each query are close to each other, leading to small influence regions. When A_q is larger, ANN retrieval and maintenance considers more cells/data points. SNN is 2.5 - 3 times better than CPM for both f = *sum* and f = *max*. Both methods are faster for *max*, because the optimization for aggregate distance calculation (presented in the last paragraph of Section 7) saves more computations than for *sum*.



(a) *f* = *sum* (b) *f* = *max*

Figure 8.6: CPU time versus area of query MBR A_q

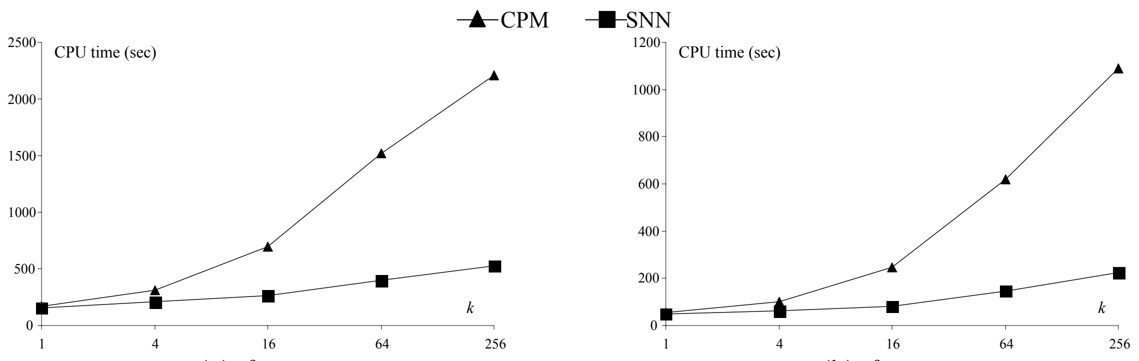
Figure 8.7 studies the effect of *n* (i.e., the number of points in each query). Aggregate distance calculations (for points and cells) are more expensive for larger *n*, leading to higher ANN computation and maintenance costs. Since the advantage of SNN over CPM is the reduced number of ANN retrievals from scratch, their difference grows as *k* (and, consequently, the cost per ANN retrieval) increases.



(a) *f* = *sum* (b) *f* = *max*

Figure 8.7: CPU time versus number of points in query *n*

Figure 8.8a (8.8b) shows the CPU time versus *k* for *f* = *sum* (*f* = *max*). The performance of both algorithms degrades with *k*, because the influence regions grow. SNN is faster in all cases. Its difference from CPM increases for larger *k* because, similar to Figure 8.7, NN computations become more costly.



(a) *f* = *sum* (b) *f* = *max*

Figure 8.8: CPU time versus number *k* of ANNs

9. CONCLUSION

This paper constitutes the first work addressing the problem of continuous *k*NN monitoring over sliding windows. As a first solution, we extend the state-of-the-art NN monitoring algorithm for update streams (CPM), to the sliding window model. Next, we present SNN, which utilizes a generalized concept of skybands for maintaining NNs. Both approaches compute the initial result of each query with an algorithm that processes the minimum number of cells. Only insertions/deletions within these cells can potentially invalidate the current *k*NN set. Therefore, the maintenance of the result considers only point arrivals and expirations therein. The difference of SNN from CPM is that it maintains a superset of the current result in the form of a *k*-skyband in the *distance-time* space. Both methods apply to time-based and count-based windows. Moreover, they can be easily adapted to other query types, such as constrained and aggregate NNs. An extensive experimental evaluation demonstrates that SNN outperforms CPM for all parameter settings, while consuming a negligible amount of extra space.

A direction for future work concerns the derivation of cost models for non-uniform data. For instance, the proposed models could be extended and combined with multi-dimensional histograms to provide accurate estimations for query optimization (in systems that involve monitoring of multiple query types). Another interesting direction would be the development of methods on non-regular grids (recall that all existing methods apply regular grids). In this case, the partitioning of the data space should take into account the data distribution, which may change with time. Although non-regular grids complicate query processing, they are expected to yield performance gains for highly skewed data. Finally, we plan to investigate distance functions that take into account freshness, in addition to distance; i.e., the data do not expire when they fall out of the window, but their utility continuously drops with time.

REFERENCES

[BBD+02] Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J. Models and Issues in Data Stream Systems. *PODS*, 2002.

[BGZ04] Balke, W., Gunzer, U., Zheng, J. Efficient Distributed Skylining for Web Information Systems. *EDBT*, 2004.

[BJKS06] Benetis, R., Jensen, C., Karciauskas, G., Saltenis, S. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. *VLDB Journal,* 15(3): 229-250, 2006.

[BKSS90] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[BKS01] Borzsonyi, S, Kossmann, D., Stocker, K. The Skyline Operator. *ICDE*, 2001.

