Chapter 3

Graph Window Query: Neighborhood Analytics on Attributed Graphs

3.1 Introduction

In this chapter, we study the neighborhood analytics on large-scale attributed graphs. Attributed graphs are prevalently adopted to model real life information networks such social networks, biological networks and phone-call networks. In the attributed graph model, the vertexes correspond to objects and the edges capture the relationships between these objects. For instance, in social networks, every user is represented by a vertex and the friendship between two users is reflected by an edge connecting the vertexes. Besides, a user's profile is maintained as the vertex's attributes. Such graphs contain a wealth of valuable information which can be analyzed to discover interesting patterns [?, ?, ?, ?]. With the increasingly larger network sizes, it is becoming challenging to query, analyze and process these graph data. Therefore, there is an urgent call for effective and efficient mechanisms to draw out information over

graph data resources.

Recent advances in graph analytics such as graph aggregation [**?**, **?**] and summarization [**?**, **?**] focus on analyzing the entire graph as a whole. In fact, it is often useful to perform *neighborhood analytics* on graph data to analyze the vicinity of vertexes. That is for each vertex, the analytics is conducted over its *neighborhoods*. For instance, in a social network, it is important to detect a person's social influence among his/her social community. The "social community" of the person is essentially his/her neighborhood vertexes representing his/her friends.

Similar neighborhood concept has been supported by window functions [**?**, **?**] in relational data analytics. Instead of performing analysis (e.g., ranking and aggregate) over the entire data set, a window function returns for each tuple a value derived from its neighboring tuples. For instance, when finding each employee's salary ranking within every department, each tuple's neighbors are basically the tuples from the same department. However, the window definition in the relational context ignores graph structures which makes it unsuitable in the graph context. Therefore, we seek to utilize the general *neighborhood analytics* to formulate the notion of *graph windows*.

We have derived two graph windows from the perspective of neighborhood functions, which are referred to as *k-hop window* and *topological window*. The semantic of these windows are first demonstrated in the following two examples.

Example 3.1.1. *(k-hop window)* In a social network (such as LinkedIn and Facebook), users are normally modeled as vertexes and connectivity relationships are modeled as edges. In this scenario, a **distance neighborhood** function, such as 2-hop neighbors, represents the most relevant connections to each user. Some analytic queries such as summarizing related connections' distribution among different companies, and computing age distribution of the related friends can be useful. In order to answer these queries, computing the distance neighborhood is necessary.

Example 3.1.2. *(Topological window)* In biological networks (such as Argocyc, Ecocyc etc. [**?**]), genes, enzymes and proteins are vertexes and their dependencies in a pathway are edges. Because these networks are directed and acyclic, **comparison neighborhood** based on the ancestry relationship helps to reveal the influences among molecules. For instance, to find out the statistics of molecule in a protein production pathway, we can traverse the graph to find every molecule that is in its upstream. Then we summarize the number of genes and enzymes among those molecules. To answer such queries, computing the ancestry based comparison neighborhood is necessary.

To support these analytics in the above examples, we propose the Graph Window Query (GWQ in short) on attributed graphs. GWQ is a neighborhood analytics which aims to facilitate vertex-centric analysis. It supports two graph windows namely *k-hop window* and *topological window*. The k-hop window of a vertex is defined by its k-hop distance (e.g., friends-of-friends in Example 3.1.1). Thus it is essentially a **distance neighborhood**. On the other hand, the topological window of a vertex contains its ancestors (e.g., upstream molecules in Example 3.1.2). Hence, it is a **comparison neighborhood** based on the ancestry relationship.

To the best of our knowledge, existing graph databases or graph query languages do not directly support our proposed GWQ. There are two major challenges in processing GWQ. First, we need an efficient scheme to calculate the window of each vertex. Second, we need efficient solutions to process the aggregation over a large number of windows that may overlap. This offers opportunities to share the computation. However, it is non-trivial to address these two challenges.

