A Generic Framework for Continuous Motion Pattern Query Evaluation

Petko Bakalov
University of California
Riverside, CA 92521
email@cs.ucr.edu

Vassilis J. Tsotras
University of California
Riverside, CA 92521
tsotras@cs.ucr.edu

Abstract

We introduce a novel query type defined over streaming moving object data, namely, the Continuous Motion Pattern *(CMP) Queries. A motion pattern is defined as a sequence of distinct spatial predicates, each attached to a temporal constraint. The spatial predicates can be of various types (range, nearest neighbor, etc.) The temporal constraints are relative to the current time instant and are used to specify the order of the spatial predicates on the time axis. A CMP query is continuously reevaluated over streaming spatiotemporal data, producing the moving objects which satisfy the query's motion pattern. We first introduce an easily maintainable indexing scheme for spatiotemporal streams that facilitates the evaluation of the spatial predicates over their temporal constraints. Using this scheme we propose a generic framework for efficiently answering a wide range of CMP queries. The effectiveness of our algorithms in reducing the query computation cost and I/O operations is revealed through a thorough experimental evaluation.*

1 Introduction

The widespread use of location detection devices (RFID, GPS etc.) has enabled the creation of complex tracking and situational awareness systems which continuously monitor the position of moving objects of interest and can thus provide complex services to their end users. Example applications include security monitoring, vehicle tracking, traffic management, etc. In a typical surveillance system architecture the set of monitored objects continuously report their position to the data collection device using data packets containing their identifier, current location and timestamp. These data packets are combined into a single spatiotemporal stream that is forwarded to a centralized server for query processing. The system users register their queries to the server and continuously receive the result based on the changing state of the data.

The described streaming architecture differs from the static spatiotemporal archives like [11, 5] in two major aspects. First, the continuous nature of the streaming data requires real time processing using relatively simple data structures for indexing. Second, in the streaming environment the query evaluation is a continuous process. Unlike the typical snapshot queries over the static archives that are evaluated only once, continuous queries require a continuous reevaluation since results become obsolete and invalid as information about the monitored objects changes.

Recent research in continuous spatial query processing has been focused mainly on *single* predicate queries, where the predicate is a Range [3, 12] or a Nearest Neighbor [6]. In this paper we focus on the continuous evaluation of "motion pattern" (i.e. CMP) queries. Here a single query is expressed with *multiple* predicates, correlated over time.

We argue that since the data produced by the monitored moving objects is "trajectorial" in nature, users of these surveillance and tracking systems should also be able to query the "behavior" of the moving objects over time. Consider for instance a criminal offenders monitoring application (like tracNET24 or ExacuTrack) which tracks the movement of law offenders (through special wearable GPS devices) inside a given area and alerts the correctional officers for any suspicious or illegal behavior. The suspicious behavior can be defined as a CMP query like: "Continuously report objects that did pass through all five bank offices in the area and have been in areas not covered with surveillance cameras for more than 20 minutes in the last half an hour". In this example, the object behavior is captured by the sequence of spatial (range) predicates ordered by time, where the temporal predicates are relative to the ever increasing current time. This type of query cannot be answered with processing methods focused only on the current state of the stream; rather, we need to maintain and query appropriate past states (the history) of the spatiotemporal stream.

Motion pattern queries [4] have been well studied for static spatiotemporal archives (i.e., when all data is known in advance and the query is evaluated once). To the best of our knowledge there is no work for this problem in the streaming environment where the query result has to be constantly reevaluated. A trivial solution involves repetitive execution of the static algorithms described in [4], i.e. every time when the result becomes obsolete it has to be recomputed. This would be very expensive and inefficient since in a typical streaming environment location updates are very frequent.

The predominant approach for effective evaluation of continuous queries is by incremental processing [20, 19]. This strategy implies that the query processor utilizes as much as possible the current intermediate results and data structures in the successive iterations of the evaluation algorithm. The query result can be kept persistent during consecutive iterations by applying positive and negative updates on it.

In this paper we first present a novel indexing scheme for answering CMP queries on spatiotemporal streams. Using this indexing structure we propose an efficient framework for incremental evaluation of a wide range of CMP queries. The effectiveness of our algorithms is revealed trough an extensive experimental evaluation.

2 Related Work

With the exception of [4] which presents a set of algorithms for pattern query evaluation in a static environment, all previous related work on spatiotemporal pattern queries deals only with modeling and language issues. [1] proposes using spatiotemporal patterns as a systematic and scalable query mechanism to characterize complex object behaviors in space and time; nevertheless no query evaluation strategy is proposed. Similarly [13] presents a powerful query language able to describe complex pattern queries using a combination of logical functions and quantifiers. This language however is of declarative nature and cannot be used the query optimization. Pattern queries have recently attracted interest for a numerical (i.e no trajectory) data in a relational DBMS.

Related to our research is also the work done on continuously querying spatiotemporal streams. Various indexing structures have been proposed in the past as well as multiple algorithms utilizing these structures to answer mainly NN and range queries. They all fall in three basic categories: (i) methods using grids, (ii) methods utilizing tree structures, and, (iii) methods using "safe" regions.

Many of the proposed indexing structures [3, 14] use a simple grid to index the location of the objects inside the spatiotemporal stream. Every cell inside the grid structure keeps a list with the objects whose location is within the boundaries of this cell. Multiple algorithms have been proposed to solve single range [3, 12]. and NN predicates [6, 14]. using this simple grid structure in distributed and centralized environments. Given its straightforward maintenance, a grid structure can handle very effectively issues like frequent updates, high arrival rates, and the continuous nature of the data, which are critical in a streaming environment. Unfortunately, because of its simplicity this structure cannot capture the temporal characteristics of the data and can be used only for a queries focused on the current state of the stream. Thus it cannot be used directly to evaluate continuous motion pattern queries.

