# Incremental Nearest-Neighbor Search in Moving Objects

Katerina Raptopoulou, Apostolos N. Papadopoulos, Yannis Manolopoulos
Department of Informatics, Aristotle University
54124 Thessaloniki, GREECE
{*katerina,apostol,manolopo*}*@delab.csd.auth.gr*

## Abstract

In databases of moving objects it is important to answer queries that concern the future positions of the objects. An important query type in such an environment is the nearest-neighbor query, which asks for the $k$ closest objects of a query object during a time interval $[t_s, t_e]$. However, there are cases where the ($k$+1)-th nearest-neighbor is requested after the execution of the $k$-NN query. In such a case, either the query must be evaluated again, or we can exploit the previous result and use an incremental method to determine the new answer. We focus on the second alternative and present efficient incremental algorithms that outperform the trivial method which is based on complete re-execution of the query. In addition, we study the problem of keeping the query result consistent in the presence of object insertions, deletions and updates which are very common in a dynamic moving-object environment.

**Keywords:** *spatiotemporal databases, nearest-neighbors, location-based services, moving objects.*

## 1 Introduction

Spatiotemporal databases combine the spatial and temporal characteristics of the stored data [22, 30, 29]. Nowadays, an important research direction in the field is the design of algorithms and access methods, towards query processing in databases of moving objects. Applications that could benefit from the efficient manipulation of moving objects include geographical information systems, navigation and tracking control, fleet management, mobile information systems and weather forecasting, to name a few. These applications require that the position of the objects at specific time instances is known (or can be computed). In this work we assume that each mobile object is capable of 1) determining its position and 2) transmitting its position to a server.

There are two different directions concerning the positions of moving objects. The first one studies the problem of processing *past queries*, which refer to past positions of objects. The trajectory of each object is stored in the database and specialized indexing schemes are used to speed up searching. The object trajectories are handled: a) by spatial access methods tuned to support time such as the 3D-Rtree [28], b) by historical access methods which use the concept of tree overlapping [12, 14, 24], or c) by specialized access methods for object trajectories [32, 15, 25]. The second direction involves the processing of *future queries* with respect to future positions of the objects. Each object is usually characterized by its reference position and its velocity vector. These two parameters are very important in order to be able to predict future object positions. Future queries are supported by access methods that either use transformations and map objects to a feature space, or they work directly in native space and use access methods that can predict the object positions according to the current motion characteristics [9, 10, 1, 20, 17, 7, 8, 11].

Among the different types of queries that can be posed to a database of moving objects, we focus on the $k$-nearest-neighbor query. Given a future time interval $[t_s, t_e]$ (where $t_s \geq t_{now}$), an integer $k$ and a query object $q$, the $k$-NN query asks for the $k$ closest objects to $q$ during $[t_s, t_e]$. Since the query object and the data objects move, it is evident that the set of nearest-neighbors may change. In other words, it is likely that the set of NNs at time $t_s$ will be different from the set of NNs at time $t_e$. An interesting variation of the problem is to compute the ($k$+1)-th nearest-neighbor, given the result of the $k$-NN query. The obvious technique that we can use is to issue a new ($k$+1)-NN query. This approach requires high computation costs both in CPU and I/O time. There are several optimizations that can be applied in order to reduce these costs and exploit the result of the $k$-NN query which has been evaluated previously.

Another issue that is of great importance in databases of moving objects is the ability to adapt the query results according to the object mobility. For example, the result of a $k$-NN query may be invalidated due to one of the following reasons:

- If a new object is inserted in the dataset we have to check whether this insertion causes changes to the query result. In this case a method is required in order to efficiently handle such a case. Otherwise the result remains as it is.
- If an object is deleted from the dataset we have to check if this deletion affects the query result. For

example, if object $o$ participates to the results of a $k$-NN query and it is deleted, we have to update the query result. Otherwise no particular action is required.

- Finally, when during the execution of a query an object update occurs, we have to check if this update invalidates the query result. An update can be represented as an object deletion followed by an object insertion.

The rest of the paper is organized as follows. Section 2 presents the appropriate background and discusses related work in the area of $k$-NN search in moving objects and incremental algorithms. In Section 3 we study incremental algorithms for moving objects whereas in Section 4 we present methods to handle object insertions, deletions and updates. Section 5 presents the performance evaluation results for the proposed methods. Concluding remarks and future directions are offered in Section 6.

## 2 Background and Related Work

The nearest-neighbor query received considerable attention in spatial databases, where data objects and queries are assumed to be static (i.e., their position remains constant). Several algorithms and cost models for $k$-NN have been proposed in the literature, such as [19, 21]. Although these algorithms are efficient in a spatial database context, they are inadequate for spatiotemporal datasets, where data objects and queries change their position with respect to time. Therefore, specialized algorithms have been proposed, taking into consideration the mobility of the dataset. Some of the proposed methods are applied in the case where only query objects are allowed to move, whereas data objects remain stable. However, several proposals handle the more general case where data objects and queries are allowed to move.

Kollios et al. [9] propose a method able to answer NN queries for moving objects in 1-D space. The method is based on the dual transformation where a line segment in the native space corresponds to a point in the transformed space, and vice-versa. The method determines the object that comes closer to the query between $[t_s, t_e]$ and not the NNs for every time instance.

Zheng et al. [33] proposed a method for computing a single NN ($k = 1$) of a moving query, applied to static points indexed by an R-tree. The method is based on Voronoi diagrams and it seems quite difficult to be extended for other values of $k$ and higher space dimensions.

In [23] a method is presented to answer such queries on moving-query, static-objects cases. Objects are indexed by an R-tree, and sampling is used to query the R-tree at specific points. However, due to the nature of sampling, the method may return incorrect results if a split point is missed. A low sampling rate yields more efficient performance, but increases the probability of incorrect results, whereas a high sampling rate poses unnecessary computational overhead, but decreases the probability of incorrect results.

