Mining, Indexing, and Querying Historical Spatiotemporal Data

Nikos Mamoulis
University of Hong Kong

Huiping Cao
University of Hong Kong

George Kollios
Boston University

Marios Hadjieleftheriou
University of California, Riverside

Yufei Tao
City University of Hong Kong

David W. Cheung
University of Hong Kong

ABSTRACT

In many applications that track and analyze spatiotemporal data, movements obey periodic patterns; the objects follow the same routes (approximately) over regular time intervals. For example, people wake up at the same time and follow more or less the same route to their work everyday. The discovery of hidden periodic patterns in spatiotemporal data, apart from unveiling important information to the data analyst, can facilitate data management substantially. Based on this observation, we propose a framework that analyzes, manages, and queries object movements that follow such patterns. We define the spatiotemporal periodic pattern mining problem and propose an effective and fast mining algorithm for retrieving maximal periodic patterns. We also devise a novel, specialized index structure that can benefit from the discovered patterns to support more efficient execution of spatiotemporal queries. We evaluate our methods experimentally using datasets with object trajectories that exhibit periodicity.

General Terms: Algorithms
Categories & Subject Descriptors: H.2.8 [Database Management]: Database Applications - Data Mining
Keywords: Spatiotemporal data, Trajectories, Pattern mining, Indexing

1. INTRODUCTION

The efficient management of spatiotemporal data has gained much interest during the past few years [10, 13, 4, 12], mainly due to the rapid advancements in telecommunications (e.g., GPS, Cellular networks, etc.), which facilitate the collection of large datasets of such information. Management and analysis of moving object trajectories is challenging due to the vast amount of collected data and novel types of spatiotemporal queries.

This work was supported by grant HKU 7149/03E from Hong Kong RGC and partially supported by NSF grants IIS-0308213 and Career Award IIS-0133825.

In many applications, the movements obey periodic patterns; i.e., the objects follow the same routes (approximately) over regular time intervals. Objects that follow approximate periodic patterns include transportation vehicles (buses, boats, airplanes, trains, etc.), animal movements, mobile phone users, etc. For example, Bob wakes up at the same time and then follows, more or less, the same route to his work everyday. Based on this observation, which has been overlooked in past research, we propose a framework for mining, indexing and querying periodic spatiotemporal data.

The problem of discovering periodic patterns from historical object movements is very challenging. Usually, the patterns are not explicitly specified, but have to be mined from the data. The patterns can be thought of as (possibly noncontiguous) sequences of object locations that reappear in the movement history periodically. Moreover, since we do not expect an object to visit *exactly* the same locations at every time instant of each period, the patterns are not rigid but differ slightly from one occurrence to the next. The pattern occurrences may also be shifted in time (e.g., due to traffic delays or Bob waking up late again). The approximate nature of patterns in the spatiotemporal domain increases the complexity of mining tasks. We need to discover, along with the patterns, a flexible description of how they variate in space and time. Previous approaches have studied the extraction of patterns from long event sequences [5, 7]. We identify the difference between the two problems and propose novel techniques for mining periodic patterns from a large historical collection of object movements.

In addition, we design a novel indexing scheme that exploits periodic pattern information to organize historical spatiotemporal data, such that spatiotemporal queries are efficiently processed. Since the patterns are accurate approximations of object trajectories, they can be managed in a lightweight index structure, which can be used for pruning large parts of the search space without having to access the actual data from storage. This index is optimized for providing fast answers to range queries with temporal predicates. Effective indexing is not the only application of the mined patterns; since they are compact summaries of the actual trajectories, we can use them to compress and replace historical data to save space. Finally, periodic patterns can predict future movements of objects that follow them.

The rest of the paper is organized as follows. Section 2 presents related work. In Section 3, we give a concrete formulation of periodic patterns in object trajectories and propose effective mining techniques. Section 4 presents the indexing scheme that exploits spatiotemporal patterns. We present a concise experimental evaluation of our techniques in Section 5. Finally, Section 6 concludes with a discussion about future work.

2. RELATED WORK

Our work is related to two research problems. The first is data mining in spatiotemporal and time-series databases. The second is management of spatiotemporal data. Previous work on spatiotemporal data mining focuses on two types of patterns: (i) frequent movements of objects over time and (ii) evolution of natural phenomena, such as forest coverage. [14] studies the discovery of frequent patterns related to changes of natural phenomena (e.g., temperature changes) in spatial regions. In general, there is limited work on spatiotemporal data mining, which has been treated as a generalization of pattern mining in time-series data (e.g., see [14, 9]). The locations of objects or the changes of natural phenomena over time are mapped to sequences of values. For instance, we can divide the map into spatial regions and replace the location of the object at each timestamp, by the region-id where it is located. Similarly, we can model the change of temperature in a spatial region as a sequence of temperature values. Continuous domains of the resulting time-series data are discretized, prior to mining. In the case of multiple moving objects (or time-series), trajectories are typically concatenated to a single long sequence. Then, an algorithm that discovers frequent subsequences in a long sequence (e.g., [16]) is applied.

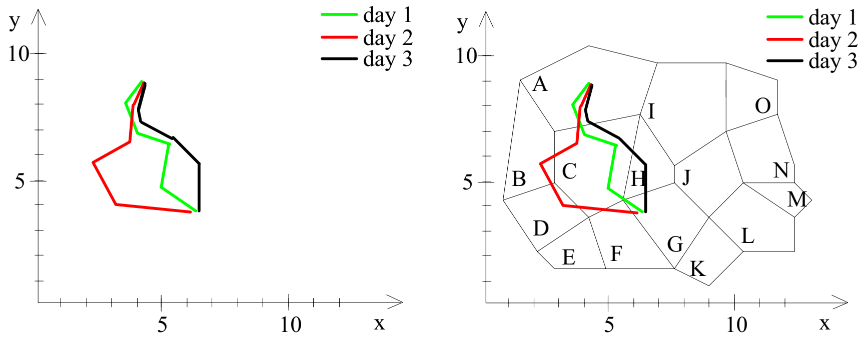
Periodicity has only been studied in the context of time-series databases. [6] addresses the following problem. Given a long sequence $\mathcal{S}$ and a period T, the aim is to discover the most representative trend that repeats itself in $\mathcal{S}$ every T timestamps. Exact search might be slow; thus, [6] proposes an approximate search technique based on sketches. However, the discovered trend for a given T is only one and spans the whole periodic interval. In [8], the problem of finding association rules that repeat themselves in *every* period of a data sequence is addressed. The discovery of multiple partial periodical patterns that do not appear in every periodic segment was first studied in [5]. A version of the well-known Apriori algorithm [1] was adapted for the problem of finding patterns of the form `*AB**C`, where `A`, `B`, and `C` are specific symbols (e.g., event types) and `*` could be any symbol ($T = 6$, in this example). This pattern may not repeat itself in every period, but it must appear at least *min_sup* times, where *min_sup* is a user-defined parameter. In [5], a faster mining method for this problem was also proposed, which uses a tree structure to count the support of multiple patterns at two database scans. [7] studies the problem of finding sets of events that appear together periodically. In each qualifying period, the set of events may not appear in exactly the same positions, but their occurrence may be shifted or disrupted, due to the presence of noise. However, this work does not consider the order of events in such patterns. On the other hand, it addresses the problem of mining patterns and their periods automatically. Finally, [15] studies the problem of finding patterns, which appear in at least a minimum number of consecutive periodic intervals and groups of such intervals are allowed to be separated by at most a time interval threshold.

