Complex Spatio-Temporal Pattern Queries

Marios Hadjieleftheriou, George Kollios
Computer Science Department
Boston University
{marioh, gkollios}@cs.bu.edu

Petko Bakalov, Vassilis J. Tsotras
Computer Science Department
University of California, Riverside
{pbakalov, tsotras}@cs.ucr.edu

Abstract

This paper introduces a novel type of query, what we name *Spatio-temporal Pattern Queries* (STP). Such a query specifies a spatio-temporal pattern as a sequence of distinct spatial predicates where the predicate temporal ordering (exact or relative) matters. STP queries can use various types of spatial predicates (range search, nearest neighbor, etc.) where each such predicate is associated (1) with an exact temporal constraint (a time-instant or a time-interval), or (2) more generally, with a relative order among the other query predicates. Using traditional spatio-temporal index structures for these types of queries would be either inefficient or not an applicable solution. Alternatively, we propose specialized query evaluation algorithms for STP queries With Time. We also present a novel index structure, suitable for STP queries With Order. Finally, we conduct a comprehensive experimental evaluation to show the merits of our techniques.

1 Introduction

Spatio-temporal data management has received a lot of attention recently, mainly due to the emergence of location based services and advances in telecommunications (cheap GPS devices, ubiquitous cellular networks, RFIDs, etc.) As a result, large amounts of spatiotemporal data is produced daily, typically in the form of trajectories. The need to efficiently analyze and query this data requires the development of sophisticated techniques. Previous research has concentrated on various spatio-temporal queries, mainly focusing on range searches and nearest neighbor variations [18, 19, 9, 16, 14], or mining tasks like extracting patterns and periodicities from spatiotemporal trajectories [17, 11]. This paper introduces a novel problem, what we term *Spatio-temporal Pattern Queries* (STP). Given a large collection of spatiotemporal trajectories, an STP query retrieves all trajectories that follow user defined movement patterns in space and time.

There are many practical applications where STP queries appear. For example, "Identify all vehicles that were very close to all three sniper attacks in Maryland (the locations and times of the attacks are known)" or "Locate products that left the factory a month ago, were stored in one of the warehouses near the dock, and loaded on a ship (locations can be tracked by using RFIDs)". While there have been various previous works addressing the problem of pattern discovery [17, 11], to the best of our knowledge, there has been no previous work on the orthogonal problem, the STP queries.

We represent a spatio-temporal pattern as a sequence of distinct spatio-temporal predicates, where the temporal ordering (exact or relative) of the predicates matters. STP queries can have arbitrary types of spatial predicates (e.g., range search, nearest neighbor, etc.), where each predicate may be associated (1) with an exact temporal constraint that is either a time-instant or a time-interval (STP Queries With Time), or (2) more generally, with a relative order (STP Queries With Order).

An STP query With Time example is: "Find objects that crossed through region A at time T_1, came as close as possible to point B at a later time T_2 and then stopped inside circle C some time during interval (T_3, T_4)". An STP query With Order is: "Find objects that first crossed through region A, then passed as close as possible from point B and finally stopped inside circle C". Here only the *relative order* of the

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

This work has been supported by NSF grants IIS-0133825, IIS-0308213, and IIS-0330481.

spatial predicates is important, independently of when exactly they occur in time.

The straightforward approach for answering STP queries is to evaluate the query pattern on all trajectories one by one using a linear scan on a sequentially stored archive. Clearly, this approach has prohibitive cost and in some cases might be infeasible due to very large database sizes and due to the expensive nature of the distance function used for STP queries With Order (as will be seen later on). When considering STP queries With Time, another straightforward approach would be to use a traditional spatio-temporal index structure [18, 19, 21, 9] to index the trajectories and utilize the index to evaluate the query predicates individually; respective answers can be combined in the end. When these STP queries consist only of range spatial predicates such a solution might work well in various practical cases. Nevertheless, this approach does not work for spatial predicates that need to be evaluated conjointly, like nearest neighbors. Consider the following example: "Find the object trajectory that crossed as close as possible from point A at time T_1 and then, as close as possible from point B at a later time T_2". Individually evaluating each predicate cannot provide the trajectory that minimizes the distance from both points. Alternatively, we show how to adapt the traditional best first search approach with a combined distance function (e.g., the sum of distances of each trajectory from the query points) to answer STP queries With Time. With careful evaluation strategies we can guarantee that each needed page from the index is loaded only once, during the evaluation of all predicates.

Nevertheless, traditional spatio-temporal index structures would be a very inefficient solution for answering STP queries With Order. Since only the relative order of the predicates is significant, the spatio-temporal index will have to retrieve the whole temporal evolution of each object. For such queries, we propose a novel indexing technique that enables efficient evaluation by storing 'order' information inside the index.

To summarize, the contributions of this paper are the following: (1) We introduce and formalize a novel query type (STP) that combines general spatial predicates with temporal constraints (STP queries With Time) and/or relative ordering (STP queries With Order). (2) We propose specialized query evaluation algorithms for these two types of STP queries, as well as a novel index structure, suitable for STP queries With Order. (3) Finally, we present an extensive experimental evaluation of the proposed techniques.

2 Problem Definition

Consider a large archive of object trajectories and a well-defined trajectory representation such that the location of the objects can be computed for any given time (the framework that will be presented is independent of the underlying trajectory representations). An

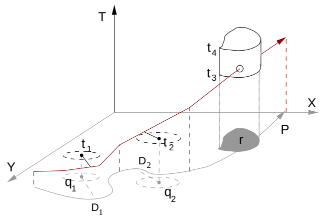


Figure 1: An example STP query With Time.

STP query is expressed as a sequence $\mathcal{Q}$ of arbitrary length m of ordered spatio-temporal predicates of the form

$$\mathcal{Q} = \{(Q_1, T_1), (Q_2, T_2), \ldots, (Q_m, T_m)\}$$

where in each pair (Q, T), Q represents a spatial predicate and T a temporal constraint. For simplicity but without loss of generality, in the rest, Q will express either a *range* (R) or a *nearest neighbor* (NN) query. As will become clear, our framework can be adapted easily to handle other spatial predicates as well. T is either a *time-instant* (t), a *time-interval* (Δt) or *empty* $(\emptyset)$. Inherently the temporal constraints impose a strict ordering on the spatial predicates. When a temporal constraint is empty, ordering will be implied by the actual position of the associated predicate in the query sequence. An alternative query expression mechanism appeared in [3], where regular expressions were used to represent mobility patterns. More details and limitations of this approach appear in the related work.

We say that a trajectory satisfies an STP query if it satisfies all spatio-temporal predicates at once:

- A range predicate is trivially, and individually, satisfied by a trajectory if the object is contained inside the given range anytime during the specified temporal constraint.

- A nearest neighbor predicate is satisfied only with respect to all other NN searches as well. We say that a trajectory satisfies the NN predicates if it minimizes the sum of the distances from those predicates during the given temporal constraints.

An example STP query With Time is shown in Figure 1. The query depicted is $\mathcal{Q} = \{(NN(q_1), t_1), (NN(q_2), t_2), (R(r), [t_3, t_4))\}$, and is satisfied by the trajectory that minimizes the sum of distances from points q_1, q_2 at times t_1, t_2, respectively, and crosses region r anytime between $[t_3, t_4)$.