For k-hop window query, the latest processing algorithm can be adopted from literature is EAGR [**?**]. EAGR leverages an overlay graph to represent the shared components among different windows. It incrementally construct the overlay graph through multiple iterations. In each iteration, it builds a Frequent-Pattern Tree [**?**] to

discover the largest shared component among vertex windows. However, to achieve efficient shared component detection, EAGR requires all vertex's k-hop neighbors to be pre-computed and resided in memory; otherwise EAGR incurs high performance overheads due to secondary storage accesses (e.g., disk I/Os). This limits the usage of EAGR in large-scale graphs. For instance, a LiveJournal social network graph[1] (4.8M vertexes, 69M edges) generates over 100GB neighborhood information for $k = 2$ in adjacency list representation. In addition, the overlay graph construction is not a one-time task, but periodically performed after a certain number of structural updates in order to maintain the quality. The high memory consumption renders the scheme impractical when k and the graph size increases.

In this chapter, we propose the *Dense Block Index (DBIndex)* to process the two graph window queries efficiently. Like EAGR, DBIndex seeks to exploit common components among different windows to salvage partial work done. However, different from EAGR, we identify the window similarity by utilizing a hash-based clustering technique. This ensures efficient memory usage, as the window information of each vertex can be computed on-the-fly. On the basis of the clusters, we develop different optimization techniques to extract the shared components which result in an efficient index construction. Moreover, we provide another *Inheritance Index (I-Index)* tailored to topological window query. I-Index differentiates itself from DBIndex by integrating additional ancestry relationships to reduce repetitive computations. This results in more efficient index construction and query processing. Our contributions of this chapter are summarized as follows:

- We study the neighborhood analytics in the graph domain and propose the *Graph Window Query* which instantiates two neighborhood functions. We formally define two graph windows: k-hop window and topological window, and illustrate how these window queries would help users better query and under-

[1]Available at http://snap.stanford.edu/data/index.html, which is used in [**?**]

stand the graphs under these different semantics.

- To support efficient query processing, we further propose two different types of indexes: *Dense Block Index* (DBIndex) and *Inheritance Index* (I-Index). The DBIndex and I-Index are specially optimized to support k-hop window and topological window query processing. We develop the indexes by integrating the window aggregation sharing techniques to salvage partial work done for efficient computation. In addition, we develop space and performance efficient techniques for index construction.

- We perform extensive experiments over both real and synthetic datasets with hundreds of millions of vertexes and edges on a single machine. Our experiments indicate that our proposed index-based algorithms outperform the naive non-index algorithm by upto four orders of magnitude. In addition, our experiments also show that DBIndex is superior over the state-of-the-art baseline (i.e., EAGR) in terms of both scalability and efficiency. In particular, DBIndx saves up to 80% of index constructing time as compared to EAGR, and performs well even when EAGR fails due to memory limitations.

The rest of the chapter is organized as follows. In Section **??**, the graph window query is formulated. In Section **??**, Dense Block Index is presented to process general window queries. A specialized index to handle topological query is presented in Section **??**. Section **??** demonstrates our experimental findings and Section 3.2 concludes this chapter.

3.2 Summary

In this chapter, we studied the neighborhood analytics in attributed graphs. We leverage the distance and comparison neighborhoods to propose two graph windows: k-hop

window and topological window. Based on the two window definitions, we proposed a new type of graph analytic query, *Graph Window Query* (GWQ). Then, we studied GWQ processing for large-scale graphs. In particular, we developed the Dense Block Index (DBIndex) to facilitate efficient processing of both types of graph windows. Moreover, we proposed the Inheritance Index (I-Index) that exploits a containment property of DAG to enhance the query performance of topological window queries. Last, we conducted extensive experimental evaluations over both large-scale real and synthetic datasets. The experimental results showed the efficiency and scalability of our proposed indexes.

