Chapter 1

Introduction

With the maturity of database technologies, nowadays applications collect data in all domains at an unprecedented scale. For example, billions of social network users and their activities are collected in the form of *graphs*; Thousand sensor reports are collected per second in the form of *time series*; Hundreds of millions of temporal-locations are collected as *trajectories*, to name just a few. Flooded by the tremendous amount of data, it is emerging to provide useful and efficient analytics for various data domains. Traditional SQL analytics which comprises of operations (such as, partition, sorting and aggregation) in the relational domain become limited in non-structured domains. In SQL context, interesting analytics such as graph traversal and pattern detection often involve complex joins which are very hard to optimize without domain knowledge. In this thesis, we explore the neighborhood data analytic, which in SQL is expressed by the window function, on different domains and demonstrate how to efficiently deploy neighborhood analytic to gain useful insights.

1.1 Neighborhood Analytic

By its self-describing name, neighborhood analytic aims to provide summaries of each object over its vicinity. In contrast to the global analytics which aggregates the entire

collection of data as a whole, neighborhood analytic provides a personalized view on each object per se. Neighborhood data analytics originates from the window function defined in SQL which is illustrated in Figure 1.1.

Season	Region	Sales	*sum()*	*avg()*
1	West	5100	***5100***	***5100***
2	West	5200	***10300***	***5150***
3	West	5200	***15500***	***5166***
4	West	4500	***20000***	***5000***
1	East	5000	***5000***	***5000***
2	East	4400	***9400***	***4700***
3	East	4800	***14200***	***4733***
4	East	5100	***19300***	***4825***

Window of season 3

```
SELECT Season, Region, Sales,
sum(), avg(), OVER(PARTITION
BY Region
ORDER BY Season DESC)
FROM employee;
```

Figure 1.1: A SQL window function computing running sum and average of sales. The window of season-3 is highlighted.

As shown in the figure, the sales report contains five attributes: "Season", "Region" and "Sales" are the facts, "`sum`()" and "`avg`()" are the analytics representing the running sum and average. A window function is represented by the `over` keyword. In this context, the window of a tuple o_i contains other tuples o_j such that o_i and o_j are in the same "region" and o_j's "season" is prior to o_i's. The window of season-3 for region-"West" is highlighted. Apart from this example, there are many other usages of the window function in the relational context. Being aware of the success of the window function, SQL 11 standard incorporates "`LEAD`" and "`LAG`" keywords which offer fine-grained specifications on a tuple's window.

Despite the usefulness, there are very few works reporting the window analytics in the non-relational domain. This may dues to the usage of *sorting* in relational windows. For example, in Figure 1.1, objects need to be sorted according to "Season", and then the window of each objects is implicitly formed. However, in non-relational context, sorting may be ambiguous and even undefined.

To generalize the window function to other domains, we propose the neighborhood analytics in a broader context. Given a set of objects (such as tuples in relational

domain or vertexes in graph domain), the neighborhood analytic is a composite function ($\mathcal{F} \circ \mathcal{N}$) applied on every object. $\mathcal{N}$ is the *neighborhood function*, which contains the related objects of an object; $\mathcal{F}$ is an *analytic function*, which could be aggregate, rank, pattern matching, etc. Apparently, the relational window function is a special case of the neighborhood analytics. For example, the window function in Figure 1.1 can be represented as $\mathcal{N}(o_i) = \{o_j | o_i.season > o_j.season \wedge o_i.region = o_j.region\}$ and $\mathcal{F} = \texttt{avg}$. Since the *sorting* requirement is relaxed, our neighborhood analytics is able to enrich the semantic of relational window notations and can be applied on many other domains.

1.2 Scope of the Thesis

In this thesis, we explore the neighborhood analytics in three prevalent data domains, namely **attributed graph**, **sequence data** and **trajectory**. Since the neighborhood analytics could be very broad, this thesis only focus on the following two intuitive neighborhood functions:

Distance Neighborhood: the neighborhood is defined based on numeric distance, that is $\mathcal{N}(o_i, K) = \{o_j | \texttt{dist}(o_i, o_j) \leq K\}$, where `dist` is a distance function and K is a distance threshold.

Comparison Neighborhood: the neighborhood is defined based on the comparison of objects, that is $\mathcal{N}(o) = \{o_i | o.a_m \texttt{ cmp } o_i.a_m\}$, where a_m is an attribute of object and `cmp` is a binary comparator.

There could be other types of neighborhood functions with different level of granularity. However, despite the simpleness of these two neighborhood functions, they are indeed versatile in representing many useful analytics.

1.3 Contributions

In brief, this thesis entitles a twofold contribution. First, by sewing different $\mathcal{N}$s and $\mathcal{F}$s, several novel neighborhood queries are proposed for *graph*, *sequence data* and *trajectory* domains respectively. Second, this thesis deals with the efficiency issues in deploying the corresponding analytic queries to handle data of nowadays scale. The roadmap of this thesis is shown in Figure 1.2.

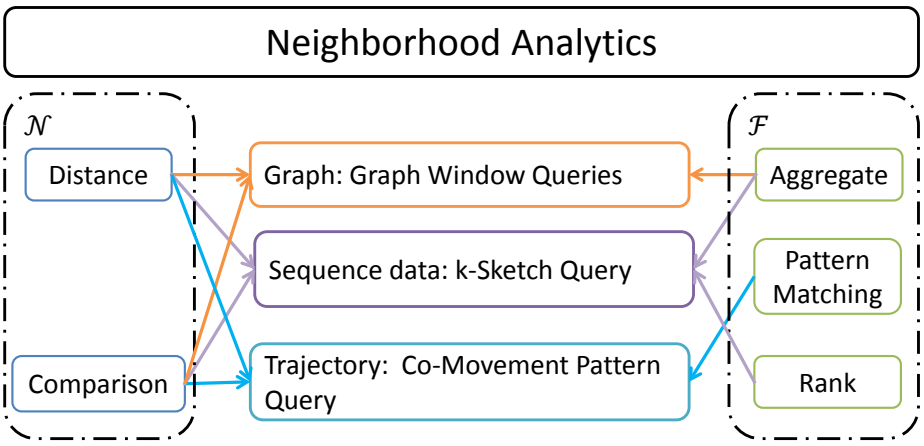


Figure 1.2: The road map of this thesis. There are three major contributions as highlighted in the center. Each contribution is a neighborhood analytic based on different $\mathcal{N}$ and $\mathcal{F}$ as indicated by arrows.

In a nutshell, we propose the neighborhood queries in three data domains. In *graph*, we identify two types of queries: *k-hop window query* is based on the *distance* neighborhood and *topological window query* is based on the *comparison* neighborhood. In *sequence data*, we propose a nested *distant* and *comparison* neighborhoods query named k-Sketch to finding striking rank-aware streaks. In *trajectory*, we analyze existing movement patterns based on *distance* and *comparison* neighborhoods and unify the existing works with a general pattern discovery query. Next, we present our contributions in detail for each of the data domains.

1.3.1 Window Queries for Graph Data

The first piece of the thesis deals with neighborhood query on graph data. Nowadays information network are typically modeled as attributed graphs where the vertexes correspond to objects and the edges capture the relationships between these objects. As vertexes embed a wealth of information (e.g., user profiles in social networks), there are emerging demands on analyzing these data to extract useful insights. We propose the concept of *window analytics* for attributed graph and identify two types of such analytics as shown in the following examples:

Example 1.3.1 (k-hop window)**.** In a social network (such as Linked-In and Facebook etc.), users are normally modeled as vertexes and connectivity relationships are modeled as edges. In the social network scenario, it is of great interest to summarize the most relevant connections to each user such as the neighbors within 2-hops. Some analytic queries such as summarizing the related connections' distribution among different companies, and computing age distribution of the related friends can be useful. In order to answer these queries, collecting data from every user's neighborhoods within 2-hop is necessary.

Example 1.3.2 (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, in order to study the protein regulating process, one may be interested to find out the statistics of molecules in each protein production pathway. For each protein,we can traverse the graph to find every other molecules that are in the upstream of its pathway. Then we can group and count the number of genes and enzymes among those molecules.

The two *windows* shown in the examples are essentially neighborhood functions defined for each vertex. Specifically, let $G = (V : E : A)$ be an attributed graph, where V is the set of vertexes, E is the set of edges, and each vertex v is associated as

a multidimensional points $a_v \in A$ called attributes. The *k-hop* window is a *distance* neighborhood function, i.e., $\mathcal{N}_1(v,k) = \{u|\texttt{dist}(v,u) \leq k\}$, which captures the vertexes that are k-hop nearby. The *topological* window, $\mathcal{N}_2(v) = \{u|u \in v.ancestor\}$, is a *comparison* neighborhood function that captures the ancestors of a vertex in a directly acyclic graph. The analytic function $\mathcal{F}$ is an aggregate function (sum, avg, etc.) on A.

Apart from demonstrating the useful use-cases on these two windows, we also investigate on supporting efficient window processing. We 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 processing. These indexes integrate the aggregation process with partial work sharing techniques to achieve efficient computation. In addition, we develop space-and-performance efficient techniques for the index construction. Notably, DBIndex saves upto 80% of indexing time as compared to the state-of-the-art competitor and upto 10x speedup in query processing.

1.3.2 k-Sketch Query for Sequence Data

The second piece of the thesis explores the neighborhood query for sequence data. In sequence data analysis, an important and revenue-generating task is to detect phenomenal patterns. An outstanding neighborhood based pattern is the *streak* which are constructed by aggregating temporally nearby data points for each subject. Streak has been found useful in many time series applications, such as network monitoring, stock analysis and computational journalism. However, the amount of streaks is too huge (i.e., $O\binom{N}{2}$ for a subject with N data points) to conduct effective analytics. Observed that many streaks share almost identical information, we are motivated to propose a k-Sketch query to effectively select k most representative streaks.

The k-sketch query is built on a core concept named *rank-aware streak*, out of

which k representatives are chosen based on a novel scoring function that balances the strikingness and the diversity. A *rank-aware streak* enriches traditional streak by including the relative position of a streak among its cohort. Such a concept is not rare in real-life. For example, journalists often adopt the rank-aware streak to promote the attractiveness of their news:

1. (Feb 26, 2003) With 32 points, Kobe Bryant saw his 40+ scoring streak end at **nine** games, tied with Michael Jordan for **fourth** place on the all-time list[1].
2. (April 14, 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**)[2].
3. (May 28, 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[3].
4. (Jun 9, 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[4].
5. (Jul 22, 2011) Pelican Point recorded a maximum rainfall of 0.32 inches for **12 months**, making it the **9th driest** places on earth[5].

In the above examples, each news is a rank-aware streak consisting of five elements: a subject (e.g., Kobe Bryant, Stocks, Delhi), an event window (e.g., nine straight games, seventh consecutive days, past month), an aggregate function on an attribute (e.g., minimum points, count of gains, average of degrees), a rank (e.g., fourth, first time, highest), and a historical dataset (e.g., all time list, seven years, since 1952). These indicators are summarized in Table 4.1.

[1] http://www.nba.com/features/kobe_40plus_030221.html
[2] http://www.cbssports.com/nba/eye-on-basketball/24525914/stephen-curry-makes-history-with-consecutive-seasons-of-250-3s
[3] http://www.zacks.com/stock/news/176469/china-stock-roundup-ctrip-buys-elong-stake-trina-sola
[4] http://www.dnaindia.com/delhi/report-delhi-records-highest-temperature-in-62-years-1994332
[5] http://www.livescience.com/30627-10-driest-places-on-earth.html

The rank-aware streak can be formally described using a joint neighborhood function. Let $e_s(t)$ denote the event of subject s at time t. Then the rank-aware streaks are generated using neighborhood analytics in a two-step manner:

(1) a *distance neighborhood* $\mathcal{N}_1(o_i, w) = \{o_j | o_i.t - o_j.t \leq w\}$ groups a consecutive w events for each event. Let $\overline{v}$ be the aggregate value associated with $\mathcal{N}_1$, then the output of this step is a set of *event windows* of the form $n = \langle o_i, w, t, \overline{v} \rangle$.

(2) a *comparison neighborhood* $\mathcal{N}_2(n_i) = \{n_j | n_j.w = n_i.w \wedge n_i.\overline{v} \geq n_j.\overline{v}\}$ ranks a subject's event window among all other event windows with the same window size. The result of the this step is a tuple $\langle o_i, w, t, r \rangle$, where r is the *rank*.