3 STP Query Algorithms

For ease of exposition we first present our solutions for STP Queries With Time, i.e., STP queries that contain spatial predicates with non-empty temporal constraints. Subsequently, we discuss solutions for the more general STP Queries With Order, i.e., queries that are ordered sequences of spatial predicates without temporal constraints.

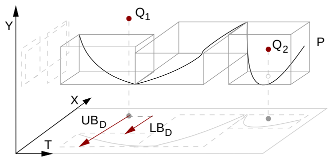


Figure 2: Approximating a trajectory with multiple MBRs.

3.1 STP Queries With Time

Given that these queries contain both spatial and temporal constraints, they can be answered using specialized evaluation strategies on existing spatio-temporal index structures. The spatio-temporal index can serve as a filtering step that will reduce the number of accesses to raw trajectory data by pruning trajectories that do not satisfy the pattern and improving query performance (against the linear search) by orders of magnitude.

For simplicity we adopt a general trajectory indexing scheme. Nevertheless, the algorithms that will be presented *do not make any assumptions about the underlying index*, as long as it can answer efficiently nearest neighbor and range queries . Without loss of generality, we assume that the trajectories are approximated using a large number of Minimum Bounding Rectangles [19, 9], which are then indexed using a spatio-temporal index structure like the R-tree [8, 18] or the MVR-tree [21, 9]. With minor modifications to our framework, other approximation techniques are applicable as well. Every MBR inserted in a leaf level of the tree is associated with the identifier of the trajectory that it belongs to, and bounds a small time-interval of the object's movement history. An example of this indexing scheme is shown in Figure 2, where a trajectory has been approximated using a total of three MBRs.

For the rest of this section we assume that the MBRs are indexed using a *secondary* spatio-temporal index structure, with data entries pointing to the raw trajectory data on disk. Raw data is stored on sequential data pages per trajectory.

Among STP queries with time, we first discuss queries that contain multiple range predicates, since they can be evaluated in a straightforward way. We then consider queries with multiple NN predicates and present various algorithmic approaches to evaluate them. Finally we discuss queries that consist of combinations of range and NN predicates.

3.1.1 Range Predicate Evaluation

Assuming a pattern query that contains only range spatio-temporal predicates, there are two obvious evaluation strategies to consider. The first approach produces the candidate results of all predicates concurrently using the index, and then loads from storage only the trajectories that belong to the intersection of the partial answer sets. The second strategy evaluates the most selective predicate first (assuming selectivity information is available), then the raw trajectory data is loaded from storage and the rest of the range predicates are answered in main memory, using the retrieved data. Which approach is better depends on the actual selectivities of the queries and the size of the trajectories.

3.1.2 Nearest Neighbor Predicate Evaluation

Consider an STP query that contains only nearest neighbor predicates. The basic idea behind our approach is to use the index structure and the trajectory MBRs to compute approximate object distances that will enable fast pruning of many trajectories, without having to load the raw trajectory data. Our technique runs one best-first-search algorithm per query predicate that returns, successively, the object with the smaller distance from that predicate. The total distance of a single trajectory from the query can be computed as the sum of individual distances from all predicates. Intuitively, by utilizing the trajectory MBRs we can compute both upper and lower-bounds of the actual distance of a trajectory from a given predicate, as shown in Figure 2. By combining approximate distances for all query predicates we will be able to prune trajectories that have lower-bounding distances larger than any known upper-bound.

A simple 1-dimensional example is shown in Figure 3(a). This STP query is expressed as $\mathcal{Q} = \{(NN(0), 1), \ldots, (NN(0), 5)\}$. Conceptually, it can be thought of as a time-interval nearest neighbor pattern: "Locate the object that stays closer to the origin during time-interval $[1, 5]$". Trajectory P_1 is the answer to this query since it minimizes the sum of distances from all five points, with distance $\mathcal{D}(Q, P_1) = 5.5$. Trajectory P_3 does not qualify since it partially intersects the query lifetime and has infinite distance for some time-instants. Using this simple example, we will illustrate two evaluation strategies, called *lazy* and *eager*, and will discuss their advantages.

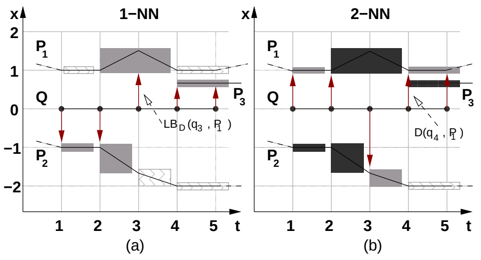


Figure 3: (a) Three trajectories and an STP query. For each query point the 1-NN MBRs (solid gray) are retrieved. (b) The MBRs belonging to P_1 cover the query, while those of P_2 do not cover points 4 and 5. Given $D(\mathcal{Q}, P_1) = 5.5$ and $LB_D(\mathcal{Q}, P_2) \geq 5.7$, trajectory P_2 can be pruned (any missing MBRs for points 4 and 5 should be at least as far as $D(q_{4,5}, P_1) = 1$ unit from the query).

Let $D(\mathcal{Q}, P)$ denote the actual distance of trajectory P from $\mathcal{Q}$. Also, let $LB_D(\mathcal{Q}, P)$ $(UB_D(\mathcal{Q}, P))$ denote a lower-bound (upper-bound) distance of P

from $\mathcal{Q}$ computed by using the distances of the MBR approximations of P from the query predicates. A threshold Λ needs to be computed so that all trajectories with $LB_D(\mathcal{Q}, P) > \Lambda$ can be pruned. In order to achieve this we incrementally locate the 1-NN, 2-NN, etc. MBRs individually for each query predicate. These MBRs should also contain the predicate in the temporal dimension. After a number of MBRs have been reported per q_i, for every discovered trajectory P (i.e., a trajectory for which at least one MBR has been reported) there are two cases: (1) the union of P's MBRs contain all query points in the time dimension and we say that P *covers* the lifespan of $\mathcal{Q}$, or (2) some points of $\mathcal{Q}$ are not covered. For example, in Figure 3(a) the 1-NN MBRs for each point are reported first (solid gray rectangles). At this step, no discovered trajectory MBRs cover all query points. In Figure 3(b) the 2-NN MBRs are retrieved. This time, the MBRs belonging to trajectory P_1 cover all query predicates.

In the first case both $LB_D(\mathcal{Q}, P)$ and $UB_D(\mathcal{Q}, P)$ can be computed without having to access the raw trajectory data. The upper-bound can be used as a pruning threshold Λ. The lower-bound can be used to prune the trajectory according to an already computed Λ. In the second case, a pessimistic approximation of $LB_D(\mathcal{Q}, P)$ can be computed. For each query predicate q_i that is covered by P the partial lower-bounding distance is equal to $D(q_i, P)$. For the query points that are not covered by P the maximum such distance $D(q_i, P')$ is used, regardless of which trajectory it corresponds to. Referring back to Figure 3(b), points 4 and 5 are not covered by P_2, thus: $LB_D(\mathcal{Q}, P_2) \geq \sum_{i=1}^{3} D(q_i, P_2) + D(q_4, P_1) + D(q_5, P_1)$. Due to the incremental discovery of trajectory MBRs, the computed approximation is still a lower-bound of the actual distance. By discovering trajectory MBRs incrementally and continuously improving the computed bounds, the number of raw trajectory data that need to be loaded from storage is reduced substantially.

