Querying Trajectories Using Flexible Patterns

Marcos R. Vieira
University of California
Riverside, CA 92507, USA
mvieira@cs.ucr.edu

Petko Bakalov
ESRI
Redlands, CA 92373, USA
pbakalov@esri.com

Vassilis J. Tsotras
University of California
Riverside, CA 92507, USA
tsotras@cs.ucr.edu

ABSTRACT

The wide adaptation of GPS and cellular technologies has created many applications that collect and maintain large repositories of data in the form of trajectories. Previous work on querying/analyzing trajectorial data typically falls into methods that either address spatial range and *NN* queries, or, similarity based queries. Nevertheless, trajectories are complex objects whose behavior over time and space can be better captured as a sequence of interesting events. We thus facilitate the use of motion "pattern" queries which allow the user to select trajectories based on specific motion patterns. Such patterns are described as regular expressions over a spatial alphabet that can be implicitly or explicitly anchored to the time domain. Moreover, we are interested in "flexible" patterns that allow the user to include "variables" in the query pattern and thus greatly increase its expressive power. In this paper we introduce a framework for efficient processing of flexible pattern queries. The framework includes an underlying indexing structure and algorithms for query processing using different evaluation strategies. An extensive performance evaluation of this framework shows significant performance improvement when compared to existing solutions.

1. INTRODUCTION

The wide availability of location and mobile technologies (cheap GPS devices, ubiquitous cellular networks) as well as the improved location accuracy (*A-GPS*) has enabled many applications that generate and maintain data in the form of *trajectories*. Examples include *AccuTracking*, *tracNET24*, Path Intelligence's *FootPath*, InSTEDD's *GeoChat*, among many others. Each trajectory has a unique identifier and consists of location data gathered for a specific moving object over an ordered sequence of time instants. Given the high data volume, more efficient techniques for query evaluation over trajectory data are needed.

Previous work on querying trajectories can be divided in two categories, **(a)** querying the future movements of moving objects based on their current positions (e.g. [12][17]), and, **(b)** querying trajectory archives, which is also the focus of this paper. Recent research efforts on querying trajectory archives has concentrated on **(i)** traditional spatiotemporal queries, such as range searches and nearest neighbors (e.g. [10][18]) (e.g. finding all trajectories that passed by *downtown LA* at *10:30am*), or **(ii)** similarity/clustering based tasks (e.g. [13][14]), such as extracting similar movement patterns and periodicities from a trajectory archive (e.g. finding all trajectories in the archive that are similar to a given query trajectory according to some similarity measure).

However, given the nature of trajectories as typically long sequences of events, a single range predicate may provide too many results (many trajectories passed through *downtown LA*) while a similarity-based query may be too restrictive (not many trajectories match the full extent or large part of the query trajectory). We thus advocate a different approach, i.e., using motion "pattern" queries. A motion pattern query specifies a combination of predicates that can thus capture only the parts of the trajectories that are of interest to the user. For example: "find all trajectories that first went by *downtown LA*, later by *West Hollywood* and ended up in *Beverly Hills*". This query simply provides a collection of range predicates that have to be satisfied in the specified order. One can also add Nearest-Neighbor (*NN*) conditions as well (in the above query: "... and they were closest to *the LAX airport*") as well as explicit time constraints: "ended up in *Beverly Hills* at 10am". Conceptually, motion pattern queries cover the query choices between the above two extremes (single predicates and similarity queries).

In this paper we introduce a general and powerful framework that describes pattern queries as regular expressions over a finite spatial alphabet. Each letter in the spatial alphabet corresponds to a non-overlapping region; their union covers the whole space where the trajectories lie. We note that there are various advantages from these choices. **(1)** The use of non-overlapping regions is natural: trajectories correspond to real entities and hence a trajectory can be in a single region at a given time. **(2)** Raw trajectory data typically come from sensors, GPS and/or RFID readers, etc. and provide extra detail that becomes cumbersome to query. Instead, the regions offer a more user-friendly way to express queries since the user is more familiar with the spatial regions [7] (for example, Downtown, LAX, etc.) **(3)** The use

EDBT 2010, March 22–26, 2010, Lausanne, Switzerland.

of spatial regions allows high-level summarization/filtering of the trajectories. The region description of a trajectory is much smaller, leading to faster query processing (while the raw data is still kept if more detail is needed). **(4)** This enables easy and effective indexing; it further enables the use of alternative evaluation algorithms (joins among inverted indexes, pattern matching, NFAs). **(5)** The region alphabet facilitates querying by regular expressions as a query language: the user can now describe complex queries over paths using this fixed alphabet.

This work is part of a larger prototype we are building for querying trajectories using regions. This system uses a hierarchical region alphabet, where the user has the ability to define queries with finer alphabet granularity (*zoom in*) for the portions of greater interest and higher granularity (*zoom out*) elsewhere. Through a GUI the user specifies a pattern query by selecting regions (using various levels of the hierarchy); this query is automatically translated into a regular expression over the finest region granularity and then executed. This paper describes the query engine of our prototype, i.e., how queries are executed at the finer granularity. Hence in the rest we assume that all query regions are at the same (finer) granularity. This finer granularity is chosen by and depends on the application needs.

Our framework further allows the use of *variables* within the pattern query. We term these *variable*-enabled queries as "flexible" patterns as they lead to a very powerful way to query the trajectory archive. Moreover, in our framework, the *fixed* and/or *variable* spatial predicates can express explicit temporal constraints ("between 10 a.m. and 11 a.m.") and/or implicit temporal ordering between them ("anytime later"). Queries can also include "numerical" conditions (nearest neighbors and their variants) over the duration of the trajectory. Using this general and powerful querying framework the user can "focus" only on the portions/events in a trajectory's lifetime that are of interest.

Novel methods are needed to efficiently process such complex queries over large trajectory repositories. We propose two query evaluation algorithms which first concentrate on trajectories that "satisfy" the fixed predicates specified in the query. As such, they prune effectively large portions of the repository that cannot lead to query answers. The first proposed algorithm uses the merge-join paradigm over the lists of trajectories associated with the query predicates. The second algorithm is based on a dynamic programming technique that finds subsequence matches between the trajectory representations and the query pattern. Efficient techniques are then used for the evaluation of the remaining *variable* predicates.

We note that patterns as effective ways to query have been examined in the past. [19][21] examine patterns over time series while [2] over event streams. Trajectories differ since they have both spatial and temporal behavior. In spatiotemporal databases patterns have been examined in [5][6][9][20]; as detailed in the related work section, these approaches either concentrate on language/modeling related issues, provide less query support (e.g. no temporal and/or numerical constraints) and have less efficient/general evaluation methods.

To summarize, the contributions of this paper are the following: **(1)** we define a simple yet powerful framework using a query language based on regular expressions; **(2)** we allow patterns to contain variables over the query space regions; **(3)** using lightweight index structures that can be easily implemented in most commercial DBMS nowadays, we propose two efficient evaluation algorithms; **(4)** finally, we present an extensive experimental evaluation of the proposed techniques against two other methods that we *extended* and implemented in our framework: a NFA-based method [2] and a *KMP*-based approach [5]. It should be noted that none of the (original) previous approaches can evaluate our proposed pattern query language. The experimental results reveal that the proposed evaluation framework achieves always better query performance over modified existing solutions, making our framework a very robust approach for querying and analyzing very large trajectory repositories.

The remainder of the paper is organized as follows: Section 2 discusses the related work; Section 3 provides the basic definitions and formal description of the spatiotemporal query language; The proposed framework is described in details in Section 4 and its experimental evaluation appears in Section 5; Section 6 concludes the paper.

2. RELATED WORK

Single predicate queries (Range and *NN* queries) for trajectory data have been well studied in the past (e.g. [18][23]). To make the evaluation process more efficient, the query predicates are typically evaluated utilizing hierarchical spatiotemporal indexing structures [10]. Most structures use the concept of Minimum Bounding Regions (MBR) to approximate the trajectories, which are then indexed using traditional spatial access methods, like the MVR-tree [22]. These solutions, however, are focused only on single predicate queries. None of them can be used for efficient evaluation of flexible pattern queries with multiple predicates. Moreover, our work is different than (and orthogonal to) approaches like [15], that can handle many single but independent predicates. In our case, all predicates appear in the same query and should all be satisfied by each trajectory in the result set. Similarity search among trajectories has been also been well studied. Work in this area focuses on the use of different distance metrics to measure the similarity between trajectories. Examples include [3][4].

Pattern queries have been used in the past for querying time-series using SQL-like query language [19][21], or event streams using a NFA-based evaluation method [2]; however, the environment in these works is different than the trajectories considered here. For moving object data, patterns have been examined in the context of query language and modeling issues [6][16][20] as well as query evaluation algorithms [9][5]. In [9] we examined incremental ranking algorithms in the case of simple spatiotemporal pattern queries. Those queries consist of range and *NN* predicates specified using only *fixed* regions. Our work differs in that we provide a more general and powerful query framework where queries can involve both fixed and *variable* regions as well as regular expression structures (repetitions, negations, optional structures, etc) and explicit ordering of the predicates along the temporal axis.

In [5], a *KMP*-based algorithm [11] is used to process patterns. This work, however, focuses only on range spatial predicates and cannot handle *explicit* and *implicit* temporal ordering of the predicates. Furthermore, this approach on evaluating patterns is effectively reduced to a sequential scanning over the list of trajectories stored in the repository: each trajectory is checked individually, which becomes prohibitive for large trajectory archives.