Benetis et al. [3] propose an algorithm capable of answering NN queries and reverse NN queries in moving-object datasets. However, the proposed method is restricted in answering only one NN per query, and no results are given for the $k$-NN query problem. In addition to the algorithms, the authors propose methods to keep the query results updated due to insertions and deletions of data objects.

In [26] the authors propose an NN query processing algorithm for moving-query moving-objects, based on the concept of time-parameterized queries. Each query result is composed of the following components: i) $R$, is the current result set of the query, ii) $T$, is the time point in which the result becomes invalid, and iii) $C$, the set of objects that influence the result at time $T$. Therefore, by continuously calculating the next set of objects that will influence the result, we determine the NNs of the query from $t_1$ to $t_2$. A TPR-tree index is used to organize the moving objects.

The main drawback of the aforementioned method is that the TPR-tree is searched several times in order to determine the next object that influences the current result. This implies additional overhead in CPU and I/O time, which is more severe as the number of requested NNs increases. In [27] the same authors present a method which is applicable for static datasets, in order to overcome the problems of repetitive NN queries. By assuming that the dataset is indexed by an R-tree structure, a single query is performed and therefore each participating tree node is accessed only once. Performance results demonstrate that NN queries are answered much more efficiently concerning query response time. However, the proposed techniques can only be applied for static datasets.

In [18] we have proposed a nearest-neighbor search algorithm in the case of moving data objects and moving query objects. Experimental results performed have been demonstrated that significant improvement is obtained in comparison to the method proposed in [26].

In [13, 31] the authors study the problem of continuous $k$-NN processing in spatiotemporal databases. They propose scalable methods for incrementally updating query results as the objects move in space.

In many cases, the user may request for the $(k+1)$-th NN of the query object, after the execution of the $k$-NN query. In such a case, an obvious approach is to perform a new $(k+1)$-NN query. However, there are several optimizations that can be applied in order to reduce the processing costs. In spatial databases several methods have been proposed for incremental computation. Hjaltason et al. [5] proposed an algorithm, which is capable of ranking the objects with respect to their distance from a query object. This algorithm is implemented on a PMR quadtree and uses a priority queue to keep track of the blocks and the objects not visited yet. Henrich [4] proposed a similar method, which is an incremental NN algorithm and can be applied on a LSD-tree. Its main difference from the previous one is that it uses two priority queues instead of one. These two queues are used for the objects and the nodes of the data structure respectively. Later, Hjaltason et al. [6] extended their initial idea and showed that their algorithm is more general and applicable not only to the PMR-tree but to the R-tree as well.

To the best of the authors' knowledge, incremental algorithms have not yet been proposed in the context of mobile objects. Therefore, in the sequel we present our ideas towards this goal. We assume that data objects are indexed by means of a TPR-tree access method [20], which is a variation of the R*-tree [2] adapted for moving objects. Moreover, we study the problem of query result consistency when insertions, deletions and updates are taking place. For deletions and updates the incremental computation of the next NN is of great importance. Query result consistency has been studied in [8, 16] and [3] in the case of reverse NN queries and range queries respectively.

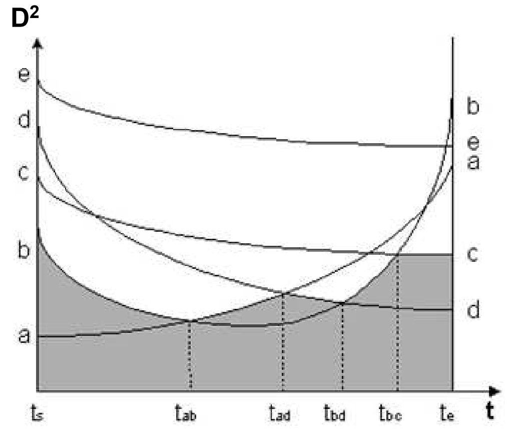
## 3 Incremental NN Algorithms

A moving nearest-neighbor query on a set of moving objects can be visualized by plotting the squared Euclidean distance ($D^2$) between each moving object and the query object. For example, Figure 1 depicts two nearest-neighbor queries for $k$=2 and $k$=3 (left and right respectively). An intersection between two object plots denotes a possible change in the result. The integral of the distance to the $k$-th nearest-neighbor in every time $t_x$ determines the area of relevance which is shown shaded in Figure 1. Evidently, this area is determined by the $k$-th nearest-neighbors of the query object. For every change in the $k$-th nearest-neighbor or a change in the order of two objects participating in the result, there is an associated split point. According to the way split points are defined, there are two types:

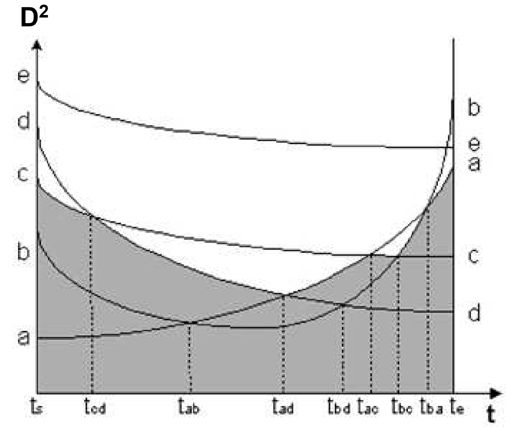
- **internal split points**, which are defined by an intersection of two objects $o1$ and $o2$ both participating in the result, and
- **external split points**, which are defined by an intersection between the current $k$-th nearest-neighbor and another object which is not participating in the result.

As an example, split points at $t_{ab}$ and $t_{bd}$ in Figure 1(a) are internal, whereas split-points at $t_{ad}$ and $t_{bc}$ are external. Note that in internal split points only the object order in the result is altered, whereas in external split points a new object is inserted in the result and takes the place of the current $k$-th nearest-neighbor (the order remains unchanged).