E.g.	**Subject**	**Aggregate function**	**Event window**	**Rank**
1	Kobe Bryant	`min`(points)	9 straight games	4
2	Stephen Curry	`sum`(shot attempts)	82 games	10
3	Stocks	`count`(gains)	7 consecutive days	1
4	Delhi	`avg`(degree)	past months (30 days)	1
5	Pelican Point	`max`(raindrops)	12 months	9

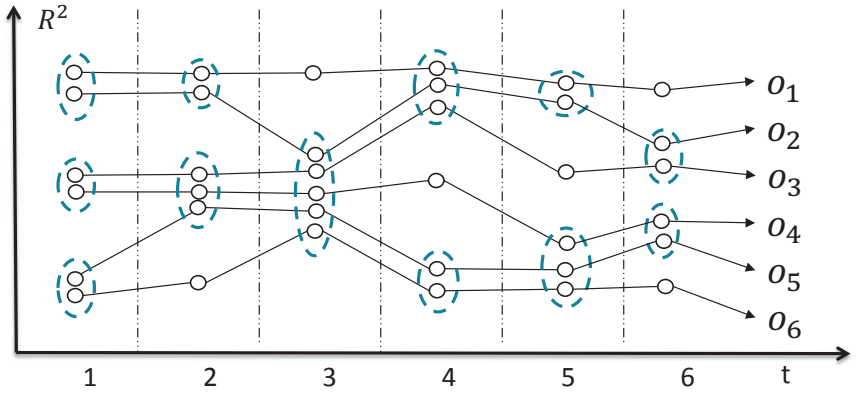
Table 1.1: News theme summary

Besides proposing the rank-aware streak, we also technically study how to efficiently support the *k-Sketch* query in both offline and online scenarios. We propose various window-level pruning techniques to find striking candidate steaks. Among those candidates, we then develop approximation methods, with theoretical bounds, to discover k-sketches for each subject. We conduct experiments on four real datasets, and the results demonstrate the efficiency and effectiveness of our proposed algorithms: the running time achieves up to 500x speedup as compared to baseline and the quality of the detected sketch is endorsed by the anonymous users from Amazon Mechanical Turk [6].

[6] `https://requester.mturk.com`

1.3.3 Co-Movement Pattern Query in Trajectory Data

The third piece of the thesis studies the neighborhood query on trajectory data. In trajectory domain, the neighborhoods of objects across multiple time snapshots collectively form a movement pattern. Due to its wide spectrum of applications, mining the movement patterns have attracted many research attention. Existing studies have identified several types of interesting movement patterns called *co-movement* patterns. A co-movement pattern refers to a group of moving objects traveling together for a certain period of time and the group of objects is normally determined by their spatial proximity. A pattern is prominent if the group size exceeds M and the length of duration exceeds K. Inspired by the basic definition and driven by different mining applications, there are a bunch of variant co-movement patterns that have been developed with more advanced constraints.



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 1.3: 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.

Figure 5.1 is an example to demonstrate the concepts of 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 black-box 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 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 $\langle o_5, o_6 : 3, 4, 5\rangle$ remains 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 shown, there are various types of co-movement patterns facilitating different application needs and it is cumbersome to deploy and optimize each of the pattern discovery algorithms. Therefore, it calls for a general framework to provide versatile and efficient support of all these pattern discoveries. We observe that these co-movement patterns can be uniformly described as a two-step neighborhood query as follows: (1) $\mathcal{N}_1$ is a *distance neighborhood* used to determine the spatial proximity of objects. For example, flock and group patterns uses the *disk-based* clustering, which is equivalent to $\mathcal{N}_1(o_i) = \{o_j | \texttt{dist}(o_i, o_j) < r\}$ for each object. Convoy, swarm and platoon patterns uses the *density-based* clustering, which is equivalent to $\mathcal{N}_1(o_i) = \{o_j | \texttt{dist}(o_j, o_k) \leq \epsilon \wedge o_k \in \mathcal{N}_1(o_i)\}$. (2) $\mathcal{N}_2$ is a comparison neighborhood used to

determine the temporal constraints, i.e., $\mathcal{N}_2(o_i) = \{o_j, T | \forall t \in T, C_t(o_i) = C_t(o_j)\}$, where $C_t(\cdot)$ returns the cluster ID of an object at time t.

Enlightened by the neighborhood unification, we propose a *General Co-Movement Pattern* (GCMP) query to capture all existing co-movement patterns. In GCMP, we treat the proximity detection (i.e., $\mathcal{N}_1$) as a black box and only focus on the pattern detection (i.e., $\mathcal{N}_2$). It is notable that the GCMP query is able to detect any of the existing co-movement patterns by adopting different analytic functions.

On the technical side, we study how to efficiently processing GCMP in a MapReduce platform to gain scalability for large-scale trajectory databases. In particular, we propose two parallel frameworks: (1) TRPM, which partitions trajectories by replicating snapshots in the temporal domain. Within each partitions, a line-sweep method is developed to find all patterns. (2) SPARE, which partitions trajectories based on object's neighborhood. Within each partitions, a variant of Apriori enumerator is applied to generate all patterns. We then show the efficiency of both our methods in the Apache Spark platform with three real trajectory datasets upto 170 million points. The results show that SPARE achieves upto 14 times efficiency as compared to TRPM, and 112 times speedup as compared to the state-of-the-art centralized schemes.

1.4 Thesis Organization

The remaining part of the thesis are organized as follows: in Chapter 2, we summarize related literature on our proposed neighborhood based queries in different data domains. In Chapter 3, we present the window function on graph data. In Chapter 4, we present the k-sketch query on sequence data. In Chapter 5, we present a pattern mining framework on trajectory data. Chapter 6 summarizes this thesis and highlights future directions.

Chapter 2

Literature Review

Our proposed neighborhood queries are inspired by the usefulness of window functions in relational analytic queries [29]. A window function in SQL specifies a set of partitioning attributes A and an aggregation function f. Its evaluation first partitions the input records based on A to compute f for each partition, and each input record is then associated with the aggregate value corresponding to the partition that contains the record. Several optimization techniques [5, 3] have also been developed to evaluate complex SQL queries involving multiple window functions.

However, the semantic and evaluation of the window function are restricted by the relational model. As been analyzed previously, SQL window functions require tuples to be sorted in order to form individual windows. However such a need is hard to meet in other data domains. Therefore, optimization techniques that are developed for the relational model become inapplicable in other domains. Nevertheless, there are quite a few works that related to the neighborhood queries that we have proposed and we summarize them in this section.

2.1 Graph Window Queries

2.1.1 Graph OLAP

Traditional graph data analytics focus on graph OnLine Analytic Processing (OLAP) [31, 25, 6, 23], which is different from Graph Window Queries. In a general model, graph OLAP applies analytics on each partition of the graph, where the partitions are created based on some attribute values of vertexes or edges. On the other hand, graph window queries aim to find the subgraph associated with each vertex and compute analytics over each subgraphs. Indeed, such differences also arise in the relational context, where different techniques are developed to evaluate OLAP and window function queries.

2.1.2 Reachability Queries and Indexes

Classic reachability queries, which answer whether two vertexes are connected, have been studied intensively in literature. To facilitate fast query processing, many indexes are proposed [28, 8]. Although our graph window queries can be built on top of the reachability queries, directly using these techniques is inefficient. For example, the most related reachability query to our k-hop window query is the k-reach query [8] which test if an input pair of vertexes is within a k-hop distance. In order to compute the k-hop window query for n vertexes, there would be $\theta(n^2)$ reachability tests. This is clearly inefficient especially when graphs are with over millions of vertexes.

2.1.3 Top-k Neighborhoods

In [27], the authors investigated the problem of finding the vertexes that have top-k highest aggregate values over their h-hop neighbors. This is similar to our k-hop queries while the difference is that they focus on providing pruning techniques to select k best vertexes and our graph window query aims to compute the analytic

values for each vertex. In our setting, their techniques degrade to the non-index approach as described in Section **??**;

2.1.4 Egocentric Networks

Egocentric networks [] refer to the neighborhood structure of a vertex in a graph. There are many works have conducted structural studies on egocentric networks however they do not focus on efficient processing on data analytics inside the egocentric networks. Recently, Jayanta et.al. [20] proposed an EAGR system to summarize attribute information among each vertex's neighborhoods. They use the Frequent-Pattern Tree heuristic [] to find the shared components among each vertex's neighborhoods. It starts by building an bipartite overlay graph to represent the vertex-neighbor mapping. Then it aims to find the bi-cliques in the overlay graph, where each bi-clique represents a set of vertexes whose neighborhood aggregates can be shared. Once a bi-clique is found, it is inserted back to the overlay graph as a virtual node to remove redundant edges. EAGR finds bi-cliques in iterations. During each iteration, it sorts each vertexes in overlay graph by their neighborhood information. Then the sorted vertices are split into equal-sized chunks. For each chunk, it then builds a FP-Tree to mining the large bi-cliques. As the algorithm iterates, the overlay graph evolves to be less dense.

The main drawback of EAGR is its demands of high memory usage on overlay construction. It requires the neighborhood information to be pre-computed, which is used in the sorting phase of each iteration. In EAGR the neighborhood information is assumed to be stored in memory. However, the assumption does not scale well for computing higher hop windows (such as $k \geq 2$). For instance, a LiveJournal social network graph [1] (4.8M nodes, 69M edges) generates over 100GB mapping information for k=2 in adjacency list representation. If the neighborhood information is resided

[1] Available at http://snap.stanford.edu/data/index.html, which is used [20]

in disk, the performance of EAGR will largely reduced. Similarly, if the neighborhood information is computed on-the-fly, EAGR needs to perform the computation in each iteration, which largely increases indexing time.

We tackle this drawback by adapting a hash based approach that clusters each vertex based on its neighborhood similarity. During the hashing, the vertex's neighborhood information is computed on-the-fly. As compared to sorting based approach, we do not require vertex's neighborhood to be pre-reside in memory. In order to reduce the repetitive computation of vertex's neighborhood, we further propose an estimation based indexing construction algorithm that only require a vertex's small hop neighborhood to be computed during clustering. As our experiments show, our proposed methods can perform well even when EAGR algorithm fails when neighborhood information overwhelms system's memory. To further reduce the neighborhood access, we adapted a Dense Block heuristic process each vertex in one pass. Experiments shows that the performance of our heuristic is comparable to EAGR's, but with much shorter indexing time.

2.2 k-Sketch Query

Our proposed k-Sketch query is closely related to three areas: automatic news detection, frequent episode mining and top-k diversity query.

2.2.1 Automatic News Detection

An prominent usage of k-Sketch query is on Computational Journalism where pioneer works include the

Earlier works on automatic news theme generation were focused on finding interesting themes from a single event. For example, Sultana et al. [21] proposed the *Situational Fact* pattern, which is modeled as a skyline point under certain dimen-

sions. Wu et al. [26] proposed the *One-of-the-Few* concept to detect news themes with some rarities. Examples of candidate news themes for the above two patterns are illustrated in Table **??**.

Method	**Example news theme**
Situational facts [21]	Ellen's tweet generates 3.3M retweets with 170,000 comments.
One-of-the-few facts [26]	Perry is one of the three candidates who received $600k
Prominent streak [30]	Kobe scored 40+ in 9 straight games!
Rank-aware theme	Kobe scored 40+ in 9 straight games ranked 4th in NBA history!

Table 2.1: Examples of different news themes

Zhang et al.[30] proposed using prominent streak to generate interesting news themes. In [30], a *Prominent Streak* is characterized by a 2D point which represents the window length and the minimum value of all events in the window. The objective is to discover the non-dominated event windows for each subject, where the dominance relationship is defined among streaks of the same subject. Our model differs from [30] in two aspects. First, we look at the global prominence (quantified by the rank) among all subjects rather than local prominence (quantified by the dominance) within one subject. Second, our model provides the best k-sketch for each subject whereas [30] returns a dominating set which could be potentially large.

2.2.2 Frequent Episode Mining

In time sequenced data mining, an episode [19, 33, 22, 15] is defined as a collection of time sequenced events which occur together within a time window. The uniqueness of an episode is determined by the contained events. The objective is to discover episodes whose occurrences exceeding a support threshold. Our sketch discovery differs from the episode mining in three major aspects. First, an episode is associated with a

categorical value while our sketch is defined on numerical values. Second, the episodes are selected based on the occurrence, while in sketch, news themes are generated in a rank-aware manner. Finally, episode mining does not restrict its output size, while sketch only outputs the best k news themes. As such, episode mining techniques cannot be straightforwardly applied to sketch discovery.