[CCC+02] Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S. Monitoring Streams - A New Class of Data Management Applications. *VLDB*, 2002.

[CHC04] Cai, Y., Hua, K., Cao, G. Processing Range-Monitoring Queries on Heterogeneous Mobile Objects. *MDM*, 2004.

[FSAA01] Ferhatosmanoglu, H., Stanoi, I., Agrawal, D., Abbadi, A. Constrained Nearest Neighbor Queries. *SSTD*, 2001.

[G84] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching, *SIGMOD*, 1984.

[GL04] Gedik, B., Liu, L. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. *EDBT*, 2004.

[H84] Henrich, A. A Distance Scan Algorithm for Spatial Access Structures. *ACM GIS*, 1984.

[HS99] Hjaltason, G., Samet, H. Distance Browsing in Spatial Databases. *ACM TODS*, 24(2): 265-318, 1999.

[KOTZ04] Koudas, N., Ooi, B., Tan, K., Zhang, R. Approximate NN queries on Streams with Guaranteed Error/performance Bounds. *VLDB*, 2004.

[KPH04] Kalashnikov, D., Prabhakar, S., Hambrusch, S. Main Memory Evaluation of Monitoring Queries Over Moving Objects. *Distributed and Parallel Databases*, (15)2:117-135, 2004.

[KRR02] Kossmann, D., Ramsak, F., Rost, S. Shooting Stars in the Sky: an Online Algorithm for Skyline Queries. *VLDB*, 2002.

[LMT+05] Li, J., Maier, D., Tufte, K., Papadimos, V., Tucker, P. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. *SIGMOD*, 2005.

[LYWL05] Lin, X., Yuan, Y., Wang, W., Lu, H. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. *ICDE*, 2005.

[MHP05] Mouratidis, K., Hadjieleftheriou, M., Papadias, D. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. *SIGMOD*, 2005.

[MXA04] Mokbel, M., Xiong, X., Aref, W. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. *SIGMOD*, 2004.

[PTFS05] Papadias, D., Tao, Y., Fu, G., Seeger, B. Progressive Skyline Computation in Database Systems. *ACM TODS*, 30(1), 41-82, 2005.

[PTMH05] Papadias, D., Tao, Y., Mouratidis, K., Hui, C. Aggregate Nearest Neighbor Queries in Spatial Databases. *ACM TODS*, 30(2), 529-576, 2005.

[PXK+02] Prabhakar, S., Xia, Y., Kalashnikov, D., Aref, W., Hambrusch, S. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 51(10): 1124-1140, 2002.

[RKV95] Roussopoulos, N., Kelly, S., Vincent, F. Nearest Neighbor Queries. *SIGMOD*, 1995.

[SR01] Song, Z., Roussopoulos, N. K-Nearest Neighbor Search for Moving Query Point. *SSTD*, 2001.

[TEO01] Tan, K., Eng, P. Ooi, B. Efficient Progressive Skyline Computation. *VLDB*, 2001.

[TP03] Tao, Y., Papadias, D. Spatial Queries in Dynamic Environments. *ACM TODS*, 28(2): 101-139, 2003.

[TP06] Tao, Y., Papadias, D. Maintaining Sliding Window Skylines on Data Streams. *IEEE TKDE*, 18(3): 377–391, 2006.

[XMA05] Xiong, X., Mokbel, M., Aref, W. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. *ICDE*, 2005.

[YPK05] Yu, X., Pu, K., Koudas, N. Monitoring K-Nearest Neighbor Queries Over Moving Objects. *ICDE*, 2005.

[ZL01] Zheng, B., Lee, D. Semantic Caching in Location-Dependent Query Processing. *SSTD*, 2001.

[ZZP+03] Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D. Location-based Spatial Queries. *SIGMOD*, 2003.

Kyriakos Mouratidis is an Assistant Professor at the School of Information Systems, Singapore Management University. He received his BSc degree from the Aristotle University of Thessaloniki, Greece, and his PhD degree in Computer Science from the Hong Kong University of Science and Technology. His research interests include spatiotemporal databases, data stream processing, and mobile computing.

Dimitris Papadias is a Professor at the Computer Science and Engineering, Hong Kong University of Science and Technology. Before joining HKUST in 1997, he worked and studied at the German National Research Center for Information Technology (GMD), the National Center for Geographic Information and Analysis (NCGIA, Maine), the University of California at San Diego, the Technical University of Vienna, the National Technical University of Athens, Queen's University (Canada), and University of Patras (Greece). He has published extensively and been involved in the program committees of all major Database Conferences, including SIGMOD, VLDB and ICDE. He is an associate editor of the VLDB Journal, the IEEE Transactions on Knowledge and Data Engineering, and on the editorial advisory board of Information Systems.