Chapter 4

k-Sketch Query: Neighborhood Analytics in Sequence Data

4.1 Introduction

In this chapter, we study the neighborhood analytics in sequence data. Sequence data are widely adopted by many applications to model events with temporal ordering such as sports games, temperature reads and stock ticks. An outstanding usage of sequence data is in the field of journalism, where the journalists strive to analyze these data to derive attention-seizing news themes. While such a task has traditionally been done manually, there is an increasing reliance on computational technology [**?**,**?**,**?**] to reduce human labor and intervention to a minimum. Recently, neighborhood analytics has also been attempted in computational journalism via the form of *streaks*. A streak is based on a **distance neighborhood** function which aggregates the last few events of an given event, such as "the average points a player scored in the last ten games prior to today's game". Based on the streaks, Zhang et. al. [**?**] studied the problem of summarizing a subject's history with *prominent streaks*. Prominent streaks are the skylines among all streaks of a subject, and thus are outstanding to represent the

subject's history.

However, there are two major drawbacks that limit the usability of prominent streaks in real applications. First, the prominent streaks generated by [**?**] may not be striking enough because they are derived from the historical data of a *single* subject without comparing to other subjects. For example, "Steve Nash has scored 15+ points in consecutive 10 games" is a prominent streak for "Steve Nash", but it is not striking given the fact that there are more than 90 players with better performance[1]. Second, the number of the prominent streaks can be overwhelming. Since prominent streaks in [**?**] are defined as skylines, a subject with n historical events may generate at most n streaks that are not dominated (i.e., prominent streaks). Therefore, there calls for a new method to automatically select a limited number of striking streaks which best summarize a subject's history.

In this chapter, we solve the problem of effectively and efficiently summarizing a subject's history by applying a novel **comparison neighborhood** function to transform streaks to the *ranked-streaks*. The comparison neighborhood function groups the streaks with the same lengths and rank them based on their aggregated values. We then propose the k-Sketch query to select k ranked-streaks which best summarize a subject's history. Compared with previous works, the ranked-streak is able to capture the strikingness of a streak which is very newsworthy as evidenced in the following news excerpts:

1. (26 Feb 2003) With 32 points, Kobe Bryant saw his 40+ scoring streak end at **nine** games, tied with Michael Jordan for the **fourth** place on the all-time list[2].

2. (14 April 2014) Stephen Curry has made 602 3-pointer attempts from beyond the arc,... are the **10th** most in NBA history in a season (**82 games**)[3].

[1] `http://www.sporcle.com/games/nbadarinh/nba-all-players-with-10-consecutive-20-point-games-90-11`

[2] `http://www.nba.com/features/kobe_40plus_030221.html`

[3] `http://www.cbssports.com/nba/eye-on-basketball/24525914/stephen-curry-makes-history-with-consecutive-seasons-of-250-3s`

3. (28 May 2015) Stocks gained for the **seventh consecutive day** on Wednesday as the benchmark moved close to the 5,000 mark for **the first** time in seven years[4].

4. (9 Jun 2014) Delhi has been witnessing a spell of hot weather over the **past month**, with temperature hovering around 45 degrees Celsius, ... **highest** ever since 1952[5].

5. (22 Jul 2011) Pelican Point recorded a maximum rainfall of 0.32 inches for **12 months**, making it the **9th driest** places on earth[6].

In the above examples, each news theme is a *ranked-streak* which consists of five components: (I) a subject (e.g., Kobe Bryant, Stocks, Delhi), (II) a streak length (e.g., nine straight games, seventh consecutive days, past month), (III) an aggregate function on an attribute (e.g., minimum points, count of gains, average of degrees), (IV) a rank (e.g., fourth, first time, highest), and (V) a historical dataset (e.g., all time list, seven years, since 1952). The indicators (I)-(IV) are summarized in Table 4.1. Among all the ranked-streaks, our k-Sketch query leverages a novel scoring function that chooses the best k ranked-streaks to summarize a subject's history. Our scoring function considers two aspects: (1) we prefer the ranked-streaks that cover as many events as possible to represent a subject's history, and (2) we prefer the ranked-streaks that have better ranks[7] as they indicate more strikingness. Our objective is to process the k-Sketch query for each subject in the domain.