2.2.3 Top-k Diversity Query

Top-k diversity queries [1, 4, 10, 7] aim to find a subset of objects to maximize a scoring function. The scoring function normally penalizes a subset if it contains similar elements. Our sketch discovery problem has two important distinctions against the top-k diversity queries. First, the inputs of the scoring function are known in advance in top-k diversity queries; whereas in our problem, the ranks of event windows are unknown. Since their calculations are expensive, we need to devise efficient methods to compute the ranks. Second, existing methods for online diversity queries [4, 10, 7] only study the update on a single result set when a new event arrives. However our online sketch maintenance incurs the problem of multiple sketch updates for each new event. Such a complex update pattern has not been studied yet and hence there is a need to develop efficient update scheme.

2.3 Movement Pattern Discovery Query

Related works can be grouped into three categories: *co-movement patterns*, *dynamic moving patterns* and *trajectory mining frameworks*. In this section, we distinguish the parallel GCMP mining from these concepts.

2.3.1 Flock and Convoy

The difference between *flock* and *convoy* lies in the object clustering methods. In *flock*, objects are clustered based on their distances. Specifically, the objects in the same cluster need to have a pair-wised distance less than *min_dist*. This essentially requires the objects to be within a disk-region of delimiter less than *min_dist*. In contrast, *convoy* clusters objects using density-based spatial clustering [11]. Technically, *flock* utilizes a m^{th}-order Voronoi diagram [14] to detect whether a subset of object with size greater than m stays in a disk-region. *Convoy* employs a trajectory simplification [9] technique to boost pairwise distance computations in the density-based clustering. After clustering, both *flock* and *convoy* use a sequential scanning method to examine each snapshots. During the scan, object groups that appear in consecutive timestamps are detected. Meanwhile, the object groups that do not match the consecutive constraint are pruned. However, such a method faces high complexity when supporting other patterns. For instance, in *swarm*, the candidate set during the sequential scanning grows exponentially, and many candidates can only be pruned after the entire snapshots are scanned.

2.3.2 Group, Swarm and Platoon

Different from *flock* and *convoy*, all the *group*,*swarm* and *platoon* patterns have more relaxed constraints on the pattern duration. Therefore, their techniques of mining are of the same skeleton. The main idea of mining is to grow object set from an empty set in a depth-first manner. During the growth, various pruning techniques are provided to prune unnecessary branches. *Group* pattern uses a VG-graph to guide the pruning of false candidates [24]. *Swarm* designs two more pruning rules called backward pruning and forward pruning [18]. *Platoon* further leverages a prefix table structure to steer the depth-first search. As shown by Li et.al. [17], *platoon* outperforms other two methods in efficiency. However, the three patterns are not

able to directly discover GCMPs. Furthermore, their pruning rules heavily rely on the depth-first search nature, which lose their efficiencies in the parallel scenario.

2.3.3 Other Related Trajectory Patterns

A closely related literature to co-movement patterns is the *dynamic moving* patterns. Instead of requiring the same set of object traveling together, *dynamic moving* patterns allow objects to temporally join or leave a group. Typical works include *moving clusters* [13], *evolving convoy* [2], *gathering* [32] etc. These works cannot model our GCMP since they enforce the global consecutiveness on the timestamps of a pattern.

2.3.4 Trajectory Mining Frameworks

Jinno et al. in [12] designed a MapReduce based algorithm to efficiently support T-pattern discovery, where a T-pattern is a set of objects visiting the same place at simliar time. Li et al. proposed a framework of processing online *evolving group* pattern [16], which focuses on supporting efficient updates of arriving objects. As these works essentially differ from co-movement pattern, their techniques cannot be directly applied to discover GCMPs.

Chapter 3

Towards Window Analytics on Large-scale Graphs

3.1 Introduction

Information networks such as social networks, biological networks and phone-call networks are typically modeled as graphs [6] where the vertices 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 between the vertices. In addition, a user's profile can be maintained as the vertex's attributes. Such graphs contain a wealth of valuable information which can be analyzed to discover interesting patterns. For example, we can find the top-k influential users who can reach the most number of friends within 2 hops. With increasingly larger network sizes, it is becoming significantly challenging to query, analyze and process these graph data. Therefore, there is an urgent need to develop effective and efficient mechanisms over graph data to draw out information from such data resources.

Traditionally, in relational DBMS, window functions have been commonly used

for data analytics [5, 3]. Instead of performing analysis (e.g. ranking, aggregate) over the entire data set, a window function returns for each input tuple a value derived from applying the function over a window of neighboring tuples. For instance, users may be interested in finding each employee's salary ranking within the department. Here, each tuple's neighbors are essentially records from the same department.

Interestingly, the notion of window functions turns out to be not uncommon in graph data. For instance, in a social network, it is important to detect a person's social position and influence among his/her social community. The "social community" of the person is essentially his/her "window" comprising neighbors derived from his/her k-hop friends. However, as illustrated in this example, the structure of a graph plays a critical role in determining the neighboring data of a vertex. In fact, it is often useful to quantify a structural range to each vertex and then perform analytics over the range. Surprisingly, though such a concept of window functions has been widely used, the notion has not been explicitly formulated. In this paper, we are motivated to extend the window queries in traditional SQL for supporting graph analysis. However, the window definition in SQL is no longer applicable in a graph context, as it does not capture the graph structure information. Thus, we seek to formulate the notion of graph windows and to develop efficient algorithms to process them over large scaled graph structures.

We have identifed two instantiations of graph windows, namely *k-hop* and *topological* windows. We first demonstrate these window semantics with the following examples.

Example 3.1.1. *(k-hop window)* In a social network (such as Linked-In and Facebook etc.), users are normally modeled as vertices and connectivity relationships are modeled as edges. In social network scenario, it is of great interest to summarize the most relevant connections to each user such as the neighbors within 2-hops. Some analytic queries such as summarizing the related connections' distribution among dif-

ferent companies, and computing age distribution of the related friends can be useful. In order to answer these queries, collecting data from every user's neighborhoods within 2-hop is necessary.

Example 3.1.2. *(Topological window)* In biological networks (such as Argocyc, Ecocyc etc.[**?**]), genes, enzymes and proteins are vertices and their dependency in a pathway are edges. Because these networks are directed and acyclic, in order to study the protein regulating process, one may be interested to find out the statistics of molecules in each protein production pathway. For each protein, we can traverse the graph to find every other molecule that is in the upstream of its pathway. Then we can group and count the number of genes and enzymes among those molecules.

A common feature among these examples is that data aggregation is needed based on a set of vertices (which is the *graph window*) defined according to each vertex. To illustrate, in example 3.1.1, every user needs to gather data from its friends and friends-of-friends. The *2-hop neighbors* form its window. Likewise, in example 3.1.2, every protein needs to count the number of particular type of genes preceding it in the regulating pathway. For every protein, the set of *preceding molecules* forms its window.

To support the analyses in the above-mentioned examples, we propose a new type of query, *Graph Window Query* (GWQ in short), over a data graph. *GWQ* is defined with respect to a graph structure and is important in a graph context. Unlike the traditional window in SQL, we identify two types of useful graph windows according to the graph structures, namely k-hop Window W_{kh} and Topological Window W_t. A k-hop window forms a window for one vertex by using its k-hop neighbors. k-hop neighbors are important to one vertex, as these are the vertices showing structural closeness as in Example 1. The k-hop neighbors window we define here is similar to the egocentric-network in network analysis [**?**] [20]. A topological window, on the other hand, forms a window for one vertex by using all its preceding vertices in a

directed acyclic graph. The preceding vertices of one vertex are normally those which influence the vertex in a network as illustrated in Example 2.

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 like query, the state-of-the-art processing algorithm is EAGR [20]. EAGR builds an overlay graph including the shared components of different windows. This is done in multiple iterations, each of which performs the following. First, EAGR sorts all the vertices according to their k-hop neighbors based on their lexicographic order. Second, the sorted vertices are split into equal sized chunks each of which is further built as one frequent-pattern tree to mine the shared components. However, EAGR requires all the vertices' k-hop neighbors to be pre-computed and resided in memory during each sorting and mining operation; otherwise, EAGR incurs high computation overhead if the pre-computed structure needs to be shuffled to/from disk. This limits the efficiency and scalability of EAGR. For instance, a LiveJournal social network graph[1] (4.8M vertices, 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 overlay quality. The high memory consumption renders the scheme impractical when k and the graph size increases.

In this paper, we propose *Dense Block Index (DBIndex)* to process queries efficiently. Like EAGR, DBIndex seeks to exploit common components among different windows to salvage partial work done. However, unlike EAGR, we identify the win-

[1]Available at http://snap.stanford.edu/data/index.html, which is used in [20]

dow similarity utilizing a hash-based clustering technique instead of sorting. 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 optimizations 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 more descendant-ancestor relationships to reduce repetitive computations. This results in more efficient index construction and query processing.

Our contributions are summarized as follows:

- We propose a new type of graph analytic query, *Graph Window Query* and formally define two graph windows: k-hop window and topological window. We illustrate how these window queries would help users better query and understand the graphs under these different semantics.

- To support efficient query processing, we further propose two different types of indices: *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 indices 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. Compared to EAGR [20], the state-of-the-art index method for k-hop window queries, our DBIndex is much more memory efficient and scalable towards handling the large-scale graphs.

- We perform extensive experiments over both real and synthetic datasets with hundreds of millions of vertices and edges on a single machine. Our experiments indicate that our proposed index-based algorithms outperform the naive non-index algorithm by up to four orders of magnitude. In addition, our experiments

also show that DBIndex is superior over EAGR in terms of both scalability and efficiency. In particular, DBIndx saves up to 80% of indexing time as compared to EAGR, and performs well even when EAGR fails due to memory limitations.

3.2 Related Work

Our proposed graph window functions (GWFs) for graph databases is inspired by the usefulness of window functions in relational analytic queries [29].

A window function in SQL typically specifies a set of partitioning attributes A and an aggregation function f. Its evaluation first partitions the input records based on A to compute f for each partition, and each input record is then associated with the aggregate value corresponding to the partition that contains the record. Several optimization techniques [5, 3] have also been developed to evaluate complex SQL queries involving multiple window functions.

However, the semantics and evaluation of window functions are very different between relational and graph contexts. Specifically, the partitions (i.e., subgraphs) associated with GWFs are not necessarily disjoint; thus, the evaluation techniques developed for relational context [5, 3] are not applicable to GWFs.

GWFs are also different from graph aggregation [31, 25, 6, 23] in graph OLAP. In graph OLAP, information in a graph are summarized by partitioning the graph's nodes/edges (based on some attribute values) and computing aggregate values for each partition. GWFs, on the other hand, compute aggregate values for each graph node w.r.t. the subgraph associated with the node. Indeed, such differences also arise in the relational context, where different techniques are developed to evaluate OLAP and window function queries.

In [27], the authors investigated the problem of finding the vertices that have top-k highest aggregate values over their h-hop neighbors. They proposed mechanisms to

prune the computation by using two properties: First, the locality between vertices is used to propagate the upper-bound of aggregation; Second, the upper-bound value of aggregates can be estimated from the distribution of attribute values. However, all these pruning techniques are not applicable in our work, as we need to compute the aggregation value for every vertex. In such a scenario, techniques in [27] degrade to the non-indexed approach as described in Section 4.

Indexing techniques have been proposed to efficiently determine whether an input pair of vertices is within a distance of k-hops (e.g. k-reach index [8]) or reachable (e.g. reachability index [28]). However, such techniques are not efficient for computing the k-hop window or topological window for a set of n vertices with a time complexity of $O(n^2)$.

[20] proposed an EAGR system, which uses the famous VNM heuristic and Frequent-Pattern Tree to find the shared component among each vertex's neighborhoods. It starts by building an overlay graph as a bipartite graph representing the vertex-neighbor mapping. Then it aims to find the bi-cliques in the overlay graph. Each bi-clique represents a set of vertices whose neighborhood aggregates can be shared. Once a bi-clique is found, it is inserted back to the overlay graph as a virtual node to remove redundant edges. EAGR find bicliques in iterations. During each iteration, it sorts each vertices in overlay graph by their neighborhood information. Then the sorted vertices are split into equal-sized chunks. For each chunk, it then builds a FP-Tree to mining the large bi-cliques. As the algorithm iterates, the overlay graph evolves to be less dense.