Methods using tree structures are either B+-tree based [9] or R-tree based [7, 8, 18, 16, 17]. The main objective here is to improve the index update performance. In [7] the amortized update cost is reduced by avoiding updates for objects that do not move outside their MBRs. [8] generalized this technique through a bottom - up update strategy. [9] propose instead the linearization of the moving object location representation by using space filling curves. Then B+ trees can be used, which have better update characteristics than R-trees.

The last group [15, 10] of query evaluation methods for spatiotemporal streams avoids the expensive maintenance of index structures over the streaming data. Instead, they introduce the notion of safe regions, created either around the moving object [15] or around the query [10]. If the object does not leave its safe region no further query processing is required. Similarly in [10] objects are considered only if they fall inside the query region or its uncertainty regions.

3 Problem Definition

Consider a situational awareness system that continuously monitors the location of a set of moving objects. Location data arrive as a stream of tuples $S \equiv \langle u_1, u_2, \ldots, u_l, \ldots \rangle$ where $u_i \equiv \langle o, l, t \rangle$, o is the moving object identifier, t is the observation timestamp and l is the object location at time t (where $l \in \mathbb{R}^d, t \in \mathbb{N}, o \in \mathbb{N}$). A trajectory $TR(o_j, T)$ of object o_j inside the stream S for time predicate T is defined as a sequence of tuples $u_1, u_2, \ldots, u_n, \ldots$ where $u_i.o = o_j$ and $\forall i\ , u_i.t \in T$. The notation used in the rest of the paper is summarized in the following table:

Notation	Meaning
Q	continuous spatial predicate
T	relative time constraint
o_j	spatial object
CNP	set of continuous numerical predicates
CBP	set of continuous binary predicates

Our definition of continuous motion pattern queries is based on the notion of pattern queries defined in [2]. However instead of absolute time constraints we use *relative* ones; in a streaming environment absolute time constraints as defined in [2] do not apply since their result never changes. A relative time constraint uses the current time instance as a reference point. (e.g.. "between 40 and 50 minutes ago"). As time advances, the value of the current timestamp changes, forcing the relative time constraint slide along the temporal axis.

More formally a CMP query $\mathcal{Q}$ is expressed as a sequence of (arbitrary number) n spatiotemporal predicates:

$$\mathcal{Q} = \{(Q_1, \langle T_1, \Psi_1\rangle), \ldots, (Q_n, \langle T_n, \Psi_n\rangle), \Theta\}$$

where Q_i is a spatial predicate, T_i is a relative time constraint, Ψ_i is a logical quantifier ($\Psi_i \equiv \{\forall, \exists\}$) which indicates if the spatial predicate is applies for the whole duration of the time constraint T_i or just for one time instance. Θ is an operator that maps a real value $\mathbb{R}$ to a boolean $\mathbb{B} \equiv \{true, false\}$. Note that in the definition above the temporal constraints T_i are optional. If there is no temporal constraint provided for spatial predicate Q_i a temporal ordering is implied by the actual position of the predicate Q_i in the query sequence $\mathcal{Q}$. For example in the query:

$$\mathcal{Q} = \{(Q_1, (30, 20), \forall), (Q_2), (Q_3), (Q_4, (10, 5), \exists), \Theta\}$$

defines a pattern where the spatial predicate Q_1 is satisfied for every time instance in the time interval between 20 and 30 minutes ago, Q_4 is satisfied for at least one time instance in the time interval between 5 and 10 minutes ago and the predicates Q_2 and Q_3 are satisfied between predicates Q_1 and Q_4 in that order.

We allow a very general class of spatial predicates to participate in a CMP query. A spatial predicate Q_i is described through a spatial object so_i (where so_i represents point or region) and can be either *binary* or *numerical*. Let CBP (respectively CNP) denote the set of all binary (respectively, numerical) predicates.

Binary spatial predicate: A predicate $Q_i \in CBP$ maps a combination between a moving object o_j and the spatial object so_i defined in this predicate to a boolean value $\mathbb{B} \equiv \{true, false\}$. That is $Q_i \equiv o_j \times so_i \rightarrow \mathbb{B}$.

An example of a binary predicate is the predicate $Inside$. that checks whether the moving object o_j is inside region so_i Range predicates belong to CBP as well. In analogy:

Numerical spatial predicate: A predicate $Q_i \in CNP$ maps the combination of moving object o_j and spatial object so_i to a real value. That is $o_j \times so_i \rightarrow \mathbb{R}$.

A numerical spatial predicate example is the function $min\ distance$ which returns the minimal distance between the moving object o_j and spatial object so_i (which can be a point or a region). Clearly NN predicates belong to CNP.

For both binary and numerical predicates the mapping is done through a **score function** $f(Q_i, TR(o_j, T_i), \Psi_i)$. Given a single spatial predicate Q_i, its relative time constraint T_i, quantifier Ψ_i and a moving object o_j in a spatiotemporal stream S, f is a score function if $f(Q_i, TR(o_j, T_i), \Psi_i) = c$ where $c \in \mathbb{R}$ for a numerical spatial predicate or $c \in \mathbb{B}$ for a binary spatial predicate.