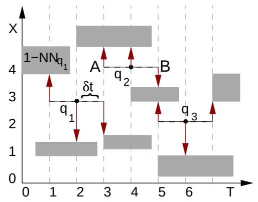


Figure 4: Evaluating time-interval predicates.

In order to evaluate time-interval temporal predicates, for every query predicate q_i we need to locate the nearest MBRs considering all time-instants contained in the given interval. An example is shown in Figure 4, for $\delta t = \pm 1$. In order to perform correct pruning, the value of $D(q, P)$ should be set to the closest MBR to q_i in Δt. This MBR is the one that minimizes the distance from the multi-dimensional region outlined by the query predicate when conceptually sweeping time-

Algorithm 1 Lazy Nearest Neighbor STP queries With Time

```
Input: Query $\mathcal{Q} = \{(NN(q_1), t_1), \ldots, (NN(q_n), t_n)\}$
Output: Nearest neighbor $P_B$
 1: Structure $\mathcal{S} \leftarrow \emptyset$ , Set $U \leftarrow \emptyset$, $PQ_{1,\ldots,n} \leftarrow \emptyset$
 2: $D_{q_1,\ldots,q_n} = 0$, $\Lambda = \infty$, $k = 1$, $P_B = \emptyset$
 3: Initialize priority queues $PQ_i$
 4: while true do
 5:     ConcurrentBestFirstSearch($k, Q, PQ_i, U, \mathcal{S}, D_{q_i}$)
 6:     for $P$ in $\mathcal{S}$ do
 7:         if $LB_D(\mathcal{Q}, P) > \Lambda$ then $\mathcal{S}$.remove($P$),
            $U$.enqueue($P$)
 8:         else if $\mathcal{Q}$ Covers $P$ then
 9:             $P_D$ = GetTrajectoryData($P$)
10:             if $D(\mathcal{Q}, P_D) < \Lambda$ then $\Lambda$ =
                $D(\mathcal{Q}, P_D), P_B = P$
11:             else $\mathcal{S}$.remove($P$), $U$.enqueue($P$)
12:     for $P$ in $\mathcal{S}$ do
13:         if $LB_D(\mathcal{Q}, P) < \Lambda$ then $stop$ = false
14:         else$\mathcal{S}$.remove($P$), $U$.enqueue($P$)
15:     if $stop$ then break
16:     $k+ = 1$
17: end while
18: Return $P_B$
```

interval Δt (e.g., the 2-dimensional line AB shown in Figure 4) and, thus, the search can be performed as a traditional NN search with an MBR query predicate.

Given the secondary index assumption, the cost of accessing the raw data is equivalent to one random disk access per trajectory. [*] In that case it is reasonable to postpone the data reads as much as possible and utilize lower-bounds instead, such that a smaller candidate trajectory set can be populated first. That way, we reduce the total number of accesses to the raw data storage, which means that query performance will depend mostly on the index access cost (depending on the size of the final candidate set). We call this the *Lazy* strategy. On the other hand, an alternative strategy is to eagerly load the raw trajectory information when evaluating each individual predicate, and directly compute the actual distance of each trajectory from the STP query. We can use the actual distances as tighter pruning thresholds that will help prune other trajectories using lower-bounds, very efficiently. We call this the *Eager* strategy. The best strategy to use depends on the characteristics of the dataset (how many pages are occupied per trajectory) and the query properties.

The lazy algorithm is shown in Algorithm 1. [†] All reported MBRs are probed into structure $\mathcal{S}$ after being removed from the priority queues during the NN searches. The structure is responsible for keeping updated the current lower-bounds of each trajectory and the query predicate coverage information. For that

[*]We assume that individual trajectories are stored sequentially on disk if they occupy more than one page.

[†]The algorithm for the eager strategy needs only simple modifications and is thus omitted.

Algorithm 2 Concurrent Best First Search

```
Input: $k, \mathcal{Q}, PQ_n, U, \mathcal{S}, D_{q_1,\dots,q_n}$
 1: for $i = 1$ to $n$ do
 2:     while $PQ_i$ not empty do
 3:         Entry $N = PQ_i$.top
 4:         if $N$ is a node then
 5:             for $j$ such that $PQ_j$.contains($N$) do
 6:                 $PQ_j$.remove($N$)
 7:                 for $C$ in $N$ do
 8:                     if $C$ not in $U$ and $C$ covers $q_j$ then
 9:                         $PQ_j$.enqueue($C$, $D(Q, C)$)
10:         else if $N$ is a data entry then
11:             S.insert($N$)
12:             Update $D_{q_i}$
13:             if $k$-NN is discovered, continue from 1
14: end for
```

purpose a hash table indexed by trajectory identifiers is constructed. The hash table contains one entry per trajectory. Each entry stores the current lower-bound distance and a bit vector, one bit per query predicate, indicating which predicates have not been covered yet by the trajectory MBRs. Every time a new MBR is discovered, the corresponding trajectory entry is retrieved, the appropriate bits are set and the lower-bound distance is updated. Examining if a trajectory covers the query is a binary AND operation on the bit vector. For trajectories that cover the query, the actual distance $D(\mathcal{Q}, P)$ is computed and stored as the lower-bound. For other trajectories, the appropriate $D(q_i, P)$ values are used.

The function `ConcurrentBestFirstSearch` (Algorithm 2) utilizes the best first search nearest neighbor algorithm to find the k-NN of every query point. The search is incremental so that the priority queues can be preserved and reused between subsequent executions. Array $D(q_i)$ is maintained by storing for each query point the distance from the last entry removed from the top of the corresponding priority queue. Both functions can be straightforwardly modified to support time-interval predicates as well as top-k searches, where more than one trajectories are retrieved. For completeness, we use the extended algorithm for our experimental evaluation, but present the simpler versions here for ease of exposition.

Since the algorithm runs a number of concurrent NN searches, it is unavoidable that some tree nodes will be retrieved more than once for a number of different predicates, even though each best first search will access the minimum required number of nodes individually [10]. In cases where there are no memory constraints LRU buffering can be used. Since many NN searches will have similar priority queues that share many common entries, especially for query predicates that lie close in space and time, locality of reference ensures that LRU buffering techniques will have a high hit ratio after the buffer becomes hot. Alternatively, the NN searches can be performed in a round robin fashion, and every time a node is retrieved from the top of one priority queue, the rest of the queues are scanned and if the node is already contained in any of them, then it is directly replaced with its children. It can be shown easily that due to the sequence of insertions in the queues, this strategy will guarantee that every node that is ever accessed by any predicate, will need to be accessed only once for all queues.

3.1.3 Combinations

For STP Queries With Time that contain combinations of range and NN searches, the evaluation order of the predicates is restricted, by construction, since the satisfiability of the NN predicates depends on the satisfiability of all the range searches. There are two alternatives: (1) Evaluate some or all of the range predicates first and then proceed with nearest neighbors evaluation and (2) evaluate all nearest neighbors first, by taking into account the satisfiability of the range searches whenever a candidate for updating threshold Λ is retrieved. If the selectivity of any range predicate is known to be very small then, after evaluating this predicate, all qualifying trajectories can be loaded from storage and the rest of the queries can be processed in main memory. Otherwise, all range predicates can be evaluated in advance and the disqualified trajectories can be pinned in set U.