3. THE QUERY LANGUAGE

We assume that a trajectory T_{id} of a moving object is stored as a sequence of w pairs $\{(ls_1, ts_1), \ldots (ls_w, ts_w)\}$, where ts_i is a timestamp and ls_i is the object location recorded at ts_i ($ls_i \in \mathbb{R}^d$, $ts_i \in \mathbb{N}$, $ts_{i-1} < ts_i$, and $0 < i \leq w$). Such raw data is collected from the application and stored in the repository. Typically, monitored objects report their position to the data collection device using data packets containing their identifier id (T_{id}), current location ls_i and timestamp ts_i. Depending on the application, objects may report continuously or simply when they change their location. We further assume that the spatial domain is partitioned to a fixed set Σ of non-overlapping regions. Regions correspond to areas of interest (e.g. *school districts*, *airports*, *city malls*, etc.) and form the alphabet used in our query pattern specification. In the following we use capital letters to represent the region alphabet, $\Sigma = \{A, B, C, ...\}$.

A general pattern query $\mathcal{Q} = (\mathcal{S}\ [\bigcup \mathcal{D}])$ consists of a sequential pattern $\mathcal{S}$ and (possibly) a set of constraints $\mathcal{D}$. Here $\mathcal{S}$ corresponds to a sequence of spatial predicates, specified using regions from Σ, while $\mathcal{D}$ represents a collection of distance functions (e.g. *NN* and their variations) that may contain regions defined in $\mathcal{S}$. A trajectory matches the pattern query $\mathcal{Q}$ if it satisfies both $\mathcal{S}$ and $\mathcal{D}$. We first describe how a pattern $\mathcal{S}$ is formed and then elaborate on the distance constraints $\mathcal{D}$. In particular, a pattern $\mathcal{S}$ is expressed as a path expression of an arbitrary number of spatiotemporal predicates P:

$$\mathcal{S} \rightarrow \ \mathcal{S}.\mathcal{S} \mid P \mid !P \mid P^{\#} \mid ?^{+} \mid ?^{*}$$

here "!" defines the **negation** operator, "#" the **optional** modifier, "+" the **one or more repetition** modifier, "*" the **zero or more repetition** modifier, and "?" the **wild-card**. The sequence of predicates in $\mathcal{S}$ is defined recursively by $\mathcal{S}.\mathcal{S}$ where the sequencer "." appears between every spatiotemporal predicate P in $\mathcal{S}$.

Each spatiotemporal predicate $P_i \in \mathcal{S}$ is defined by a triplet $P_i = \langle op_i, \mathcal{R}_i, [int_i] \rangle$. Here $\mathcal{R}_i$ corresponds to a predefined spatial region or a *variable*, i.e., $\mathcal{R}_i \in \Sigma \cup \Gamma$ (where Γ is the set of variables, to be discussed later). The operator op_i describes the topological relationship that a trajectory and the spatial region must satisfy over the (optional) time interval int_i.

In particular, we use the topological relationships described in [6]; examples of such operators are the relations *Equal*, *Inside*, *Touch*, *Meet*, among others. Given a trajectory T_j and a region $\mathcal{R}_i$, the operator op_i returns a boolean value $\mathbb{B} \equiv \{true, false\}$ whether the trajectory T_j and the region $\mathcal{R}_i$ satisfy the topological relationship op_i (e.g., an *Inside* operator will be *true* if the trajectory was sometime inside region $\mathcal{R}_i$ during time interval int_i). For simplicity in the following we assume that the spatial operator is set to *Inside* and it is thus omitted from the query examples.

Within the pattern $\mathcal{S}$, the wild-card "?" is used to specify "don't care" parts in a trajectory's lifetime and can be of two types: **(i)** "$?^{+}$": one or more occurrences of any region predicate (e.g. $P_i.?^{+}.P_{i+1}$ implies that the predicate P_{i+1} is satisfied after predicate P_i with one or more regions visited between them); or, **(ii)** "$?^{*}$": zero or more occurrences of any region visit (e.g. $P_i.?^{*}.P_{i+1}$ which implies that the predicate P_{i+1} can be satisfied any time after predicate P_i).

A predefined region (i.e., $\mathcal{R}_i \in \Sigma$) is explicitly specified by the user in the query predicate (e.g. "the convention center"). In contrary, a *variable* denotes an arbitrary region and it is denoted by a lowercase letter preceded by the "@" symbol (e.g. "@x"). A variable region is defined using symbols in Γ, where $\Gamma = \{@a, @b, @c, ...\}$. Unless otherwise specified, a *variable* takes a single value (instance) from Σ (e.g. @a=C); however, in general, one can also specify the possible values of a *variable* as a subset of Σ (e.g., "any city district with museums"). Conceptually, *variables* work as placeholders for explicit spatial regions and can become instantiated (bound to a specific region) during the query evaluation in a process similar to unification in logical programming.

Moreover, the same *variable* "@x" can appear in several different predicates of pattern $\mathcal{S}$, referencing to the same region everywhere it occurs. This is useful for specifying complex queries that involve revisiting the same region many times. For example, a query like "$@x.?^{*}.B.@x$" finds trajectories that started from some region (denoted by variable "@x") , then at some point passed by region B and immediately after they visited the same region they started from. Note that for our purposes, wild-card "?" is also considered a variable; however it refers to any region (and not necessarily the same region if it occurs multiple times within a pattern).

Finally, a predicate P_i may include an explicit temporal constraint int_i in the form of an interval, which implies that the spatial relationship op_i between a trajectory and region $\mathcal{R}_i$ should be satisfied in the specified time interval int_i (e.g. "passed by area B between 10am and 11am"). If the temporal constraint is missing, we assume that the spatial relationship can be satisfied any time in the duration of a trajectory. For simplicity we assume that if two predicates P_i, P_j occur within pattern $\mathcal{S}$ (where $i < j$) and have temporal constraints int_i, int_j, then these intervals do not overlap and int_i occurs before int_j on the time axis.

Spatiotemporal predicates however cannot answer queries with constraints (for example, "best-fit" type of queries – like NN and the related – that find trajectories which best match a specified pattern). This is because topological predicates are binary and thus cannot capture distance based properties of the trajectories. The optional $\mathcal{D}$ part of a general query $\mathcal{Q}$ is thus used to describe distance-based or other constraints among the *variables* used in the $\mathcal{S}$ part. A simple kind of constraint can involve comparisons among the used variables (e.g., @x!=@y). More interesting is the distance-based constraint which have the form $(AGGR(d_1, d_2, ...); \theta)$ and is described below.

For simplicity in the following we assume Euclidean distance (L_2) but other distances, like *Manhattan* (L_1), *Infinity* (L_∞), among others, can also be used. Consider for example a $\mathcal{Q}$ query whose pattern $\mathcal{S}$ contains three *variables* $@x$, $@y$, $@z$, i.e., $\mathcal{S} \equiv A.?^*.B.@x.@y.C.?^*.@z$. Among the trajectories that satisfy $\mathcal{S}$, the user may specify that in addition, the sum of the distance between regions $@x$ and $@y$ and the distance between $@z$ and a fixed region E is less than 100 feet. Hence $\mathcal{D}$ contains a collection of distance terms $d_1, d_2, ...$, where term d_i represents the distance between two *variable* regions or between a *variable* region and a fixed one. In our example there are two distance terms: $d_1 = d(@x, @y)$ and $d_2 = d(@z, E)$.

Distance terms need to be aggregated into a single numerical value using an aggregation function (depicted as $AGGR$ in the formal definition of $\mathcal{D}$). In the previous example $AGGR = SUM$, but other aggregators like AVG, $SDEV$, MIN, MAX, etc., can also be used. The aggregated numerical score for each trajectory still needs to be mapped to a binary value so as to determine if the trajectory satisfies $\mathcal{D}$. This is done by the θ operator defined in $\mathcal{D}$. This operator can be a simple check function (using $=, \leq$ and others). In our example θ corresponds to "$<$ 100 feet" and returns true for all trajectories whose aggregate distance is less than 100 feet. It is also possible to use other θ operators, e.g. *MIN*, *MAX*, *Top-k*, etc. In the previous example, if the θ operator is changed to *Top-k*, the query will return true only for the trajectories with the *Top-k* aggregated distances. For simplicity of the description, in the remainder of this paper we use $AGGR = SUM$ and $\theta = MIN$ (which corresponds to a *NN* query).

The use of *variables* in describing both the topological predicates and the numerical conditions provides a very powerful language to query trajectories. To describe a query, the user can use fixed regions for the portions of the trajectory where the behavior should satisfy known (strict) requirements, and *variables* for portions where the exact behavior is not known (but can be described by a sequence of *variables* and the constraints between them). The ability to use the same *variable* many times in the query allows for revisiting areas, while the ability to refer to these *variables* in the distance functions allows for easy description of *NN* and related queries. It is exactly this "flexibility" allowed by the use of *variables* in selecting trajectories that led to the term "flexible pattern queries".

4. QUERY EVALUATION FRAMEWORK

To simplify the presentation we first start with the evaluation of the spatial predicates for a pattern $\mathcal{S}$. Later we extend the discussion to cover queries that in addition contain distance constraints $\mathcal{D}$. Finally we present the incorporation of time constraints inside the query $\mathcal{Q}$.