Given a pattern query $\mathcal{Q}$ and a stream S our goal is to find the moving objects $o_j \in S$ which satisfy the query $\mathcal{Q}$. If the query $\mathcal{Q}$ contains binary predicates CBP we have to find all objects o_j which satisfy all binary predicates (i.e. $\forall Q_i \in CBP \rightarrow f(Q_i, TR(o_j, T_i), \Psi_i) = true$). The binary predicates are aggregated using the operator $\cap$ (AND). If the query Q contains numerical predicates CNP, their scores on object o_j are aggregated using a summation to produce an object numerical score μ_j.

$$\mu_j = \sum_{Q_i \in CNP} f(Q_i, TR(o_j, T_i), \Psi_i)$$

To determine if an object o_j satisfies the CNP predicates of query Q its numerical score has to be mapped to a binary value. This is done by the Θ operator defined in the query $\mathcal{Q}$. An example of Θ operator can be the simple check function "≤ 4". In this example Θ will return $true$ for all objects o_j that have a sum of the numerical scoring functions less or equal than 4 (i.e., $\mu_j \leq 4$). It is also possible to use more sophisticated Θ operators like min or max. In the first case the operator will return $true$ only for the object which has the smallest numerical score μ_j. In analogy $\Theta \equiv max$ will return $true$ only for the object with the highest numerical score. To summarize, in order to check if an object o_j satisfies Q we compute $\lambda(Q, o_j)$ where:

$$\lambda(Q, o_j) \equiv \bigcap_{Q_i \in CBP} f(Q_i, TR(o_j, T_i), \Psi_i)$$

$$\cap \quad \Theta(\sum_{Q_i \in CNP} f(Q_i, TR(o_j, T_i), \Psi_i))$$

and $\lambda \equiv \{true, false\}$.

4 Index Structure

Given that incremental processing has been shown to be the most efficient approach for continuous query evaluation, we need an appropriate spatiotemporal indexing structure that can accommodate positive and negative updates. Such structure should answer efficiently questions of the type "given area A, provide all objects that are *not* in A at the previous time instant but appear in A at the current instant" (a positive update), or, "provide all objects that do not appear in A in the current time instant but were in A at the previous one" (a negative update).

Generating positive or negative updates every time an object changes its location would produce an intractable number of such updates. Instead we generate positive and

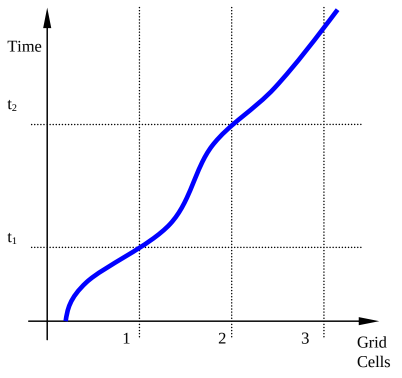


Figure 1. 2-dimensional example.

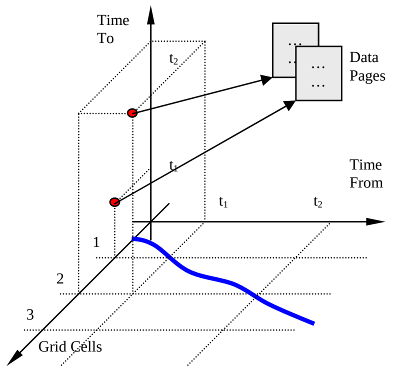


Figure 2. Indexing space.

negative updates every time an object enters or leaves a spatial cell. Hence we utilize a uniform grid structure to discretize the spatial domain. This grid is combined with two temporal axes ("from" and "to") to form a $d + 2$ dimensional index space (where d is the number of dimensions in the spatial domain).

In particular, for every time period $(t_{from}; t_{to})$ during which an object o_j was inside grid cell g_i we place a $d + 2$ *index point* I_i inside the index space. The projected coordinate of I_i on the temporal "from" axis is t_{from} and on the "to" axis is t_{to}. I_i keeps the period during which object o_j was within cell g_i, and points to a secondary storage (disk page) where the actual (raw) object movement data for the specified time period and the specified grid cell are stored. Hence, an **index point** carries a tuple $I_i = \langle o_j, g_i, t_{from}, t_{to}, p\rangle$, where o_j is an object in the stream S, g_i is the grid cell that contains object o_j (i.e., $\wedge(g_i, TR_{o_j}) = true$ for $\forall t \in \langle t_{from}, \ldots t_{to}\rangle$) and p is a pointer pointing to a place in secondary storage which stores the sequence of pairs $\{\langle l_1, t_1\rangle, \ldots, \langle l_n, t_n\rangle\}$, where $l_i \in g_i$ and $t_i \in \langle t_{from}, \ldots, t_{to}\rangle$.

As a result, an object trajectory can be abstracted as a "sequence of index points" in the indexing space.

Figure 1 shows a 1-dimensional trajectory which stays within grid cell 1 during interval $(0; t_1)$; it then stays inside

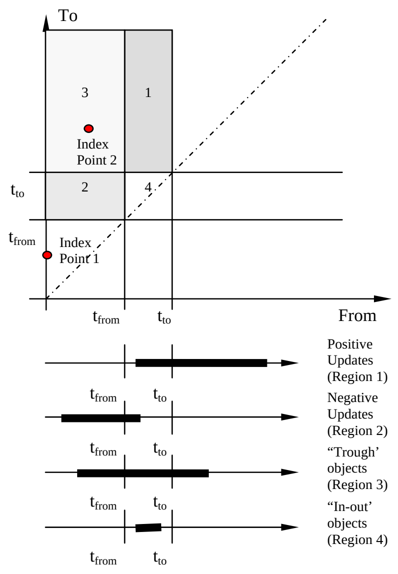

To
3
1
Index Point 2
2
4
t_to
t_from
Index Point 1
t_from
t_to
From
Positive Updates (Region 1)
t_from
t_to
Negative Updates (Region 2)
t_from
t_to
"Trough' objects (Region 3)
t_from
t_to
"In-out' objects (Region 4)
t_from
t_to

Figure 3. Indexing space projection.

grid cell 2 from t_1 to t_2, and finally moves inside grid cell 3. Figure 2 depicts the 3-dimensional indexing space for this example.

We now describe how the above indexing structure leads to effective incremental evaluation of continuous queries. To simplify the discussion we ignore the spatial axes in this indexing space and focus only on the plane formed by the two temporal axes i.e., the "from" and "to" axes. Figure 3 shows the projection of the index points of figure 2 on the time axes.