3.2 STP Queries With Order

Assume now that the user is not interested in the exact times that the spatial predicates were satisfied, but only in the order in which they did. Hence these STP queries can be expressed as ordered sequences of spatial predicates of the form $\mathcal{Q} = \{Q_1, Q_2, \dots, Q_n\}$. Let $S_{Q_m}(P, t_k)$ denote the fact that trajectory P, during its lifetime, satisfied predicate Q_m at time-instant t_k. A single trajectory may satisfy any given predicate for multiple time-instants. Formally, we say that trajectory P satisfies query $\mathcal{Q}$ if there exists an n-length sequence $S_{Q_1}(P, t_1), \dots, S_{Q_n}(P, t_n)$ such that $t_i \le t_j$, for all $1 \le i < j \le n$. Note that the time-interval between any two consecutive t_i, t_j can be arbitrary.

A solution that utilizes existing spatio-temporal index structures (as in the previous section) will certainly be inefficient for STP queries with order, since no temporal constraints are specified. In order to evaluate a single predicate, the complete evolution of the index on the temporal dimension will have to be examined. That is, a range query will need to encompass the whole lifespan of the dataset on its temporal dimension (which for large trajectory archives becomes prohibitively expensive). Instead, we propose efficient solutions to STP queries With Order by using specialized index structures.

3.2.1 Range Predicate Evaluation

Instead of using a (3-dimensional) spatio-temporal index one could project out the temporal dimension from each trajectory and index the resulting objects with a 2-dimensional spatial index. A spatial range query can then easily retrieve all trajectories that satisfy it, irrespective of the time that they did so. Intuitively, to find if a trajectory satisfies the STP query, all range predicates would be evaluated first, the intersection of the individual result sets would be computed, and the remaining trajectories would be retrieved in order to check if they really do satisfy the predicates in the correct order.

There are however clear disadvantages with this solution. First, this structure does not inherently preserve order. Moreover, the quality of the 2-dimensional index is expected to be worse than the 3-dimensional spatio-temporal one, due to increased overlapping of projected MBRs.

Ideally, we would like to have an index structure that maintains the order with which each trajectory satisfies an arbitrary sequence of range predicates. One way to accomplish this would be to create a 'predicate index', that is, an index on the range predicate themselves, instead of the object trajectories. For each range predicate r_m, the structure associates an ordered list of $S_{r_m}(P_l, t_k)$ entries that contains all trajectories that satisfy the predicate. Entries in such list are ordered by trajectory identifier (P_l). Since a trajectory can satisfy predicate r_m many times throughout its duration, the entries of the same trajectory are further ordered by time (t_k). Then, given an STP query $\mathcal{Q} = \{R(r_1), R(r_2), \ldots, R(r_n)\}$, all range predicates can be evaluated concurrently using an operation similar to a "merge-join" among the n lists associated with these predicates. Using this structure the correct answers are retrieved in *sorted trajectory identifier order*.

There are two cases where entries from the lists can be skipped (thus resulting in faster processing of the merge-join). First, whenever in a given predicate list r_m a trajectory identifier (say P_s) is encountered that is larger than all the trajectory identifiers currently at the top of the other lists, entries from these lists corresponding to trajectories P_r ($r < s$) can be effectively skipped. Essentially, predicate r_m cannot be satisfied by any of the trajectories with smaller identifiers (simply because such identifiers did not appear in the list of r_m). Second, the time-instant at which a trajectory satisfied a corresponding predicate, would assist in skipping list entries whenever the ordering of the query predicates was not obeyed.

Clearly not all possible (ad hoc) spatial range predicates can be indexed. Instead, a space partitioning grid can be used such that an arbitrary range query can be represented with reasonable accuracy as a list of cells. Predicate list are created for each cell, storing the trajectories that intersect with the cell.

For ease of exposition assume that a regular grid is used to partition the space. Each cell is represented with a unique cell identifier. An illustrative example

Algorithm 3 Range STP queries With Order

```
Input: Query $\mathcal{Q} = \{R(r_1), \ldots, R(r_n)\}$
Output: Trajectories satisfying the predicates
 1: for $i = 1$ to $n$ do
 2:     $L_i$ = combined list of cells intersected by $r_i$
 3: Candidate set $U \leftarrow \emptyset$
 4: for $i = 1$ to $n$ do
 5:     for $j = 1$ to $n$ do
 6:         if $L_1[i].id \notin L_j$ then break
 7:         else
 8:             Let $k$ be the first entry for $L_1[i].id$ in $L_j$
 9:             while $L_1[i].id = L_j[k].id \bigwedge L_1[i].t > L_j[k].t$
                    do $k+ = 1$
10:             if $L_1[i].id \neq L_j[k].id$ then break
11:     $U = U \cup L_1[i].id$
12: end for
13: Retrieve trajectories in $U$ and verify the results
```

is shown in Figure 5. Trajectory P_3 crossed cell B at time-instant 3. At time-instant 4 trajectory P_1 entered the cell. Trajectory P_2 followed at time-instant 9 and returned at time-instant 13. For simplicity, in this example we assumed that each trajectory remained in the cell for a single time-instant. Then the list for cell B becomes $\{(P_1, 4), (P_2, 9), (P_2, 13), (P_3, 3)\}$.

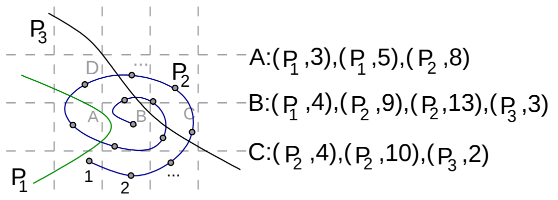

P3
P2
P1
D
...
A
B
C
1
2
...
A:(P1 ,3),(P1 ,5),(P2 ,8)
B:(P1 ,4),(P2 ,9),(P2,13),(P3 ,3)
C:(P2 ,4),(P2 ,10),(P3 ,2)

Figure 5: A uniform partitioning and the representation of cells as lists of sorted trajectory identifiers.

The proposed approach is shown in Algorithm 3. If a range predicate is contained within a cell, we overestimate the result by using the cell's list instead. If a range predicate is larger than the cell size, we approximate it by the smallest enclosing range of cells. In order to use the above join algorithm a sorted list needs to be materialized first for the combined cells, which requires time linear to the total number of entries in the lists since they are already sorted. With this approach a verification step is needed to remove false positives that are not actually contained in the original range predicate. A fine grid granularity will lead to a small number of such false positives. Using a grid resolves the overlapping issues of the traditional MBR indices while it also prunes many object trajectories from the search by preserving temporal ordering. Moreover, the elimination of trajectories that do not satisfy some of the predicates can happen without having to access the raw data from storage, contrary to the available alternatives for spatio-temporal index structures. Furthermore, it achieves substantial space savings since it maintains small representations of trajectories, approximated within the grid granularity. At the same time, it discretizes the space, keeping only a fixed number of lists.