The main drawback of EAGR is its demands of high memory usage on overlay construction. It requires the neighborhood information to be pre-computed, which is used in the sorting phase of each iteration. In EAGR the neighborhood information is assumed to be stored in memory. However, the assumption does not scale well for computing higher hop windows (such as k $\geq$ 2). For instance, a LiveJournal social

network graph [2] (4.8M nodes, 69M edges) generates over 100GB mapping information for k=2 in adjacency list representation. If the neighborhood information is resided in disk, the performance of EAGR will largely reduced. Similarly, if the neighborhood information is computed on-the-fly, EAGR needs to perform the computation in each iteration, which largely increases indexing time.

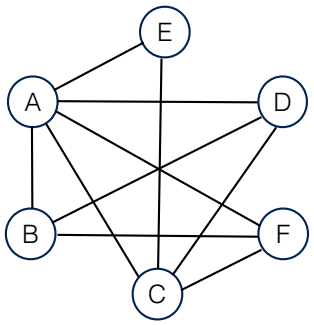
We tackle this drawback by adapting a hash based approach that clusters each vertex based on its neighborhood similarity. During the hashing, the vertex's neighborhood information is computed on-the-fly. As compared to sorting based approach, we do not require vertex's neighborhood to be pre-reside in memory. In order to reduce the repetitive computation of vertex's neighborhood, we further propose an estimation based indexing construction algorithm that only require a vertex's small hop neighborhood to be computed during clustering. As our experiments show, our proposed methods can perform well even when EAGR algorithm fails when neighborhood information overwhelms system's memory. To further reduce the neighborhood access, we adapted a Dense Block heuristic process each vertex in one pass. Experiments shows that the performance of our heuristic is comparable to EAGR's, but with much shorter indexing time.

3.3 Problem Formulation

In this section, we provide the formal definition of graph window query. We use $G = (V, E)$ to denote a directed/undirected data graph, where V is its vertex set and E is its edge set. Each node/edge is associated with a (possibly empty) set of attribute-value pairs.

Fig. 3.1 shows an undirected graph representing a social network that we will use as our running example in this paper. The table shows the values of the five attributes (User, Age, Gender, Industry, and Number of posts) associated with each vertex. For

[2]Available at http://snap.stanford.edu/data/index.html, which is used [20]



(a)

User	Age	Gender	Industry	Posts
A	21	M	IT	12
B	26	F	IT	15
C	30	F	Finance	28
D	22	M	Finance	23
E	28	M	Power	26
F	23	F	Power	14

(b)

Figure 3.1: Running Example of Social Graph. (a) provides the graph structure; (b) provides the attributes associated with the vertices of (a).

convenience, each node is labeled with its user attribute value; and there is one edge between a user X and another user Y if X and Y are connected in the social network.

Given a data graph $G = (V, E)$, a *Graph Window Function (GWF)* over G can be expressed as a quadruple (G, W, Σ, A), where $W(v)$ denotes a *window specification* for a vertex $v \in V$ that determines the set of vertices in some subgraph of G, Σ denotes an *aggregation function*, and A denotes a *vertex attribute*. The evaluation of a GWF (G, W, Σ, A) on G computes for each vertex v in G, the aggregation Σ on the values of attribute A over all the vertices in $W(v)$, which we denote by $\Sigma_{v' in W(v)} v'.A$.

Note that, in this paper, we focus on the attribute-based aggregation with distributive or algebraic aggregation functions. In other words, $W(v)$ refers to a set of vertices, and the aggregation function Σ operates on the values of attribute A over all the vertices in $W(v)$. Meanwhile, the aggregation function Σ is distributive or algebraic (e.g., sum, count, average), as these aggregation functions are widely used in practice.

In the following, we introduce two useful types of window specification (i.e., W), namely, *k-hop window* and *topological window*.

Definition 3.3.1 (K-hop Window)**.** Given a vertex v in a data graph G, the k-hop window of v, denoted by $W_{kh}(v)$ (or $W(v)$ when there is no ambiguity), is the set of

neighbors of v in G which can be reached within k hops. For an undirected graph G, a vertex u is in $W_{kh}(v)$ iff there is a α-hop path between u and v where $\alpha \leqslant k$. For a directed graph G, a vertex u is in $W_{kh}(v)$ iff there is a α-hop directed path from v to u [3] where $\alpha \leqslant k$.

Intuitively, a k-hop window selects the neighboring vertices of a vertex within a k-hop distance. These neighboring vertices typically represent the most important vertices to a vertex with regard to their structural relationship in a graph. Thus, k-hop windows provide meaningful specifications for many applications, such as customer behavior analysis [**?**, **?**] , digital marketing [**?**] etc.

As an example, in Fig. 3.1, the 1-hop window of vertex E is $\{A, C, E\}$ and the 2-hop window of vertex E is $\{A, B, C, D, E, F\}$.

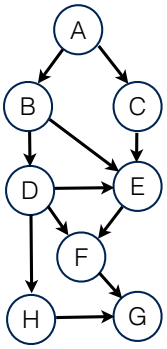
Definition 3.3.2 (Topological Window)**.** Given a vertex v in a DAG G, the topological window of v, denoted by $W_t(v)$, refers to the set of ancestor vertices of v in G; i.e., a vertex u is in $W_t(v)$ iff there is directed path from u to v in G.

There are many directed acyclic graphs (DAGs) in real-world applications (such as biological networks, citation networks and dependency networks) where topological windows represent meaningful relationships that are of interest. For example, in a citation network where (X,Y) is an edge iff paper X cites paper Y, the topological window of a paper represents the citation impact of that paper [**?**, **?**, **?**].

As an example, Fig. 3.2 shows a small example of a Pathway Graph from a biological network. The topological window of E $W_t(E)$ is $\{A, B, C, D, E\}$ and $W_t(H)$ is $\{A, B, D, H\}$.

Definition 3.3.3 (Graph Window Query)**.** A graph window query on a data graph G is of the form $GWQ(G, W_1, \Sigma_1, A_1, \cdots$,

[3]Other variants of k-hop window for directed graphs are possible; e.g., a vertex u is in $W_{kh}(v)$ iff there is a α-hop directed path from u to v where $\alpha \leqslant k$.



ID	Type
A	Enzyme
B	Cytokine
C	Transporter
D	Enzyme
E	Enzyme
F	Cytokine
G	Enzyme
H	Transporter

(a) (b)

Figure 3.2: Running Example of Pathway DAG. (a) provides the DAG structure; (b) provides the attributes associated with the vertices of (a).

$W_m, \Sigma_m, A_m)$, where $m \geq 1$ and each quadruple (G, W_i, Σ_i, A_i) is a graph window function on G.

In this paper, we focus on graph window queries with a single window function that is either a k-hop or topological window. The evaluation of complex graph window queries with multiple window functions can be naively processed as a sequence of window functions one after another. We leave the optimization of processing multiple window functions for further study in the future.

3.4 Dense Block Index

A straightforward approach to process a graph window query $Q = (G, W, \Sigma, A)$, where $G = (V, E)$, is to dynamically compute the window $W(v)$ for each vertex $v \in V$ and its aggregation $\Sigma_{v' \in W(v)} v'.A$ independently from other vertices. We refer to this approach as *Non-Indexed* method.

Given that many of the windows would share many common nodes (e.g., the k-hop windows of two adjacent nodes), such a simple approach would be very inefficient due to the lack of sharing of the aggregation computations.

To efficiently evaluate graph window queries, we propose an indexing technique

named *dense block index* (*DBIndex*), which is both space and query efficient. The main idea of DBIndex is to try to reduce the aggregation computation cost by identifying subsets of nodes that are shared by more than one window so that the aggregation for the shared nodes could be computed only once instead of multiple times.

For example, consider a graph window query on the social graph in Fig. 3.1 using the 1-hop window function. We have $W(B) = \{A, B, D, F\}$ and $W(C) = \{A, C, D, E, F\}$ sharing three common nodes A, D, and F. By identifying the set of common nodes $S = \{A, D, F\}$, its aggregation $\Sigma_{v \in S} v.A$ can be computed only once and then reuse to compute the aggregation for $\Sigma_{v \in W(B)} v.A$ and $\Sigma_{v \in W(C)} v.A$.

Given a window function W and a graph $G = (V, E)$, we refer to a non-empty subset $B \subseteq V$ as a *block*. Moreover, if B contains at least two nodes and B is contained by at least two different windows (i.e., there exists $v_1, v_2 \in V$, $v_1 \neq v_2$, $B \subseteq W(v_1)$, and $B \subseteq W(v_2)$), then B is referred to as a *dense block*. Thus, in the last example, $\{A, D, F\}$ is a dense block.

We say that a window $W(X)$ is *covered* by a collection of disjoint blocks $\{B_1, \cdots, B_n\}$ if the set of nodes in the window $W(X)$ is equal to the union of all nodes in the collection of disjoint blocks; i.e., $W(X) = \bigcup_{i=1}^{n} B_i$ and $B_i \cap B_j = \emptyset$ if $i \neq j$.

To maximize the sharing of aggregation computations for a graph window query, the objective of DBIndex is to identify a small set of blocks $\mathcal{B}$ such that for each $v \in V$, $W(v)$ is covered by a small subset of disjoint blocks in $\mathcal{B}$. Clearly, the cardinality of $\mathcal{B}$ is minimized if $\mathcal{B}$ contains a few large dense blocks.

Thus, given a window function W and a graph $G = (V, E)$, a DBIndex to evaluate W on G consists of three components in the form of a bipartite graph. The first component is a collection of nodes (i.e., V); the second component is a collection of blocks; i.e., $\mathcal{B} = \{B_1, \cdots, B_n\}$ where each $B_i \subseteq V$; and the third component is a collection of links from blocks to nodes such that if a set of blocks $B(v) \subseteq \mathcal{B}$ is linked to a node $v \in V$, then $W(v)$ is covered by $B(v)$. Note that a DBIndex is independent

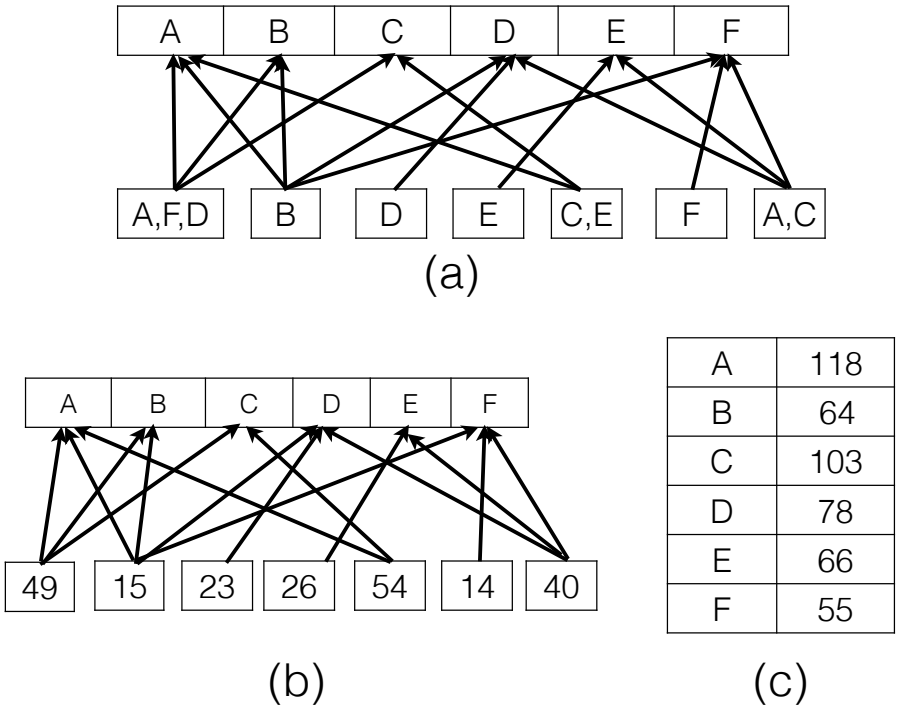


Figure 3.3: Window Query Processing using DBIndex. (a) provides the DBIndex for 1-hop window query in Fig. 3.1; (b) shows the partial aggregate results based on the dense block; (c) provides the final aggregate value of each window.

of both the aggregation function (i.e., Σ) and the attribute to be aggregated (i.e., A).

Fig. 3.3(a) shows an example of a DBIndex wrt the social graph in Fig. 3.1 and the 1-hop window function. Note that the index consists of a total of seven blocks of which three of them are dense blocks.