We study the k-Sketch query processing under both offline and online scenarios. In the offline scenario, our objective is to efficiently discover the sketch for each subject

[4]`http://www.zacks.com/stock/news/176469/china-stock-roundup-ctrip-buys-elong-stake-trina-solar-beats-estimates`

[5]`http://www.dnaindia.com/delhi/report-delhi-records-highest-temperature-in-62-years-1994332`

[6]`http://www.livescience.com/30627-10-driest-places-on-earth.html`

[7]We consider a ranked-streak with rank i to be more attractive than j if $i < j$ and the other fields are the same.

Table 4.1: Indicators of ranked-streaks.

E.g.	**Subject**	**Aggregate Function**	**Streak Length**	**Rank**
1	Kobe	min(points)	9 straight games	4
2	Stephen	sum(shot attempts)	82 games	10
3	Stock Index	count(gains)	7 consecutive days	1
4	Delhi	average(degree)	past months (30 days)	1
5	Pelican Point	max(raindrops)	12 months	9

from historical data. The major challenge lies in generating the rank information of streaks. Since the number of streaks is quadratic with respect to the number of events, enumerating all of them is not scalable. By leveraging the subadditivity among the upper bounds of streaks, we design two effective pruning techniques to facilitate efficient ranked-streak generation. Furthermore, we notice that generating exact sketches from ranked-streaks is computationally expensive. Thus, we design an efficient $(1 - 1/e)$-approximate algorithm by exploiting the submodularity of the k-Sketch query.

In the online scenario, fresh events are continuously fed into the system and our goal is to maintain the sketches for each subject uptodate. When a new event about subject s arrives, many ranked-streaks of various lengths can be derived. For each derived ranked-streak, not only the sketch of s but also the sketches of other subjects may be affected. Dealing with such a complex updating pattern is non-trivial. To efficiently support the update while maintaining the quality of sketches, we propose a 1/8-approximate algorithm which only examines $2k$ ranked-streaks for each subject whose sketch is affected.

Our contributions of this chapter are hereby summarized as follows:

- We study the neighborhood analytics in sequence data to tackle the problem of automatic summarization of a subject's history. We use both the distance and comparison neighborhood functions to model the ranked-streak, which is a common news theme in real-life reports but has not been addressed in previous

works. We formulate the summarization problem as a k-Sketch query under a novel scoring function that considers both strikingness and coverage.

- We study the k-Sketch query processing in both offline and online scenarios. In the offline scenario, we propose two novel pruning techniques to efficiently generate ranked-streaks. Then we design a $(1 - 1/e)$-approximate algorithm to compute the sketches for each subject. In the online scenario, we propose a 1/8-approximate algorithm to efficiently support the complex updating patterns as new event arrives.

- We conduct extensive experiments with four real datasets to evaluate the effectiveness and the efficiency of our proposed algorithms. In the offline scenario, our solution is three orders of magnitude faster than baseline algorithms. While in the online scenario, our solution achieves up to 500x speedup. In addition, we also perform an anonymous user study via Amazon Mechanical Turk[8] platform, which validates the effectiveness of our k-Sketch query.

The rest of this chapter is organized as follows. In Section **??**, we formulate the k-Sketch query. Section **??** presents the algorithms for processing the k-Sketch query in the offline scenario. Section **??** describes the algorithms for maintaining k-Sketches in the online scenario. In Section **??**, comprehensive experimental studies on both the efficiency and the effectiveness of our algorithms are conducted. Section A.1 discusses the extension of our methods. Finally, Section 4.2 concludes our paper.