Even though in the preceding analysis we use uniform grids for simplicity, in practice, we do not need to maintain a simple regular grid over the data. Alternatively, we may use any dynamic space partitioning structure like the adaptive grid files, kdb-trees, etc. [6]. This would guarantee that all the cells of the structure contain approximately the same number of data, and the corresponding lists have similar sizes.

3.2.2 Nearest Neighbor Predicate Evaluation

We now consider STP queries With Order that contain only nearest neighbor predicates. An object trajectory satisfies the query if it minimizes the sum of the distances from the query predicates and in the correct order. One straightforward approach is to use the incremental ranking nearest neighbor algorithm introduced for STP queries With Time with two needed modifications: (1) The pruning threshold Λ needs to be updated only if a candidate trajectory truly minimizes the distance in the correct order and not arbitrarily; and (2) the best first search queues should be populated even with entries that do not contain the predicates in the temporal dimension. However, this solution is expected to yield poor query performance since it does not inherently prune using predicate ordering but needs to postpone ordering verifications until a trajectory is loaded from storage. Also, the increased number of nodes that will be inserted in the queues will slow down execution, intensify the search cost, and increase the memory requirements. Moreover, similar with the case of range predicates, an approach that first uses a 2-dimensional projection to eliminate the temporal dimension will not improve performance. Finally, all discovered MBRs during evaluation need to be retained, since they might be part of the best overall answer given the ordering of the predicates, even if they lie farther than other MBRs from a specific query predicate. This means that trajectory lower-bounds cannot be incrementally improved during evaluation, thus, no trajectories can be pruned unless a truly small Λ is computed. Hence, this technique will need to load an excessive number of MBRs in order to prune all candidates and terminate the search. This is corroborated by our experimental evaluation.

On the other hand, the space partitioning index structure proposed in the previous section can be used to speed up the execution of these queries as well. Once more, we utilize the incremental ranking algorithm described above. However, instead of using the best first search strategy per query predicate, the algorithm iteratively en-queues and examines all the cells adjacent to the cell containing the query. For every query predicate q_m a sorted queue of satisfiability predicates $S_{q_m}(P, t_i)$ is maintained. This queue is populated with all the entries contained in adjacent cells. In each phase, the process increases the number of adjacent cells examined by moving one cell further away from the query in every direction (in the beginning it examines the cell containing the query, then the eight cells adjacent to the query, and so on). For all new entries $S_{q_m}(P, t_i)$ that are added in the queue at each step, the lower-bound distance $LB_D(q_i, P)$ is computed pessimistically; i.e., assuming that the actual trajectory would be as close as possible to the query.

Then, all the predicates are evaluated in order. For every new cell added in any query predicate queue, the algorithm joins it with the queues of all other predicates. Trajectories that visit the predicates in the correct order are loaded from storage as soon as they have appeared in all queues, and the exact distances are computed. The pruning threshold Λ is updated accordingly. The exact distance computation is postponed for trajectories that do not follow the right order. Next, the lower bound distance from the query needs to be computed for all candidate trajectories that appear in at least one of the queues. Assume that a trajectory entry $S_{q_m}(P, t)$ exists in the queue for q_m. The minimum partial distance of P from q_m is equal to $MinDist(q_m, C)$, where C is the cell that contains P at time-instant t. Assume that no entries for trajectory P are contained in the queue of predicate q_m. Then, P has not been discovered yet for q_m, thus it has to lie at least as far as the farthest explored point for q_m in every direction. This distance is equal to the minimum of the distances of q_m from the sides of the rectangle defined by the perimeter of the explored cells. Given the total lower bounding distance of each trajectory from the query and Λ, trajectories can be safely pruned in every step.

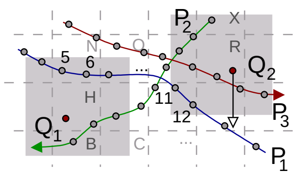


Figure 6: An example of the incremental nearest neighbor algorithm.

The actual algorithm is omitted since the modifications can be straightforwardly applied in Algorithm 1. The en-queue, join and merge procedure is shown in Algorithm 4. We further illustrate the details of the algorithm using an example. Figure 6 depicts an STP query with two NN predicates q_1, q_2 in that order, a 6×4 grid, and three trajectories. The grid cells have identifiers $A, B, \ldots, X, Y$ starting from the lower left corner (not all cell identifiers are depicted). In the first phase of processing, the combined list of q_1 consists only of the entries in cell H, which contains element $S_{q_1}(P_2, 8)$. Similarly, the list of q_2 contains entry $S_{q_2}(P_3, 7)$. The partial distance of P_2 from q_1 is set to 0, while the pessimistic distance from q_2 is equal to the minimum distance of q_2 from the sides

Algorithm 4 En-queue, Join and Merge Operation

```
Input: $l, Q, CL_n, \mathcal{S}, D_{q_1,\ldots,q_n}$
1: for $i = 1$ to $n$ do
2:     Locate next level of cells $l$
3:     for each new cell list $L$ do
4:         for $T \in L$ do
5:             S.insert($T$)
6:             Join with lists $CL_j, j \neq i$ to see if T satisfies the order (similarly to the range query algorithm), and if yes tag it in $S$ for subsequent retrieval
7:             Insert $T$ in list $CL_i$
8:     Update $D_{q_i}$
9: end for
```

of cell R. The total lower bound $LB_D(\mathcal{Q}, P_2)$ from the query is thus equal to $0 + MinDist(q_2, R)$ (similarly $LB_D(\mathcal{Q}, P_3) = 0 + MinDist(q_1, H)$). In the next phase the algorithm evaluates all cells adjacent to q_1 and q_2. Starting with q_1 the process merges the lists of cells $A, B, \ldots, N, O$ (the shaded region around q_1), with the list of q_1 (cell H). Before the merging operation, each cell is joined with q_2, and entries that follow the query ordering are loaded from storage for verification. For example, consider cell B being merged with query list q_1. The list of B contains element $S_B(P_2, 9)$. First, we join B with q_2 and deduce that there is no entry in q_2 with the same identifier yet, so no action needs to be taken. Likewise, when merging cell N, element $S_N(P_3, 2)$ is probed into list q_2, where it is already contained with a satisfiability predicate corresponding to time-instant 7. Thus, P_3 satisfies the order and, in addition, it has appeared in the lists of all predicates, so it is loaded from storage for an exact distance computation and Λ is updated accordingly.

Trajectories for which the computed lower bounds are larger than Λ can be safely pruned (none in this case yet). The rest of the cells are merged in the same way. The list for q_1 becomes $(P_1, 3-8), (P_2, 6-9), (P_3, 2-4)$. In the same phase, when merging cell X with query q_2, we join element $S_X(P_2, 1)$ with the list of q_1, where four entries for P_2 are already present. Since all entries are associated with time-instants larger than 1, we deduce that no sequence of satisfaction predicates follows the order of the query, so no action needs to be taken. After all cells around q_2 have been merged, the new partial distances of all trajectories can be computed. For trajectory P_3 the exact distance is known. For the rest of the trajectories, since they did not satisfy the order, the partial distance is equal to the minimum distance of the query point from the sides of the rectangle corresponding to the perimeter of the newly added cells (the far side of cell K in this case as shown with the arrow). The intuition is that since the points found so far do not satisfy the ordering yet, any point that does must be at least as far as this distance from the query After computing the new lower bounds it can be deduced that trajectories P_1 and P_2 can be pruned given the current Λ, and the algorithm stops.