A number of spatial access methods, which are variants of the R–tree [3] have developed for the management of moving object trajectories. [10] proposes 3D variants of this access method, suitable for indexing historical spatiotemporal data. Time is modeled as a third dimension and each moving object trajectory is mapped to a polyline in this 3D space. The polyline is then decomposed into a sequence of 3D line segments, tagged with the object-id they correspond to. The segments, in turn, are indexed by variants of the 3D R–tree, which differ in the criteria they use to split their nodes. Although this generic method is always applicable, it stores redundant information if the positions of the objects do not constantly change. Other works [13, 4] propose multi-version variants of the R–tree, which share similar concepts to access methods for time-evolving data [11]. Recently [12], there is an increasing interest in (approximate) aggregate queries on spatiotemporal data, e.g., "find the distinct number of objects that were in region r during a specific time interval".

3. PERIODIC PATTERNS IN OBJECT TRAJECTORIES

In our model, we assume that the locations of objects are sampled over a long history. In other words, the movement of an object is tracked as an n-length sequence $\mathcal{S}$ of spatial locations, one for each timestamp in the history, of the form $\{(l_0, t_0), (l_1, t_1), \ldots, (l_{n-1}, t_{n-1})\}$, where l_i is the object's location at time t_i. If the difference between consecutive timestamps is fixed (locations are sampled every regular time interval), we can represent the movement by a simple sequence of locations l_i (i.e., by dropping the timestamps t_i, since they can be implied). Each location l_i is expressed in terms of spatial coordinates. Figure 1a, for example, illustrates the movement of an object in three consecutive days (assuming that it is tracked only during specific hours, e.g., working hours). We can model it with sequence $\mathcal{S} = \{\langle 4, 9\rangle, \langle 3.5, 8\rangle, \ldots, \langle 6.5, 3.9\rangle, \langle 4.1, 9\rangle, \ldots\}$. Given such a sequence, a minimum support *min_sup*, and an integer T, called *period*, our problem is to discover movement patterns that repeat themselves every T timestamps. A discovered pattern P is a T-length sequence of the form $r_0 r_1 \ldots r_{T-1}$, where r_i is a *spatial region* or the special character `*`, indicating the whole spatial universe. For instance, pattern `AB*C**` implies that at the beginning of the cycle the object is in region `A`, at the next timestamp it is found in region `B`, then it moves irregularly (it can be anywhere), then it goes to region `C`, and after that it can go anywhere, until the beginning of the next cycle, when it can be found again in region `A`. The patterns are required to be followed by the object in at least *min_sup* periodic intervals in $\mathcal{S}$.

Existing algorithms for mining periodic patterns (e.g., [5]) operate on event sequences and discover patterns of the above form. However, in this case, the elements r_i of a pattern are events (or sets of events). As a result, we cannot directly apply these techniques for our problem, unless we treat the exact locations l_i as discrete categorical values. Nevertheless it is highly unlikely that an object will repeat an identical sequence of $\langle x, y\rangle$ locations precisely. Even if the spatial route is precise, the location transmissions at each timestamp are unlikely to be perfectly synchronized. Thus, the object will not reach the same location at the same time every day, and as a result the sampled locations at specific timestamps (e.g., at 9:00 a.m. sharp, every day), will be different. In Figure 1a, for example, the first daily locations of the object are very close to each other, however, they will be treated differently by a straightforward mining algorithm.



events sequence:
A A C C C G | A A C B D G | A A A C H G

some partial periodic patterns:
support(AA***G) = 3
support(AAC**G) = 2
support(AA*C*G) = 2

(a) an object's movement (b) a set of predefined regions (c) event-based patterns

Figure 1: Periodic patterns in with respect to pre-defined spatial regions

One way to handle the noise in object movement is to replace the exact locations of the objects by the regions (e.g., districts, mobile communication cells, or cells of a synthetic grid) which contain them. Figure 1b shows an example of an area's division into such regions. Sequence {`A`, `A`, `C`, `C`, `C`, `G`, `A`,...} can now summarize the object's movement and periodic sequence pattern mining algorithms, like [5], can directly be applied. Figure 1c shows three (closed) discovered patterns for T=6, and $min\_sup = 2$. A disadvantage of this approach is that the discovered patterns may not be very descriptive, if the space division is not very detailed. For example, regions `A` and `C` are too large to capture in detail the first three positions of the object in each periodic instance. On the other hand, with detailed space divisions, the same (approximate) object location may span more than one different regions. For example, in Figure 1b, observe that the third object positions for the three days are close to each other, however, they fall into different regions (`A` and `C`) at different days. Therefore, we are interested in the *automated* discovering of patterns *and their descriptive regions*. Before we present methods for this problem, we will first define it formally.

3.1 Problem definition

Let $\mathcal{S}$ be a sequence of n spatial locations $\{l_0, l_1, \dots, l_{n-1}\}$, representing the movement of an object over a long history. Let $T \ll n$ be an integer called *period* (e.g., day, week, month). A *periodic segment* s is defined by a subsequence $l_i l_{i+1} \dots l_{i+T-1}$ of $\mathcal{S}$, such that i modulo $T = 0$. Thus, segments start at positions $0, T, \dots, (\lfloor \frac{n}{T} \rfloor - 1) \cdot T$, and there are exactly $m = \lfloor \frac{n}{T} \rfloor$ periodic segments in $\mathcal{S}$.* Let s^j denote the segment starting at position $l_{j \cdot T}$ of $\mathcal{S}$, for $0 \le j < m$, and let $s_i^j = l_{j \cdot T + i}$, for $0 \le i < T$. A *periodic pattern* P is defined by a sequence $r_0 r_1 \dots r_{T-1}$ of length T, such that r_i is either a spatial region or $*$. The *length* of a periodic pattern P is the number of non-$*$ regions in P. A segment s^j is said to *comply with* P, if for each $r_i \in P$, $r_i = *$ or s_i^j is *inside* region r_i. The *support* $|P|$ of a pattern P in $\mathcal{S}$ is defined by the number of periodic segments in $\mathcal{S}$ that comply with P. We sometimes use the same symbol P to refer to a pattern and the set of segments that comply with it. Let $min\_sup \le m$ be a positive integer (*minimum support*). A pattern P is *frequent*, if its support is larger than min_sup.

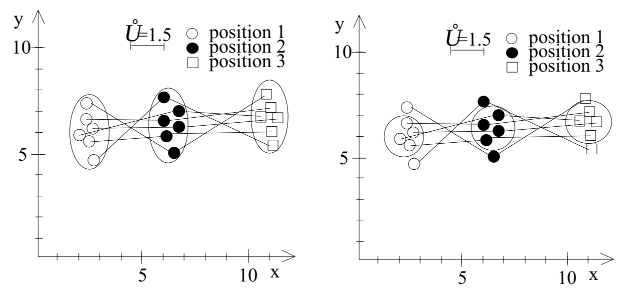
*If n is not a multiple of T, then the last n modulo T locations are truncated and the length n of sequence $\mathcal{S}$ is reduced accordingly.