3.4.1 Query Processing using DBIndex

Given a DBIndex wrt a graph G and a window function W, a graph window query $Q = (G, W, \Sigma, A)$ is processed by the following two steps. First, for each block B_i in the index, we compute the aggregation (denoted by T_i) over all the nodes in B_i; i.e., $T_i = \Sigma_{v \in B_i} v.A$. Thus, each T_i is a partial aggregate value. Next, for each window $W(v)$, $v \in V$, the aggregation for the window is computed by aggregating over all the partial aggregates associated with the blocks linked to $W(v)$; i.e., if $B(v)$

is the collection of blocks linked to $W(v)$, then the aggregation for $W(v)$ is given by $\Sigma_{B_i \in B(v)} T_i$.

Consider again the DBIndex shown in Fig. 3.3(a) defined wrt the social graph in Fig. 3.1 and the 1-hop window function. Fig. 3.3(b) shows how the index is used to evaluate the graph window query $(G, W, sum, Posts)$ where each block is labeled with its partial aggregate value; and Fig. 3.3(c) shows the final query results.

3.4.2 DBIndex Construction

In this section, we discuss the construction of the DBIndex (wrt a graph $G = (V, E)$ and window function W) which has two key challenges.

The first challenge is the time complexity of the index construction. From our discussion of query processing using DBIndex, we note that the number of aggregation computations is determined by both the number of blocks as well as the number of links in the index; the former determines the number of partial aggregates to compute while the latter determines the number of aggregations of the partial aggregate values. Thus, to maximize the shared aggregation computations using DBIndex, both the number of blocks in the index as well as the number of blocks covering each window should be minimized. However, finding the optimal DBIndex to minimize this objective is NP-hard[4]. Therefore, efficient heuristics are needed to construct the DBIndex.

The second challenge is the space complexity of the index construction. In order to identify large dense blocks to optimize for query efficiency, a straightforward approach is to first derive the window $W(v)$ for each node $v \in V$ and then use this derived information to identify large dense blocks. However, this direct approach incurs a high space complexity of $O(|V|^2)$. Therefore, a more space-efficient approach is needed in order to scale to handle large graphs.

[4]Note that a simpler variation of our optimization problem has been proven to be NP-hard [**?**].

In this section, we present two heuristic approaches, namely *MC* and *EMC*, to construct DBIndex. The second approach EMC is designed to improve the efficiency of the first approach MC for constructing DBIndex (wrt k-hop window function) by using some approxmation techique at the expense of possibly sacrificing the "quality" of the dense blocks (in terms of their sizes).

3.4.2.1 MinHash Clustering (MC)

To reduce both the time and space complexities for the index construction, instead of trying to identify large dense blocks among a large collection of windows, we first partition all the windows into a number of smaller clusters of similar windows and then identify large dense blocks for each of the smaller clusters. Intuitively, two windows are considered to be highly similar if they share a larger subset of nodes. We apply the well-known *MinHash based Clustering (MC)* algorithm [**?**] to partition the windows into clusters of similar windows. The MinHash clustering algorithm is based on using the *Jaccard Coefficient* to measure the similarity of two sets. Given the two window $W(v)$ and $W(u)$, $u, v \in V$, their *Jaccard Coefficient* is given by

$$J(u, v) = \frac{|W(u) \cap W(v)|}{|W(u) \cup W(v)|} \tag{3.1}$$

The *Jaccard Coefficient* ranges from 0 to 1, where a larger value means that the windows are more similar.

Our heuristic approach to construct DBIndex I operates as follows. Let $nodes(I)$, $blocks(I)$, and $links(I)$ denote, respectively, the collection of nodes, blocks, and links that form I. Initially, we have $nodes(I) = V$, $blocks(I) = \emptyset$, and $links(I) = \emptyset$.

The first step is to partition the nodes in V into clusters using MinHash algorithm such that nodes with similar windows belong to the same cluster. For each node $v \in V$, we first derive its window $W(v)$ by an appropriate traversal of the graph G.

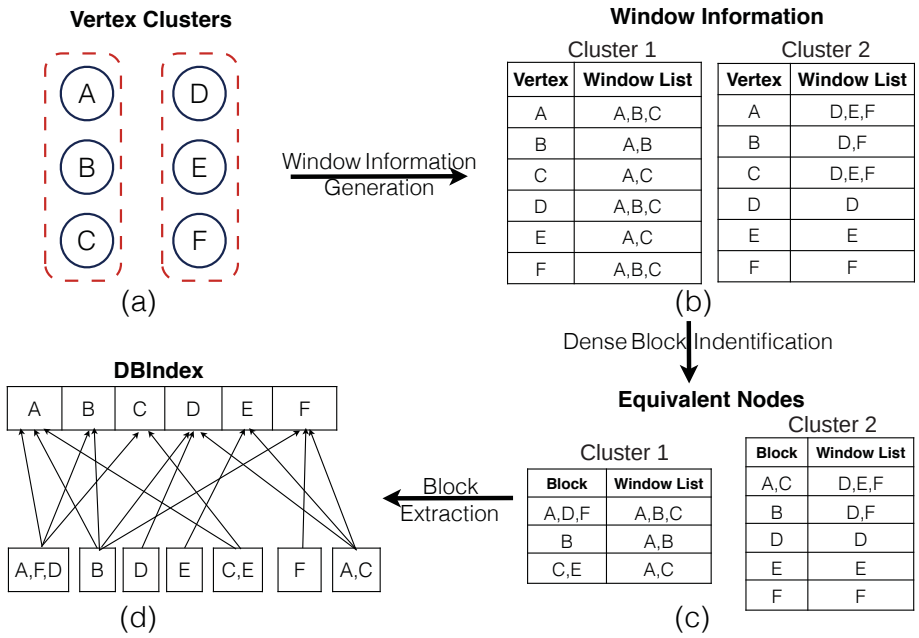


Figure 3.4: DBIndex Construction over Social Graph in Fig. 3.1. (a) shows two clusters after MinHash clustering; (b) shows the window information of involved vertices within each cluster; (c) shows the dense blocks within each cluster; (d) provides the final DBIndex.

Next, we compute a hash signature (denoted by $H(v)$) for $W(v)$ based on applying m hash functions on the set $W(v)$. Nodes with identical hash signatures are considered to have highly similar windows and are grouped into the same cluster. To ensure that our approach is scalable, we do not retain $W(v)$ in memory after its hash signature $H(v)$ has been computed and used to cluster v; i.e., our approach does not materialize all the windows in the memory to avoid high space complexity. Let $\mathcal{C} = \{C_1, C_2, \cdots\}$ denote the collection of clusters obtained from the first step, where each C_i is a subset of nodes.

The second step is to identify dense blocks from each of the clusters computed in the first step. The identification of dense blocks in each cluster C_i is based on the notion of node equivalence defined as follows. Two distinct nodes $u, v \in C_i$ are defined to be equivalent (denoted by $u \equiv v$) if u and v are both contained in the same set of windows; i.e., for every window $W(x), x \in C_i$, $u \in W(x)$ iff $v \in W(x)$.

Algorithm 1 CreateDBIndex

Input: Graph $G = (V, E)$, window function W
Ensure: DBIndex I
1: Initialize DBIndex I: $nodes(I) = V$, $blocks(I) = \emptyset$, $links(I) = \emptyset$
2: **for all** $v \in V$ **do**
3: Traverse G to determine $W(v)$
4: Compute the hash signature $H(v)$ for $W(v)$
5: **end for**
6: Partition V into clusters $\mathcal{C} = \{C_1, C_2, \cdots\}$ based on hash signatures $H(v)$
7: **for all** $C_i \in \mathcal{C}$ **do**
8: **for all** $v \in C_i$ **do**
9: Traverse G to determine $W(v)$
10: **end for**
11: `IdentifyDenseBlocks` (I, G, W, C_i)
12: **end forreturn** I

Based on this notion of node equivalence, C_i is partitioned into blocks of equivalent nodes. To perform this partitioning, we need to again traverse the graph for each node $v \in C_i$ to determine its window $W(v)$[5].

However, since C_i is now a smaller cluster of nodes, we can now materialize all the windows for the nodes in C_i in memory without exceeding the memory space. In the event that a cluster C_i is still too large for all its vertex windows to be materialized in main memory, we can further partition C_i into equal size sub-clusters. This re-partition process can be recursively performed until the sub clusters created are small enough such that the windows for all nodes in the sub clusters fit in memory.

Recall that a block B is a dense block if B contains at least two nodes and B is contained in at least two windows. Thus, we can classify the nodes in each C_i as either dense or non-dense nodes: a node $v \in C_i$ is classified as a *dense node* if v is contained in a dense block; otherwise, v is a non-dense node.

For each dense block B in C_i, we update the blocks and links in the DBIndex I as follows: we insert B into $blocks(I)$ if $B \notin blocks(I)$, and we insert (B, v) into $links(I)$

[5]Note that although we could have avoided deriving $W(v)$ a second time if we had materialzed all the derived windows the first time, our approach is designed to avoid the space complexity of materializing all the windows in memory at the cost of computing each $W(v)$ twice. We present an optimization in Section 3.4.2.2 to avoid the recomputation cost.

for each $v \in C_i$ where $B \subseteq W(v)$. If all the blocks in C_i are dense blocks, then we are done with identifying dense blocks in C_i; otherwise, there are two cases to consider. For the first case, if all the nodes in C_i are non-dense nodes, then we also terminate the process of identifying dense blocks in C_i and update the blocks and links in the DBIndex I as before: we insert each non-dense block B into $blocks(I)$, and we insert (B, v) into $links(I)$ for each $v \in C_i$ where $B \subseteq W(v)$. For the second case, if C_i has a mixture of dense and non-dense nodes, we remove the dense nodes from C_i and recursively identify dense blocks in C_i following the above two-step procedure.

Note that since the blocks are identified independently from each cluster, it might be possible for the same block to be identified from different clusters. We avoid duplicating the same block in $blocks(I)$ by checking that a block B is not already in $blocks(I)$ before inserting it into $blocks(I)$. The details of the construction algorithm are shown as Algorithms 1, 2, and 3.

Algorithm 2 IdentifyDenseBlocks

Input: DBIndex I, Graph $G = (V, E)$, window function W, a cluster $C_i \subseteq V$

```
1: Partition C_i into blocks based on node equivalence
2: Initialize DenseNodes = ∅
3: for all dense block B do
4:     Insert B into blocks(I) if B ∉ blocks(I)
5:     Insert (B, v) into links(I) for each v ∈ C_i where B ⊆ W(v)
6:     DenseNodes = DenseNodes ∪ B
7: end for
8: if (DenseNodes = ∅) then
9:     for all block B do
10:        Insert B into blocks(I) if B ∉ blocks(I)
11:        Insert (B, v) into links(I) for each v ∈ C_i where B ⊆ W(v)
12:    end for
13: else if (C_i − DenseNodes ≠ ∅) then
14:    if (C_i ≠ DenseNodes) then
15:        RefineCluster (I, G, W, C_i − DenseNodes)
16:    end if
17: end if
```

Fig. 3.4 illustrates the construction of the DBIndex with respect to the social graph in Fig. 3.1(a) and 1-hop window using the MC algorithm. First, the set of

graph vertices are partitioned into clusters using MinHash clustering; Fig. 3.4(a) shows that the set of vertices $V = \{A, B, C, D, E, F\}$ are partitioned into two clusters $C_1 = \{A, B, C\}$ and $C_2 = \{D, E, F\}$.

For convenience, cluster 1 in Fig. 3.4(b) shows for each $v \in C_1$, the set of vertices whose windows contain v; i.e., $\{u|\ v \in W(u)\}$. Similarly, Cluster 2 in Fig. 3.4 (b) shows for each $v \in C_2$, the set of vertices whose windows contain v. Consider the identification of dense blocks in cluster C_1. As shown in Fig. 3.4 (c), based on the notion of equivalence nodes, cluster C_1 is partitioned into three blocks of equivalent nodes: $B_1 = \{A, D, F\}$, $B_2 = \{B\}$, and $B_3\{C, E\}$. Among these three blocks, only B_1 and B_3 are dense blocks. The MC algorithm then tries to repartition the window A, B, C using non-dense nodes in C_1, (i.e., B_2). Since B_2 is the only non-dense node, it directly outputs. At the end of processing cluster C_1, the DBIndex I is updated as follows: $blocks(I) = \{B_1, B_2, B_3\}$ and $links(I) = \{(B_1, \{A, B, C\}), (B_2, \{A, B\}),, (B_3, \{A, C\})\}$. The identification of dense blocks in cluster C_2 is of similar process.