The result of a nearest-neighbor query is represented by a split-list. The split-list contains the time points at



(a) $k = 2$



(b) $k = 3$

Figure 1: Visualization of a $k$-NN query result.

which there is a change in the result, and the object IDs for each time subinterval in increasing distance order from the query object. If we are interested in both internal and external split points then both types are present in the split-list. Otherwise, only external split-points are represented. Therefore the split-list contains all the necessary information to describe the result of a $k$-NN query. The split-lists for the results of Figure 1 are depicted in Figure 2. Note that both internal and external split-points are represented. Above each split-point the pair of intersecting objects is presented. The nearest-neighbors for each time subinterval are given in increasing distance order with respect to the query object.

The challenge is given this information to determine the $k$+1 nearest-neighbor, by avoiding the query re-execution from the beginning. More specifically, we aim at:

- reducing the number of disk accesses in comparison to a newly executed ($k$+1)-NN query and
- reducing the required CPU time in comparison to

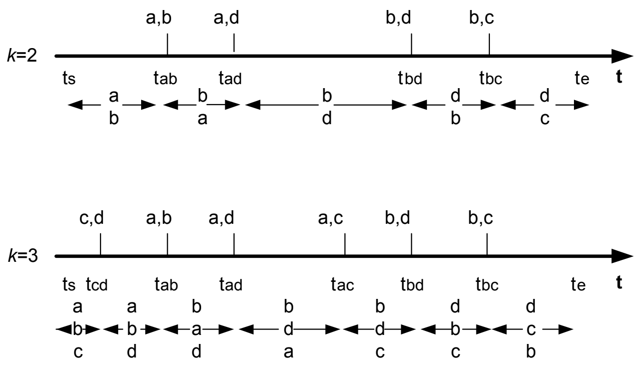


Figure 2: Split-list examples (top $k$=2, bottom $k = 3$).

a newly executed ($k$+1)-NN query.

The target is to exploit the split-list in order to define a starting point that will help in determining the ($k$+1)-th nearest-neighbor. We assume for the time being that there is at least one external split-point present in the split-list of the $k$-NN result. The case where all split-points are internal will be discussed later. External split-points provide valuable information for the determination of the ($k$+1)-th nearest-neighbor, since they correspond to time instances where there is a change in the result set. For example, by observing Figure 1(a) it is evident that at time $t_{ad}$+$dt$ (where $dt$ is a sufficiently small time interval), the ($k$+1)-th neighbor is object $a$. Similarly, at time point $t_{bc}$+$dt$ the ($k$+1)-th neighbor is object $b$. Therefore, for the time instances that correspond to external split-points, the ($k$+1)-th neighbor can be determined directly from the split-list. However, this does not solve the problem for the whole interval $[t_s, t_e]$, since we can not be certain about what is happening between external split-points. For example, the intersection of objects $a$ and $c$ (Figure 1(a)) has not been recorded in the split-list.

The above discussion suggests that we must determine a way to discover new split-points that denote a change in the result of the ($k$+1)-NN query. Let $t_x$ be an external split-point in the split-list of the $k$-NN query result, which denotes an intersection between objects $o_1$ and $o_2$. Without loss of generality, we assume that at time $t_x$+$dt$ the $k$-th NN is $o_2$, whereas at time $t_x$-$dt$ the $k$-th NN is $o_1$. The time interval $[t_s, t_e]$ is therefore partitioned into two subintervals $[t_s, t_x)$ and $(t_x, t_e]$. Note that at $t_x$ the ($k$+1)-th NN is already known (either $o_1$ or $o_2$). The method proceeds as follows:

1. New external split-points are determined for the subinterval $[t_s, t_x)$.

2. New external split-points are determined for the subinterval $(t_x, t_e]$.

3. A new split-list is generated by combining the split-points of the $k$-NN result with the new set of split-points

In the sequel we describe the process of generating new split-points and combining them with the split-points that are present in the split-list of the $k$-NN result.

### 3.1 Generating new split-points

There are two alternatives for the generation of new split-points. The first alternative determines split-points in increasing order with respect to time, by continuously querying the TPR-tree. The second alternative, searches the TPR-tree only once, but it is possible to fetch objects that subsequently will be discarded. We examine each alternative in detail, by using Figure 1(a) for illustration purposes, assuming that the external split-point at $t_{ad}$ is the starting point.

At time $t_{ad}$ objects $a$ and $d$ intersect. According to the first alternative, new splits points are determined one-by-one for the two subintervals $[t_s, t_{ad})$ and $(t_{ad}, t_e]$. Using object $a$ the TPR-tree is searched for the next possible intersection between $a$ and other objects. If no intersection is found, then we deduce that object $a$ is the ($k$+1)-th nearest-neighbor during $(t_{ad}, t_e]$. In our case, an intersection between $a$ and $c$ is determined at time $t_{ac}$. A new split-point is generated, and the same method is applied for object $c$. At time $t_{cb}$ objects $c$ and $b$ intersect and therefore another split-point is generated. Finally, object $b$ does not intersect any other object before $t_e$ and thus no more split-points can be generated for the subinterval $(t_2, t_e]$. The method for the subinterval $[t_s, t_{ad})$ is the same. There is only one new split-point for this subinterval at time $t_{cd}$, denoting the intersection between objects $c$ and $d$. The algorithm for the generation of a new split-point is similar to the algorithm reported in [27], which uses a simplified version of the *mindist* distance between the query object and the MBR of an internal node. After the algorithm termination, a new split-list is generated for the ($k$+1)-th nearest-neighbor. The outline of algorithm INCNN-REP is presented in Figure 3.