4.2 Summary

In this chapter, we looked at the neighborhood analytics in sequence data. We leverage the joint distance and comparison neighborhood functions to design the novel

[8]https://requester.mturk.com

ranked-streak which quantifies the strikingness of a streak. We then formulated the *k-Sketch* query which aims to best summarize a subject's history using k ranked-streaks. We studied the k-Sketch query processing in both offline and online scenarios, and propose efficient solutions to cope each scenario. In particular, we designed novel streak-level pruning techniques and a $(1 - 1/e)$-approximate algorithm to achieve efficient processing in offline. Moreover, we designed a 1/8-approximate algorithm for the online sketch maintenance. Our comprehensive experiments demonstrated the efficiency of our solutions and a human study confirmed the effectiveness of the k-Sketch query.

Chapter 5

GCMP Query: Neighborhood Analytics in Trajectories

5.1 Introduction

In this chapter, we look at the neighborhood analytics on large-scale trajectories. A trajectory is the spatial trace of a moving object which contains a sequence of spatial-temporal records. With the prevalence of positioning devices, the scale and spectrum of the trajectory collection has been drastically boosted to an unprecedented level. Tremendous amounts of trajectories are continually being generated from animal telemetry chips, vehicle GPSs and wearable devices. Data analysis on large-scale trajectories benefits a wide range of applications and services, including traffic planning [**?**], animal analysis [**?**], location-aware advertising [**?**], and social recommendations [**?**], to name just a few.

An important and outstanding instance of neighborhood analytics on top of trajectories is to discover co-moving objects. A *co-movement* pattern [**?**, **?**] refers to a group of objects traveling together for a certain period of time, which can be formulated as two-step neighborhood functions. First, in the spatial domain, a **distance**

neighborhood function clusters spatially nearby objects as groups (i.e., objects are neighbors under a distance function). Second, in the temporal domain, a **comparison neighborhood** function defines the co-moving pattern as the group that appears in neighboring timestamps (i.e., the group lasts for certain duration). A pattern is prominent if the size of the group exceeds M and the length of the duration exceeds K, where M and K are parameters specified by users. Rooted from such a basic definition and driven by different mining applications, there are many variants of co-movement patterns that have been developed with additional constraints.

Table 5.1 summarizes several popular co-movement patterns with different constraints with respect to spatial neighborhood, temporal constraints in consecutiveness and computational complexity. In terms of spatial neighborhood, the *flock* [**?**] and the *group* [**?**] patterns adopt disk-based clustering which requires all the objects in a group to be enclosed by a disk with radius r[1]; whereas the *convoy* [**?**], the *swarm* [**?**] and the *platoon* [**?**] patterns resort to density-based spatial clustering[2]. In terms of temporal constraints, the *flock* and the *convoy* require all the timestamps of each detected spatial group to be consecutive, which is referred to as *global consecutiveness*; whereas the *swarm* does not impose any restriction. The *group* and the *platoon* adopt a compromised approach by allowing arbitrary gaps between consecutive segments, which is called *local consecutiveness*. They introduce a parameter L to control the minimum length of each local consecutive segment.

Figure 5.1 is an example to demonstrate the concepts of the various co-movement patterns. The trajectory database consists of six moving objects and the temporal dimension is discretized into six snapshots. In each snapshot, we treat the clustering method as a blackbox and assume that they generate the same clusters. Objects in proximity are grouped in the dotted circles. As aforementioned, there are three parameters to determine the co-movement patterns and the default settings in this