A problem with the definition above is that it imposes no control over the density of the pattern regions r_i. In other words, if the pattern regions are too relaxed (e.g., each r_i is the whole map), the pattern may always be frequent. Therefore, we impose an additional constraint as follows. Let $\mathcal{S}^P$ be the set of segments that comply with a pattern P. Then each region r_i of P is *valid* if the set of locations $R_i^P := \{s_i^j \mid s^j \in \mathcal{S}^P\}$ form a *dense cluster*. To define a dense cluster, we borrow the definitions from [2] and use two parameters ϵ and $MinPts$. A point p in the spatial dataset R_i^P is a *core* point if the circular range centered at p with radius ϵ contains at least $MinPts$ points. If a point q is within distance ϵ from a core point p, it is assigned in the same cluster as p. If q is a core point itself, then all points within distance ϵ from q are assigned in the same cluster as p and q. If R_i^P forms a single, dense cluster with respect to some values of parameters ϵ and $MinPts$, we say that region r_i is valid. If all non-`*` regions of P are valid, then P is a valid pattern. We are interested in the discovery of valid patterns only. In the following, we use the terms *valid region* and *dense cluster* interchangeably; i.e., we will often use the term dense region to refer to a spatial dense cluster and the points in it.

Figure 2a shows an example of a valid pattern, if $\epsilon = 1.5$ and $MinPts = 4$. Each region at positions 1, 2, and 3 forms a single, dense cluster and is therefore a dense region. Notice, however, that it is possible that two valid patterns P and P' of the same length (i) have the same `*` positions, (ii) every segment that complies with P', complies with P, and (iii) $|P'| < |P|$. In other words, P *implies* P'. For example, the pattern of Figure 2a implies the one of Figure 2b (denoted by the three circles). A frequent pattern P' is *redundant* if it is implied by some other frequent pattern P. The **mining periodic patterns problem** searches for all valid periodic patterns P in $\mathcal{S}$, which are frequent and non-redundant with respect to a minimum support min_sup. For simplicity, we will use 'frequent pattern' to refer to a valid, non-redundant frequent pattern.

3.2 Mining periodic patterns

In this section, we present techniques for mining frequent periodic patterns and their associated regions in a long history of object trajectories. We first address the problem of finding frequent 1-patterns (i.e., of length 1). Then, we propose two methods to find longer patterns; a bottom-up, level-wise technique and a faster top-down approach.

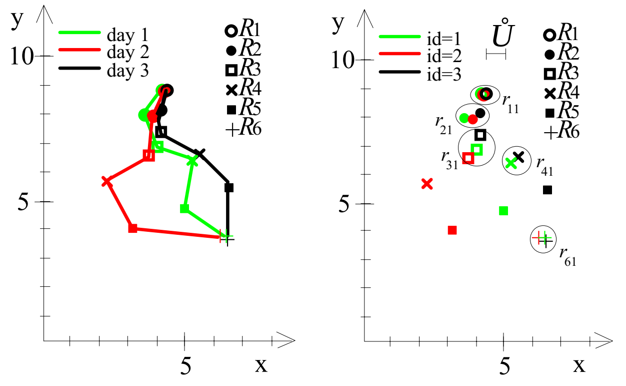


(a) a valid pattern (b) a redundant pattern

Figure 2: Redundancy of patterns

3.2.1 Obtaining frequent 1-patterns

Including automatic discovery of regions in the mining task does not allow for the direct application of techniques that find patterns in sequences (e.g., [5]), as discussed. In order to tackle this problem, we propose the following methodology. We divide the sequence $\mathcal{S}$ of locations into T spatial datasets, *one for each offset* of the period T. In other words, locations $\{l_i, l_{i+T}, \ldots, l_{i+(m-1)\cdot T}\}$ go to set R_i, for each $0 \leq i < T$. Each location is tagged by the id $j \in [0, \ldots, m-1]$ of the segment that contains it. Figure 3a shows the spatial datasets obtained after decomposing the object trajectory of Figure 1a. We use a different symbol to denote locations that correspond to different periodic offsets and different colors for different segment-ids.



(a) T-based decomposition (b) dense clusters in R_i's

Figure 3: locations and regions per periodic offset

Observe that a dense cluster r in dataset R_i corresponds to a frequent pattern, having $*$ at all positions and r at position i. Figure 3b shows examples of five clusters discovered in datasets R_1, R_2, R_3, R_4, and R_6. These correspond to five 1-patterns (i.e., $r_{11}*****$, $*r_{21}****$, etc.). In order to identify the dense clusters for each R_i, we can apply a density-based clustering algorithm like DBSCAN [2]. Clusters with less than *min_sup* points are discarded, since they are not frequent 1-patterns according to our definition.

Clustering is quite expensive and it is a frequently used module of the mining algorithms, as we will see later. DBSCAN [2] has quadratic cost to the number of clustered points, unless an index (e.g., R–tree) is available. Since R–trees are not available for every set of arbitrary points to be clustered, we use a hash-based method, that divides the 2D space using a regular grid with cell area $\epsilon\sqrt{2} \times \epsilon\sqrt{2}$. This grid is used to hash the points into buckets according to the cell that contains them. The rationale of choosing this cell size is that if one cell contains at least $MinPts$ points, we know for sure that it is dense and need not perform any range queries for the objects in it. The remainder of the algorithm merges dense cells that contain points within distance ϵ (using inexpensive minimum bounding rectangle tests or spatial join, if required) and applies ϵ-range queries from objects located in sparse cells to assign them to clusters and potentially merge clusters. Our clustering technique is fast because not only does it avoid R–tree construction, but it also minimizes expensive distance computations. The details of this algorithm are omitted for the sake of readability.

3.2.2 A level-wise, bottom-up approach

Starting from the discovered 1-patterns (i.e., clusters for each R_i), we can apply a variant of the level-wise Apriori-TID algorithm [1] to discover longer ones, as shown in Figure 4. The input of our algorithm is a collection $\mathcal{L}_1$ of frequent 1-patterns, discovered as described in the previous paragraph; for each R_i, $0 \leq i < T$, and each dense region $r \in R_i$, there is a 1-pattern in $\mathcal{L}_1$. Pairs $\langle P_1, P_2 \rangle$ of $(k-1)$-patterns in $\mathcal{L}_{k-1}$, with their first $k-2$ non-$*$ regions in the same position and different $(k-1)$-th non-$*$ position create candidate k-patterns (lines 4–6). For each candidate pattern P_{cand}, we then perform a segment-id join between P_1 and P_2 and if the number of segments that comply with both patterns is at least *min_sup*, we run a pattern validation function to check whether the regions of P_{cand} are still clusters. After the patterns of length k have been discovered, we find the patterns at the next level, until there are no more patterns at the current level, or there are no more levels.