We find it is non-trivial to precisely analyze the complexity of Algorithm 1. Here, we only offer a brief analysis. Suppose the MinHash cost is H and the total cost for k-bounded BFS for all vertex is B, Lines 1-5 has the complexity of $O(H+B)$. Lines 7-10 has the complexity of $O(B)$. A single execution of Algorithm 2 has the complexity of $O(|V|)$, since we can simply partition nodes using hashing. Suppose the iteration runs for K times, the total cost for Algorithm 2 and Algorithm 3 is $O(K|V|)$. Therefore the overall complexity of Algorithm 1 is $O(H + 2 * B + O(K|V|))$. H depends on the number of vertex-window mappings for a given query and B depends on the graph structure and number of hops. As we demonstrate in Section 6, the H and B are the major contribution of the indexing time. To reduce the index time, we provide further optimization techniques.

Algorithm 3 RefineCluster

Input: DBIndex I, Graph $G = (V, E)$, window function W, a cluster $C \subseteq V$

1: **for all** $v \in C$ **do**
2: Compute the hash signature $H(v)$ for $W(v) \cap C$
3: **end for**
4: Partition C into clusters $\mathcal{C} = \{C_1, C_2, \cdots\}$ based on hash signatures $H(v)$
5: **for all** $C_i \in \mathcal{C}$ **do**
6: `IdentifyDenseBlocks` (I, G, W, C_i)
7: **end for**

3.4.2.2 Estimated MinHash Clustering (EMC)

The MC approach described in the previous section requires the window of each node (i.e., $W(v), v \in V$) to be computed twice in order to avoid the high space complexity of materializing all the windows in main memory. For k-hop window function with a large value of k, the cost of graph traverals to compute the k-hop windows could incur a high computation overhead. Moreover, the cost of initial MinHash in MC approach equals to the initial number of vertex-window mappings, which is of the same order as graph traversal. For the larger hops, MinHash clustering would incur high computation cost.

To address these issues, we present an even more efficient approach, referred to as *Estimated MinHash Clustering (EMC)*, to optimize the construction of the DBIndex for k-hop window function with larger k.

The key idea behind EMC is based on the observation that for any two nodes $u, v \in V$, if their m-hop windows, $W_m(u)$ and $W_m(v)$, are highly similar and they are grouped into the same cluster, then it is likely that the n-hop windows of these two nodes, where $n > m$, would also be highly similar and grouped into the same cluster.

Using the above observation, we could reduce the overhead cost for constructing a DBIndex wrt a k-hop window function by clustering the nodes based on their k'-hop windows, where $k' < k$, instead of their k-hop windows.

To reduce the overhead of window computations, our EMC approach is similar

to the MC approach except for the first round of window computations (line 3 in Algorithm 1): EMC uses lower hop windows to approximate k-hop windows for the purpose of clustering the nodes in V. Thus, the hash signatures used for partitioning V are based on lower hop windows. This approximation clearly has the advantage of improved time-efficiency as traversing and Minhashing on lower hop window is of order of magnitude faster. For the extreme case, adapting 1-hop window of a node v requires only accessing the adjacent nodes of v. The tradeoff for this improved efficiency is that the "quality" of the dense blocks might be reduced (in terms of their sizes). However, our experimental results show that this reduction in quality is actually only marginal which makes this approximation a worthy tradeoff.

3.4.2.3 Justification of Heuristic

In the following, we show the theoretical justification of our heuristic: the Jaccard coefficient is increasing wrt number of hops for a large class of graphs. We assume that the degree of vertices follows the same distribution, which is true in most real-networks and random graph models[6]. This implies we can analyze vertices with their neighborhoods structure using a unified way.

We use d_i to indicate the degree of vertex i. For any vertex pair (u, v), their intersection on k-hop window consists of three part. We name them using $A = W_k(u) - W_k(v)$, $B = W_k(v) - W_k(u)$ and $C_k = W_k(u) \cap W_k(v)$. Clearly the Jaccard coefficient at k-hops can be expressed as follows:

$$J_k(u, v) = \frac{|C_k|}{|A_k| + |B_k| + |C_k|} \tag{3.2}$$

To deduct the relationship between $J_k(u, v)$ and $J_{k+1}(u, v)$, we prove the following lemma first:

[6]E.g. Social network, Preferential Attachment model etc. follow power-law distribution. However, our analysis do not restrict on power-law distribution.

Theorem 3.4.1. Let S be a collection of connected vertices, the number of newly discovered vertices by one-hop expansion from S is bounded by a function on $|S|$.

Proof. Consider a random variable Y_i indicate the newly discovered vertices from one-hop expansion from vertex i. Then the probability of $|Y_i| = y$ is can be analyzed as follows: there are d_i edges for vertex i. Since $|Y_i|$ is connected with S, one edge is fixed to link with a vertex in S. There are remaining $d_i - 1$ edges with y edges linked to the new vertices. In total, there are $\binom{|V|-1}{d_i-1}$ combinations with d_i edges. Therefore, the probability can be written as:

$$Prob(Y_i = y | v_i \in S) = \frac{\binom{|S|-1}{d_i-y-1}\binom{|V|-|S|}{y}}{\binom{|V|-1}{d_i-1}} \tag{3.3}$$

Thus, the expectation of Y_i is:

$$\begin{aligned}
E(Y_i|v_i \in S) &= \Sigma(y * Prob(Y_i = y|v_i \in S)) \\
&= \Sigma_{y=1}^{y=d_i-1}\left(\frac{\binom{|S|-1}{d_i-y-1}\binom{|V|-|S|}{y}}{\binom{|V|-1}{d_i-1}} * y\right) \\
&= \Sigma_{y=1}^{y=d_i-1}\left(\frac{\binom{|S|-1}{d_i-y-1}\binom{|V|-|S|-1}{y-1}}{\binom{|V|-1}{d_i-1}} * (|V| - |S|)\right) \\
&= (|V| - |S|) * \Sigma_{y=1}^{y=d_i-1}\frac{\binom{|S|-1}{d_i-y-1}\binom{|V|-|S|-1}{y-1}}{\binom{|V|-1}{d_i-1}} \\
&= (|V| - |S|) * \frac{\binom{|V|-2}{d_i-2}}{\binom{|V|-1}{d_i-1}} = \frac{(|V| - |S|) * (d_i - 1)}{|V| - 1}
\end{aligned} \tag{3.4}$$

Taking the expectation over all vertices in S, we can find the expectation of $E(Y|S) = \frac{(|V|-|S|)*(\overline{d}-1)}{|V|-1}$, where $\overline{d}$ is the average degree of the graph. We then define the event $X = \cup_{i=1}^{i=|S|} Y_i$, i.e. X is the number of newly discovered vertices for one-hop expansion

of entire S. By union bound, the expectation of X is:

$$\begin{aligned} E(X|S) &= E(\cup_{i=1}^{i=|S|} Y_i|S) \\ &\leq \Sigma_{i=1}^{i=|S|} E(Y_i|S) \\ &= \frac{|S|(|V|-|S|)(\overline{d}-1)}{|V|-1} = f(|S|) \end{aligned} \tag{3.5}$$

The bound is achieved when each vertices in S discovers non-overlapping neighbors, such as in the case of tree structure. Therefore, the newly discovered vertices are tightly bounded by a quadratic function on $|S|$. □ □

We thus use $f(m)$ to denote the number of newly discovered vertices from a base set of m connected vertices. Since u, v have identical degree distribution, their expected value $S_u = E(|W_k(u)|)$ and $S_v = E(|W_k(v)|)$ is likely to be the same, i.e. $S_u \cong S_v$. We further use $\alpha = \frac{|C|}{|A|+|C|}$ to denote the portion of shared components in u's k-hop neighborhood. Likewise, we use $\beta = \frac{|B|}{|B|+|C|}$ for $W_k(v)$. Since vertices have identical degree distribution, α and β are likely to be the same, i.e. $\alpha \cong \beta$. Now, the $J_{k+1}(u, v)$ for $(k+1)$-hop can be represented as follows:

$$\begin{aligned} J_{k+1}(u,v) &= \frac{|C_{k+1}|}{|A_{k+1}|+|B_{k+1}|+|C_{k+1}|} \\ &= \frac{|C_k| + \alpha * f(S_u) + \beta * f(S_v)}{|A_k|+|B_k|+|C_k|+f(S_u)+f(S_v)-\Delta} \end{aligned} \tag{3.6}$$

, the Δ here is to compensate the doubly counted portion on the overlapping: $(A_{k+1} \cup B_{k+1}) \cap C_{k+1}$. Therefore, it subsequently follows:

$$\begin{aligned} J_{k+1}(u,v) &\geq \frac{|C_k| + \alpha * f(S_u) + \beta * f(S_v)}{|A_k|+|B_k|+|C_k|+f(S_u)+f(S_v)} \\ &\cong \frac{|C_k| + 2\alpha * f(S_u)}{|A_k|+|B_k|+|C_k|+2*f(S_u)} \end{aligned} \tag{3.7}$$

Due to the fact that $\alpha = \frac{|C_k|}{|C_k|+|A_k|} \geq \frac{|C_k|}{|A_k|+|B_k|+|C_k|}$, it follows:

$$\begin{aligned} \frac{\alpha f(S_u)}{f(S_u)} &\geq \frac{|C_k|}{|A_k|+|B_k|+|C_k|} \Leftrightarrow \\ \frac{|C_k|+2\alpha * f(S_u)}{|A_k|+|B_k|+|C_k|+2*f(S_u)} &\geq \frac{|C_k|}{|A_k|+|B_k|+|C_k|} \\ &= J_k(u,v) \end{aligned} \tag{3.8}$$

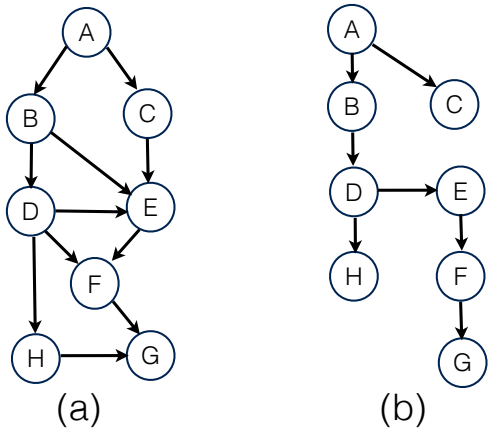
Therefore, our analysis shows that $J_k(u, v)$ is most likely increasing for random graphs with identical degree distribution.

3.4.3 Handling Updates

In this section, we overview how our DBIndex is maintained when there are updates to the input graph. There are two types of updates for graph data: updates to the attribute values associated with the nodes/edges and updates to the graph structure (e.g., addition/removal of nodes/edges). Since the DBIndex is an index on the graph structure which is independent of the attribute values in the graph, the DBIndex is not affected by updates to the graph's attribute values.

The efficient maintenance of the DBIndex in the presence of structural updates is challenging as a single structural change (e.g., adding an edge) could affect many vertex windows. To balance the tradeoff between efficiency of index update and efficiency of query processing, we have adopted a two-phase approach to maintain the DBIndex. The first phase is designed to optimize update efficiency where the DBIndex is updated incrementally whenever there are structural updates to the graph. The incremental index update ensures that the updated index functions correctly but does not fully optimize the query efficiency of the updated index in terms of maximizing the shared computations. The second phase is designed to optimize query efficiency where the DBIndex is periodically re-organized to maximize share computations.

As an example of how the DBIndex is updated incrementally, consider a structural



ID	PID	WD
A	nil	nil
B	A	nil
C	A	nil
D	B	nil
E	D	C
F	E	nil
G	F	H
H	D	nil

(c)

Figure 3.5: I-Index Construction over the Pathway DAG in Fig. 3.2. (a) shows the DAG structure; (b) provides the inheritance relationship discovered during the index construction; (c) shows the final I-Index.

change where a new edge is added to the input graph. Let S denote the subset of graph vertices whose windows have expanded (with additional vertices) as a result of the insertion of the new edge. Let $W'(v)$ denote the set of additional vertices in the vertex window of v for each vertex $v \in S$. Based on the identified changes to the vertex windows (i.e., S and $\{W'(v)|\ v \in S\}$), we construct a secondary DBIndex which is then merged into the primary DBIndex. As the identified changes are small relative to the entire graph and collection of vertex windows, the construction and merging of the secondary index can be processed efficiently relative to an index reorganization to fully optimize query efficiency.

3.5 Inheritance Index

DBIndex is a general index that can support both k-hop as well as topological window queries. However, the evaluation of a topological window function, W_t, can be further optimized due to its containment feature. In other words, the window of a descendant vertex completely covers that of one of its ancestors. This feature can be formally formulated in the following theorem.

Theorem 3.5.1. In a DAG, if vertex u is the ancestor of vertex v, the topological window of v, $W_t(v)$ completely contains the window of u, $W_t(u)$, i.e., $W_t(u) \subset W_t(v)$.