For simplicity we assume that the space is partitioned into 2-dimensional non-overlapping regions (Figure 1). To efficiently evaluate flexible pattern queries we will facilitate two lightweight index structures in the form of ordered lists, that are stored in addition to the raw trajectory data. There is one *region-list* per region and one *trajectory-list* per trajectory. The *region-list* $\mathcal{L}_A$ of a given region $\mathcal{A}$ acts as an inverted index that contains all trajectories that passed by

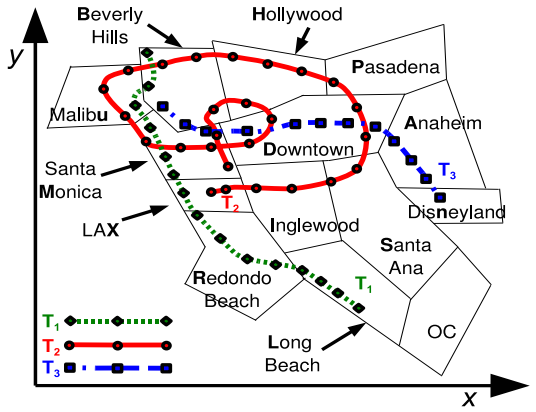

Figure 1: Region-based trajectory representation.

region $\mathcal{A}$. Each entry in $\mathcal{L}_A$ is a record that contains a trajectory identifier T_{id}, the time interval (*ts-entry*:*ts-exit*) during which the moving object was inside $\mathcal{A}$, and a pointer to the *trajectory-list* of T_{id}. If a trajectory visits a given region $\mathcal{A}$ multiple times in different time intervals, we store a record for each visit. Records in a *region-list* are ordered first by the trajectory-id T_{id} and then by *ts-entry*. For example, in Figure 1 the *region-list* entry for the region D (***D**owntown*) is $\{T_2(7, 9); T_2(21, 23); T_3(5, 10); ...\}$.

In order to fast prune trajectories that do not satisfy the query $\mathcal{S}$, each trajectory is approximated by the sequence of regions it visited. A record in the *trajectory-list* of trajectory T_{id} contains the region and the time interval (*ts-entry*:*ts-exit*) during which this region was visited by T_i, ordered by *ts-entry*. In the above Figure 1 the *trajectory-list* entry for T_2 is $\{X(1, 3); I(3, 5); S(5, 7); D(7, 9); P(9, 10); H(10, 13); B(13, 15); U(15, 18); M(18, 21); D(21, 23); H(23, 24); B(24, 25); M(25, 27)\}$. Note that records from a *region-list* index point to the corresponding records in a *trajectory-list* index. For example, the record T_2(21,23) in the *region-list* $\mathcal{L}_D$ (***D**owntown*) contains a pointer to the page in the *trajectory-list* of T_2 that contains the corresponding record D(21,23).

The only requirement for the region partitioning is that regions should be non-overlapping. In practice, there may be a difference between the regions presented to the user and what lists are created. In such scenarios we use uniform grid and overestimate a region by approximating it with the smallest collection of grid cells which completely encloses it. False positives may be generated from regions that do not completely fit the set of covering grid cells, however, they can be removed with a verification step using the original trajectory data. Finding the best grid granularity can be done by an optimization process which combines the number of grid cells and the total overestimated area into a single objective function. Moreover, instead of a uniform grid, one could facilitate instead a dynamic space partitioning structure (e.g. adaptive grid files, kdb-trees, among many others) that assigns grid cells sizes according to the data density. Then, dense areas will have more, finer cells which in return allow for better approximation of the regions and thus fewer false positives are generated.

For evaluating pattern queries we propose two different strategies. The *Index Join Pattern* (*IJP*) is based on a merge join

operation performed over the *region-lists* corresponding to every fixed predicate in the query pattern $\mathcal{S}$. The *Dynamic Programming Pattern* (*DPP*) performs subsequence matching between the query pattern $\mathcal{S}$ and the trajectory approximations stored as the *trajectory-lists*. Both algorithms use the same two indexing structures for pruning purposes, but in different ways: *IJP* uses the *region-lists* for pruning and the *trajectory-lists* for the variable binding; *DPP* uses mainly the *trajectory-lists* for the subsequence matching and performs an intersection-based pruning on the *region-lists*. Which algorithm would behave better will thus depend on the pruning capabilities provided by its main index; this in turn depends on the trajectory archive and the query characteristics.

4.1 The Index-Join Pattern Algorithm (IJP)

4.1.1 Spatial Predicate Evaluation

We start with the case where the pattern $\mathcal{S}$ does not contain any explicit temporal constraints. In this scenario, the pattern specifies the order by which its predicates (whether fixed or variable) need to be satisfied. Assume $\mathcal{S}$ contains m predicates and let $\mathcal{S}_f$ denote the set of n fixed predicates, while $\mathcal{S}_v$ denotes the set of r variable predicates (m=n+r). The evaluation of $\mathcal{S}$ with the *IJP* Algorithm can be divided in two steps: **(i)** the algorithm evaluates the set $\mathcal{S}_f$ using the *region-list* index to fast prune trajectories that do not qualify for the answer; **(ii)** then the collection of *candidate* trajectories is further refined by evaluating the set of $\mathcal{S}_v$.

(i) Fixed predicate evaluation: All n fixed predicates in $\mathcal{S}_f$ can be evaluated *concurrently* using an operation similar to a "merge-join" among their *region-lists* $\mathcal{L}_i, i \in 1..n$. Records from these n lists are retrieved in sorted T_{id} order and then joined by their T_{id}'s. Records are pruned using the trajectory *ids* and the temporal intervals (*ts-entry*:*ts-exit*). In each list $\mathcal{L}_i$ we keep a pointer p_i that points to the record currently considered for the join. This pointer scans the list starting from the top.

If the same region appears more than once in the pattern $\mathcal{S}$, a separate pointer traversing that *region-list* is used for each region appearance in the pattern. For example, to process the pattern $?^*.M.D.M$ the *region-lists* of M and D are accessed using one pointer for *region-list* $\mathcal{L}_D$ (p_D) and two pointers for traversing *region-list* M (p_{M_1} and p_{M_2}). If a trajectory-id T_{id} appears in all of the n *region-lists* involved in the query pattern, and their corresponding time intervals in all n *region-lists* satisfy the ordering of the predicates in $\mathcal{S}$, this T_{id} is saved as a possible solution. The pseudo code is shown in Algorithm 1.

During the merge-join, there are cases where records from a *region-list* can be skipped, thus resulting in faster processing. For example, assume that predicate $P_i \in \mathcal{S}$ (corresponding to the *region-list* $\mathcal{L}_i$) is before predicate $P_j \in \mathcal{S}$ (corresponding to $\mathcal{L}_j$). Further assume that in list $\mathcal{L}_i$ the current record considered for the join has trajectory identifier T_r, while in list $\mathcal{L}_j$ the current record considered has trajectory identifier T_s. If $T_s < T_r$, processing in list $\mathcal{L}_j$ can skip all its records with $T_{id} < T_r$. That is, the pointer p_j in list $\mathcal{L}_j$ can advance to the first record with $T_{id} \geq T_r$. Essentially, predicate $\mathcal{P}_i$ cannot be satisfied by any of the trajectories in $\mathcal{L}_j$ with smaller T_{id} than T_r. Since records in a *region-list* are sorted by T_{id}, $\mathcal{L}_i$ does not contain trajectories with smaller identifiers than r.

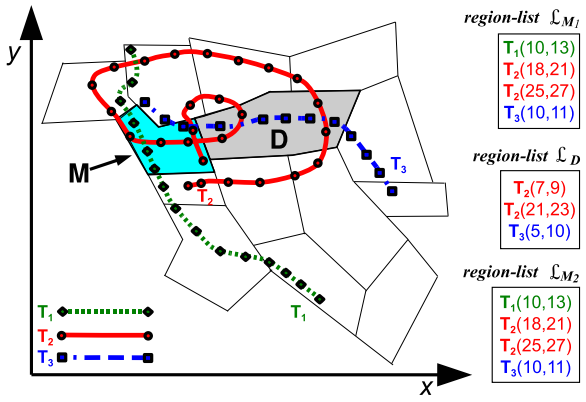

Figure 2: Trajectory examples T_1, T_2 and T_3.

Similarly, when a record from the same trajectory (e.g. T_s) is found in two *region-lists* (e.g. $\mathcal{L}_i, \mathcal{L}_j$), the algorithm checks whether the corresponding time intervals of the records match the order of predicates in the pattern $\mathcal{S}$. Hence a trajectory that satisfies $\mathcal{S}$ should visit the region of $\mathcal{L}_i$ before visiting the region of $\mathcal{L}_j$. If the record of T_s in $\mathcal{L}_i$ has *ts-entry* that falls after the corresponding *ts-entry* of T_s in list $\mathcal{L}_j$, this record can be skipped in $\mathcal{L}_i$, since it cannot satisfy the query. Since *region-lists* are stored in ordered way, advancing a *region-list* forward to a specific location stamp by T_{id} or by (T_{id}, *ts-entry*) can be easily implemented using an index B$^+$-tree on the (T_{id}, *ts-entry*) composite attribute.

Example: The first step of *IJP* algorithm is illustrated using the example in Figure 2. Assume the pattern $\mathcal{S}$ in the query $\mathcal{Q}$ contains three fixed (M, D, M) and three *variable* predicates $(?^+, @x, @x)$, as in:

$$\mathcal{S} = \{?^+.@x.?^*.M.?^*.D.?^*.@x.?^*.M\}$$

This pattern looks for trajectories that first visited an arbitrary region (denoted by $?^+$) one or more times, then visited some region denoted by *variable* $@x$, then (after visiting zero or more regions) it visited region M, then region D and then visited again the same region $@x$ before finally returning to M. The first step of the join algorithm uses the *region-list* for M and D ($\mathcal{L}_M$ and $\mathcal{L}_D$). For simplicity, instead of using two separate pointers in list $\mathcal{L}_M$, Figure 2 depicts two copies of list $\mathcal{L}_M$, namely $\mathcal{L}_{M_1}$ and $\mathcal{L}_{M_2}$. Conceptually, $\mathcal{L}_{M_1}$ represents the first occurrence of M in $\mathcal{S}$ (before D) and $\mathcal{L}_{M_2}$ the second occurrence of M (after D).