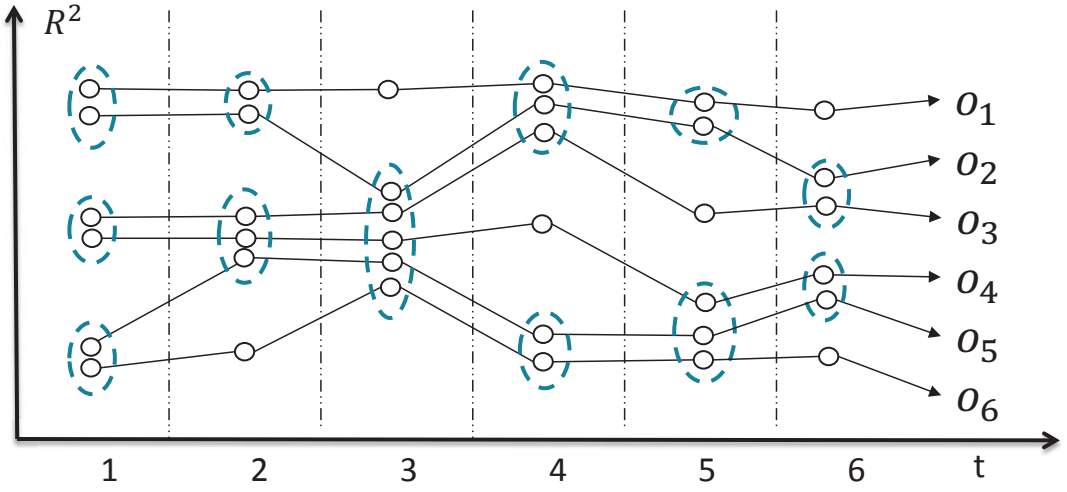
[1]Disk-based clustering is equivalent to $\mathcal{N}(o_i) = \{o_j | \texttt{dist}(o_i, o_j) < r\}$.
[2]Density-based clustering is equivalent to $\mathcal{N}(o_i) = \{o_j | \texttt{dist}(o_j, o_k) \leq \epsilon \wedge o_k \in \mathcal{N}(o_i)\}$.

Table 5.1: Constraints and complexities of co-movement patterns. The time complexity indicates the performance wrt. $|\mathbb{O}|$, $|\mathbb{T}|$ in the worst case, where $|\mathbb{O}|$ is the number of objects, and $|\mathbb{T}|$ is the number of discretized timestamps.

Pattern	**Spatial Neighborhood**	**Temporal Constraint**	**Time Complexity**
flock [**?**]	disk based	global consecutive	$O(\|\mathbb{O}\|\|\mathbb{T}\|\log(\|\mathbb{O}\|))$
convoy [**?**]	density based	global consecutive	$O(\|\mathbb{O}\|^2 + \|\mathbb{O}\|\|\mathbb{T}\|)$
swarm [**?**]	density based	-	$O(2^{\|\mathbb{O}\|}\|\mathbb{O}\|\|\mathbb{T}\|)$
group [**?**]	disk based	local consecutive	$O(\|\mathbb{O}\|^2\|\mathbb{T}\|)$
platoon [**?**]	density based	local consecutive	$O(2^{\|\mathbb{O}\|}\|\mathbb{O}\|\|\mathbb{T}\|)$

example are $M = 2$, $K = 3$ and $L = 2$. Both the *flock* and the *convoy* require the spatial clusters to last for at least K consecutive timestamps. Hence, $\langle o_3, o_4 : 1, 2, 3\rangle$ and $\langle o_5, o_6 : 3, 4, 5\rangle$ are the only two candidates matching the patterns. The *swarm* relaxes the pattern matching by discarding the temporal consecutiveness constraint. Thus, it generates many more candidates than the *flock* and the *convoy*. The *group* and the *platoon* add another constraint on local consecutiveness to retain meaningful patterns. For instance, $\langle o_1, o_2 : 1, 2, 4, 5\rangle$ is a pattern matching local consecutiveness because timestamps $(1, 2)$ and $(4, 5)$ are two segments with length no smaller than $L = 2$. The difference between the *group* and the *platoon* is that the *platoon* has an additional parameter K to specify the minimum number of snapshots for the spatial clusters. This explains why $\langle o_3, o_4, o_5 : 2, 3\rangle$ is a *group* pattern but not a *platoon* pattern.