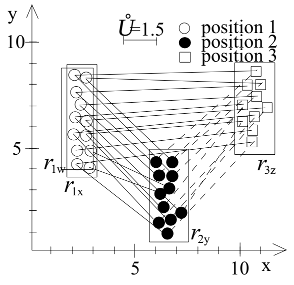
```
Algorithm STPMine1($\mathcal{L}_1$, $T$, $min\_sup$);
1).  $k$:=2;
2).  while ($\mathcal{L}_{k-1} \neq \emptyset \wedge k < T$)
3).      $\mathcal{L}_k$:=$\emptyset$;
4).      for each pair of patterns $(P_1, P_2) \in \mathcal{L}_{k-1}$
5).              such that $P_1$ and $P_2$ agree on the first $k-2$
6).              and have different $(k-1)$-th non-$*$ position
7).          $P_{cand}$:=candidate_gen($P_1$, $P_2$);
8).          if ($P_{cand} \neq null$) then
9).              $P_{cand}$:=$P_1 \bowtie_{P_1.sid=P_2.sid} P_2$; //segment-id join
10).             if $|P_{cand}| \geq min\_sup$ then
11).                 validate_pattern($P_{cand}$, $\mathcal{L}_k$, $min\_sup$);
12).     $k$:=$k+1$;
13). return $\mathcal{P}$:=$\bigcup \mathcal{L}_k, \forall 1 \leq k < T$;
```

Figure 4: Level-wise pattern mining

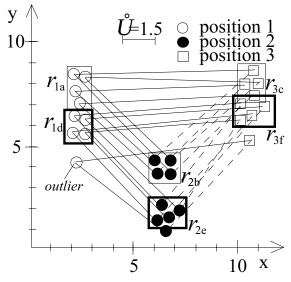
In order to facilitate fast and effective candidate generation, we use the MBRs (i.e., *minimum bounding rectangles*) of the pattern regions. For each common non-$*$ position i the intersection of the MBRs of the regions for P_1 and P_2 must be non-empty, otherwise a valid superpattern cannot exist. The intersection is adopted as an approximation for the new pattern P_{cand} at each such position i. During candidate pruning, we check for every $(k-1)$-subpattern of P_{cand} if there is at least one pattern in $\mathcal{L}_{k-1}$, which agrees in the non-$*$ positions with the subpattern and the MBR-intersection with it is non-empty at all those positions. In such a case, we accept P_{cand} as a candidate pattern. Otherwise, we know that P_{cand} cannot be a valid pattern, since some of its subpatterns (with common space covered by the non-$*$ regions) are not included in $\mathcal{L}_{k-1}$.

Function **validate_pattern** takes as input a k-length candidate pattern P_{cand} and computes a number of actual k-length patterns from it. The rationale is that the points at all non-$*$ positions of P_{cand} may not form a cluster anymore after the join of P_1 and P_2. Thus, for each non-$*$ position of P_{cand} we re-cluster the points. If for some position the points can be grouped to more than one clusters, we create a new candidate pattern for each cluster and validate it. Note that, from a candidate pattern P_{cand}, it is possible to generate more than one actual patterns eventually. If no position of P_{cand} is split to multiple clusters, we may need to re-cluster the non-$*$ positions of P_{cand}, since some points (and segment-ids) may be eliminated during clustering at some position.

To illustrate the algorithm, consider the 2-length patterns $P_1 = r_{1x}r_{2y}*$ and $P_2 = r_{1w}*r_{3z}$ of Figure 5a. Assume that $MinPts = 4$ and $\epsilon = 1.5$. The two patterns have common first non-$*$ position and $MBR(r_{1x})$ overlaps $MBR(r_{1w})$. Therefore, a candidate 3-length pattern P_{cand} is generated. During candidate pruning, we verify that there is a 2-length pattern with non-$*$ positions 2 and 3 which is in $\mathcal{L}_2$. Indeed, such a pattern can be spotted at the figure (see the dashed lines). After joining the segment-ids in P_1 and P_2 at line 9 of STPMine1, P_{cand} contains the trajectories shown in Figure 5b. Notice that the locations of the segment-ids in the intersection may not form clusters any more at some positions of P_{cand}. This is why we have to call **validate_pattern**, in order to identify the valid patterns included in P_{cand}. Observe that, the segment-id corresponding to the lowermost location of the first position is eliminated from the cluster as an outlier. Then, while clustering at position 2, we identify two dense clusters, which define the final patterns $r_{1a}r_{2b}r_{3c}$ and $r_{1d}r_{2e}r_{3f}$.



(a) 2-length patterns



(b) generated 3-length patterns

Figure 5: Example of STPMine1

3.2.3 A two-phase, top-down algorithm

Although the algorithm of Figure 4 can find all partial periodic patterns correctly, it can be very slow due to the huge number of region combinations to be joined. If the actual patterns are long, all their subpatterns have to be computed and validated. In addition, a potentially huge number of candidates need to be checked and evaluated. In this section, we propose a top-down method that can discover long patterns more efficiently.

After applying clustering on each R_i (as described in Section 3.2.1), we have discovered the frequent 1-patterns with their segment-ids. The first phase of STPMine2 algorithm replaces each location in $\mathcal{S}$ with the cluster-id it belongs to or with an 'empty' value (e.g., $*$) if the location belongs to no cluster. For example, assume that we have discovered clusters $\{r_{11}, r_{12}\}$ at position 1, $\{r_{21}\}$ at position 2, and $\{r_{31}, r_{32}\}$ at position 3. A segment $\{l_1, l_2, l_3\}$, such that $l_1 \in r_{12}$, $l_2 \notin r_{21}$, and $l_3 \in r_{31}$ is transformed to subsequence $\{r_{12}*r_{31}\}$. Therefore, the original spatiotemporal sequence $\mathcal{S}$ is transformed to a symbol-sequence $\mathcal{S}'$.

Now, we could use the mining algorithm of [5] to discover fast all frequent patterns of the form $r_0r_1 \ldots r_{T-1}$, where each r_i is a cluster in R_i or $*$. However, we do not know whether the results of the sequence-based algorithm are actual patterns, since the contents of each non-$*$ position may not form a cluster. For example, $\{r_{12}*r_{31}\}$ may be frequent, however if we consider only the segment-ids that qualify this pattern, r_{12} may no longer be a cluster or may form different actual clusters (as illustrated in Figure 5). We call the patterns P' which can be discovered by the algorithm of [5] *pseudopatterns*, since they may not be valid.

To discover the actual patterns, we apply some changes in the original algorithm of [5]. While creating the *max-subpattern tree*, we store with each tree node the segment-ids that correspond to the pseudopattern of the node after the transformation. In this way, one segment-id goes to exactly one node of the tree. However, $\mathcal{S}$ could be too large to manage in memory. In order to alleviate this problem, while scanning $\mathcal{S}$, for every segment s we encounter we perform the following operations.

- First, we insert the segment to the max-subpattern tree, as in [5], increasing the counter of the candidate pseudopattern P' that s corresponds to after the transformation. An example of such a tree is shown in Figure 6. This node can be found by finding the (first) maximal pseudopattern that is a superpattern of P' and following its children, recursively. If the node corresponding to P' does not exist, it is created (together with any non-existent ancestors). Notice that the dotted lines are not implemented and not followed during insertion (thus, we materialize the tree instead of a lattice). For instance, for segment with $P' = \{*r_{21}r_{31}\}$, we increase the counter of the corresponding node at the second level of the tree.
- Second, we insert an entry $\langle P'.id, s.sid\rangle$ to a file F, where $P'.id$ is the id of the node of the lattice that corresponds to pseudopattern P' and $s.sid$ is the id of segment s. At the end, file F is sorted on $P'.id$ to bring together segment-ids that comply to the same (maximal) pseudopattern. For each pseudopattern with at least one segment, we insert a pointer to the file position, where the first segment-id is located. Nodes of the tree are labeled in breadth-first search order for reasons we will explain shortly.