It is evident that for each new split-point determined, a query must be issued to the TPR-tree (lines 3 and 10 of Figure 3). Although the repetitive queries are likely to retrieve similar disk pages, the total number of TPR-tree node accesses is high. Therefore, we present another alternative concerning the determination of new split-points, which issues only two queries (one for each subinterval) and fetches the relevant objects. However, some of the retrieved objects may be discarded because they may not contribute to the ($k$+1)-NN query result. When querying the TPR-tree, the algorithm does not stop when the first intersection is determined, but continues to search the tree for more intersections. After searching the TPR-tree for both subintervals, the algorithm returns a set of ob-

```
Algorithm INCNN-REP
Input: the $k$-NN split-list
Output: the ($k$+1)-NN split-list
1.    let ($o_1$,$o_2$,$t_x$) be any external split-point
2.    do /* subinterval $(t_x, t_e]$ */
3.        search TPR-tree for first intersection of $o_1$ in $(t_x, t_e]$
4.        set $o_x$ := intersecting object
5.        create new split point ($o_1$,$o_x$,$t_x$)
6.        update ($k$+1)-NN split-list
7.        set $o_1 := o_x$
8.    while (intersection found)
9.    do /* subinterval $[t_s, t_x)$ */
10.       search TPR-tree for first intersection of $o_2$ in $[t_s, t_x)$
11.       set $o_x$ := intersecting object
12.       create new split point ($o_2$,$o_x$,$t_x$)
13.       update ($k$+1)-NN split-list
14.       set $o_1 := o_x$
15.   while (intersection found)
```

Figure 3: The INCNN-REP algorithm

jects which comprise the possible candidates for the ($k$+1)-th nearest-neighbor. In our example, these objects are $b$, $c$ and $e$. Therefore, objects $b$, $c$, $d$ and $e$ are participating for the ($k$+1)-th nearest-neighbor. Using the set of candidates, a new split-list is generated for the ($k$+1)-th nearest-neighbor. Evidently, this method computes the result in two phases: a) the candidate determination phase and b) the new split-list generation phase. The outline of algorithm INCNN-2P is illustrated in Figures 4, 5 and 6.

```
Algorithm INCNN-2P
Input: the $k$-NN split-list
Output: the ($k$+1)-NN split-list
1.    call DetermineCandidates
2.    set $C$ := set of candidate objects
3.    call GenerateSplitList($C$)
```

Figure 4: The INCNN-2P algorithm

```
Algorithm DetermineCandidates
Input: the $k$-NN split-list
Output: the set of candidates for the ($k$+1)-th NN
1.    let ($o_1$,$o_2$,$t_x$) be any external split-point
2.    search the TPR-tree for intersections of $o_2$ in $(t_x, t_e]$
3.    search the TPR-tree for intersections of $o_1$ in $[t_s, t_x)$
4.    return the candidate objects
```

Figure 5: The DetermineCandidates algorithm

If we are interested only in external split-points, the new split-list determined by the aforementioned algorithms corresponds to the split-list for the whole ($k$+1)-NN query. In the case where internal split-points are also considered, a combination of two split-lists must be performed. The combination of the new split-list with

```
Algorithm GenerateSplitList
Input: the set $C$ of candidates
Output: a split-list for the ($k$+1)-th NN
1.    set $o_s$ := the NN at $t_s$
2.    while (TRUE) do
3.        find first intersection of $o_s$ with $o_x \in C$ at $t_x$
4.        if ($t_x > t_e$) break
5.        record $t_x$ as a split-point
6.        set $o_s := o_x$
7.    end
```

Figure 6: The GenerateSplitList algorithm

the split-list of the $k$-NN query result determines the final split-list of the ($k$+1)-NN query. This is performed by a merging process with respect to the time instance of each split-point. Since the two lists are sorted with respect to time, the merging is performed easily.

## 3.2 Absence of external split-points

In the previous section we assumed that there is at least one external split-point in the split-list of the $k$-NN query result. However, we must cover the case where all split-points are internal. An example of this situation is depicted in Figure 7.

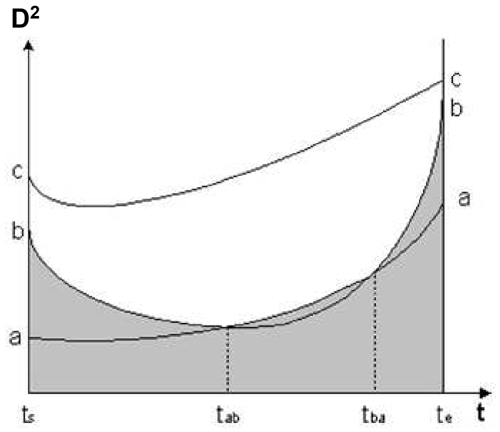


Figure 7: Query result with no external split-points.

In such a case, a starting point for the ($k$+1)-th NN investigation can not be determined directly from the split-list. This happens because either the split-list is empty (if only external split-points are recorded) or contains internal split-points only. In other words, the set of objects that correspond to the $k$ nearest-neighbors at $t_s$ are also the nearest-neighbors at $t_e$. Only the object order changes. We need at least one object which does not belong to the set of nearest-neighbors. This object is determined by computing the ($k$+1)-NN at any time instance in $[t_s, t_e]$. Then, this object is used as a candidate and either INCNN-REP or INCNN-2P can be used to provide the complete answer.

## 4 Insertions, Deletions and Updates

In dynamically changing environments, there are operations that alter the status of the database. A new object may be inserted, an already existing object may be deleted or updated. For example, in an aircraft monitoring system the appearance of a new aircraft in the vicinity of the radar corresponds to an object insertion. Similarly, switching off a mobile phone corresponds to an object deletion. Finally, a change of direction or speed of a car corresponds to an object update. In a set of moving objects, the queries that have been specified and correspond to future time intervals (future queries) must be kept up to date. This implies that the result of a future query must be valid, according to the changes applied to the database. Therefore, objects participating in insertions, deletions and updates must be examined in order to determine if they affect any of the specified queries.