As can be seen, there are various co-movement patterns requested by different applications and it is cumbersome to design a tailored solution for each type. In addition, despite the generality of the *platoon* (i.e., it can be reduced to other types of patterns via proper parameter settings), it suffers from the so-called *loose-connection* anomaly. We use two objects o_1 and o_2 in Figure 5.2 to illustrate the scenario. These two objects form a *platoon* pattern in timestamps $(1, 2, 3, 102, 103, 104)$. However, the two consecutive segments are 98 timestamps apart, resulting in a false positive co-movement pattern. In reality, such an anomaly may be caused by the periodic



Flock (M=2, K=3)	$\langle o_3, o_4: 1,2,3\rangle$ $\langle o_5, o_6: 3,4,5\rangle$
Convoy (M=2, K=3)	$\langle o_3, o_4: 1,2,3\rangle$ $\langle o_5, o_6: 3,4,5\rangle$
Group (M=2, L=2)	$\langle o_1, o_2: 1,2,4,5\rangle$ $\langle o_3, o_4, o_5: 2,3\rangle$ $\langle o_5, o_6: 3,4,5\rangle$
Platoon (M=2,K=3,L=2)	$\langle o_1, o_2: 1,2,4,5\rangle$ $\langle o_3, o_4 : 1,2,3\rangle$ $\langle o_4, o_5: 2,3,5,6\rangle$ $\langle o_5, o_6: 3,4,5\rangle$
Swarm (M=2,K=3)	$\langle o_1, o_2: 1,2,4,5\rangle$ $\langle o_3, o_4 : 1,2,3\rangle$ $\langle o_4, o_5: 2,3,5,6\rangle$ $\langle o_5, o_6: 1,3,4,5\rangle$ $\langle o_2, o_3: 3,4,6\rangle$

Figure 5.1: Trajectories and co-movement patterns. The example consists of six trajectories across six snapshots. Objects in spatial clusters are enclosed by dotted circles. M is the minimum cluster cardinality; K denotes the minimum number of snapshots for the occurrence of a spatial cluster; and L denotes the minimum length for local consecutiveness.

movements of unrelated objects, such as vehicles stopping at the same petrol station or animals pausing at the same water source. Unfortunately, none of the existing patterns have directly addressed this anomaly.

The other issue with existing methods is that they are built on top of centralized indexes which may not be scalable. Table 5.1 shows their theoretical complexities in the worst cases and the largest real dataset ever evaluated in previous studies is up to million-scale points collected from hundreds of moving objects. In practice, the dataset is of much higher scale and the scalability of existing methods is left unknown. Thus, we conduct an experimental evaluation with 4000 objects moving for 2500 timestamps to examine the scalability. Results in Figure 5.3 show that their performances degrade dramatically as the dataset scales up. For instance, the

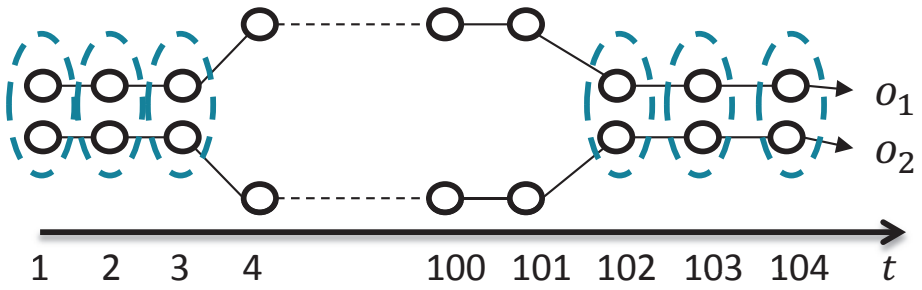
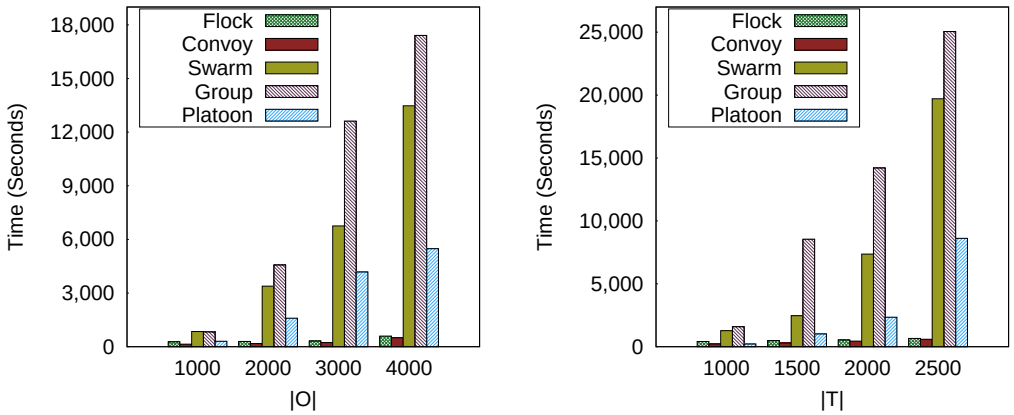