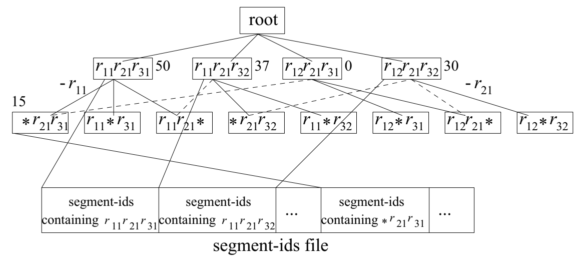


Figure 6: Example of max-subpattern tree

Now, instead of finding frequent patterns in a bottom-up fashion, we traverse the tree in a top-down, breadth-first order. For every pseudopattern with at least min_sup segment-ids, we apply the **validate_pattern** function of Figure 4 to recover potential valid patterns. All segment-ids that belong to a discovered pattern are removed from the current pseudopattern. The rationale is that we are interested in patterns that are not spatially contained in some superpattern, so we use only those segment-ids that are not included in a pattern to verify subpatterns of it.

Thus, after scanning the first level of the lattice, we may have discovered some patterns and we may have shrunk segment-id lists of the pseudopatterns. Then, we move to the next level of the lattice. The support of a pseudopattern P' at each level is the recorded support of P' plus the supports of all its superpatterns (recall that a segment-id is assigned to the *maximal* pattern it complies with). The supports of the superpatterns can be immediately accessed from the lattice. If the total support of the candidate is at least min_sup, then the segment-ids have to be loaded for application of **validate_pattern**. The segment-ids of a superpattern may already be in memory from previous level executions. If not, they are loaded from the file F. After validation, only the disqualified segment-ids are kept to be used at lower level patterns. Traversal continues until there are no more patterns or it is not possible to find more patterns at lower levels of the lattice.

The fact that segment-ids are clustered in F according to the breadth-first traversal of the lattice, minimizes random accesses and restricts the number of loaded blocks to memory. The segment-ids for a superpattern remain in memory to be used at lower level validations. If we run out of memory, the segment-ids of the uppermost lattice levels are rewritten to disk, but this time possibly to a smaller file if there were some deletions.

A pseudo code for STPMine2 is shown in Figure 7. Initially, the tree and the segment-ids file are created and linked. Then for each level, we find the support of a pseudopattern $|P'|$ at level k by accessing only the supports of its superpatterns $P'' \supset P$ at level $k+1$, since we are accessing the tree in breadth first order. If $|P'| \geq min\_sup$, we validate the pattern as in STPMine1 and if some pattern is discovered, we remove from P' all those segment-ids that comply with the discovered pattern. Thus, the number of segment-ids decrease as we go down the levels of the tree, until it is not possible to discover any more patterns, or there are no more levels. Notice that the patterns discovered here are only maximal, as opposed to STPMine1, which discovers *all* frequent patterns. However, we argue that maximal patterns are more useful, compared to the huge set of all patterns. In addition, as we show in the experimental section, STPMine2 is much faster than STPMine1 for data, which contain long patterns.

```
Algorithm STPMine2($\mathcal{L}_1$, $T$, $min\_sup$);
1).  build max-subpattern tree $\mathcal{T}$ and pattern-file $F$;
2).  sort $F$ on $P'.id$ and connect it to the nodes of $\mathcal{T}$;
3).  for $k = T$ downto 2
4).      for each pattern $P'$ at level $k$ of $\mathcal{T}$
5).          $|P'|$:=$P'.counter + \sum_{P'' \supset P', length(P'')=k+1} |P''|$;
6).          if $|P'| \geq min\_sup$ then
7).              $P_{cand}$:=$\bigcup_{P'' \supseteq P'} P''.sids$;
8).              validate_pattern($P_{cand}, \mathcal{L}, min\_sup$);
9).              if $\mathcal{P}$ has changed then
10).                 remove from $P'$ those $sids$ in new patterns of $\mathcal{P}$;
11).                 if unassigned $sids$ less than $min\_sup$ then
12).                     return $\mathcal{P}$;
13). return $\mathcal{P}$;
```

Figure 7: Top-down pattern mining

3.3 Mining shifted/distorted patterns

We have discussed how to deal with non-rigid pattern instances in space, using clustering. However, pattern instances may be also shifted/distorted in time. For example, even though Bob follows more or less the same route from his house to his work, some days he may delay, because he wakes up later than usual, or due to traffic. Shifted/distorted pattern instances can be counted by our algorithms, as follows. For a single object location at offset position i, instead of generating a single point in the corresponding R_i, as before, we generate a point at all neighbor offset positions $R_{(i-\tau)\ mod\ T}$, $R_{(i-\tau+1)\ mod\ T}$, ..., $R_{(i+\tau)\ mod\ T}$, where τ is a maximum shifting/distortion threshold. Consider, for instance, the 5th position of day 1, in Figure 3a and assume that $\tau = 1$. Instead of generating a single '■' point at that location, we generate one '■' point (to file R_5), one '+' point (to file R_4), and one '×' point (to file R_6). All these points have the same coordinates, but they are considered part of multiple periodic positions. In other words, there is a data replication with a factor $2 \cdot \tau + 1$, however, this ensures that shifted patterns will be counted in the supports of the actual positions. In practice, most of the replicated points will be discarded as noise, after the discovery of the 1-patterns, thus the overhead will not increase significantly. Obviously, segment-ids that appear multiple times in the same pattern (due to point replication), are counted only once. As a variant of this idea we can *weigh* the replicated points with a number anti-proportional to their distance from their actual temporal positions, in order to penalize distortion and increase accuracy.

4. INDEXING USING PERIODIC PATTERNS

The aim of a good spatio-temporal index structure is to manage the trajectories of a set of m moving objects $S = \{p_1, p_2, \ldots, p_m\}$, in order to efficiently process spatiotemporal range queries, like "find all objects which were in Central district between 2:00 p.m. and 3:00 p.m. yesterday". In this section, we present an indexing method that exploits the discovered periodic patterns. First, we present the proposed index structures and then we discuss query processing algorithms that use them.

4.1 Indexing scheme

For each object $p \in S$, we first apply the mining techniques to extract their periodic patterns and then organize all of them (for all objects) into a special index structure called *Period Index* (PI). Only objects that follow periodic patterns are stored in the PI and each pattern is stored only once. In particular, the PI consists of the following two structures; (i) a structure called *Pattern Index* that stores, for each object $p \in S$ a concise representation of its periodic pattern(s) P and (ii) an index called *Location Index* that stores for each object p with some pattern(s) in PI, the *actual locations* of p. Furthermore, we use a traditional spatiotemporal index (e.g., a 3D R–tree [10]) to store the locations of objects that do not follow any periodic movement (outliers). We call this structure the *Exception Index* (EI). We expect that EI is small (compared to the database size); otherwise the dataset is not periodic, in which case our technique degrades to a traditional spatiotemporal indexing method. Next, we discuss possible implementations of the Pattern and Location Indexes.

Pattern Index: Consider a periodic pattern $P = r_0 r_1 \ldots r_{T-1}$ for an object p. Let us assume for now that there is no $*$ position in P and the patterns for all periodic objects have the same length T (we discuss the more general case later). For each valid region $r_i \in P$, we compute the two dimensional *region MBR* M_i that encloses this region (e.g., see Figure 5). The area of each M_i is expected to be small, depending on the density parameters ϵ and $MinPts$ that are used in the clustering phase. An important property of each M_i is that it encloses all locations with offset i that belong to segments which comply with P. That is, for each location l_j of p at timestamp j, we have $l_j \in M_i$, where $i = j \ mod \ T$. Of course that is true only if l_j belongs to a periodic segment. The Pattern Index is a two dimensional index (in our case a 2D R–tree [3]) that contains all the region MBRs M_i for all periodic objects. In addition, with each M_i of an object p, we store the offset i and the object-id $p.oid$.