In [16] the authors present efficient methods for updating the query results in the case of range queries in a database of moving objects, whereas in [3] the authors study the consistency problem if reverse nearest-neighbor queries. In this section we present methods in the case of $k$-NN queries. Let $w$ be a moving object that is either inserted, deleted or updated. There are two steps in the process of query update:

1. the determination of the queries that may be affected by $w$, and

2. the update of the query result

The first step can be handled by applying indexing mechanisms on the queries, as it is suggested in [16]. Using this method, the role of queries and data is exchanged. Every time an object is modified, the queries that may be possibly affected are determined by consulting the index on the queries. The second step involves the refinement of the set of queries determined, and the update of the query result if this is necessary. In the sequel we focus on the efficient update of the query results. The query indexing problem can be solved by creating a TPR-tree for the active queries. If the number of these queries is small, the index can be managed in main memory.

Let $w$ be a new moving object inserted in the database, and $q$ be a $k$-NN query defined by a moving point at $(q_x, q_y)$, a time interval $[t_s, t_e]$, an integer $k$ and the corresponding result stored in the split-list. Also, let $S$ be the set of objects that participate in the result of $q$. Note that set $S$ must contain at least $k$ objects. We distinguish among the following cases, which describe the relation of $w$ to the current answer:

**case 1:** $w$ does not intersect any of the objects in $\mathcal{S}$ between $t_s$ and $t_e$, and it is "above" the area of relevance. In this case, $w$ is ignored, since it can not contribute to the NNs. The number of split points remains the same.

**case 2:** $w$ does not intersect any of the objects in $\mathcal{S}$ between $t_s$ and $t_e$, and it is completely "inside" the area of relevance. In this case $w$ must be taken into account, since it affects the answer from $t_s$ to $t_e$ (Proposition 4). The number of split points may be reduced.

**case 3:** $w$ intersects at least one object $v \in \mathcal{S}$ at time $t_s \leq t_x \leq t_e$, but at time $t_x$ $v$ is not contained in the set of NNs. In this case, again $w$ is ignored, since this intersection can not be considered as a split point because the answer is not affected. Therefore, no new split points are generated.

**case 4:** $w$ intersects at least one object $v \in \mathcal{S}$ at time $t_s \leq t_x \leq t_e$, and object $v$ is contained in the set of NNs at time $t_x$. In this case $w$ must be considered because at least one new split point is generated. However, some of the previous split points may be discarded.

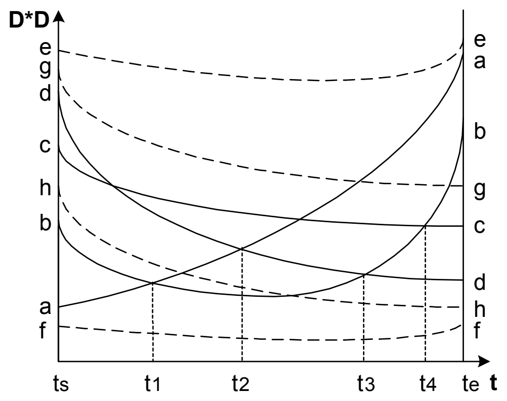


Figure 8: The four different cases that show the relation of a new object to the current NNs

The aforementioned cases are depicted in Figure 8. Object $e$ corresponds to *case 1*, since it is above the area of interest. Object $f$ corresponds to *case 2*, because it is completely covered by the relevant area. Object $g$ although intersects some objects, the time of these intersections are irrelevant to the answer, and therefore the situation corresponds to *case 3*. Finally, object $h$ intersects a number of objects at time points that are critical to the answer and therefore corresponds to *case 4*.

Let us examine now the case where a moving object is deleted from the database. Let $w$ be the deleted object and $q$ a $k$-NN query which contains $w$ in its result set. Updating the query result involves the following operations:

1. the removal of $w$ from the split-list, and

2. the update of the split-list

During the first operation, the split-list is scanned, and object $w$ is removed from the corresponding time intervals between $t_s$ and $t_e$ where it participates to the query result. This removal yields in a split-list that, for some time intervals, represents the result of a ($k$-1)-NN query. In order to determine the $k$-th NN for these intervals, one of the incremental algorithms (INCNN-REP, INCNN-2P) discussed in the previous section can be applied.

The last operation we examine involves object update. In real-life applications is very unusual for a moving object to maintain the same mobility characteristics for long time periods. Concerning moving objects modeled as moving points, there are two operations that may request for an update: change of speed, and change of direction. When a moving object reports one or both of these changes the result of the queries that are affected must be updated to reflect the changes. Let $w$ be an object that changes its mobility characteristics, and $q$ a $k$-NN query which contains $w$ in its result set. The object update is equivalent to the deletion of $w$ followed by the insertion of a new object $w'$. Therefore, the query update problem can be solved by the following steps:

1. object $w$ is deleted and the split-list is updated,
2. the new $k$-th NN of the query is computed incrementally, and
3. object $w'$ is inserted and the split-list is updated

The above discussion shows that the incremental computation described in the previous section is an important building block for updating the result of a query. In the upcoming section we study the efficiency of incremental algorithms through an experimental performance evaluation.