The algorithm starts from the first record in list $\mathcal{L}_{M_1}$, namely T_1(10,13). It then checks the first record in list $\mathcal{L}_D$, i.e., trajectory T_2. We can deduce immediately that T_1 is not a candidate trajectory, since it does not appear in the list of $\mathcal{L}_D$, so we can skip T_1 from the $\mathcal{L}_{M_1}$ list and continue with the next record there, trajectory T_2(18,21). Since T_2(7,9) in list $\mathcal{L}_D$ has interval before (18,21), list $\mathcal{L}_D$ moves to its next record T_2(21:23). These two occurrences of T_2 coincide with the pattern $M.?^*.D$ of $\mathcal{S}$ so we need to check if T_2 passes again by region M. Thus we consider the first record of list $\mathcal{L}_{M_2}$, namely trajectory T_1(10:13). Since it is not from T_2 it cannot be an answer so list $\mathcal{L}_{M_2}$ advances to the next

Algorithm 1 *IJP*: Fixed Spatial Predicates

Require: Query $\mathcal{S}$
Ensure: Trajectories satisfying $\mathcal{S}_f$
1: $n \leftarrow |\mathcal{S}_f|$ ▷ number of fixed predicates in $\mathcal{S}$
2: **for** $i \leftarrow 1$ to n **do** ▷ for each $\mathcal{S}_f$
3: Initialize $\mathcal{L}_i$ with the *cell-list* of $\mathcal{P}_i$
4: Candidate Set $U \leftarrow \emptyset$
5: **for** $w \leftarrow 1$ to $|\mathcal{L}_1|$ **do** ▷ analyze each entry in $\mathcal{L}_1$
6: $p_1 = w$ ▷ set the pointer for $\mathcal{L}_1$
7: **for** $j \leftarrow 2$ to n **do** ▷ examine all other lists
8: **if** $\mathcal{L}_1[w].id \notin \mathcal{L}_j$ **then**
9: **break** ▷ $\mathcal{L}_1[w].id$ does not qualify
10: Let k be the first entry for $\mathcal{L}_1[w].id$ in $\mathcal{L}_j$
11: **while** $\mathcal{L}_1[w].id = \mathcal{L}_j[k].id$ **and** $\mathcal{L}_{j-1}[p_{j-1}].t > \mathcal{L}_j[k].t$ **do**
12: $k \leftarrow k + 1$ ▷ align $\mathcal{L}_{j-1}[p_{j-1}].t$ and $\mathcal{L}_j[k].t$
13: **if** $\mathcal{L}_1[w].id \neq \mathcal{L}_j[k].id$ **then**
14: **break** ▷ $\mathcal{L}_1[w]$ does not qualify
15: **else** $p_j = k$ ▷ set the pointer for $\mathcal{L}_j$
16: **if** $\mathcal{L}_1[w]$ qualifies **then**
17: $U \leftarrow U \cup \mathcal{L}_1[w].id$ ▷ $\mathcal{L}_1[w]$ satisfy all $\mathcal{S}_f$

record T_2(18,21). Now pointers in all lists point to records of T_2. However, T_2(18,21) in $\mathcal{L}_{M_2}$ does not satisfy the pattern since its time interval should follow the interval (21,23) of T_2 in D. Hence $\mathcal{L}_{M_2}$ is advanced to the next record, which happens to be T_2(25,27). Again we have a record from the same trajectory T_2 in all lists and this occurrence of T_2 satisfies the temporal constraints and thus the pattern $\mathcal{S}$. As a result, trajectory T_2 is kept as a candidate in U. The processing moves to the next record in list $\mathcal{L}_{M_1}$, namely T_2(25,27). However, this record cannot satisfy the pattern $\mathcal{S}$ so it is skipped. Eventually $\mathcal{L}_{M_1}$ will consider T_3(10,11) which causes list $\mathcal{L}_D$ to move to T_3(5,10). Trajectory T_3 cannot satisfy the temporal constraint, so it is skipped from list $\mathcal{L}_D$ and the algorithm terminates since one of the lists reached its end. □

(ii) Variable predicate evaluation: The second step of the *IJP* algorithm evaluates the *variable* predicates r in $\mathcal{S}_v$, over the set of candidate trajectories U generated in the first step. For a fixed predicate its corresponding *region-list* contains all trajectories that satisfy it. However, *variable* predicates can be bound to any region, so one would have to look at all *region-lists*, which is not realistic. We will again need one list per each *variable* predicate (termed *variable-list*), however such *variable-lists* are not precomputed (like the *region-lists*). Rather they are created on the fly using the candidate trajectories filtered from the fixed predicate evaluation step.

To populate a *variable-list* for a *variable* predicate $P_j \in \mathcal{S}_v$ we compute the possible assignments for *variable* P_j by analyzing the *trajectory-list* for each candidate trajectory. In particular, we use the time intervals in a candidate trajectory to identify which portions of the trajectory can be assigned to this particular *variable* predicate. An example is shown in Figure 3, using the candidate trajectory T_2 from Figure 2. From the previous step we know that T_2 satisfies the fixed predicates at the following regions: M(18,21), D(21,23), M(25,27) (shown in bold in the *trajectory-list* of T_2). Using the pointers from the *region-lists* of the previous step, we know where the matching regions are in the *trajectory-list* of T_2. As a result, T_2 can be conceptually partitioned is three segments ($Seg1, Seg2, Seg3$) shown in

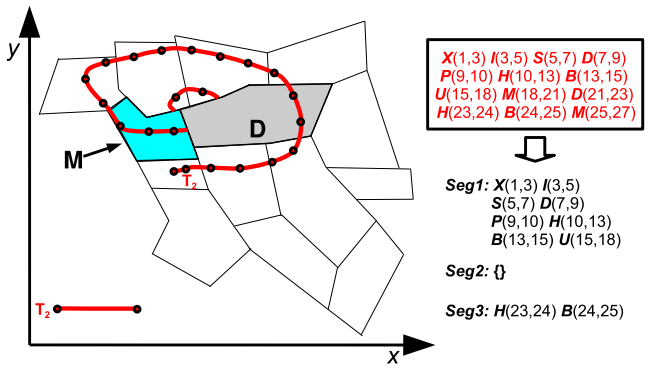

Figure 3: Segmentation of T_2 for *IJP* ($Seg2 = \emptyset$).

Figure 3. Note that $Seg2$ is empty since there is no region between M(18,21) and D(21,23).

These trajectory segments are used to create the *variable-lists* by identifying the possible assignments for every *variable*. Since a *variable*'s assignments need to maintain the pattern, each *variable* is restricted by the two fixed predicates that appear before and after the *variable* in the pattern. All *variables* between two fixed predicates are first grouped together. Then for every group of *variables* the corresponding trajectory segment (the segment between the fixed predicates) is used to generate the *variable-lists* for this group. Grouping is advantageous, since it can create *variable* lists for multiple *variables* through the same pass over the trajectory segments. Moreover, it ensures that the *variables* in the group maintain their order consistent with the pattern $\mathcal{S}$.

Assume that a group of *variable* predicates has w members. Each trajectory segment that affects the *variables* of this group is then streamed through a window of size w. The first w elements of the trajectory segment are placed in the corresponding predicate lists for the *variables*. The first element in the segment is then removed and the window shifts by one position. This proceeds until the end of the segment is reached. In the above example there are two groups of *variables*: the first consists of *variables* "?$^+$" and "@x" in that order (i.e., w=2), while the second group has a single member "@x" (w=1). Figure 4 depicts the first three steps in the *variable* list generation for the group of *variables* "?$^+$" and "@x". This group streams through segment $Seg1$, since it is restricted on the right by the fixed predicate M in pattern $\mathcal{S}$. Each list is shown under the appropriate *variable*. A different *variable* list will be created for the second group with *variable* "@x", since this group streams through segment $Seg3$ (the second "@x" *variable* is restricted by fixed predicates D and M).

The generated *variable-lists* are then joined in a way similar to the previous step. Because the *variable-lists* are populated by trajectory segments coming from the same trajectory (trajectory T_2 in our example) the join criteria checks only if the ordering of pattern $\mathcal{S}$ is obeyed. In addition, if the pattern contains *variables* with the same name (like @x) the join condition verifies that they are matched to the same region and time interval.

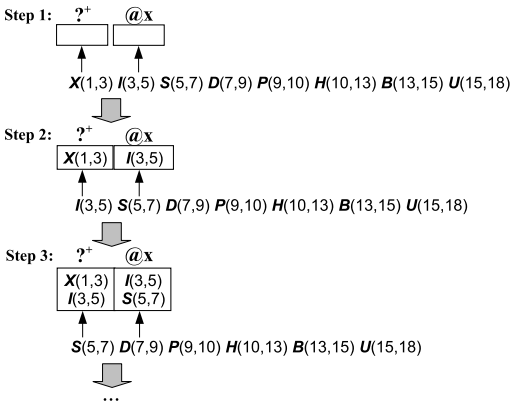

Figure 4: *Variable* list generation for *IJP*.

Complexity Analysis for variable predicate evaluation: Assume that the fixed predicate evaluation step generates k candidate trajectories in U and let l denote the maximum trajectory segment length. The worst case scenario is when all *variable* lists have length l. Thus the *variable* predicate evaluation in the worst case scenario is $\mathcal{O}(klr)$.

Explicit Temporal Constraints: The *IJP* algorithm can easily support explicit temporal constraints (assigned to the spatial predicates) by incorporating them as extra conditions in the join evaluations among the list records. Due to lack of space we omit further discussion.