Figure 5.2: *Loose-connection* anomaly. Even though $\langle o_1, o_2 \rangle$ is considered as a valid *platoon* pattern, it is highly probable that these two objects are not related as the two consecutive segments are 98 timestamps apart.

detection time of *group* drops twenty times as the number of objects grows from *1k* to *4k*. Similarly, the performance of *swarm* drops over fifteen times as the number of snapshots grows from *1k* to *2.5k*. These observations imply that existing methods are not scalable to support large-scale trajectory databases.



(a) Varying No. of objects (b) Varying No. of timestamps

Figure 5.3: Performance measures on existing co-movement patterns. A sampled GeoLife dataset is used with upto 2.4 million data points. Default parameters are $M = 15$, $K = 180$, $L = 30$.

In this work, we close these two gaps by making the following contributions. First, we propose the *general co-movement pattern* (GCMP) which models various co-moment patterns in a unified way and can avoid the *loose-connection* anomaly. In GCMP, we introduce a new gap parameter G to pose a constraint on the temporal gap between two consecutive segments. By setting a feasible G, the loose-connection

anomaly can be effectively controlled. In addition, our GCMP is also general. It can be reduced to any of the previous pattern by customizing its parameters.

Second, we investigate deploying our GCMP detector on the modern MapReduce platform (i.e., Apache Spark) to tackle the scalability issue. Our technical contributions are threefold. First, we design a baseline solution by replicating the snapshots to support effective parallel mining. Second, we devise a novel *Star Partitioning and ApRiori Enumerator* (SPARE) framework to resolve limitations of the baseline. SPARE achieves workload balance by partitioning objects into fine granular stars. For each partition, an Apriori Enumerator is adopted to mine the co-movement patterns. Third, we leverage the *temporal monotonicity* property of GCMP to design several optimization techniques including *sequence simplification*, *monotonicity pruning* and *forward closure check* to further reduce the number of candidates enumerated in SPARE.

We conduct a set of extensive experiments on three large-scale real datasets with hundreds of millions of temporal points. The results show that both our parallel schemes efficiently support GCMP mining in large datasets. In particular, with over 170 million trajectory points, SPARE achieves upto 112 times speedup using 162 cores as compared to the state-of-the-art centralized schemes. Moreover, SPARE further achieves almost linear scalability with upto 14 times efficiency as compared to the baseline algorithm.

The rest of this chapter is organized as follows: Section **??** states the problem of general co-movement pattern mining. Section **??** provides a baseline solution. An advanced solution named *Star Partitioning and ApRiori Enumerator* (SPARE) is presented in Section **??**. Section **??** reports our experimental evaluation. Finally, Section 5.2 summarizes this chapter.

5.2 Summary

In this chapter, we studied one of the neighborhood analytics, namely the co-movement pattern discovery, on trajectory data. We proposed a generalized co-movement pattern query to unify those proposed in the past literature. We then devised two types of parallel frameworks on Apache Spark that can scale to support pattern detection in trajectory databases with hundreds of millions of points. The efficiency and scalability were verified by extensive experiments on three real datasets.