## 5 Performance Evaluation

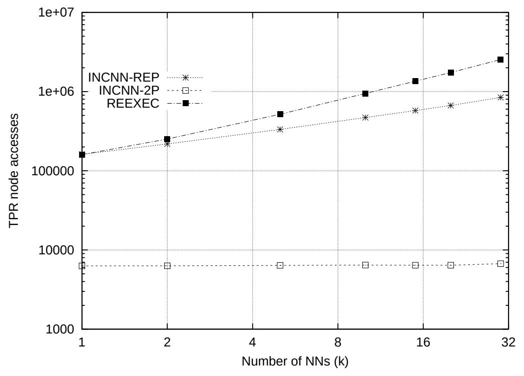
The purpose of this section is to study and evaluate the efficiency of the incremental NN processing algorithms INCNN-REP and INCNN-2P, and compare their performance with algorithm REEXEC (described in [27]), which is based on query re-execution. The database is composed of 50,000 to 1,000,000 moving objects whose reference positions are uniformly distributed in an address space of 1,000 x 1,000 km. Each moving object is assigned a speed randomly selected between 0 and 30 m/sec for each dimension. The direction of movement is randomly selected. All experiments have been conducted using a Pentium IV at 2.4 GHz processor system. The varying parameters used for the experimental evaluation are as follows:

- the number $k$ of requested nearest-neighbors,
- the query duration $[t_s, t_e]$ (query travel time),
- the size of the available memory for each query, and
- the number of database objects.

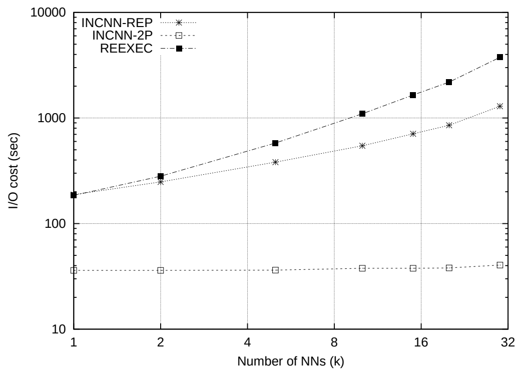
Using the aforementioned dataset, a TPR-tree is constructed for indexing purposes. The TPR-tree page size is set to 2KBytes for all the experiments conducted. The efficiency of the algorithms is expressed by the number of TPR-tree node accesses, the CPU time required to process each query, and the number of disk accesses. In order to measure the latter, an LRU buffer is used, and the number of page faults is computed for the query processing duration. A time overhead of 8ms is assigned for every page fault occurred. In the experimental evaluation that follows, when $k$=5 it means that the incremental algorithms search for the 6-th nearest-neighbor of the query, whereas algorithm REEXEC computes the 6-NN query.

Figure 9 illustrates the performance of the algorithms assuming that the varying parameter is the number $k$ of requested nearest-neighbors, which varies between 1 and 30. For each query we assume an available buffer which is equal to 5% of the total database size. The database contains 1,000,000 moving objects. Since the total number of pages of the TPR-tree is 77,926 the buffer has 3,897 available pages. The query travel time is set to 524 seconds (8.7 minutes). The first observation is that algorithm REEXEC issues many repetitive queries to the TPR-tree and this highly affects the number of accessed TPR-tree nodes. As Figure 9(a) illustrates, the number of node accesses for the REEXEC algorithm is by factors larger than for the INCNN-2P algorithm. Algorithm INCNN-REP although accesses less nodes than REEXEC, its performance is by factors worse than that of INCNN-2P. The existence of a buffer is very important since it reduces the number of page faults occurred. This is illustrated in Figure 9(b) where the I/O cost is presented for all algorithms. The number of disk accesses is significantly less than the number of node accesses and INCNN-2P outperforms the other methods. It is evident that for all algorithms the total cost is dominated by the I/O cost (Figures 9(b), (c) and (d)).

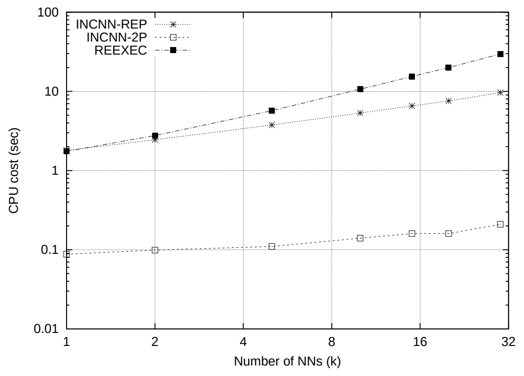
Figure 10 illustrates the impact of the buffer size which varies from 1% to 20% of the total database size. The number of database objects is set to 500,000, and the number of the requested nearest-neighbors $k$ is set to 10. The query travel time is again set to 524 seconds. We observe that the performance of all methods is highly affected by the buffer size. INCNN-2P however is less affected, and shows a fairly stable performance. For small buffer sizes, INCNN-REP and REEXEC issue a large number of disk accesses which degrades their performance. When the number of buffer pages is more than 2000 we observe a clear improvement of the performance. In the extreme case where all database objects are maintained in main memory (this is not shown in the figure), the total cost is dominated by the CPU time. Therefore, since INCNN-2P shows better performance with respect to the CPU time (Figure 9(c)) it can be used for main-memory databases as



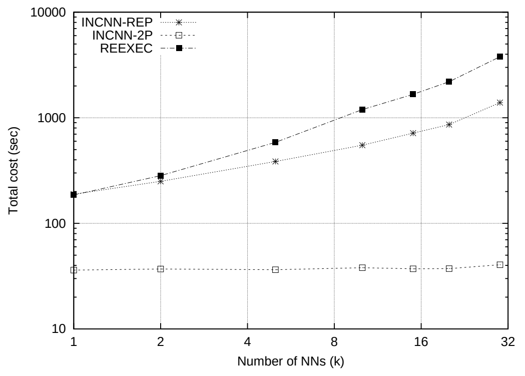
(a) Tree node accesses



(b) I/O cost (sec)

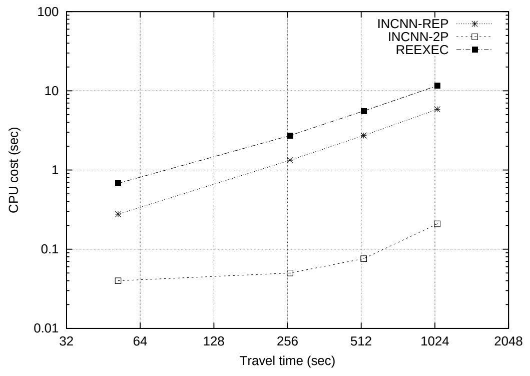


(c) CPU cost (sec)