4.1.2 Adding Distance-based Constraints

The evaluation of distance constraints $\mathcal{D}$ inside a query $\mathcal{Q}$ is performed as a post filtering step after the pattern $\mathcal{S}$ evaluation. The intuition is that the spatial predicates in $\mathcal{S}$ will greatly reduce the number of candidate trajectories which need to be examined by the distance-based algorithm. Nevertheless, since the distance terms contain *variables*, there are still many possibilities to bound the values of these *variables*. The *IJP* algorithm has the advantage of re-using the *variable* lists created during the spatial predicate search. These lists effectively enumerate all possible value bindings. However, instead of using a brute force approach that will examine all possible bindings, the *IJP* approach uses a variation of the *Threshold Algorithm* [8] and examines these possibilities in an incremental ordered fashion. As a result, it avoids examining all possible bindings.

Regarding the *IJP* approach, assume that the $\mathcal{S}$ evaluation has returned a collection of trajectories $\mathcal{T}$. For each *variable* in $\mathcal{S}$ one *variable-list* per trajectory in $\mathcal{T}$ is also created. All *variable-lists* for a given *variable* are concatenated and sorted, first by region and then by trajectory *id*. Note that the same region may be associated with different trajectory *ids*. For simplicity consider the scenario where the distance terms are combinations of a *variable* with a fixed region (i.e., $d(@x, A)$). The case where the distance term contains two *variables* is omitted for brevity.

For each distance term in $\mathcal{D}$ a separate list is created. As with the *variable-lists*, *distance-lists* are also computed *on-the-fly*. The idea is to incrementally examine the vicinity around the fixed *variable* of each distance term d_i. To evaluate distances between regions, we use the uniform grid that has been introduced in Section 3. We will use the distance between grid cells to lower bound the Euclidean distance between regions.

For example, given a term $d(@x, A)$, in the first iteration we examine the grid cells (and the regions approximated with those grid cells) that are one cell away from the grid approximation of region $\mathcal{A}$. The next iteration will expand the vicinity by one cell, and so on. When we discover a region which appears also in the sorted concatenated list for $@x$, we load all the corresponding trajectory *ids* and place them in the list for this distance term.

As the lists for all distance terms in $\mathcal{D}$ have been created incrementally, the *TA* algorithm finds the trajectory that appears in all *distance-lists* and minimizes the sum of the distances.

4.2 The Dynamic Programming Pattern Algorithm (DPP)

The *DPP* algorithm is divided into two steps: ***(i) Trajectory Selection*** and ***(ii) Matching***. Using the *trajectory-lists* the first step selects a candidate set of trajectories $\bar{T}$ based on the fixed predicates in $\mathcal{S}$. The second step uses pattern matching to eliminate trajectories that do not match the sequence order in $\mathcal{S}$. It also checks for appropriate *variable* bindings with possible verification on duplicate *variables* in $\mathcal{S}$. The pseudo code for the *DPP* algorithm is shown in Algorithm 2.

(i) Trajectory Selection: For each *region-list* of a fixed region that appears in $\mathcal{S}$, we select the *ids* T_{id} for all trajectories that visited this region. Candidate set $\bar{T}$ is computed by intersecting the collected *ids* (per region). That is, $\bar{T}$ contains *ids* of the trajectories that have visited (independently of what order) all the regions in $\mathcal{S}$. Nevertheless, since no order of these appearances has been verified, a further verification step must be performed on each $T' \in \bar{T}$ to enforce the *order* of $\mathcal{S}$. This verification step is performed using dynamic programming.

(ii) Matching: For each trajectory $T' \in \bar{T}$ a dynamic programming matrix $\mathcal{M}$ (function *BuildDPM*) is first created; it will later retrieve the matches of $\mathcal{S}$ in the trajectory T' (function *ScanDPM*). The $\mathcal{M}$ matrix enables the *DPP* algorithm to match **all** occurrences of the pattern $\mathcal{S}$ in T' in the specified order defined in $\mathcal{S}$. Matrix $\mathcal{M}$ has a column j for each region visited by the trajectory T'. Multiple visits to the same region are represented with multiple columns in $\mathcal{M}$, as it is stored the same way in the *trajectory-list* index. The rows i in the matrix correspond to the predicates $P_i \in \mathcal{S}$. Therefore, the size of $\mathcal{M}$ is $|\mathcal{S}|.|T'|$. The value in each entry in $\mathcal{M}[i][j]$ is computed based on the predicate P_i and the *j-th* element in the region approximation of the trajectory T' denoted as T'_j. (This is the *j-th* element in the *trajectory-list* of T').

It should be noted that if pattern $\mathcal{S}$ contains only fixed spatial predicates, the matrix $\mathcal{M}$ can be shrunk by eliminating the regions in T' that are not present in $\mathcal{S}$. This optimization does not compromise the sequence of patterns found because for each R_j in T', the attribute ($ts\text{-}entry_j$:$ts\text{-}exit_j$) is also kept.

Algorithm 2 *DPP*: Fixed and Variable Spatial Predicates

Require: Query $\mathcal{S}$ which consists of predicates P_i
Ensure: Trajectories satisfying $\mathcal{S}$
1: Let $\bar{T}$ be the set of candidate trajectories from *trajectory-list* having all fixed predicates in $\mathcal{S}$
2: Answer Set $\mathcal{A} \leftarrow \emptyset$ ▷ initialize the answer set
3: **for each** trajectory $T' \in |\bar{T}|$ **do**
4: *BuildDPM*(T', $\mathcal{S}$) ▷ construct matrix $\mathcal{M}$
5: **if** $ABS(\mathcal{M}[|\mathcal{S}|][|T'|]) \geq P_{|\mathcal{S}|}.idx$ **then**
6: *ScanDPM*(|$\mathcal{S}$|,|T'|) ▷ analyze matrix $\mathcal{M}$

Function: *BuildDPM*(T, $\mathcal{S}$)
1: **for** $i \leftarrow 0$ to $|\mathcal{S}|$ **do** ▷ for each row of $\mathcal{M}$
2: **for** $j \leftarrow 0$ to $|T|$ **do** ▷ for each column of $\mathcal{M}$
3: **if** $i = 0$ **or** $j = 0$ **then** $\mathcal{M}[i][j] \leftarrow 0$ ▷ trivial case
4: **else**
5: **if** $P_i.type$ is a Fixed Spatial Predicate **then**
6: **if** $P_i.R = T.R_j$ **then**
7: $\mathcal{M}[i][j] \leftarrow (-(ABS(\mathcal{M}[i-1][j-1]) + 1))$
8: **else**
9: $\mathcal{M}[i][j] \leftarrow MAX(ABS(\mathcal{M}[i-1][j]), ABS(\mathcal{M}[i][j-1]))$
10: **else** ▷ $P_i.type$ is a variable or wild-card
11: **if** $P_i.type = \{?^+, @\}$ **then**
12: $\mathcal{M}[i][j] \leftarrow (-(ABS(\mathcal{M}[i-1][j-1]) + 1))$
13: **else** ▷ case where $P_i.type = \{?^*\}$
14: **if** $i = P_i.idx$ **then**
15: $\mathcal{M}[i][j] \leftarrow ABS(\mathcal{M}[i-1][j])$
16: **else** $\mathcal{M}[i][j] \leftarrow (-(ABS(\mathcal{M}[i-1][j-1]) + 1))$

Function: *ScanDPM*(i,j)
1: **if** $i > 0$ **then** ▷ valid column in $\mathcal{M}$
2: **for** $k \leftarrow j$ to $k \geq P_i.idx$ **downto** 1 **do**
3: **if** $ABS(\mathcal{M}[i][k]) \geq P_i.idx$ **then**
4: **if** $\mathcal{M}[i][k] \leq 0$ **then** ▷ found a match in $\mathcal{M}$
5: **if** $P_i.type = \{@\}$ **and** $Match[P_i.link] \neq T'.R_k$ **then continue**
6: $Match[i] \leftarrow T'.R_k$ ▷ found a match for $T'.R_k$
7: **if** $P_{i-1}.type = \{?^*\}$ **then**
8: *ScanDPM*($i-1$, k) ▷ next iteration
9: **else**
10: *ScanDPM*($i-1$, $k-1$) ▷ next iteration
11: **else** $\mathcal{A} \leftarrow \mathcal{A} \cup T'.id$ ▷ found $T'.id$ to the answer set

Each matrix entry can a take numerical value in the range (-|$\mathcal{S}$|;|$\mathcal{S}$|). The absolute value stored in the matrix entries corresponds to the length of the longest match between the pattern $\mathcal{S}$ and the trajectory approximation T' discovered so far. A negative number in $\mathcal{M}[i][j]$ denotes a match between the pattern P_i and the trajectory region R_i, and its absolute value is the length of the longest match found so far. In this way, the matrix $\mathcal{M}$ is used to store both the match occurrences, represented with negative value, and the length of each match, the absolute values in $\mathcal{M}[i][j]$.

The matrix $\mathcal{M}$ is computed row by row, column by column starting from the $\mathcal{M}[0][0]$ entry until the $\mathcal{M}[|\mathcal{S}|][|T'|]$ entry. At every step the *BuildDPM* function compares the values of the current predicate P_i and the current region from the trajectory approximation T_j (the same as the T'_j). If there is no match between P_i and T_j, then the value of $\mathcal{M}[i][j]$ is the biggest absolute value among the neighbors ($\mathcal{M}[i-1][j]$ or $\mathcal{M}[i][j-1]$). If there is a match between P_i and T_j then the entry $\mathcal{M}[i][j]$ takes the value $|\mathcal{M}[i-1][j-1]| + 1$, but it is stored as a negative number indicating that the current pair P_i, T_j participates in the match.