Given a time period $(t_{start}; t_{end})$ we can partition the index space in four regions as shown on figure 3. To locate all positive updates for time period $(t_{start}; t_{end})$ (e.g. all objects which enter some grid cell) we need to locate all index points I_i such that: $(t_{start} \leq I_i.t_{from} \leq t_{end}) \cap (I_i.t_{to} \geq t_{end}))$. These are the index points which reside inside region 1 in figure 3. Similarly, for the negative updates we need the index points I_i such that: $(I_i.t_{from} \leq t_{end}) \cap (t_{start} \leq I_i.t_{to} \leq t_{end}))$. These are the index points inside region 2 in figure 3. Region 3 contains objects which were inside the given grid cell during the whole time period $(\forall I_i \rightarrow (I_i.t_{from} \leq t_{start}) \cap (t_{end} \leq I_i.t_{to})))$ and region 4 contains objects which moved in and then moved out of some grid cell g_i during the time period.

As a result, the process of finding objects which change their location and move to another grid cell is equivalent to issuing a range query in the indexing space which retrieves the indexing points in the appropriate partition. Such range queries can be efficiently resolved with a spatial index (an R tree or kdb tree) build on top of the index space.

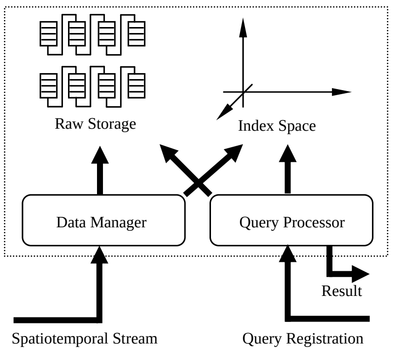


Figure 4. General Framework.

5 CMP Query Evaluation

We proceed with the evaluation algorithms for answering CMP queries assuming that we have a spatiotemporal stream S and the indexing described in the previous section.

5.1 General Framework

As depicted in Figure 4 there are two major processes in a CMP query evaluation framework that work in parallel. The Data Manager keeps the raw data storage and the indexed space on the server side consistent with the spatiotemporal stream S. The Query Processor is responsible for the continuous reevaluation of the motion pattern queries $\mathcal{Q}$ in the system. The clients submit their CMP queries and they are registered inside the Query Processor. During its lifetime, a registered motion pattern query goes trough two distinctive phases, namely: the Initialization phase and the Reevaluation phase. During the first phase the initial result for a CMP query is computed from scratch and then reported to the client. The result and the intermediate data structures are preserved inside the query processor. After the formation of the initial result, the evaluation of the continuous query moves to its second phase (and stays until it is removed from the system). In this phase the result is kept persistent through consecutive executions of an incremental evaluation algorithm. On a regular time basis the result is reported to the client. Given the different approaches for evaluating binary and numerical predicates, we present their algorithms separately.

5.2 Binary predicates

We will illustrate the evaluation algorithm for continuous binary predicates CBP using the predicate $inside$ as an example.

For simplicity first we assume that the spatial object so_i related to a binary spatial predicate $Q_i \in CBP$ can be covered precisely with the grid cells or, equivalently, that the

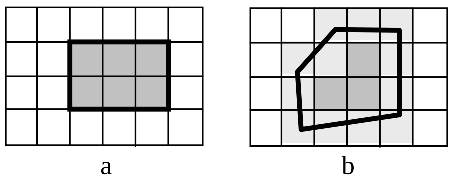


Figure 5. Query object snapping.

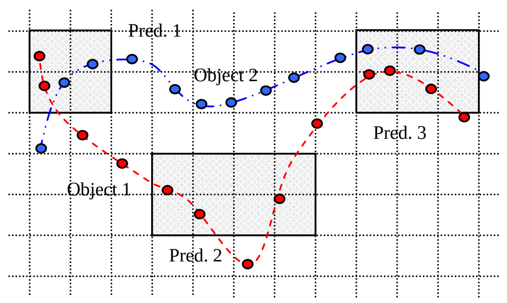
so_i borders are "snapped" to the grid (fig 5.a). (An extension that handles arbitrary object shapes like the one in fig 5.b will be discussed later). For every single predicate object so_i we have two types of grid cells: (1) grid cells entirely covered by object so_i and (2) grid cells not covered by so_i at all. The set of grid cells covered by object so_i form the *search area* of the binary predicate Q_i. (This depends on the type of the spatial predicate - for example in the predicate $disjoint$ the search area will be formed by the set of grid cells NOT covered by the so_i while for the predicate $inside$ this is the set of grid cells covered by so_i). Using the described index structure we locate all index points I_i inside the search area for the specified time constraint T_i.

A hash table called Binary Hash Table (BHT) is created. This hashing table is indexed by the object identifiers and has a column for every binary predicate $Q_i \in CBP$ in the query $\mathcal{Q}$. For each moving object o_j and for each predicate Q_i the table contains a list of index points I_i that have been discovered for this object inside the search area of the predicate Q_i. Checking if an object satisfies all predicates can be done on the fly while inserting the index points in the structure. If an object o_j covers all predicates (there are index points in all columns for this object) then this object satisfies the query $\mathcal{Q}$ and is placed inside the result set. Figure 6 shows an example of such a hash table. Algorithm 1 describes the first (initialization) phase.

During the second phase (Algorithm 2) the result is kept consistent by applying positive and negative updates. To do so we use the BHT table produced in the first phase. Assume that the last query reevaluation was at timestamp t_{prev} and the current timestamp is t_{now}. Using the partitioning of the index space as shown in Figure 3 inside the Q_i search area for a time period $(t_{prev}; t_{now})$ we can compute the list of positive and negative updates which occur in the Q_i search area for time interval $(t_{prev}; t_{now})$.

The set of index points inside region 2 forms the negative updates and the index points inside region 3 - the positive ones. We apply these updates to the BHT for predicate Q_i and adjust the result set accordingly.