Another interesting problem is computing the minimum ordered distance of a trajectory from a given query efficiently. Assuming discrete time or approximate trajectory representations, one way is to exhaustively check all possible sequences of locations. A better way is to use the Threshold Algorithm (TA) [4]. Given a query $\mathcal{Q}$ and a trajectory P, the n-length sequence of locations that minimizes the distance from all predicates and also satisfies the query order can be found by using the incremental top-k ranking concept of TA. Our algorithm will perform even better, since whenever the actual query to trajectory distance needs to be evaluated, the following information is already computed: (1) That the trajectory definitely satisfies the predicate order, and (2) that all the trajectory points that are candidates for minimizing the total distance are already available. Given the above and the expensive nature of this distance function, it is clear that our technique will offer substantial improvement over the exhaustive search or any other technique that needs to evaluate this distance from scratch.

3.2.3 Combinations

Using our proposed grid structure combinations of range and nearest neighbor STP queries can be answered as well. The same approach as with STP queries With Time can be utilized. The incremental algorithm can be modified to take into account the range predicates during the joining phase of the lists. More specifically, in the beginning, for every range predicate we create a combined list of the entries of the constituent cells which remain fixed throughout execution. Then, while evaluating the NN predicates, every time a new cell is added in the list of a specific predicate, we join the sorted list of the cell with all other lists of nearest neighbor and range predicates and instantly prune the entries that do not satisfy any of the range predicates with the correct order. The added benefit of the algorithm is that, depending on the selectivity of the range queries, it is expected that the candidate entries for inclusion in the lists will decrease substantially during evaluation.

4 Experimental Evaluation

We proceed with a comprehensive experimental evaluation of the proposed algorithms. We run various experiments with synthetic datasets to test the behavior of each technique under different settings. All experiments were run on an Intel Pentium-4 2.4 GHz processor running Linux, with 512 MBytes main memory. We generated synthetic datasets of moving object trajectories. The dataset represents the freeway network of Indiana and Illinois. The 2-dimensional spatial universe is 1,000 miles long in each direction and contains up to 500,000 objects. Object velocities follow a Normal distribution with mean 60 mph, and standard deviation 15 mph. We run simulations for 400 minutes (time-instants). To illustrate the generality of our framework, we index the data using two different indexing techniques, the R-tree and the MVR-tree.

4.1 STP Queries With Time

For this type of queries trajectories were split into multiple MBRs and then indexed using an R-tree and an MVR-tree. For our experiments we approximated the datasets using on average 20 MBRs per trajectory (for a total of approximately 6,000,000 MBRs) [9]. We set the page size for all indices to 4 KBytes, and used a 256 pages LRU buffer (e.g., in comparison to a total of 61,477 pages present in an R-tree indexing 300,000 trajectories). For clarity we present the results only for the R-tree in the graphs, but very similar conclusions were drawn for the MVR-tree as well.

We use four query sets: RandomPattern, RelevantPattern, CombinedPattern and IntervalPattern. All query sets contain sets of NN predicates only, except from the CombinedPattern that contains combinations of NN and range predicates. All sets have 100 queries and each query consists of a number of predicates at increasing time instants. For all queries we provide the top-20 results. Queries in the RandomPattern set have predicates that lie on consecutive nodes of the network, reachable from each other given the maximum velocity of objects and the temporal constraints of the query. The RelevantPattern set contains queries that are formed from partial segments of object trajectories already contained in the dataset, slightly skewed in time and space from the original, with random location/time-instant tuples dropped in-between. With the CombinedPattern set, we evaluate the algorithms using query sets that are a mix of NN and range predicates. In particular, we randomly replace a number of NN predicates with range searches centered at the same positions. Finally, the IntervalPattern set is generated similarly to the RelevantPattern, and includes only NN predicates but with time-interval temporal constraints.

Comparison vs. Linear Scan. Since our query sets contain NN predicates, we can only compare against a linear trajectory scan (since the traditional search algorithms for spatial index structures cannot be applied). We compared the linear scan against our techniques for all subsequent experiments. Although, since the total I/O of this straightforward approach was, on average, three orders of magnitude higher than the other algorithms, we exclude it from the graphs in order to preserve detail. Hence in the rest we concentrate on the relative performance of the lazy and eager algorithms.

Performance vs. Number of Predicates. We test our algorithms for queries with increasing number of predicates. The results are shown in Figure 7 for the RandomPatterns and Figure 8 for the RelevantPatterns, where we plot the index I/O, the trajectory I/O and the total I/O for each technique. Clearly, for RandomPatterns and numerous predicates the lazy algorithm deteriorates substantially. Many index nodes need to be accessed before a tight threshold can be computed, hence, the NN searches become very expensive. On the contrary, for RelevantPatterns all algorithms remain practically unaffected, since a nearest neighbor is discovered fast and the searches terminate faster.

In terms of average trajectory loads (LR-T and ER-T), for RandomPatterns all algorithms need to load an increasing number of trajectories, as the number of predicates increases. In order to find the top-20 results the probability that many trajectories satisfy the query decreases proportionately and the searches start expanding correspondingly, thus many trajectories are being discovered. The eager algorithm loads a larger number of trajectories in all cases. Assuming one random I/O per trajectory access, the eager achieves the best performance overall, balancing the trajectory accesses and the index overhead, especially for large numbers of predicates, where the searches become expensive. For RelevantPatterns all algorithms load the same number of candidate trajectories. Since there exist many trajectories similar to the given query, the top-20 candidates are found very fast, and only few extra trajectories cannot be pruned using the lower bounds.

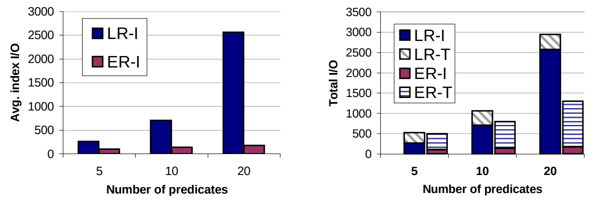


Figure 7: RandomPattern: Number of predicates.

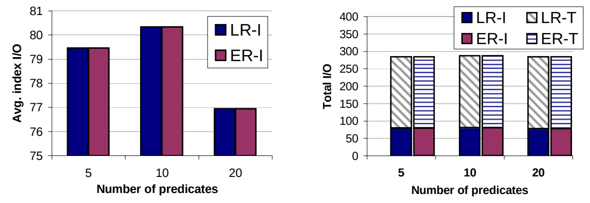


Figure 8: RelevantPattern: Number of predicates.

Performance vs. Dataset Size. Next, we run scale-up experiments with increasing numbers of trajectories. Figures 9 and 10 summarize the results. We used both RandomPattern and RelevantPattern query sets with 10 predicates per query. As expected, we observe that for all index structures the average number of node accesses per query increases, since the total number of nodes in the structures increases as well, and the trees become deeper. The total number of trajectory accesses remains stable for RelevantPatterns, since the top-20 results per query are discovered very fast, irrespective of the total number of objects in the dataset. On the contrary, for RandomPatterns there is a linear increase in the number of trajectory access, since with a larger dataset size and looser thresholds, more trajectories are discovered during the NN searches. In terms of total I/O the eager algorithm is once more the best choice overall, providing better results for the RandomPatterns.