The previous description applies only for fixed spatial predicates. For wild-card ($?^+$,$?^*$) and variable (@) spatial predicates, the computation of the entry $\mathcal{M}[i][j]$ is done differently. Because such variables can be *bound* with any value of T_j, the value of $\mathcal{M}[i][j]$ is computed as a "match". Therefore, the entry value is $-(|\mathcal{M}[i-1][j-1]|+1)$, as previously described. This phase does not handle the case where a pattern $\mathcal{S}$ contains *variables* which appear multiple times. This verification step is performed in the *ScanDPM* function. Instances of the same variable are "linked" in a backward way using a "pointer" (*link*) with the following constraint: $P_i.link \leftarrow P_j$ if $P_i = P_j$ and $i < j$. Because matrix $\mathcal{M}$ is verified for matching in a "backward" way (from $\mathcal{M}[|\mathcal{S}|][|T'|]$ to $\mathcal{M}[1][1]$ entry), the pointers are associated to the next occurrence in the pattern $\mathcal{S}$.

There is also a special case where the predicate P_i is *optional* in the pattern $\mathcal{S}$. In this case, the computation and further verification of matrix $\mathcal{M}$ has to consider the case where P_i does not match T_j. To deal with this, another attribute $P_i.idx$ is associated with each predicate in $\mathcal{S}$. Basically, this attribute stores the position of each predicate P_i in cases the optional predicate does not match with any T_j. This *idx* attribute is defined in the following manner:

$$P_i.idx \leftarrow \begin{cases} 1 & \textbf{if } i = 1 \\ P_{i-1}.idx & \textbf{if } P_i.type = \{?^*, ?^\#\} \\ P_{i-1}.idx + 1 & \textbf{otherwise} \end{cases}$$

After the matrix $\mathcal{M}$ is computed, the matches on it have to be searched. This is performed by the *ScanDPM* function which "searches" for negative numbers stored in $\mathcal{M}$; such numbers denote the occurrence of a match. The operation goes row by row, column by column in a direction opposite to the direction of construction, starting with the bottom right entry. If the last matrix entry $\mathcal{M}[i][|T'|]$ has an absolute value greater than the last *idx* in P (i.e. $ABS(\mathcal{M}[|\mathcal{S}|][|T'|]) \geq P_{|\mathcal{S}|}.idx$), then there is at least one match between $\mathcal{S}$ and T'. Otherwise we can safely prune the trajectory avoiding further processing. Because we are only interested in finding the longest and complete match between $\mathcal{S}$ and T', we only look for entries that have values greater or equal than the $\mathcal{S}_i.idx$ index (smaller values indicate that there is a partial match but not a complete one). If the cell value is less than the current pattern index $\mathcal{S}_i.idx$, then the function *ScanDPM* aborts the processing of the current row i.

If there is a match in $\mathcal{M}[i][j]$, then the function *ScanDPM* is called recursively to process the sub-matrix with bottom right corner $\mathcal{M}[i-1][j-1]$. If the predicate P_i is optional ($^\#$ and *) then the function is called for the $\mathcal{M}[i-1][j]$ entry instead. The algorithm stops when all predicates in $\mathcal{S}$ are processed (i=0), thus finding *all* possible matches of $\mathcal{S}$ in T'.

Complexity Analysis: The *BuildDPM* function calculates the value for each matrix entry just once. Let s denote the length of a trajectory T' in terms of number of regions visited. Then the matrix $\mathcal{M}$ has m rows (|$\mathcal{S}$|) and s columns, and the complexity of this method is $\mathcal{O}(sm)$. The complexity of *ScanDPM* is $\mathcal{O}(m+s)$ because at each step we move one step left-up diagonally or up (e.g., at least one of i and j is decremented). Therefore, the time complexity for

Table 1: Matrix $\mathcal{M}$ for T_2 and $\mathcal{S}$ example

			T_2	X	I	S	D	P	H	B	U	M	D	H	B	M
			j	1	2	3	4	5	6	7	8	9	10	11	12	13
$\mathcal{S}$	i	idx	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$?^+$	1	1	0	-1$^{\circledast}$	-1$^{\circledast}$	-1$^{\circledast}$	-1$^{\circledast}$	-1$^{\circledast}$	-1*	-1	-1	-1	-1	-1	-1	-1
$@x$	2	2	0	0	-2	-2	-2	-2	-2°	-2*	-2	-2	-2	-2	-2	-2
$?^*$	3	2	0	0	2	-3	-3	-3	-3	-3°	-3$^{\circledast}$	-3	-3	-3	-3	-3
M	4	3	0	0	0	3	3	3	3	3	3	-4$^{\circledast}$	4	4	4	-4
$?^*$	5	3	0	0	0	3	-4	-4	-4	-4	-4	-4	-5	-5	-5	-5
D	6	4	0	0	0	0	-4	4	4	4	4	4	-5$^{\circledast}$	5	5	5
$?^*$	7	4	0	0	0	0	4	-5	-5	-5	-5	-5	-5	-6*	-6	-6
$@x$	8	5	0	0	0	0	0	-5	-6	-6	-6	-6	-6	-6°	-7*	-7
$?^*$	9	5	0	0	0	0	0	5	-6	-7	-7	-7	-7	-7	-7°	-8
M	10	6	0	0	0	0	0	0	6	7	7	-8	8	8	8	-8$^{\circledast}$

processing a single trajectory T' with the *DPP* algorithm is $\mathcal{O}(vsm)$, where $v = |\bar{T}|$ (i.e., the number of candidate trajectories produced from the ***trajectory selection*** step). The reader should note that the two algorithms produce candidate trajectory sets using different methods (*IJP* considers the temporal order and *DPP* does not); hence in the complexity analysis they are represented as k and v.

Explicit Temporal Constraints: When the pattern query $\mathcal{S}$ has explicit temporal constrains int_i in its definition, the *DPP* algorithm only performs a check along with the match checks in order to satisfies int_i too (not shown in Algorithm 2). If only one of the above conditions is satisfied, then the value of $\mathcal{M}[i][j]$ is computed as not a match. Otherwise, it is computed as a match.

Example: We use the same example of pattern $\mathcal{S}$ in Figure 2 to illustrate how the *DPP* algorithm works. Using the *region-list* the trajectory identifiers that have all the grids M and D are in $\bar{T} = \{T_2, T_3\}$. For each trajectory T' in $\bar{T}$, the matrix $\mathcal{M}$ is computed using the function *BuildDPM*. The computation of matrix $\mathcal{M}$ for T_2 and $\mathcal{S}$ appears in Table 1. Since $P_{|\mathcal{S}|}.idx$ is 6, the *ScanDPM* function looks for entry values equal to $\mathcal{M}[10][j] \geq |-6|$ in the 10-*th* row of matrix $\mathcal{M}$. In *ScanDPM*, the entry $\mathcal{M}[10][13]$ passes the checks of the algorithm and the entry $\mathcal{M}[10][13]$ is stored as a match in $Match[10]$ (M was found in the 13*th* column of T_2) and then the function *ScanDPM* is called for the $\mathcal{M}[9][12]$ matrix. Again, entry $\mathcal{M}[9][12]$ passes all the checks and it is called for $\mathcal{M}[8][12]$. Because P_8 is a *variable* (i.e., *variable* $@x$) and it is the first *variable* encountered so far, it passes the bounded value check (*link* test) and then it is bounded to the grid B. Then the function *ScanDPM* is called in the following sequence for entries in $\mathcal{M}$: $\mathcal{M}[7][11]$, $\mathcal{M}[6][10]$, $\mathcal{M}[5][10]$, $\mathcal{M}[4][9]$, $\mathcal{M}[3][8]$ and then for $\mathcal{M}[2][8]$, but it fails for this last one because the *link* test does not pass ($\mathcal{M}[2][8] \neq \mathcal{M}[8][12]$). Then it is called for $\mathcal{M}[2][7]$, and the *link* test satisfies because variable $@x$ is bounded to grid B ($\mathcal{M}[2][7] = \mathcal{M}[8][12]$). Then *ScanDPM* is called for $\mathcal{M}[1][6]$ until j is 0. In the end, the pattern $?^+.B.?^*.M.?^*.D.?^*.B.?^*.M$ is found and added to $\mathcal{A}$. The backtracking also evaluates the entry $\mathcal{M}[8][11]$ and finds pattern $?^+.H.?^*.M.?^*.D.?^*.H.?^*.M$. Other calls for other entries are called, e.g. $\mathcal{M}[10][9]$ (-8), but they all fails to bound to other predicates in $\mathcal{S}$. The 2 patterns found for the query pattern $\mathcal{S}$ in trajectory T_2 are highlighted in Table 1 (yellow* for the first pattern found, blue$^{\circ}$ for the second, and green$^{\circledast}$ when the entries are found for both of patterns). □

4.2.1 Adding Distance-based Constraints

The evaluation of distance constraints $\mathcal{D}$ inside a query $\mathcal{Q}$ is performed as a post filtering step after the pattern $\mathcal{S}$ evaluation. The *DPP* algorithm can only use a brute force approach since it maintains a trajectory as a sequence of regions but loses the spatial properties of these regions. Therefore, the *DPP* algorithm can only compute the distance for the constraint as a final step.

5. EXPERIMENTAL EVALUATION

We run various experiments with real world and synthetic datasets to test the behavior of each technique under different settings. All experiments were run on an Intel Pentium-4 2.6 GHz processor running Linux 2.6.22 with 1 GBytes main memory. All implementations used the same disk manager framework with disk page size set to 4KB for each index (*region-list* and *trajectory-list* indexes) and 16KB for the raw trajectory archive.