A more general situation is when the spatial object so_i describing the spatial predicate O_i does not snap precisely to the grid. In this case there are three categories of grid cells, as following: (1) cells which are entirely covered by so_i (2) cells that are partially covered by so_i (3) cells not covered by so_i (Figure 5.b). We process the partially cov-



Obj.	Predicates		
	1	2	3
1	1 2	5 6 8	9 10 11
2	2 3	Not covered	11 12

Figure 6. Binary hash table example.

Algorithm 1 Binary query - phase 1

```
Require: Query $Q = \{(RP(r_1), T_1), \ldots, (RP(r_n), T_n)\}$,
 1: $C \leftarrow \emptyset$; $R \leftarrow \emptyset$; $H \leftarrow \emptyset$
 2: for $i = 1$ to $n$ do
 3:     $IdxPoints \leftarrow$ GetAllIndedxPoints$(r_i, T_i)$;
 4:     $GreyIdxPoints \leftarrow$ GetAllGreyIndedxPoints$(r_i, T_i)$;
 5:     while $IdxPoints \cup GreyIdxPoints$ not empty do
 6:         Entry $ip = IdxPoints \cup GreyIdxPoints$.pop
 7:         InsertIntoHash$(H, ip.obj, i, ip)$
 8:         if $H$.allPredicatesCovered$(ip.obj)$ then
 9:             if $H$.coveredByGrey$(ip.obj)$ then
10:                 $C$.push$(ip.obj)$;
11:             else $R$.push$(ip.obj)$;
12:             end if
13:         end if
14: while $C$ not empty do
15:     Entry $id = U$.pop
16:     $P =$ getTrajectoryData$(id)$
17:     if $P$ satisfies $Q$ then $R$.push$(id)$
18: end for
```

ered grid cells similarly as the fully covered grid cells with one major difference. If an object inside the hash table covers all the predicates but some of the index points I_i are generated from partially covered grid cells (we call them gray index points) we have to load the actual trajectory data from the secondary storage for this predicate and verify if the predicate is indeed satisfied.

5.3 Numerical predicates

We describe the algorithm for continuous numerical predicate CNP using as example predicate the $min\ dist$ while as Θ operator in $\mathcal{Q}$ we use min. This operator returns $true$ only for the moving object o_j with minimal sum of the distances to the query predicates $Q_i \in CNP$.

First we discuss the initialization phase (Algorithm 3). The general idea here is to use the index structure and the grid it contains to compute an upper bound $\mu_j.u$ and a lower bound $\mu_j.l$, of the object numerical score μ_j for every object o_j. Using these bounds we then prune as many object trajectories as possible.

For the case of a $min\ dist$ predicate we use the index structure described in section 3 (in particular the spatial grid inside it), to compute the upper and the lower bound distances of the actual distance between an object location and the spatial object so_i. By summing together the lower bound distances for all predicates we get a lower bound object score $\mu_j.l$. The sum of the upper bound distances generates the object upper bound score $\mu_j.u$. We can successfully prune and avoid the raw trajectory access for all objects which have a lower bound distance $\mu_j.l > \mu_j.u$ bigger than the upper bound distance of another object.

The algorithm starts from the grid cells containing the spatial object so_i associated with the predicate O_i and interactively examines all cells adjacent to them. (This is the case when the Θ operator minimizes the object scores μ_j. If Θ maximizes the object scores we start from the most distant grid cell from the query). In each step the process increases the number of adjacent cells examined by moving one step further away from the query point (see Figure 7). We maintain two hash tables in main memory - the Lower bound table (LBT) and the Actual Score Table (AST). The structures are populated with lower bound object scores $\mu_j.l$ and actual object numerical score μ_j as we visit the adjacent to the spatial object so_i cells.

Table LBT has a column for every query predicate $Q_i \in CNP$ and one row for every single object o_j discovered so far. It contains the lower bound score $f.l(Q_i, o_j[T_i])$ per predicate for every object. If the given predicate has not been covered by an object o_j in the corresponding column, we put the maximal lower bound distance for this predicate regardless of which object trajectory it corresponds to. Due to the incremental visit of grid cells the computed approximation is still a lower bound to the actual object score μ_j. In each iteration and for each predicate $Q_i \in CNP$ we add the grid cells one hop away from the grid cells accessed the previous step to the vicinity of the predicate Q_i. We query the index points I_i in the index space for these newly added grid cells, compute the lower bound score of the corresponding grid cell to the predicate Q_i and place them inside LBT. The sum of the lower bound distances in the LBT columns for a given object forms the object lower bound score $\mu_j.l$. The rows inside LBT are also sorted in increasing order of $\mu_j.l$.

The AST table stores the actual object score $f(Q_i, o_j[T_i])$ per predicate and is organized in the

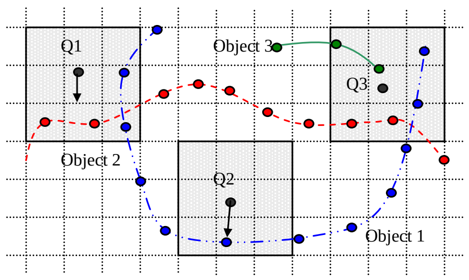
Algorithm 2 Binary query - phase 2

```
Require: Query $Q = \{(RP(r_1), T_1), \ldots, (RP(r_n), T_n)\}$, $R$, $C$, $H$
 1: for $i = 1$ to $n$ do
 2:     $PU \leftarrow$ GetPosUpdate$(r_i, T_i)$
 3:     $NU \leftarrow$ GetNegUpdate$(r_i, T_i)$
 4:     while $NU$ not empty do
 5:         Entry $ip = NU$.pop
 6:         RemoveFromHash$(H, ip.obj, i, ip)$
 7:         if !$H$.allPredicatesCovered$(ip.obj)$ then
 8:             if $ip.obj \in C$ then $C$.pop$(ip.obj)$;
 9:             else $R$.pop$(ip.obj)$;
10:             end if
11:         end if
12:     while $PU$ not empty do
13:         Entry $ip = PU$.pop
14:         InsertIntoHash$(H, ip.obj, i, ip)$
15:         if $H$.allPredicatesCovered$(ip.obj)$ then
16:             if $H$.coveredByGrey$(ip.obj)$ then
17:                 $C$.push$(ip.obj)$;
18:             else $R$.push$(ip.obj)$
19:             end if
20:         end if
21: while $U$ not empty do
22:     Entry $id = U$.pop
23:     $P$ = getTrajectoryData$(id)$
24:     if $P$ satisifes $Q$ then $R$.push$(id)$
25: end for
```