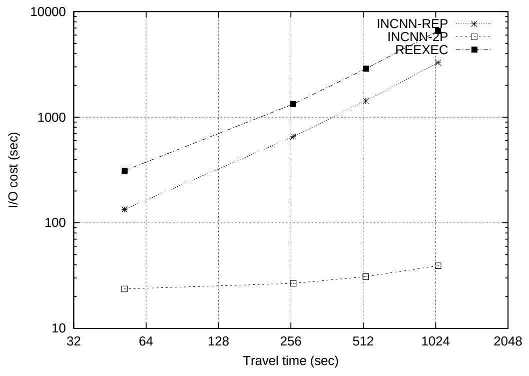


(d) Total cost (sec)

Figure 9: Results for different values of the $k$.



(a) CPU cost



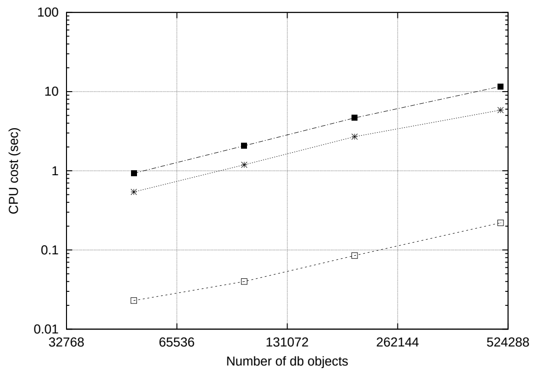
(b) I/O cost (sec)

Figure 11: Results for different query travel time values.

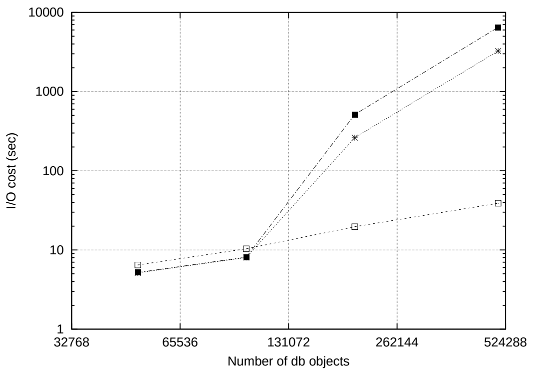
well.

Figure 11 presents the impact of the query travel time which varies from 52 seconds to 1048 seconds (17.4 minutes). The database contains 500,000 moving objects. The number $k$ is set to 10 and the size of the buffer is set to 5% of the database size (1952 pages). Again we observe that algorithm INCNN-2P performs better than the other methods for different query duration intervals ($[t_s, t_e]$).

Finally, Figure 12 presents the behavior of the methods for several database sizes. The database size varies between 50,000 and 500,000 objects. The size of the buffer is fixed at 2,000 pages, whereas the number of $k$ is set to 10. The query travel time is set to 1048 seconds. We observe that generally INCNN-2P shows the best performance. However, for small databases sizes



(a) CPU cost



(b) I/O cost (sec)

Figure 12: Results for different database sizes.

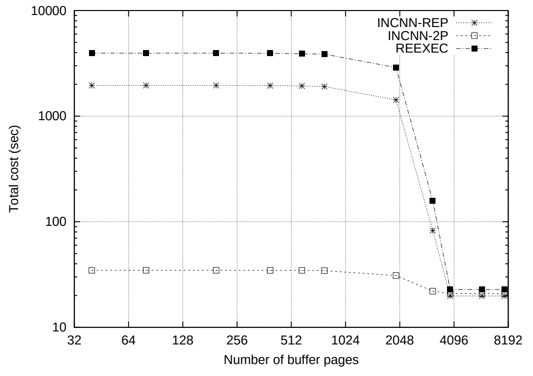


Figure 10: Results for different buffer sizes.

the I/O performance of INCNN-2P may be marginally worse than REEXEC. This happens because there are cases where the number of disk accesses that INCNN-2P issues is slightly larger than REEXEC. INCNN-2P follows a filter-refinement approach, and some of the retrieved objects may not participate to the final answer.

By observing the previous results we can state that, generally, INCNN-2P algorithm outperforms INCNN-REP and REEXEC and therefore, can be used for the incremental computation of the $(k+1)$-th nearest-neighbor. As described in a previous section, the computation of the $(k+1)$-th closest object (with respect to a query object) can be used in order to update the query results after a deletion or an update. On the other hand, if an insertion of a new object affects the result of an already computed $k$-NN query, the new result can be computed by using the split-list of the $k$-NN query and the new object.

## 6 Concluding Remarks

An interesting research direction in spatiotemporal databases is the design of efficient query processing techniques for future queries, i.e. queries that refer to a future time interval $[t_s, t_e]$. In this work, we focused on $k$-NN query processing for datasets composed of moving objects. Particularly, we have studied the problem of incremental computation of the $(k+1)$-th nearest-neighbor given the result of the already executed $k$-NN query. The proposed algorithm INCNN-2P outperforms INCNN-REP which is based on repetitive queries and REEXEC which issues a new $(k+1)$-NN query from the beginning without considering the result of the previously executed $k$-NN query.

Another important issue in moving-object databases is the ability to keep the query results consistent after insertions, deletions and updates. We have shown that the incremental computation method can be used if a deletion or an update affects the query results. For object insertions only the available split-list and the new object are needed to compute the new result. Future research in the area may include:

- The use of a priority queue in a spatiotemporal context. A problem that may arise, is that objects and tree nodes may become invalid after insertions, deletions and updates.
- The consideration of all available external split-points instead of just one.
- The study of cost estimations for incremental $k$-NN processing.