For comparison purposes, we examined two previous pattern matching approaches. In particular, we modified [5] and [2] (called here *Extended-KMP* (*E-KMP*) and *Extended-NFA* (*E-NFA*) respectively) and implemented them in our proposed framework in order to fair compare them against the *IJP* and *DPP* algorithms. The *E-KMP* contains extensions to handle the variable predicates ($?^*$, $?^+$) as well as the implicit/explicit temporal constraints. The *NFA* used in [2] finds simple event patterns in streaming data. Hence it is not formulated to evaluate topological relations or temporal constraints as described in this paper. We thus extend it to cover these as well, as to process queries with variables. To this end, a stack is created for each variable "$@x$". If a variable appears in the query many times, a post processing check is performed at the accept state of the NFA. For fairness, all algorithms were tested using the same index framework (i.e., the *E-KMP* and the *E-NFA* algorithms receive a candidate set of trajectories similar to the *DPP* approach).

For real datasets, we use the *Truck* and *Buses* trajectorial data from [1]. Both datasets represent moving objects in the metropolitan area of Athens, Greece. The *Truck* dataset has 276 trajectories of 50 trucks where the longest trajectory timestamp is 13,540 time units. The *Buses* has 145 trajectories of school buses with maximum timestamp 992. For simplicity of the experimental evaluation, we do not use real regions; instead we assume that the spatial domain (area of Athens) is partitioned into (artificial) regions using a uniform grid. These grid cells become the alphabet for our queries; hence in the rest the terms "region" and "cell" have the same meaning. To examine the effect of the alphabet size on the index structures we experiment with grid granularity starting from 25×25 up to 100×100.

For the synthetic datasets, we generated datasets of moving object trajectories. The dataset represents the freeway network of Indiana and Illinois states together. The 2-dimensional spatial universe is 1,000 miles long in each direction and contains up to 200,000 objects. Objects start at random positions on predefined routes in a road network and follow a Normal distribution with mean 60 mph and standard deviation 15 mph. We run simulations for 500 minutes (timestamps). For these datasets, the spatial universe was

Table 2: Query time (s) for real datasets.

P	Dataset	$\|\mathcal{S}\|$	$\|\mathcal{S}_f\|$	$\|\mathcal{A}\|$	*E-NFA*	*E-KMP*	*DPP*	*IJP*
$\mathcal{S}_1$	*Buses*	10	3	57	2.46	1.90	**1.11**	1.53
$\mathcal{S}_2$	*Buses*	20	7	29	89.62	62.75	28.99	**3.03**
$\mathcal{S}_3$	*Trucks*	20	7	76	111.91	54.68	30.28	**10.57**
$\mathcal{S}_4$	*Trucks*	46	29	11	3.06	0.73	**0.22**	1.56

partitioned with a grid 100×100.

In order to generate relevant query patterns, we randomly sample and fragment 100 trajectories. The length and location of each fragment are randomly chosen. These fragments are then concatenated to create a query. For the synthetic datasets we used pattern length from 5 up to 10 predicates. We also generated sets of query patterns with different number of variable predicates (from 0 to 2). The location of each variable inside the query was randomly chosen. For queries with 2 variables, half of the patterns have the same variable twice, and the other half use two different variables (i.e. $@x$ and $@y$). For each experiment, we measured the average total time (in seconds) and the average I/O for a set of 100 queries. The query cost shown consists of the CPU time and the I/O time.

5.1 Queries with Spatial Predicates

The first experiment, shown in Table 2, evaluates the total time (in seconds) required to execute four complex pattern queries on the *Buses* and *Truck* datasets. Since in the real datasets objects move in relatively similar ways, we experimented with larger number of predicates so as to create more selective queries. Moreover, queries $\mathcal{S}_1$ - $\mathcal{S}_4$ contain between 2 and 4 variables and several wild-cards $?^+$ and $?^*$. The total number of predicates is specified by $|\mathcal{S}|$, the number of fixed predicates is $|\mathcal{S}_f|$, the number of trajectories returned is shown under $|\mathcal{A}|$.

The results show that the *E-NFA* algorithm performs worse for all queries. This is because it cannot take advantage of the existing indexing structures so as to focus the search only on those parts of the trajectories that might contain answer (except from the original trajectory pruning using the region-list intersection). This is to be expected since the method has been designed for identifying patterns over streaming (non-archived) data. We experienced a similar behavior with the other real and also the synthetic datasets; hence we remove the *E-NFA* method from the following comparisons. Among the rest, the *DPP* and *IJP* algorithms, have typically more robust behavior; nevertheless, *E-KMP* still shows competitive behavior for some queries.

To examine the effect of the size of the alphabet on the index size, we experimented with the real datasets and different alphabets (by changing the grid size). As expected the increased number of letters in the alphabet increases the size of the index (see Figure 6). Each trajectory visits more regions (which have smaller size) during its lifetime and thus generates more records in the index structure. Note that in this experiment, the size of index was very small compared to the raw data size (varying between 4% and 6% for the *Buses* and 2% and 5% for the *Trucks* dataset). The number of I/Os during the query evaluation however stays the same because each predicate in the query still corresponds to a single (though smaller) grid cell. As a result, the observed query times remain similar to the ones shown in Table 2 (and thus omitted).

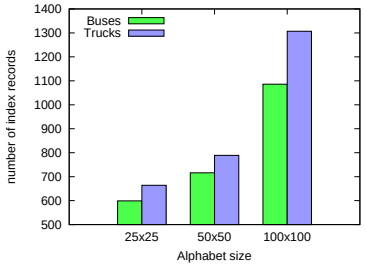

Figure 6: Total number of index records for different alphabet sizes

To further examine the performance of the *DPP*, *IJP* and *E-KMP* algorithms we use the (controlled) synthetic datasets. The next experiment evaluates the total time required to execute 100 pattern queries while varying the number of spatial predicates in $\mathcal{P}$ from 5 to 10. It uses a synthetic dataset with 50,000 trajectories. The results appear in Figure 5 (in log-scale) and for patterns with: no variable, 1 variable and 2 variables (Figures 5(a), (b) and (c) respectively).

As observed from these experiments, when increasing the number of predicates in the pattern, the query time of the *DPP* and *E-KMP* algorithms increases. For the *DPP* the larger pattern implies a larger matrix and thus more processing. The *E-KMP* is very sensitive to the number of "?*" in the query; as the pattern increases in size the probability of more "?*" increases (this effect will be examined further later). Nevertheless, the *DPP* algorithm is always more efficient than the *E-KMP* (typically by an order of magnitude).

The *IJP* algorithm is affected the least by the number of predicates. This is because processing in the *IJP* algorithm is guided by the *region-lists* of the first few predicates in the pattern (for example, the third list is accessed after a match in the first two lists is found, etc.). Hence, adding more lists does not directly affect the processing. As more predicates are added, the processing of the *E-KMP* and *DPP* starts increasing making the *IJP* a faster solution.

For the same experiments, Figure 7 depicts the average I/O's for 1 variable and 2 variables. In particular *E-KMP* and *DPP* have identical I/O behavior since they are using the same approach to pick candidate *trajectory-lists* (without using the time constraints). Even though all three algorithms use the same indexes to retrieve objects, the *IJP* uses a different strategy (as described in Section 4) which results in a different I/O behavior. Nevertheless, all algorithms have comparable I/O behavior, leading us to the conclusion that the major differences in the overall processing time among the algorithms are not I/O based but mainly CPU bound.

We also performed experiments comparing the proposed index structure with R-trees. The R-tree was outperformed by our simpler grid structure (not reported here for lack of space). Since R-trees are data-driven structures the overlapping implies that several sub-trees need to be analyzed. Fur-

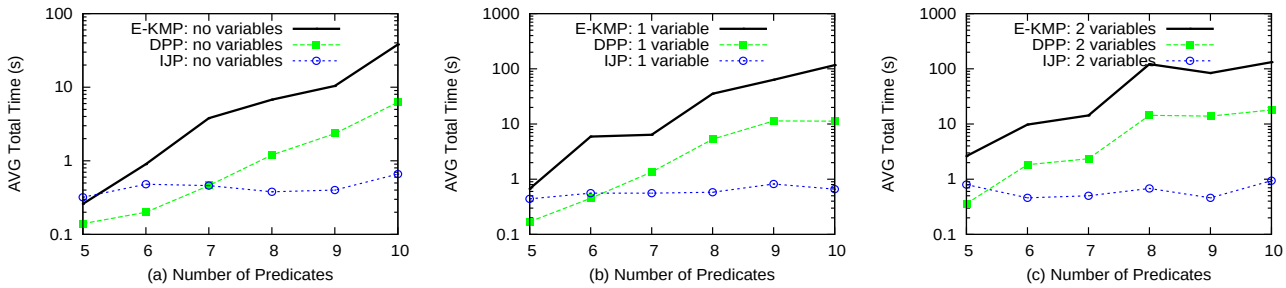

Figure 5: Query Time for pattern with (a) no variable, (b) 1 variable, and (c) 2 variables

thermore, when MBRs over-approximate regions, the verification step at the end of the algorithm had to process significant amount of false positives.

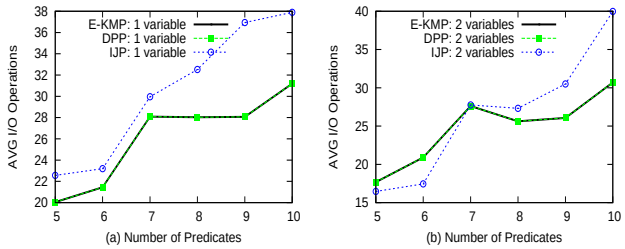

Figure 7: Query I/O for 1 and 2 variables

5.2 Scalability Experiments

Number of wild-cards in $\mathcal{P}$: We next examined the performance of the three algorithms when varying the number of "?*" wild-cards in the pattern. For these experiments, we randomly sampled 100 trajectories from the previous synthetic dataset and then extracted query patterns of length 10. These patterns contain only the "." sequencer (i.e., no "?*") and 2 variables. Using this query set we created a new set that has queries with one ".?*.". This set was created by randomly replacing one "." by a ".?*.". We continued in the same way creating a new query set with queries having two "?*" by replacing an additional ".", etc. Figure 8(a) shows the average total time (in log-scale) to execute 100 queries varying the total number of "?*" (from 0 to 8) in each pattern with 2 variables.