same way as LBT. There is one column for every query predicate $Q_i \in CNP$ and one row for every object, which covers all predicates in $\mathcal{Q}$. The sum of the scores in the columns forms the actual object numerical score μ_j. The rows inside AST are sorted in increasing order of μ_j.

We continuously compare the best lower bound in LBT and the best numerical score μ_j in AST. (these are the first rows in both tables). If the best lower bound in LBT is larger than the best numerical score μ_j in AST (for Θ maximizing scores it is the opposite) and the corresponding object in LBT covers all predicates in $\mathcal{Q}$, then for this moving object, the raw trajectory data is loaded, μ_j is computed and placed inside the AST. The algorithm stops when the best lower bound $\mu_j.l$ in LBT is bigger than the best numerical score μ_j in AST.

For the second phase (reevaluation) of the algorithm (Algorithm 4) we keep both tables - the LBT and AST and the vicinity discovered so far for each numerical predicate $Q_i \in CNP$ in $\mathcal{Q}$. Assume that the last query reevaluation was at time-stamp t_{prev} and the current time-stamp is t_{now}. Using the partitioning of the index space shown on Figure 3 we locate the positive and negative updates in each predicate vicinity for the time period since the last reevaluation. These updates are applied to LBT to keep it persistent. We keep AST persistent by loading the raw trajectory data for



Actual Score Table

Obj.	Predicate			Numerical score
	1	2	3	
1	1.5	1.3	1.1	3.9

Lower Bound Table

Obj.	Predicate			Lower Bound Score
	1	2	3	
3	0	1	1	2
2	0.7	1	0.4	2.1

Figure 7. NN query example.

the time period since the last reevaluation, and recomputing μ_j.

Both update types - positive or negative modify the lower bounds in LBT. When the updates are applied, it may happen that the best actual score in AST is bigger than the best lower bound score LBT. In this scenario, phase 1 is reevaluated increasing the vicinity circle for the predicates $Q_i \in CNP$. Elements from the LBT are popped and added to AST until the condition is satisfied again.

5.4 Predicates without Temporal Constraints

For CMP queries where the temporal constraints T_i are not specified, an ordering will be implied by the actual position of the predicate Q_i in the query sequence. For such CMP queries we need to verify that the predicates are satisfied in the proper order. To do so during the insertion of the index points I_i for predicate Q_i, which is without temporal constraint T_i, we check if $I_i.t_{from}$ and $I_i.t_{to}$ satisfy the order (e.g. $(\exists I_j\ for\ Q_{i-1} \rightarrow I_j.t_{to} \leq I_i.t_{from}) \cap (\exists I_k\ for\ Q_{i+1} \rightarrow I_k.t_{from} \geq I_i.t_{to})$) Only if I_i satisfies the order we increase the satisfied predicate's counter. To speed up this process the index points can be kept sorted during the insertions and deletions in BHT by using a heap (details are omitted due to lack of space).

Algorithm 3 Numerical query - phase 1

```
Require: Query $Q = \{(NNP(q_1), T_1), \ldots, (NNP(q_n), T_n)\}$
 1: $AST \leftarrow \emptyset$; $LBT \leftarrow \emptyset$; $r = 0$
 2: while $AST.best \geq LBT.best$ do
 3:     for $i = 1$ to $n$ do
 4:         $r = r + 1$;
 5:         $SA \leftarrow$IncreaseSearchArea$(q_i, r)$;
 6:         $IdxPoints \leftarrow$ GetAllIndedxPoints$(SA, T_i)$;
 7:         while $IdxPoints$ not empty do
 8:             Entry $ip = IdxPoints$.pop
 9:             if $ip.obj \in AST$ thenInsertIntoAST$(ip, i)$;
10:             else InsertIntoLBT$(ip, i)$;
11:             end if
12:         end while
13:     end for
14:     while $LBT.best$ covers all predicates do
15:         Entry $id = LBT.best$;
16:         $P =$ getTrajectoryData$(id)$;
17:         $AST$.LoadIntoAST$(P)$;
18:     end while
19: end while
```

Algorithm 4 Numerical query - phase 2

```
Require: Query $Q = \{(RP(r_1), T_1), \ldots, (RP(r_n), T_n)\}$, $AST$, $LBT$, $r$
 1: for $i = 1$ to $n$ do
 2:     $SA \leftarrow$getSearchArea$(q_i, r)$;
 3:     $PU \leftarrow$ GetPosUpdate$(SA, T_i)$
 4:     $NU \leftarrow$ GetNegUpdate$(SA, T_i)$
 5:     while $NU$ not empty do
 6:         Entry $ip = NU$.pop
 7:         DeleteFromLBT$(ip, i)$;
 8:         end if
 9:     end while
10:     while $PU$ not empty do
11:         Entry $ip = PU$.pop
12:         InsertIntoLBT$(ip, i)$;
13:         end if
14:     end while
15: end for
16: RefreshAST
17: if $AST.best \geq LBT.best$ then goto phase 1;
18: end if
```

6 Experimental Evaluation

In our experiments we use synthetic data to test the behavior of each algorithm under different settings. We have created up to 150,000 objects moving in a 2-dimensional spatial universe which is 1,000 miles long in each direction. Objects follow random routes on a freeway network traveling through a number of consecutive intersections and report their positions every time-instant. Query reevaluation is done every 2 minutes. On the top of this data we build the indexing space as it is described in section 4. We used a standard R tree (with utilization factor 64%) and a KDB tree as the indexing structures build on top of the index points. To test the proposed techniques we use two measures, namely: (i) the average number of index node accesses, which is mainly CPU related and, (ii) the average number of data pages per query that need to be retrieved from secondary storage for verification of the result (the I/O cost of the algorithms). We refer to the first phase as ”initial” and to the second as ”continuous”.