Location Index: In this index, we store the locations of all the periodic objects in the database. One approach is to implement it using a hash table indexed on the object ids. Each entry $h(p.oid)$ in the hash table contains the period T of the object p and a pointer to the first disk page that contains the locations of p. The locations are organized as an array ordered by the location timestamps and stored in sequential disk pages, e.g., in the following order: $l_0, l_1, l_2, \ldots, l_{n-1}$. Therefore, to find the location l_t of p at a specific timestamp t, we just need to calculate the disk page that contains this entry, which can be done in constant time. The size of the hash table is proportional to the number of objects and for typical applications this table can be easily kept in main memory.

In general, the patterns discovered by the data mining algorithms may contain $*$ positions. All these locations are considered outliers and they are inserted into EI. Therefore, no MBRs for these elements are inserted into PI. In addition, the locations of the segments of periodic objects that do not comply with the periodic pattern are inserted into EI. Note that if the Location Index is implemented using the hash-based approach, the above insertions introduce some replication since they are stored in both indexes. However, the redundancy is expected to be small, especially if the discovered patterns have high support.

4.2 Query processing

Here we discuss how to evaluate spatiotemporal range queries using the Period and Exception Index structures discussed above. Given a query region in space q_R and time interval $q_T = [t_s, t_e]$, we are interested in finding the objects that are contained in q_R at some point during q_T.

The query processing algorithm first runs the (spatiotemporal) query on the Exception Index and retrieves the objects that satisfy the query. Let A be the set of these objects. The next step, is to run the query on the Pattern index using only the spatial extent q_R. For each region MBR that intersects the query, we keep the object id and the offset of this MBR. Let B be the set of objects that correspond to these MBRs. We compute the set $C = B - A$ (set difference). This set contains all the objects that must be checked using the Location Index in order to validate if a particular object satisfies the query. Indeed, if an object appears in A, it means that it has already been discovered using the Exception Index, and therefore it does not need to be checked again. Using the hash table, for each object $p \in C$, we compute the disk page that contains the location of p at t_s. Next, we examine the consecutive locations of this object sequentially, starting from t_s until a qualifying location is found or the end of the query time interval t_e is reached. Finally, the answer is the union of A and the objects from C that pass the verification step.

4.3 Other compression/indexing schemes

In some applications the vast amount of historical spatiotemporal data can render their storage in secondary memory devices impractical. Typically, very old data are deleted, or in the best case simply archived in sequential tertiary storage devices (e.g., tapes), making efficient search impossible. Instead of storing the actual object movements, we can *compress* this information by keeping only the patterns, their occurrences, and the exception movements. More specifically, for each object we keep (i) the periodic patterns, (ii) the segment-ids that comply with each pattern, and (iii) the exception movements. This information can be managed using the structures described in the previous section, but now the exact locations corresponding to periodic positions (i.e., the Location Index) are discarded. This can greatly reduce the storage requirements. Essentially, query evaluation becomes inexact since a set of locations corresponding to a non-$*$ offset position are approximated by their MBR, but the error is small due to the descriptiveness of the patterns.

Another, more aggressive approach is to discard the exceptions, as well. The lightweight Pattern Index described in Section 4.1 can be used by itself to provide approximate answers to queries referring to the past movements of objects. If for each periodic pattern of an object we also store its validity lifetime, the Period Index could filter objects not only based on their spatial relationship with the query, but using the temporal dimension as well.

5. EXPERIMENTAL EVALUATION

We implemented and evaluated the mining and indexing techniques presented in the paper. The language used was C++ and the experiments were performed on a Pentium III 700MHz workstation with 1GB of memory, running Unix.

In order to test the effectiveness and efficiency of the techniques under various conditions, we designed a generator for long object trajectories which exhibit periodicity according to a set of parameter values. These parameters are the length n of the time history (in timestamps), the period T, the length ℓ of the maximal frequent patterns followed by the object ($\ell \leq T$), and a probability f for a periodic segment in the object's movement to comply with no hidden patterns (i.e., the movement during this segment is irregular).

Before generating the movement, the approximate regions for the maximal periodic patterns are determined. Let P be a generated pattern. A random circular route is generated in space and for each non-$*$ position i in P, a spatial location l_{P_i} (i.e., point) on that route is determined, such that the distance between two non-$*$ positions on the route is proportional to their temporal distance in the pattern. Afterwards, the movement of the object is generated. For every periodic segment s, we initially determine whether s should be a noise (i.e., irregular) segment or not, given the probability f.

If s is a regular segment, a random maximal pattern P is selected and the object's movement is generated as follows. If the next segment location to be generated corresponds to a non-$*$ position i of P, the location l_i is generated randomly and within a distance E from the spatial location l_{P_i} of the non-$*$ position. E ranges from 0 to 2% of the map size. Otherwise (i.e., l corresponds to an $*$ position), l_i is generated randomly, but such that the movement is "targeted" to the next periodic location. In other words, (i) l_i "moves" with respect to the previous segment location l_{i-1} towards the next non-$*$ position j and (ii) its distance from the previous location l_{i-1} is the spatial distance between l_{i-1} and l_{P_j} divided by $j-i+1$, i.e., the temporal distance between these two positions. In order to prevent regular movements, both the distance and direction angle are distorted. In specific, we add to the angle (in radians) a random number in $[-1,1]$ and the distance is multiplied by a number between $[1.5, 0.8]$.[†]

If s is a noise segment, the object can move everywhere in space. The movement is determined by a random direction angle (with respect to the previous location) and a random distance in $[0, maxwalk]$, where $maxwalk$ is used to control the maximum "walking" distance of the object between two timestamps. In order to avoid extreme jumps, after half of the movements in a noise segment, the rest are generated to "target" to the next periodic position, using the method described above.

5.1 Mining Effectiveness

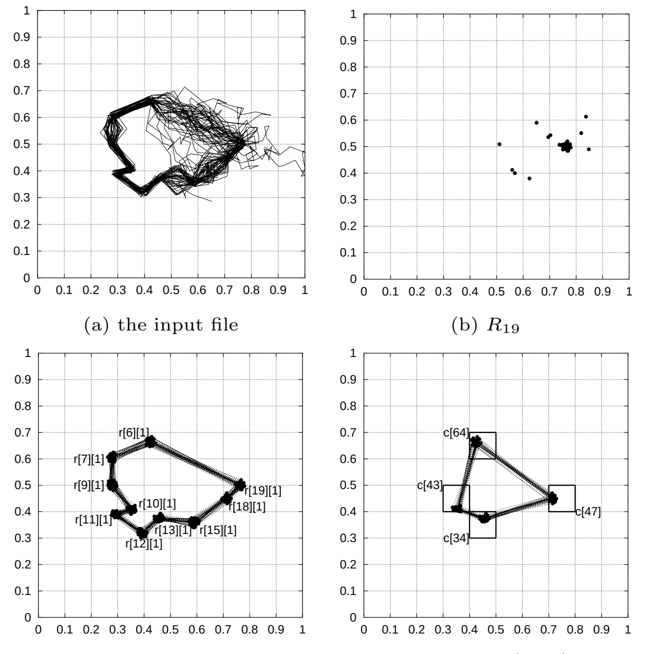
The first experiment demonstrates the effectiveness of the mining techniques proposed in Section 3.2. We generated a small problem, with $n = 1000$ (i.e., there are only 1000 locations in the object's trajectory). T is set to 20 and the object follows a single periodic pattern P at 39 out of 50 segments, whereas the movement is irregular in 11 segments. Figure 8a shows the objects trajectory, where the periodic movement can roughly be observed. For this dataset $\ell = 10$, i.e., there are 10 non-$*$ positions in P. Figure 8b shows the locations of the object if we consider only the last position in each of the 50 periodic segments. This corresponds to file R_{19}. A cluster, corresponding to a frequent 1-pattern, can easily be spotted on the figure.