Proof. In a DAG, if u is the ancestor of v, then $u \rightsquigarrow v$. $\forall w \in W_t(u)$, then $w \rightsquigarrow u$. As $u \rightsquigarrow v$, then $w \rightsquigarrow v$. Thus, $w \in W_t(v)$ and the theorem is proved. □

Let us consider the BioPathway graph in Fig. 3.2 as an example. Fig. 3.5 (a) shows its abstract DAG. In (a), D is the ancestor of E. In addition, we can see that the window of D, $W_t(D)$ is $\{A, B, D\}$ and the window of E, $W_t(E)$ is $\{A, B, D, C, E\}$. It is easy to see that $W_t(D) \subset W_t(E)$.

Now, Theorem 3.5.1 provides us with opportunities for optimizing the space and computation of topological window queries. First, since the set of vertices corresponding to the window of a node, say u, is a superset of the set of vertices of its parent node, say v, there is no need to maintain the full set of vertices of the window at u. Instead, we only need to maintain the difference between $W_t(u)$ and $W_t(v)$. We note that in a DAG, it is possible for u to have multiple parents, $v_1, \cdots, v_k$. In this case, the parent which has the smallest difference with u can be used; where there is a tie, it is arbitrarily broken. We refer to this parent as the *closest* parent. For instance, in Fig. 3.5 (a), instead of maintaining $\{A, B, D, C\}$ for $W_t(E)$, it can simply maintain the difference to $W_t(D)$ which is $\{C\}$. This is clearly more space efficient.

Second, using a similar logic, the aggregate computation at a node u can actually reuse the aggregate result of its closest parent, v. Referring to our example, the aggregation result of $W_t(D)$ can be simply passed or inherited to $W_t(E)$ and further aggregated with the difference set ($\{C\}$) in $W_t(E)$ to generate the aggregate value for $W_t(E)$. Fig. 3.5 (b) indicates the inheritance relationship that the values of the father can be inherited to the child in the tree.

Thus, we propose a new structure, called the **inheritance index**, *I-Index*, to support efficient processing of topological window queries. In *I-Index*, each vertex v maintains two information.

- The first information is the ID of the closest parent (say u) of v. We denote this as PID(v).
- The second information is the difference between $W_t(v)$ and $W_t(u)$. We denote this as WD(v).

With PID(v), we can retrieve $W_t(u)$, and combining with WD(v), we can derive $W_t(v)$. Likewise, we can retrieve the aggregation result of u which can be reused to compute v's aggregation result.

Fig. 3.5(c) shows the I-index of our example in Fig. 3.5(a). In the figure, I-Index is represented in a table format; the second column is the PID and the third indicates the WD.

3.5.1 Index Construction

Building an I-Index for a DAG can be done efficiently. This is because the containment relationship can be easily discovered using a topological scan. Algorithm 4 lists the pseudo code for index creation. The scheme iterates through all the vertices in a topological order. For vertex v, the processing involves two steps. In the first step, we determine the closest parent of v. This is done by comparing the cardinalities of the windows of v's parents, and find the parent with largest value. The corresponding PID is recorded in the *PID* field of I-Index (Lines 7-12). In the second step, the window of v, $W_t(v)$, is pushed to its children (Lines 16-18). When the processing of v finishes, its window can be discarded. This frees up the memory space, which makes the scheme memory efficient.

We not that the complexity of Algorithm 4 is non-trivial to analyze. This dues to the difficulty of analyzing of the number of ancestors of each vertex. Suppose the average number of ancestors for each vertex is H, then Algorithm 4 is of complexity $O(H|V| * d)$, where d is the average degree of the graph. This complexity is close

to the output complexity. That is to gather the all vertex-window mapping, at least $O(H|V|)$ elements needs to be outputted. Thus the indexing time complexity is reasonably efficient.

We further note that the size of *I-Index* is hard to be precisely evaluated. This dues to the difficulty of analyzing the window difference. Assume the average size of window difference is D, then the size of *I-Index* is $O(D|V|)$. Although D can be as large $O(|V|)$, our experimental results indicate that the index size is always comparable to the graph size. We defer this discussion to section 3.6. Furthermore, it is possible to reduce the index size (should it be a concern) by employing compression techniques.

Algorithm 4 CreateI-Index

Input: Input graph: G
Ensure: Inheritance Index: $IIndex$

```
1: IIndex ← ()
2: p ← ()                                   ▷ stores the window for each vertex
3: c ← ()                                   ▷ stores the cardinality of window for each vertex
4: for all v ∈ topological order do
5:     WD ← −∞                              ▷ the window difference
6:     bestu ← nil
7:     for all u ∈ v.parent do
8:         if c[u] > diff then
9:             diff ← c[u]
10:            bestu ← u
11:        end if
12:    end for
13:    IIndex[v].WD ← WD
14:    IIndex[v].PID ← bestu
15:    p[v] ← p[v] ∪ v
16:    for all u ∈ v.child do
17:        p[u] ← p[u] ∪ p[v]
18:    end for
19:    c[v] ← |p[v]|                        ▷ update window cardinality
20:    p[v] ← ()                            ▷ release memory
21: end for
```

3.5.2 Query Processing using I-Index

By employing the I-Index, window aggregation can be processed efficiently for each vertex according to the topological order. Algorithm 5 provides the pseudo code for the query processing. Each vertex v's window aggregation value can be calculated by using the following formula:

$$\Sigma(W_t(v)) = \Sigma(W_t(v.PID), \Sigma(v.WD)) \tag{3.9}$$

where Σ is the aggregate function. As the vertex is processed according to the topological order, $W_t(v.PID)$ would have already been calculated while processing v's parent and thus can be directly used for v without any recompution. In general, v's window aggregation is achieved by utilizing its parent's aggregate value and window difference sets. This avoids repeated aggregate computation and achieves the goal of computation sharing between a vertex and its parent. In so doing, the computation overhead can be further reduced. Take the index provided in Fig. 3.5 (c) as an example, assume the query wants to calculate the sum value over each window for every vertex. As a comparison, the number of add operations are 33, 22, 16 for the cases without any index, with DBIndex and with I-Index index respectively.

Algorithm 5 QueryProcessingOverIIndex

Input: Input graph G, aggregate function Σ, inheritance index $IIndex$
Ensure: w ▷ The aggregation result of each vertex

```
1: w ← ()
2: for all v ∈ topological order do
3:     u ← IIndex[v].PID
4:     WD ← IIndex[w].WD
5:     S ← v.val
6:     S ← Σ(S, w[u])
7:     for all t ∈ WD do
8:         S ← Σ(S, t.val)
9:     end for
10:    w[v] ← S
11: end forreturn w
```

As the query processing in Algorithm 5 basically scans the *I-Index*, the query complexity essentially correlates to the index size. As we shown in the experiment session, the query can be performed efficiently in various graph conditions. We defer the discussion to Section 6.

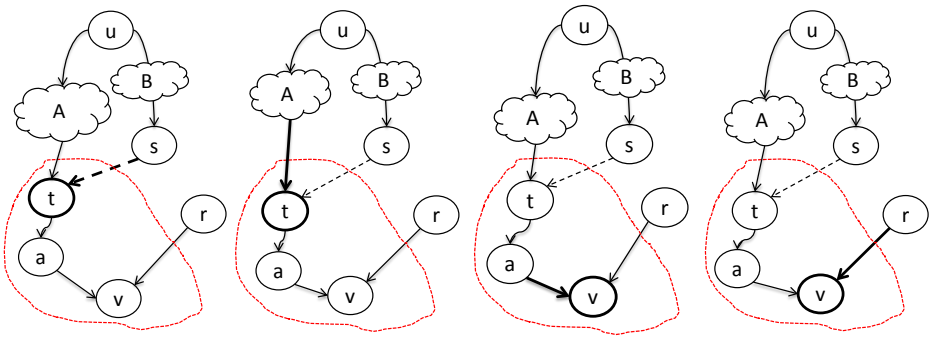
3.5.3 Handling Updates

We cover the updates handling in this section. For attribute updates, *I-Index* is not affected since *I-Index* is only structure related. Structure updates on *I-Index* consists of node updates and edge updates. It is easy to handle the case where an isolated node is added or delete, since it does not affect any other nodes in the graph. Adding (resp. deleting) a node with edges can be done via edge insertions (resp. deletions). Here we focus on describing single edge insertion and deletion. We use I to denote the *I-index* and $I(v)$ to denote the index entry of v.

During an update of edge $e(s,t)$, there are two types of vertices are affected. The first type contains single node t, which is the endpoint of edge e. The second type of nodes contains all the descendants of t. There are altogether four special cases needs to be consider during the updates. We illustrate the four cases with aids of Fig. 3.6. In Fig. 3.6, the dashed edge $e(s,t)$ is the edge to be updated (added or deleted). The node u is the *lowest common ancestor*(LCA) of t and s. The cloud shape A and B are nodes in between of u,s and u,t. Since u is the LCA, A and B are thus disjoint. Bold edge $e(c,d)$ indicate that $I(d).PID$ is c. We distinguish and handle the four cases as follows:

Case I ($I(t).PID = s$): As shown in Fig. 3.6 (a), during deletion, t needs to choose a parent from A to be its $I(t).PID$. $I(t).WD$ needs to be updated accordingly. In this case, no insertion needs to be considered since if there is no edge between s and t, $I(t).PID$ cannot be s.

Case II ($I(t).PID \neq s$): As shown in Fig. 3.6 (b), during insertion, B needs



(a) Edge Update Case 1 (b) Edge Update Case 2 (c) Edge Update Case 3 (d) Edge Update Case 4

Figure 3.6: Updates on *I-Index*. Cloud shape indicate the nodes in the subgraph between the endpoint nodes. The dashed circle indicate the affected range of updates. The bold arrow indicates the PID field *I-Index*

to be excluded from $I(t).WD$. During deletion, any node in B that cannot reach t needs to be included in $I(t).WD$. Since A and B are disjoint, every node in B needs to be removed from $I(t).WD$

Case III ($t \rightsquigarrow I(v).PID$): As shown in Fig. 3.6 (c), during insertion, any node in B needs to be removed from $I(v).WD$. During deletion, any node in B that reaches v but cannot reach a needs to be added to $I(v).WD$.

Case IV ($t \nrightarrow I(v).PID$): As shown in Fig. 3.6 (d), during insertion, any node in B that cannot reach r needs to be included into $I(v).WD$. During deletion, any node in B that cannot reach v needs to be excluded from $I(v).WD$.

During structure updates, essential operations are computing A, B and performing reachability queries. Computing A, B can be done via existing techniques such as [**?**, **?**] while reachability query can be supported by indexing methods such as [**?**]. We defer exploring for more efficient updating algorithms to future work

3.6 Experimental Evaluation

In this section, we present a comprehensive experimental evaluation of our solutions using several real-world information networks and various synthetic datasets. Since

the focus of this paper is on query processing efficiency, we do not evaluate the efficiency of index updates. All experiments are conducted on an Amazon EC2 r3.2xlarge machine[7], with an 8-core 2.5GHz CPU, 60GB memory and 320GB hard drive running with 64-bit Ubuntu 12.04. As the source code of EAGR is not available, we implemented it and used it as a reference in our comparative study. All algorithms are implemented in Java and run under JRE 1.6.

Name	Type	# of Vertices	# of Edges
LiveJournal1	undirected	3,997,962	34,681,189
Pokec	directed	1,632,803	30,622,564
Orkut	undirected	3,072,441	117,185,083
DBLP	undirected	317,080	1,049,866
YouTube	undirected	1,134,890	2,987,624
Google	directed	875,713	5,105,039
Amazon	undirected	334,863	925,872
Stanford-web	directed	281,903	2,312,497

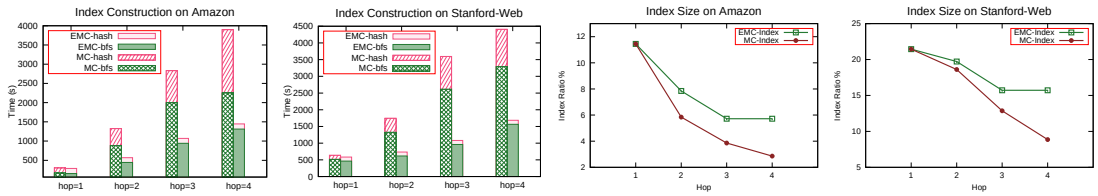
Table 3.1: Large Scale Real Data

Datasets. For real datasets, we use 8 information networks which are available at the Stanford *SNAP* website [8]: LiveJournal1, Pokec, Orkut, DBLP, YouTube, Google, Amazon and Stanford-web. The detail description of these datasets is provided in Table 3.1.