6.1 Comparison with the brute force approach

First we compare our algorithms with a brute-force approach where all trajectories from the repository are examined sequentially. The results are shown in Figure 8. Note the logarithmic scale. We can see that the proposed index structure and algorithms help reduce the total I/O cost by orders of magnitude for both the numerical and the binary predicates. The brute force approach is shown to be computationally very expensive and is thus not depicted in the

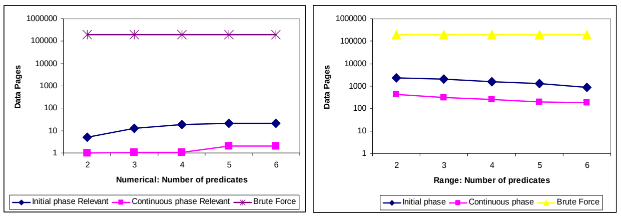


Figure 8. RANGE AND NN QUERIES: Number of predicates.

remaining experiments.

6.2 Binary Predicates

6.2.1 Performance vs. Number of Predicates

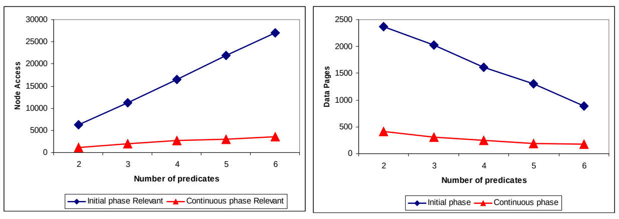


Figure 9. RANGE QUERIES: Number of predicates.

This set of experiments examines how the algorithms perform for queries with increasing number of predicates. Figure 9 depicts the average number of index node and data pages accesses for different number of predicates. A dataset with size 100,000 objects was used. Clearly the continuous phase is much faster than the initial phase. As expected, the number of index points accessed in both the initial and continuous phases increases with the number of the query predicates because of the increased total search area. The number of data pages accessed decreases linearly as the number of predicates increases. With the increase in the number of predicates it becomes less likely to find objects which satisfy all predicates; thus there is a smaller number of candidates which must be evaluated using the raw trajectory data.

6.2.2 Performance vs. Dataset Size

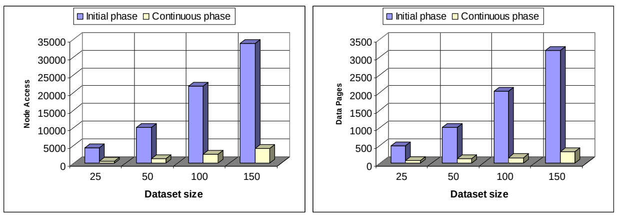


Figure 10. RANGE QUERIES: Dataset Size.

We next evaluate the performance scale-up for various dataset sizes. We use queries with five range predicates. Figure 10 shows the results for the four different datasets (the size is given in thousands). Again the continuous phase is an order magnitude faster than the initial one. As expected, the average number of node accesses per query increases as the dataset size increases. This is because the density in the index space is increased and the total number of points inside the query regions also grows. The number of objects in the result set also increases and this causes the increase of the data I/Os needed for the verification step. Nevertheless, the number of index points and data pages accessed during the reevaluation step is much smaller than the same number in the initial phase. This is due to the incremental evaluation and because during the reevaluation we access only partitions 1 and 2 (see Figure 3). Given the small reevaluation period these partitions have relatively small area and therefore generate limited number of index points.

6.3 Numerical Predicates

6.3.1 Performance vs. Number and Type of Predicates

For the nearest neighbor queries we generate two query sets, namely: (i) a RandomPattern set where the location of the query predicate and its time interval are chosen in random, and, (ii) a RelevantPattern set which is generated using existing object trajectories, that are slightly skewed in space as compared with the original. The results for both datasets are shown on Figure 11. We first discuss node accesses Our incremental algorithm is expected to work best when the

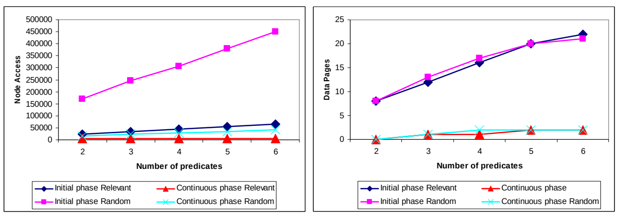


Figure 11. NN QUERIES: Number of predicates.

spatial stream contains object trajectories that are very similar to the query definition (RelevantPattern). This enables fast pruning because the algorithm quickly finds an object which covers all numerical predicates. Then its numerical score is used for pruning the other objects. For the RandomPattern set as the number of predicates increases the probability that a given object matches closely the query predicate decreases dramatically. As a result, the discovered areas for each predicate need to grow large until an object which satisfies all predicates is found. This results in a large number of index points which have to be retrieved. For the data page access the behavior of the algorithms in the initial and continuous phase is similar. This is because few candidate trajectories have been discovered by the end of the initial phase resulting in a limited number of verification steps. As with the binary predicates the continuous phase is faster.