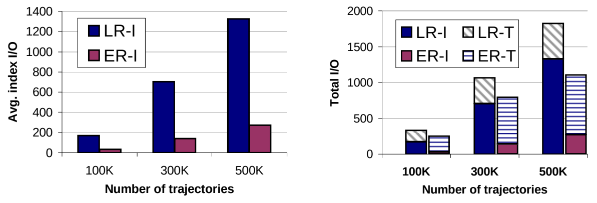


Figure 9: RandomPattern: Number of trajectories.

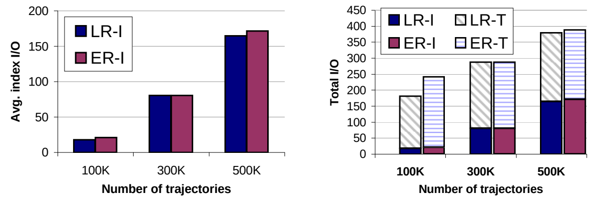


Figure 10: RelevantPattern: Number of trajectories.

Query Runtime vs. Number of Predicates. Figure 11 reports the average computational cost (as the total wall-clock time, averaged over a number of executions) for top-20 queries of various numbers of predicates. Doubling the number of predicates doubles the amount of time required to answer the RandomPattern queries, as expected due to lack of good candidates. In contrast, the number of predicates does not affect RelevantPattern queries, which are easier to answer.

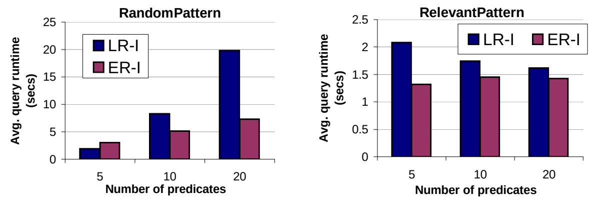


Figure 11: Query runtime.

Performance vs. Buffer Size. Figure 12 plots the effect of the buffer size on query performance. We use 10 predicates per query and top-20 nearest neighbors. Since we run multiple NN searches concurrently we expect some searches to access the same nodes at approximately the same time. Larger buffer sizes help alleviate the problem of loading these nodes multiple times. We can see from the graph that an 1 Mbyte buffer is adequate for eliminating most superfluous node accesses. As expected, the lazy algorithm benefits the most, since it is the one with the heaviest index access cost.

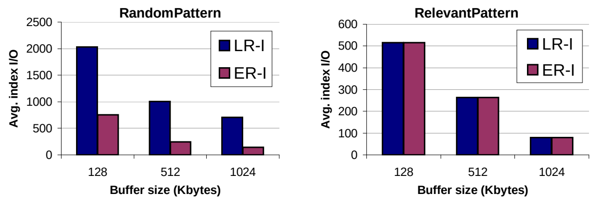


Figure 12: Buffer size.

Interval Queries. We performed experiments using the IntervalPattern queries. We used a fixed time-interval Δt equal to ten time-instants for all query predicates. For comparison purposes Figure 13 shows the results of the eager algorithm for queries with time-instant predicates and interval predicates (IER) (the same set of queries was used, where each time-interval constraint was replaced with a time-instant). As expected, interval queries have the larger index access cost in comparison with time-instant queries, due to the larger number of nodes that intersect with the temporal constraints of the predicates. A surprising result is that interval queries load much fewer candidate trajectories from storage. Intuitively, the time-interval algorithm locates very similar trajectories to the query a lot faster, by performing a more robust searching on the time dimension, thus a very tight threshold is computed earlier during the evaluation process.

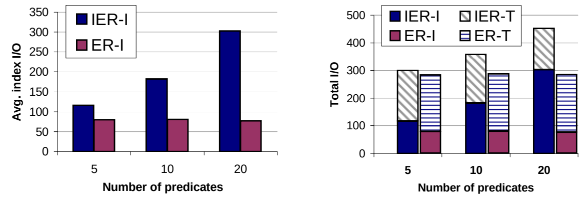


Figure 13: IntervalPattern: Number of predicates.

Combined Queries. Finally, we tested our algorithms for CombinedPattern queries that contain both NN and range predicates. Figure 14 shows the effect of increasing the range query selectivity to the index and trajectory access cost, when using top-20 queries. Line ER-I corresponds to the cost of answering a query that contains only NN predicates. RER-1 and RER-2 correspond to answering the same query, where one NN predicate has been replaced with a range predicate. Clearly, for very small range queries that do not contain the actual nearest neighbors of the original query, the remaining NN searches need to expand the search substantially in order to retrieve their closest candidates that also satisfy the range predicate. Here, the cost for range query side 2.5 has being clipped in order to preserve the detail in the graph. The actual cost is 3,000 index I/Os on average. The problem is exacerbated since a large number of nearest neighbors (twenty in this case) is requested. As the range query increases in size, since the original twenty nearest neighbor trajectories are all contained in the range, the rest of the searches discover them very fast. In this case, the index cost for RER-1 increases due to the extra cost for answering the range query in the beginning. On the contrary, the trajectory access cost decreases since many new trajectories discovered during the NN searches can be pruned, given the answer of the range query. Alternative RER-2 gave similar results to the original query, as expected, since the range predicate has no effect on the execution of the ranking algorithm, pruning entries only after they have been loaded from storage.

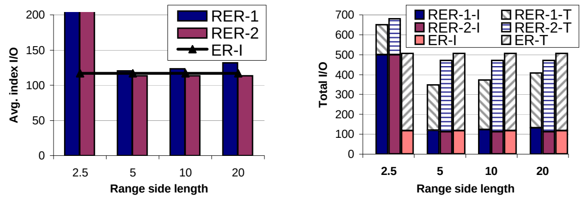


Figure 14: CombinedPattern: Range selectivity.

Conclusion. In general our algorithms yielded very efficient pruning for STP queries, especially when considering that the only available alternative is the linear scan. The eager strategy performed the best overall, in all cases, balancing the trajectory access cost and the index overhead.

4.2 STP Queries With Order

For this type of queries we selected a 100×100 uniform grid and created the corresponding sorted lists per cell, as described in Section 3.2. We compare our grid based index against the straightforward approach of using a 2-dimensional R-tree.

4.2.1 Range Queries

For this experiment we created a query set that contains only range predicates. The queries were created similarly to the RelevantPatterns set, by replacing the NN predicates with range predicates. For the R-tree index, assuming that the selectivity of the predicates is not known, all predicates were evaluated in sequence and the intersection of the individual result sets was returned.

Performance vs. Number of Predicates. First, we evaluate the techniques using varying numbers of range predicates, given a fixed range query side length equal to 15 miles (with the universe size being $1,000 \times 1,000$ miles). The results are shown in Figure 15. The *CellList* in the graph corresponds to our proposed technique, and the *R-tree* to the 2-dimensional R-tree approach. The CellList has significantly reduced index I/O, due to more efficient pruning during the list join step. Moreover, another benefit of our approach is the reduced number of raw trajectory data that need to be retrieved. Our approach yields from two up to three orders of magnitude less trajectory accesses than the R-tree (the graph shows the results in a logarithmic scale). This is expected, since the R-tree has no way of pruning trajectories due to predicate ordering constraints; it does not store such information in the index.