Figure 8c shows the maximal frequent pattern P of length 10, successfully discovered by STPMine1 and STPMine2, when $min\_sup = 30$. The non-$*$positions are 6, 7, 9, 10, 11, 12, 13, 15, 18, and 19. We plot the object's movement, interpolated using only the non-$*$ positions. The discovered pattern is identical to the generated one. The dense regions are successfully detected by the clustering module, and the spatial extents of the pattern are minimal.

We also developed and tested a technique that applies directly the data mining algorithm for event sequence data [5]. The space is divided using a regular $M \times M$ grid. Then, each location of $\mathcal{S}$ is transformed to the cell-id which encloses the location. For instance, if we assume that all locations are in a unit $[0,1] \times [0,1]$ space, a location $l = \langle x, y \rangle$ is transformed to a cell with id $\lfloor y \cdot M \rfloor \cdot M + \lfloor x \cdot M \rfloor$. Then, we use the algorithm of [5] to find partial patterns that are described by cell-ids. We call this the *grid-based* mining method.

Figure 8d shows a maximal pattern P' discovered by this grid-based technique, when using a 10×10 grid. P' has the largest length among all discovered patterns, however it is only 4 (whereas the actual pattern P has 10 non-$*$ positions). The non-$*$ positions of P' are 6, 10, 13, and 18, captured by cells c_{64}, c_{43}, c_{34}, and c_{47}, respectively. Most frequent positions are lost because the locations in the respective clusters are split into more than one cells. For instance, the cluster of Figure 8b is split between cells c_{47} and c_{57}. Neither of these cells has higher support than min_sup for position 19, thus the frequent 1-pattern is missed. Other grid size/position settings also produce similar results; the pattern regions are either split and missed or found and overestimated by larger cells. From this small example, we can see the importance of discovering the periodic patterns *and their descriptive regions* effectively.

[†]These values were tuned to match realistic object movements and at the same time to disallow falsely generated periodic patterns.



(a) the input file (b) R_{19}

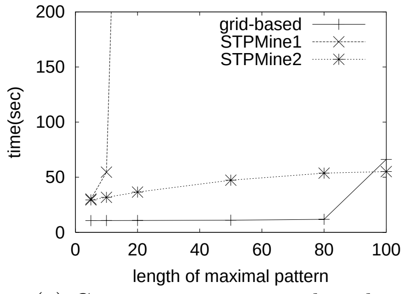
(c) P by STPMine ($|P| = 39$) (d) a grid-based P' ($|P'| = 30$)

Figure 8: Example of a dataset and discovered patterns ($n = 50, min\_sup = 30$)

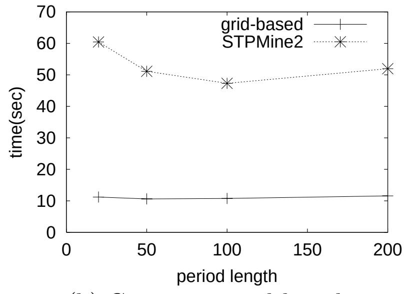
5.2 Mining Efficiency

In the next set of experiments, we validate the efficiency of the proposed techniques under various data settings. First, we compare the cost of the (ineffective) grid-based method, STPMine1, and STPMine2 as a function of the length of the maximal hidden pattern. We generated a sequence $\mathcal{S}$ of n=1M object locations, and set $T = 100$ and $min\_sup = 0.7 \cdot n$. For this and subsequent experiments we used $\epsilon = 0.005$ and $MinPts = 200$ in the clustering module.

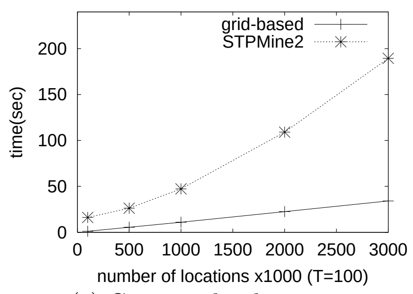
Figure 9a plots the results. Naturally, the grid-based approach is the fastest method, since it performs no clustering and no refinement of the discovered regions. However, as exemplified in the previous section, it misses the long patterns in all tested cases. Moreover, its efficiency is due to the fact that a large fraction of actual 1-patterns are missed and the search space is pruned. STPMine1 is very slow, when the hidden patterns are long. Like most bottom-up mining techniques, it suffers from the huge number of candidates that need to be generated and validated, and therefore it is inapplicable for the tested cases where the hidden patterns have more than 10 non-$*$ positions. STPMine2 is very efficient because it uses the first phase to identify fast large patterns



(a) Cost vs. max-pattern length



(b) Cost vs. period length



(c) Cost vs. database size

Figure 9: Mining efficiency under various conditions

that are potentially useful. Even when re-clustering fails for the maximal candidate patterns, the actual patterns are discovered usually only after few hops down the max-subpattern tree. Observe that, even though STPMine2 performs clustering a large number of times, it is not significantly slower than the ineffective grid-based approach. Interestingly, it outperforms the grid-based method when there is a single hidden pattern with length equal to T. In this case, the grid method spans many actual clusters between grid cells and splits the actual pattern to multiple maximal frequent patterns, the support of which is expensive to count in the large lattice.

In the next experiment, we test the effects of period length on the same database size, but with different values of T. The length of the maximal hidden pattern is $0.5 \cdot T$ in all cases. Again, n=1M and $min\_sup = 0.7 \cdot n$. Figure 9b compares the costs of the grid-based approach and STPMine2; we do not include the cost of STPMine1, since this method is very slow for long patterns. The figure shows that the costs of the two methods are almost invariant to T for a constant database size n. If T is small, then there are few, but large files to be clustered by STPMine1. On the other hand, for large T, there are many but small R_i to be clustered. In the final experiment, we test the scalability of STPMine2 to the length n of the spatiotemporal sequence $\mathcal{S}$. Figure 9c shows the costs of STPMine2 and the grid-based approach as a function of n, when $T = 100$ and the maximal pattern length is 50. Observe that STPMine2 is scalable, since the database size is only linearly related to the cost of finding and validating the maximal patterns. In summary, STPMine2 is an effective and efficient technique for mining periodic patterns *and* their accurate descriptive regions in spatiotemporal data.

5.3 Indexing effectiveness

To test the effectiveness of the Period Index scheme, we generated synthetic datasets of up to 200,000 objects. For every object we set $n = 1000$, whereas (unless otherwise specified) 80% of the objects follow a single periodic pattern with period $T = 10$ and length $l = 9$. The rest of the objects follow random walks. We also generated an assortment of query workloads. Every set consists of 100 range queries uniformly distributed in space that cover a fixed area q_R equal to 1% of the universe. We varied the temporal predicate q_T of the queries from 5 up to 20 time instants.