6.3.2 Performance vs. Dataset Size

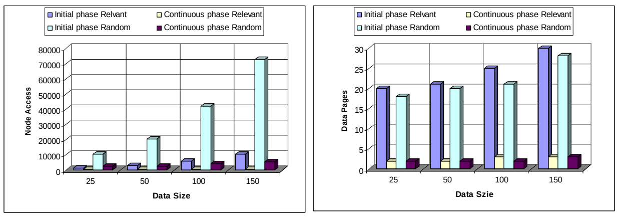


Figure 12. NN QUERIES: Dataset Size.

To test the scalability of the numerical predicate algorithm we use the same dataset sizes (given in thousands) used in the scale-up test for the binary predicates. The results are shown in Figure 12. As it can be seen for the RandomPattern dataset the average number of node accesses per query starts growing very fast with the increase of the dataset size compared with the growth in the RelevantPattern query set. This is because in the RelevantPattern the algorithm discovers the candidate objects very fast and the vicinity of the numerical predicates is thus relatively small. As a result, the search for positive and negative updates during the continuous phase reaches only a limited number of index points. As for the data pages, in the initial phase for both query sets there is an increase in the number of pages accessed with the increase of the dataset size. Nevertheless

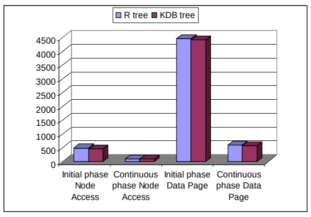


Figure 13. Index type. R tree v.s. KDB tree

the actual number is relatively small since for NN queries few candidate trajectories are typically found.

6.4 Performance using KDB and R tree

We tested the algorithms performance using KDB tree and R tree structures for efficient access to the content of the indexing space. We use dataset containing 25K trajectories and query with 5 range predicates. The results are shown on Figure 13. As it can be depicted the algorithmic performance does not depend of the type of index structure chosen. The KDB tree performs slightly better mainly because there is no overlapping between the point bounding boxes which is advantageous for point data indexing.

6.5 Performance vs. Buffer Size

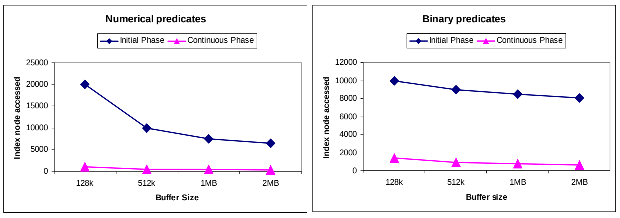


Figure 14. Buffer Size.

Figure 14 shows the effect of different buffer sizes on the query performance for 50K objects. Since we run multiple nearest neighbor searches concurrently we expect that there will be a large number of pages accessed in two or more consecutive iterations. Larger buffer sizes can help alleviate loading these nodes multiple times. As expected numerical predicate queries benefit the most since they iteratively visit grid cells which are spatially close.

7 Conclusions

In this paper we define a novel type of complex continuous queries called Continuous Motion Pattern Queries. We present a framework for efficient evaluation of the CMP queries, starting with an index structure for spatiotemporal streams, oriented towards incremental evaluation of continuous queries. The results show that our algorithms are able to do fast pruning and thus achieve very good performance. Overall the continuous phase in our system is very efficient which leads to very fast incremental query evaluation.

References

[1] M. Erwig. Toward spatiotemporal patterns. *Spatio-Temporal Databases*, pages 29–54, 2004.

[2] M. Erwig and M. Schneider. Spatio-temporal predicates. *IEEE Transactions on Knowledge and Data Engineering*, 2002.

[3] B. Gedik and L. Liu. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *Proc. of Extending Database Technology (EDBT)*, pages 67–87, 2004.

[4] M. Hadjieleftheriou, G. Kollios, P. Bakalov, and V. J. Tsotras. Complex spatio-temporal pattern queries. In *VLDB*, pages 877–888, 2005.

[5] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatiotemporal objects. In *Proc. of Extending Database Technology (EDBT)*, pages 251–268, 2002.

[6] N. Koudas, B. C. Ooi, K-L. Tan, and R. Zhang. Approximate nn queries on streams with guaranteed error/performance bounds. In *VLDB*, 2004.

[7] D. Kwon, S. Lee, and S. Lee. Indexing the current positions of moving objects using the lazy update r-tree. In *MDM*, pages 113–120, 2002.

[8] M.-L. Lee, W. Hsu, C. S. Jensen, and K. L. Teo. Supporting frequent updates in R-Trees: A bottom-up approach. In *VLDB*, 2003.

[9] D. Lin, C. S. Jensen, B. C. Ooi, and S. Saltenis. Efficient indexing of the historical, present, and future positions of moving objects. In *MDM*, pages 59–66, 2005.

[10] M. F. Mokbel and W. G. Aref. Gpac: generic and progressive processing of mobile queries over mobile data. In *MDM*, pages 155–163, 2005.

[11] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal access methods. *IEEE Data Engineering Bulletin*, 26(2):40–49, 2003.

[12] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatiotemporal databases. In *Proc. of ACM Management of Data (SIGMOD)*, 2004.

[13] H. Mokhtar, J. Su, and O. Ibarra. On moving object queries: (extended abstract). In *Proc. of ACM Symposium on Principles of Database Systems (PODS)*, pages 188–198, 2002.

[14] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *Proc. of ACM Management of Data (SIGMOD)*, pages 634–645, 2005.

[15] S. Prabhakar, Y. Xia, D. Kalashnikov, W. Aref, and S. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Trans. Comput.*, 51(10):1124–1140, 2002.

[16] S. Saltenis and C. S. Jensen. Indexing of moving objects for location-based services. In *ICDE*, pages 463–472, 2002.

[17] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. *SIGMOD Record*, 29(2):331–342, 2000.

[18] Y. Tao, D. Papadias, and J. Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, pages 790–801, 2003.

[19] X. Xiong, M.. Mokbel, and W. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005.

[20] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, pages 631–642, 2005.