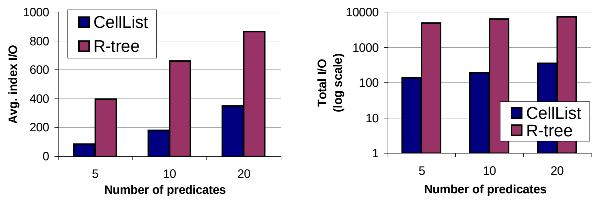


Figure 15: RelevantPattern: Number of predicates.

Performance vs. Range Selectivity. The next experiment evaluates the performance with respect to increasing range predicate selectivity. Figure 16 presents the results. As expected, when the size of the range increases the index I/O rises proportionately, since a larger number of candidates are retrieved. Similar observations hold for trajectory accesses as well. Clearly, the CellList can answer STP queries With Order, by far more efficiently than the R-tree, and with a structure that is smaller in size

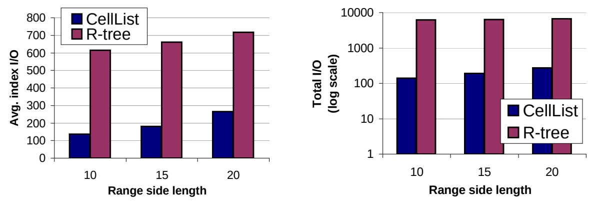
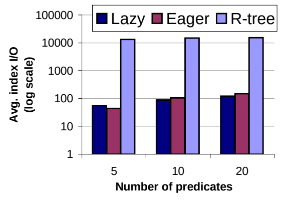


Figure 16: RelevantPattern: Range selectivity .

4.2.2 Nearest Neighbor Queries

For these queries we used the *RelevantPatterns* to evaluate the CellList both using the lazy and the eager strategies. We also run a simple eager algorithm on a 2-dimensional R-tree, with the ranking algorithm described in Section 3.2.2. We compared each technique for an increasing number of predicates. The results are reported in Figure 17. The lazy strategy worked the best in this case, requiring a small number of index accesses as well as a very small number of trajectory accesses. Naturally, delaying trajectory retrieval in this case is beneficial due to the double pruning based both on lower-bounds and the predicate ordering. It is evident that, contrariwise, the eager strategy has three orders of magnitude more trajectory accesses, since it unnecessarily retrieves trajectories that do not satisfy the order. The R-tree approach, as predicted in Section 3.2.2, deteriorated to almost a sequential scan of both the index structure and the trajectory storage, especially for larger numbers of predicates.

For STP Queries with Order, the CellList is a robust solution with excellent performance, both for range and nearest neighbor predicates. In particular, the lazy strategy for these queries works the best, in contrast to STP queries With Time, where the eager strategy was the best alternative.



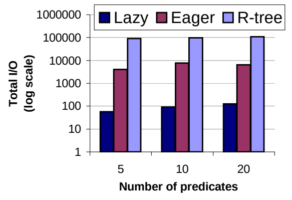


Figure 17: RelevantPattern: Number of predicates.

5 Related Work

Modelling and expressing complex spatio-temporal queries has been investigated in [7]. In this paper we use a very general expression mechanism, that can utilize any previous definition of spatio-temporal predicates. Applying the incremental ranking algorithms for answering STP queries with NN predicates has been inspired by the Threshold Algorithm (TA) proposed in [4]. The TA algorithm is used for ranking objects with multiple features given any monotonic preference function. Here, we show how this algorithm can be applied in the case of spatio-temporal data using index structures instead of materialized sorted lists. Traditional nearest neighbor queries have been addressed in previous work including [1, 5, 10, 20]. None of these approaches has focused on efficient evaluation of combinations of NN predicates (with the exception of GNN queries [16]). To the best of our knowledge, no work has appeared for answering combinations of spatial predicates with order and without temporal constraints.

Related to the STP range queries with order (after the grid reduction) is the work in [12] which considers non-contiguous pattern queries on string sequences. However, our structures are more suitable for spatio-temporal data and can answer more general queries. Mouza and Rigaux [3] introduced mobility pattern queries, where patterns are expressed using regular expressions. This work is a special case of STP queries, concentrating on STP queries With Order and Range predicates. The techniques introduced therein cannot handle distance based predicates, neither explicit temporal constraints. Furthermore, the patterns are limited to predefined ranges, which have been determined in advance by partitioning the space into areas of interest. In contrast, here we present more general techniques that can handle arbitrary query ranges, distance based predicates, and temporal constraints.

Any trajectory indexing scheme that can answer nearest neighbor, range queries, and other spatial predicates can be used with the STP query algorithm described here [22, 15, 2, 13, 18, 21]. were there is an inherent uncertainty in moving object trajectories [23].

6 Conclusions

We introduced and formalized a novel type of spatio-temporal pattern query for trajectories. We designed specialized spatio-temporal index structures and algorithms to effectively reduce the computation and I/O operations per query, when compared with existing approaches. Finally, we presented a thorough experimental evaluation. For future work we plan to extend the techniques for answering pattern queries that impose constraints between groups of predicates, and for including relative temporal constraints.

References

[1] R. Benetis, C. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *IDEAS*, pages 44–53, 2002.

[2] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with seti. In *CIDR*, 2003.

[3] C. du Mouza and P. Rigaux. Mobility patterns. In *STDBM*, pages 1–8, 2004.

[4] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[5] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A. El Abbadi. Constrained nearest neighbor queries. In *SSTD*, pages 257–278, 2001.

[6] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[7] R. H. Güting, M. H. Bhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *TODS*, 25(1):1–42, 2000.

[8] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[9] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatiotemporal objects. In *EDBT*, pages 251–268, 2002.

[10] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.

[11] N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, and D. W. Cheung. Mining, indexing, and querying historical spatiotemporal data. In *SIGKDD*, pages 236–245, 2004.

[12] N. Mamoulis and M.L. Yiu. Non-contiguous sequence pattern queries. In *EDBT*, pages 783–800, 2004.

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

[14] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatiotemporal databases. In *SIGMOD*, 2004.

[15] M. Nascimento, J. Silva, and Y. Theodoridis. Evaluation of access structures for discretely moving points. *Spatio-Temporal Database Management*, pages 171–188, 1999.

[16] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. In *ICDE*, pages 301–312, 2004.

[17] W.-C. Peng and M.-S. Chen. Developing data allocation schemes by incremental mining of user moving patterns in a mobile computing system. *TKDE*, 15(1):70–85, 2003.

[18] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, pages 395–406, 2000.

[19] K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying mobile objects in spatio-temporal databases. In *SSTD*, pages 59–78, 2001.

[20] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.

[21] Y. Tao and D. Papadias. MV3R-Tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB*, pages 431–440, 2001.

[22] Y. Theodoridis, T. Sellis, A. Papadopoulos, and Y. Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. In *SSDBM*, pages 123–132, 1998.

[23] G. Trajcevski, O. Wolfson, K. Hinrichs, and S. Chamberlain. Managing uncertainty in moving objects databases. *TODS*, 29(3):463–507, 2004.