We implemented the *Periodic Index* (PI) by using a 2D R–tree as the *Pattern Index*, and a 3D R–tree as the *Exception Index*. We used a main memory hash table with pointers to the stored object data on disk for the *Location Index*. For comparison, we also implemented a 3D R–tree that indexes the trajectories without considering the periodic patterns.

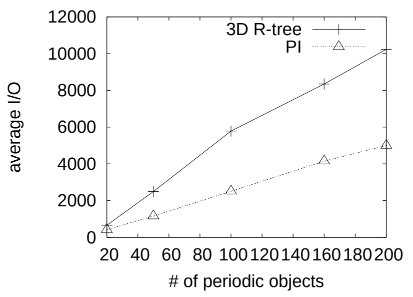
To compare the two approaches we count the average I/O cost per query for all query workloads. First, we ran a scale-up experiment for increasing dataset sizes. In Figure 10a we observe that PI scales well, yielding less than a 2-fold increase in average query I/O when doubling the size of the dataset. On the other hand, the 3D R–tree degrades slightly as the size of the database increases. For all cases, PI has at least 2 times fewer I/Os than the 3D R–tree.

Figure 10b shows how the index adapts to datasets with increasing number of object segments (60%–90%) that exhibit periodicity. Clearly, PI incurs much fewer I/Os compared to the 3D R–tree even for datasets that do not contain a very large number of periodic segments. Next, we tested how the algorithms adapt to datasets with increasing numbers of periodic objects. We generated datasets where 60% up to 90% of the objects follow periodic patterns, while the rest do random walks. Figure 10c plots the average query I/O. From the trend we can see that the fewer the periodic objects, the closer PI tends to the 3D R–tree and vice versa.

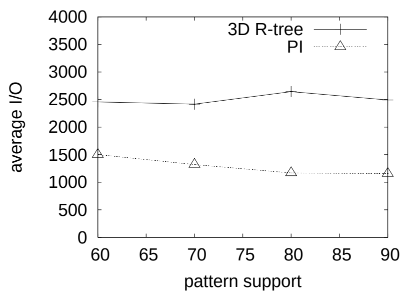
The next experiment tests the efficiency of the indexes with increasing query lengths q_T (see Figure 10d). It is apparent that PI is not affected at all from the length of the query time interval. On the other hand, the 3D R–tree degrades linearly. For $|q_T| = 20$, PI gives a 4-fold improvement over the 3D R–tree. This is expected, since the *Period Index* is a 2-dimensional R–tree that is independent of the time dimension. In contrast, the 3D R–tree needs to access an ever increasing number of nodes on the time dimension as the query becomes larger.

Finally, Figure 10e plots the total size of both structures (main memory and disk resident parts for PI) versus the percentage of periodic segments contained in the dataset. Furthermore, we plot the size of only the Pattern Index. As expected PI's size decreases as periodicity increases. The 3D R–tree, of course, is oblivious to the existence of object periodic patterns. The Pattern Index is orders of magnitude smaller than the total database size (16 MBs vs. 2.3GBs). It is worth pointing out that the Pattern Index is a small approximate (lossy) representation of the dataset and can be used by itself to answer approximate queries with some error guarantees based on the support of the object patterns.

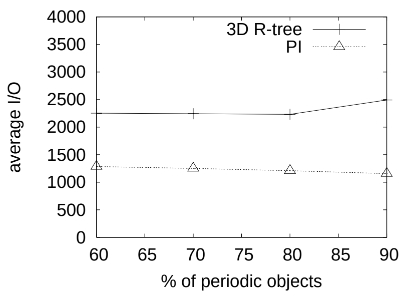
To conclude, PI yields a consistent improvement over the straightforward 3D R–tree (4-fold in the best case). For longer periodicities we expect PI's performance to further improve, since there are more chances for compressing hidden patterns.



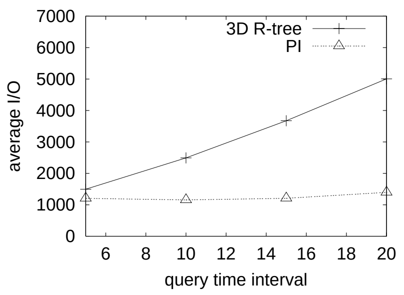
(a) Cost vs. # of periodic objects



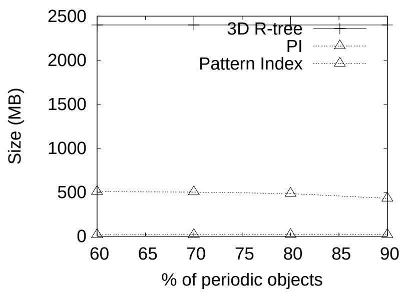
(b) Cost vs. pattern support



(c) Cost vs. % periodic objects



(d) Cost vs. length of q_T



(e) Size vs. % periodic objects

Figure 10: Comparison between Period Index and 3D R–tree

6. CONCLUSIONS

In this paper we presented a framework for mining partial periodic patterns from historical spatiotemporal data and use them to build an effective index for object movements. Our contributions can be summarized as follows:

- We define the important problem of periodic pattern mining in spatiotemporal databases. We identify several important applications of the mined patterns, including data management, data compression, approximate query processing, and probabilistic future movement prediction.
- We propose effective techniques for discovering the periodic patterns *and* their descriptive spatial regions from a long history of object movements. A top-down technique (STPMine2), in specific, is very efficient, having cost comparable to (ineffective) methods for event-sequence data.
- We propose an indexing scheme that uses the discovered patterns to effectively manage spatiotemporal data. As shown in the experiments, it is much faster compared to a conventional index that does not take periodicity into account.

7. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of Very Large Data Bases*, pages 487–499, 1994.

[2] M. Ester, H. P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. of ACM Knowledge Discovery and Data Mining*, pages 226–231, 1996.

[3] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM Management of Data*, pages 47–57, 1984.

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

[5] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *Proc. of International Conference on Data Engineering*, pages 106–115, 1999.

[6] P. Indyk, N. Koudas, and S. Muthukrishnan. Identifying representative trends in massive time series data sets using sketches. In *Proc. of Very Large Data Bases*, pages 363–372, 2000.

[7] S. Ma and J. L. Hellerstein. Mining partially periodic event patterns with unknown periods. In *Proc. of International Conference on Data Engineering*, pages 205–214, 2001.

[8] B. Özden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *Proc. of International Conference on Data Engineering*, pages 94–101, 1998.

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

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

[11] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.

[12] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-temporal aggregation using sketches. In *Proc. of International Conference on Data Engineering*, pages 449–460, 2004.

[13] Y. Tao and D. Papadias. MV3R–tree: A spatio-temporal access method for timestamp and interval queries. In *Proc. of Very Large Data Bases*, pages 431–440, 2001.

[14] I. Tsoukatos and D. Gunopulos. Efficient mining of spatiotemporal patterns. In *Proc. of Symposium on Advances in Spatial and Temporal Databases*, pages 425–442, 2001.

[15] J. Yang, W. Wang, and P. S. Yu. Mining asynchronous periodic patterns in time series data. In *KDD*, 2000.

[16] M. J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1/2):31–60, 2001.