## References

[1] P.K. Agarwal, L. Arge, J. Erickson: "Indexing Moving Points", *ACM PODS*, pp.175-186, 2000.

[2] N. Beckmann, H.P. Kriegel and B. Seeger: "The R*-tree: an Efficient and Robust Method for Points and Rectangles", *ACM SIGMOD*, pp.322-331, 1990.

[3] R. Benetis, C.S. Jensen, G. Karciauskas, S. Saltenis: "Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects", *IDEAS*, pp.44-53, 2002.

[4] A. Henrich: "A Distance-Scan Algorithm for Spatial Access Structures", *ACM-GIS*, pp. 136143, Gaithersburg, MD, December 1994.

[5] G. Hjaltason, H. Samet. "Ranking in Spatial Databases", *SSD*, pp. 83-95, 1995.

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

[7] Y. Ishikawa, H. Kitagawa, T. Kawashima: "Continual Neighborhood Tracking for Moving Objects Using Adaptive Distances",*IDEAS*, pp.54-63, 2002.

[8] D.V. Kalashnikov, S. Prabhakar, S.E. Hambrusch, W.G. Aref: "Efficient Evaluation of Continuous Range Queries on Moving Objects, *DEXA*, pp.731-740, 2002.

[9] G. Kollios, D. Gounopoulos and V.J. Tsotras: "Nearest Neighbor Queries in a Mobile Environment", *Proceedings of the International Workshop on Spatiotemporal Database Management*, pp.119-134, 1999.

[10] G. Kollios, D. Gunopoulos, V. Tsotras: "On Indexing Mobile Objects", *ACM PODS*, pp.261-272, 1999.

[11] I. Lazaridis, K. Porkaew, S. Mehrotra: "Dynamic Queries over Mobile Objects", *EDBT*, pp.269-286, 2002.

[12] D. Lomet, B. Salsberg: "Access Methods for Multiversion Data", *ACM SIGMOD*, pp.315-324, 1989.

[13] M.F. Mokbel, X. Xiong and W.G. Aref: "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-Temporal Databases", *ACM SIGMOD*, 2004.

[14] M.A. Nascimento, J.R.O. Silva: "Towards Historical R-trees", *ACM SAC*, 1998.

[15] D. Pfoser, C.S. Jensen, Y. Theodoridis: "Novel Approaches to the Indexing of Moving Object Trajectories", *VLDB*, pp.395-406, 2000.

[16] S. Prabhakar, Y. Xia, D.V. Kalashnikov, W.G. Aref, S.E. Hambrusch: "Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects", *IEEE TOCS*, 51(10), pp.1124-1140, 2002.

[17] C.M. Procopiuc, P.K. Agarwal, S. Har-Peled: "STAR-Tree: An Efficient Self-Adjusting Index for Moving Objects", *ALENEX*, pp.178-193, 2002.

[18] K. Raptopoulou, A.N. Papadopoulos and Y. Manolopoulos: "Fast Nearest-Neighbor Search in Moving-Object Databases", *Geoiformatica*, 7(2), pp.113-137, 2003.

[19] N. Roussopoulos, S. Kelley, F. Vincent. "Nearest Neighbor Queries". *ACM SIGMOD*, pp. 71-79, 1995.

[20] S. Saltenis, C.S. Jensen, S. Leutenegger, M. Lopez: "Indexing the Positions of Continuously Moving Objects", *ACM SIGMOD*, pp.331-342, 2000.

[21] T. Seidl and H.Kriegel: "Optimal Multi-Step k-Nearest-Neighbor Search", *ACM SIGMOD*, Seattle, USA, 1998.

[22] A.P. Sistla, O. Wolfson, S. Chamberlain, S. Dao: "Modeling and Querying Moving Objects", *IEEE ICDE*, pp.422-432, 1997.

[23] Z. Song, N. Roussopoulos: "K-NN Search for Moving Query Point", *SSTD*, pp.79-96, 2001.

[24] Y. Tao, D. Papadias: "Efficient Historical R-trees", *SSDBM*, 2001.

[25] Y. Tao and D. Papadias: "MV3R-tree - a Spatio-Temporal Access Method for Timestamp and Interval Queries", *VLDB*, pp.431- 440, 2001.

[26] Y. Tao, D. Papadias, Q. Shen: "Continuous Nearest Neighbor Search", *VLDB*, pp.287-298, 2002.

[27] Y. Tao, D. Papadias: "Time-Parameterized Queries in Spatio-Temporal Databases" *ACM SIGMOD*, pp. 334-345, 2002.

[28] Y. Theodoridis, M. Vazirgiannis, T. Sellis: "Spatio-temporal Indexing for Large Multimedia Applications", *Proceedings of 3rd IEEE Conference on Multimedia Computing and Systems*, pp.441-448, 1996.

[29] Y. Theodoridis, T. Sellis, A.N. Papadopoulos, Y. Manolopoulos: "Specifications for Efficient Indexing in Spatio-Temporal Databases", *SSDBM*, pp.123-132, 1998.

[30] O. Wolfson, B. Xu, S. Chamberlain, L. Jiang: "Moving Objects Databases: Issues and Solutions", *SSDBM*, pp.111-122, 1998.

[31] X. Xiong, M.F. Mikbel and W.G. Aref: "SEA-CNN: Scalable Processing of Continuous k-Nearest Neighbor Queries in Spatio-Temporal Databases", *IEEE ICDE*, 2005.

[32] X. Xu, J. Han, W. Lu: "RT-tree: an Improved R-tree Index Structure for Spatio-Temporal Databases", *SDH*, pp.1040-1049, 1990.

[33] B. Zheng, D. Lee: "Semantic Caching in Location-Dependent Query Processing", *SSTD*, pp.97-116, 2001.