Again we observed that in all experiments the *DPP* algorithm is always faster than the *E-KMP*. As the number of "?*" increases, the performance of *E-KMP* deteriorates drastically, showing the dependence of *E-KMP* to the "?*" wild-cards. This is because each such wild-card forces the *E-KMP* approach to run more, shorter queries. More queries add to the processing time but also since these queries are smaller, the shifting function of *E-KMP* is not as effective. The *DPP* is up to 4 times faster than the *E-KMP* when there are 8 "?*" wild-cards (Figure 8(a)). For the *DPP*, the total processing time increases because more matches qualify as an answer. The performance of the *IJP* algorithm is independent of the number of the "?*" wild-cards, since they are evaluated in the same way as the "." sequencers. As a result, as more wild-cards appear in the query, *IJP* will eventually become faster than the *DPP*.

Number of Trajectories: We then varied the indexed dataset size to examine the scalability of the proposed algorithms. For these experiments, we used a synthetic dataset of 200,000 trajectories. We started with inserting the first 50,000 trajectories in the indexes and measured the query time (for an average of 100 queries each with 5 predicates, including 0, 1 and 2 variables). We repeated the experiment after adding an additional 25,000 trajectories. This incremental process continued with increments of 25,000 trajectories until the total of 200,000 trajectories in the archive.

The behavior of all algorithms grows linearly with the dataset size, as shown in Figure 8(b). Recall that from our complexity analysis, both the *IJP* and *DPP* algorithms are proportional to the number of candidate trajectories; as more trajectories are added, this number increases thus affecting the overall performance accordingly. Again, the *DPP* algorithm behaves consistently better than the *E-KMP*. Among all algorithms, the *IJP* has the faster rate of increase. This is because, the larger datasets create large *region-lists* which directly affects the join processing cost. Moreover, *IJP* performs two join operations (one in the *region-lists* and one in the *variable-lists*) and both of them are directly affected by the size of the lists.

5.3 Patterns with Spatial Predicates and Nearest Neighbors

We also performed experiments to examine how the algorithms behave when adding nearest neighbor predicates (i.e., pattern queries that contain both $\mathcal{P}$ and $\mathcal{D}$). We examined four query datasets varying the number of distance terms from one to four. Each distance term uses two variables (i.e., it is of the form $d(@x, @y)$ which corresponds to the very processing demanding *NN* query). All variables in each query pattern are different and their positions were randomly chosen. Figure 8(c) shows the results for queries using 10 predicates while increasing the number of distance terms. Clearly, the *IJP* approach outperforms the "brute-force" approach of the *DPP* (up to two orders of magnitude). This is because *IJP* maintains the spatial properties of trajectories and can thus reuse the variable lists to avoid examining all possible bindings.

Discussion: In all our experiments the previous *E-KMP*-based approach (even optimized to use indexes) was outperformed by the *DPP* algorithm. Furthermore, its performance deteriorates drastically as the number of "?*" wild-cards increases. Similarly, the *E-NFA* was outperformed by all algorithms. When comparing our two new algorithms, we

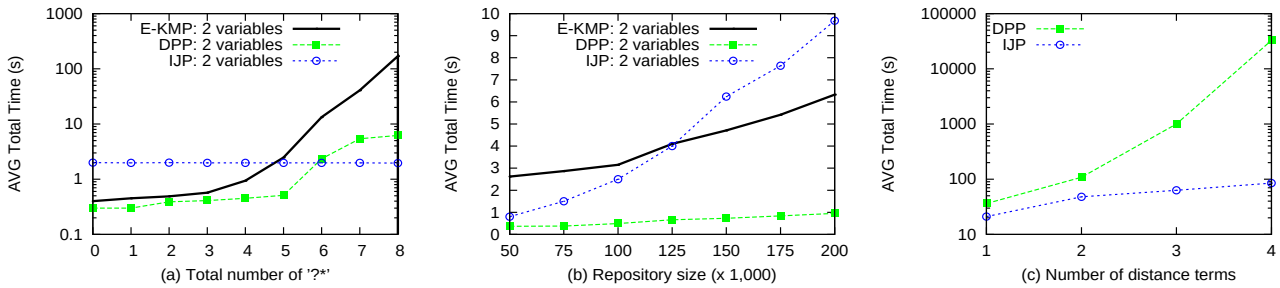

Figure 8: Total time (s) when increasing (a) the number of "?*" in $\mathcal{P}$; (b) the number of trajectories indexed, and (c) the number of distance terms in $\mathcal{D}$

observed the following: **(i)** For a small number of predicates the *DPP* algorithm is faster than the *IJP* algorithm. This is because the matrix M is small and thus it is processed very fast. **(ii)** For larger number of predicates the *IJP* algorithm becomes faster since its performance is not affected by the increase in predicates, while the *DPP* is affected by the increase in the matrix size. **(iii)** On the other hand, *IJP* is a join-based algorithm, hence the larger the dataset, the more expensive is the join step. **(iv)** Nevertheless, *IJP* has more robust performance when considering distance-based queries (*NN*) as well, while the *DPP* (and *E-KMP*) algorithm needs to use a very consuming "brute-force" approach.

6. CONCLUSIONS AND FUTURE WORK

We introduced a framework for processing "flexible pattern queries" over trajectory archives. Such queries combine the ability of fixed and variable predicates, with explicit or implicit temporal constraints and distance-based constraints. Previous works have considered only subsets of this framework and are based on variations of the *KMP* algorithm or use finite automata. We introduced two query processing techniques, one based on merge joins (*IJP*) and one based on subsequence matching (*DPP*). The experimental evaluation shows that our techniques improve substantially even over *optimized* (using indexing and preprocessing techniques) *KMP* and *NFA* approaches. Among our approaches, *IJP* is more robust in that it can easily support *NN* queries, while *DPP* is better for patterns with smaller number of predicates or wild-cards. Since however both approaches use the same indexing schemes, they can both be available to the user. As a next step in this research we are examining cost models that will enable a query optimizer to pick the best technique based on the query parameters (size of the query pattern, number of variables, wild-cards, etc). Furthermore, we plan to extend our framework to support complex pattern trajectory joins and density-based pattern queries.

Acknowledgements: This research was partially supported by NSF grants IIS-0534781 and IIS-0803410. M. Vieira's work has been funded by a CAPES (Brazilian Federal Agency for Post-Graduate Education)/Fulbright Ph.D. fellowship.

7. REFERENCES

[1] http://www.rtreeportal.org.
[2] J. Agrawal and et al. Efficient pattern matching over event streams. *SIGMOD*, pages 147–159, 2008.
[3] A. Anagnostopoulos and et al. Global distance-based segmentation of trajectories. In *ACM KDD*, 2006.
[4] Y. Cai and R. Ng. Indexing spatio-temporal trajectories with Chebyshev polynomials. In *ACM SIGMOD*, pages 599–610, 2004.
[5] C. du Mouza, P. Rigaux, and M. Scholl. Efficient evaluation of parameterized pattern queries. In *CIKM*, pages 728–735, 2005.
[6] M. Erwig and M. Schneider. Spatio-temporal predicates. *IEEE TKDE*, pages 881–901, 2002.
[7] ESRI. ArcGIS. www.esri.com.
[8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.
[9] M. Hadjieleftheriou, G. Kollios, P. Bakalov, and V. Tsotras. Complex spatio-temporal pattern queries. In *VLDB*, pages 877–888, 2005.
[10] M. Hadjieleftheriou, G. Kollios, V. Tsotras, and D. Gunopulos. Indexing spatiotemporal archives. *VLDB J.*, pages 143–164, 2006.
[11] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. on Computing*, 1977.
[12] G. Kollios, D. Papadopoulos, D. Gunopulos, and V. Tsotras. Indexing mobile objects using dual transformations. *VLDB J.*, pages 238–256, 2005.
[13] J.-G. Lee, J. Han, and K.-Y. Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, pages 593–604, 2007.
[14] N. Mamoulis and et al. Mining, indexing, and querying historical spatiotemporal data. In *ACM SIGKDD*, pages 236–245, 2004.
[15] M. Mokbel and W. Aref. SOLE: scalable on-line execution of continuous queries on spatio-temporal data streams. In *VLDB J.*, pages 971–995, 2008.
[16] H. Mokhtar, J. Su, and O. Ibarra. On moving object queries. In *ACM PODS*, pages 188–198, 2002.
[17] M. Pelanis, S. Saltenis, and C. Jensen. Indexing the past, present, and anticipated future positions of moving objects. *ACM TODS*, pages 255–298, 2006.
[18] D. Pfoser, C. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, pages 395–406, 2000.
[19] R. Sadri and et al. Expressing and optimizing sequence queries in database systems. *ACM TODS*, 2004.
[20] M. A. Sakr and R. H. Güting. Spatiotemporal pattern queries in SECONDO. In *SSTD*, pages 422–426, 2009.
[21] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A model for sequence databases. In *IEEE ICDE*, 1995.
[22] Y. Tao and D. Papadias. MV3R-Tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB*, pages 431–440, 2001.
[23] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *VLDB*, pages 287–298, 2002.