For synthetic datasets, we use two widely used graph data generators. We use the *DAGGER* generator [**?**] to generate all the synthetic DAGs and the SNAP graph data generator at the Stanford SNAP website to generate non-DAG datasets. For each dataset, each vertex is associated with an integer attribute.

Query. In all the experiments, the window query is conducted by using the SUM() as the aggregate function over the integer attribute in each dataset.

[7]http://aws.amazon.com/ec2/pricing/
[8]http://snap.stanford.edu/snap/index.html



(a) Index Built on Amazon (b) Index Built on Stanford-web (c) Index Size on Amazon (d) Index Size on Stanford-web

Figure 3.7: Index Construction Analysis for EMC and MC. (a) and (b) depict the index time for the Amazon and Stanford-web networks; (c) and (d) shows the index size for the Amazon and Stanford-web datasets

3.6.1 Comparison between MC and EMC

We first compare the effectiveness of the two DBIndex construction algorithms: Min-Hash Clustering (MC) and Estimated MinHash Clustering (EMC). We look at the index construction time, index sizes and query performance. All these experiments are conducted based on two real world datasets: Amazon and Stanford-web.For both datasets, we run a series of k-hop queries.[9] For queries with hop count larger than 1, EMC uses 1-hop information for the initial clustering.

Index Construction. Figs. 3.7 (a) and (b) compare the index construction time between MC and EMC when we vary the windows from 1-hop to 4-hop for the Amazon and Stanford-web graphs respectively. To better understand the time difference, the construction time is split into two parts: the MinHash cost (EMC-hash or MC-hash) and the breadth-first-search traversal (to compute the k-hop window) cost (EMC-bfs or MC-bfs). The results show the same trend for the two datasets. We made several observations. First, as the number of hops increases, the indexing time increases as well. This is expected as a larger hop count results in a larger window size and the BFS and computation time increase correspondingly. Second, as the hop count increases, the difference between the index time of EMC and that of MC widens. For instance, as shown in Figs. 3.7(a) and (b), for the 4-hop window

[9]For the Stanford-web graph, which is directed, the k-hop windows are directed k-hop windows where $u \in W(k)$ if there is a directed path of at most k hops from vertex v to vertex u.

queries, compared to MC, EMC can save 62% and 66% construction time for the Amazon and Stanford-Web datasets respectively. EMC benefits from both the low MinHash cost and low BFS cost. From Figs. 3.7 (a) and (b), we can see that the MinHash cost of MC increases as the number of hops increases, while that for EMC remains almost the same as the 1-hop case. This shows that the cost of MinHash becomes more significant for larger windows. Thus, using 1-hop clustering for larger hop counts reduces the MinHash cost in EMC. Similarly, as EMC saves on BFS cost for k-hop queries where $k > 1$, the BFS cost of EMC is much smaller than that of MC as well.

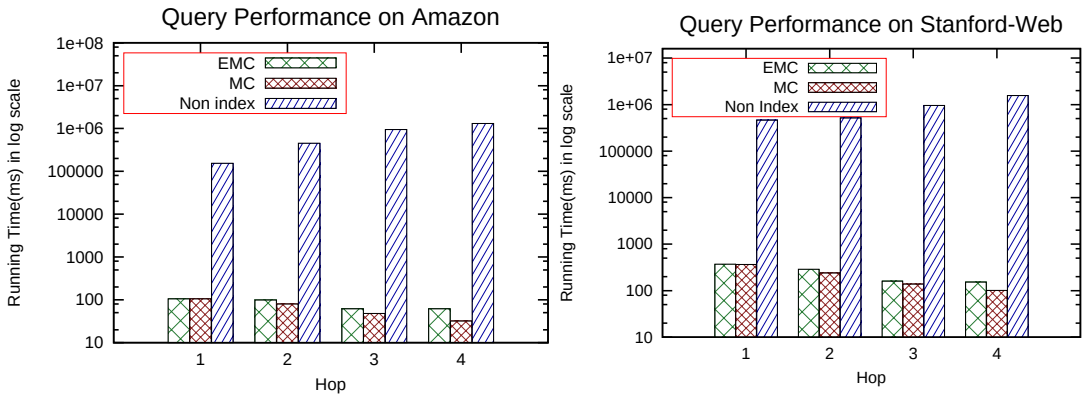
Index Size. Figs. 3.7 (c) and (d) present the effect of hop counts on the index size for the Amazon and Stanford-web datasets respectively. The y-axis shows the index ratio which is the index size over the original graph size. The insights we derive are: First, the index size is rather small compared to the original graph - it varies from 3% to 12% of the original graph for the Amazon dataset and from 8% to 22% for the Stanford-web dataset. Second, the index size decreases as the number of hops increases. While this appears counter-intuitive initially, it is actually reasonable - a larger hop results in a bigger window, which leads to more dense blocks. Third, the index ratio of EMC is slightly larger than that of MC for larger hop count. This indicates that MC can find more dense blocks than EMC to reduce the index size. Fourth, the index ratio on the Amazon dataset is much smaller than the ones on the Stanford-web dataset. This is because the Amazon dataset is undirected while the Stanford-web dataset is directed. For the Stanford-web dataset, since we use directed k-hop windows, the window size is naturally smaller.

Query Performance. Figs. 3.8 (a) and (b) present the query time of MC and EMC on the two datasets respectively as we vary the number of hops from 1 to 4. To appreciate the benefits of an index-based scheme, we also implemented a *Non-indexed* algorithm which computes window aggregate by performing k-bounded breadth first

search for each vertex individually in real time. In Figures 3.8 (a) and (b), the execution time shown on the y-axis is in log scale. The results show that the index-based schemes outperform the non-index approach by four orders of magnitude. For instance, for the 4-hop query over the Amazon graph, our algorithm is 13,000 times faster than the non-index approach. This confirms that it is necessary to have well-designed index support for efficient window query processing. By utilizing DBIndex, for these graphs with millions of edges, every aggregation query can be processed in just between 30ms to 100ms for the Amazon graph and between 60ms to 360ms for the Stanford-web graph. In addition, we can see that as the number of hops increases, the query time decreases. This is the case because a larger hop count eventually results in a larger number of dense blocks where more (shared) computation can be salvaged. Furthermore, we can see that the query time of EMC is slightly longer than that of MC when the number of hops is large. This is expected as EMC does not cluster based on the complete window information; instead, it uses only partial information derived from the 1-hop windows. However, the performance difference is quite small even for 4-hop queries- for the Amazon dataset, the difference is only 20ms; and for the Stanford-web graph, the difference is 35ms. For small number of hops, the time difference is even smaller. This performance penalty is acceptable as tens of milliseconds time difference will not affect user's experience. As EMC is significantly more efficient than MC in index construction, EMC may still be a promising solution to many applications. As such, in the following sections, we adopt EMC for DBIndex in our experimental evaluations.

3.6.2 Comparison between DBIndex and EAGR

In this set of experiments, we compare DBIndex and EAGR [20] using both large-scale real and synthetic datasets. Like [20], for each dataset, EAGR is run for 10 iterations in the index construction.



(a) Amazon Graph

(b) Stanford-web Graph

Figure 3.8: Query Performance Comparison of MC and EMC

3.6.2.1 Real Datasets

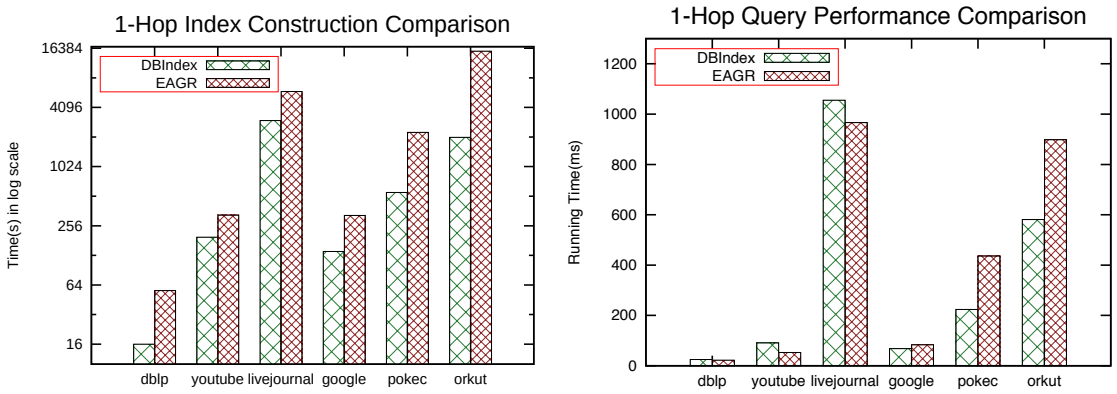
We first study the index construction and query time performance of DBIndex and EAGR for 1-hop and 2-hop windows using 6 real datasets: DBLP, Youtube, Livejournal, Google, Pokec and Orkut. The results for 1-hop window and 2-hop window are presented in Figs. 3.9 and 3.10 respectively. As shown in Figs. 3.9(a) and 3.10(a), both DBIndex and EAGR can build the index for all the real datasets for 1-hop but EAGR ran out of the memory for 2-hop window queries on LiveJournal and Orkut datasets. This further confirms that EAGR incurs high memory usage as it needs to build the FPT and maintain the vertex-window mapping information. We also observe that DBIndex is significantly faster than EAGR in index creation. We emphasize that the time is shown in logarithmic scale. For instance, for Orkut dataset, EAGR takes 4 hours to build the index while DBIndex only takes 33 minutes.

Fig. 3.9 (b) and Fig. 3.10 (b) show the query performance for 1-hop and 2-hop queries respectively. The results indicate that the query performance is comparable. For most of the datasets, DBIndex is faster than EAGR. In some datasets (e.g. Orkut and Pokec), DBIndex performs 30% faster than the EAGR. We see that, for 1-hop queries on Youtube and LiveJournal datasets and 2-hop queries on Youtube dataset, DBIndex is slightly slower than EAGR. We observe that these datasets are very sparse

graphs where the intersections among windows are naturally small. For very sparse graphs, both DBIndex and EAGR are unable to find much computation sharing. In this case, the performance of DBIndex and EAGR is very close. For instance, in the worse case for livejournal, DBIndex is 9% slower than EAGR where the actual time difference remains tens of millionseconds.

Another insight we gain is that as expected, compared to Fig. 3.9 (b), 2-hop query runs faster for both algorithms. This is because there is more computation sharing for 2-hop window query.

In summary, DBIndex takes much shorter time to build but offers comparable, if not much faster, query performance than EAGR.



(a) Index Construction (b) Query Performance

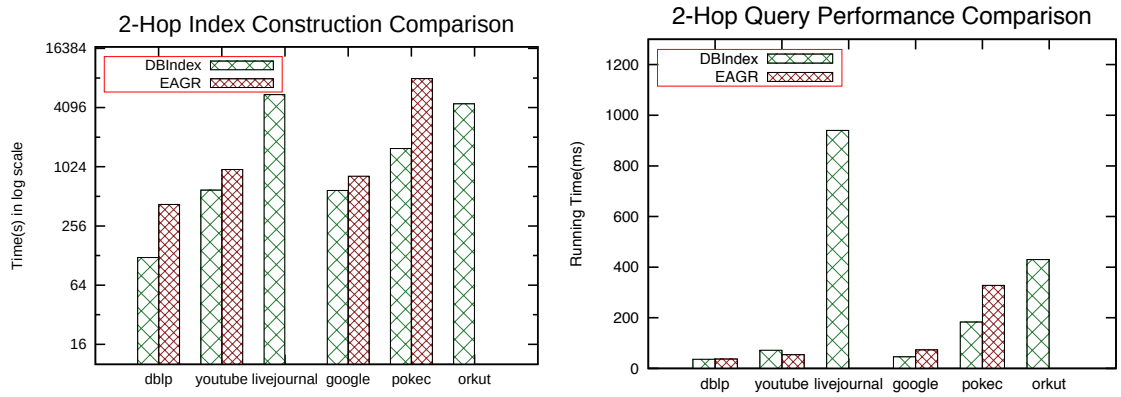
Figure 3.9: Comparison between DBIndex and EAGR for 1-hop query

3.6.2.2 Synthetic Datasets

To study the scalability of DBIndex under large-scale networks, we generated synthetic datasets using the SNAP generator.

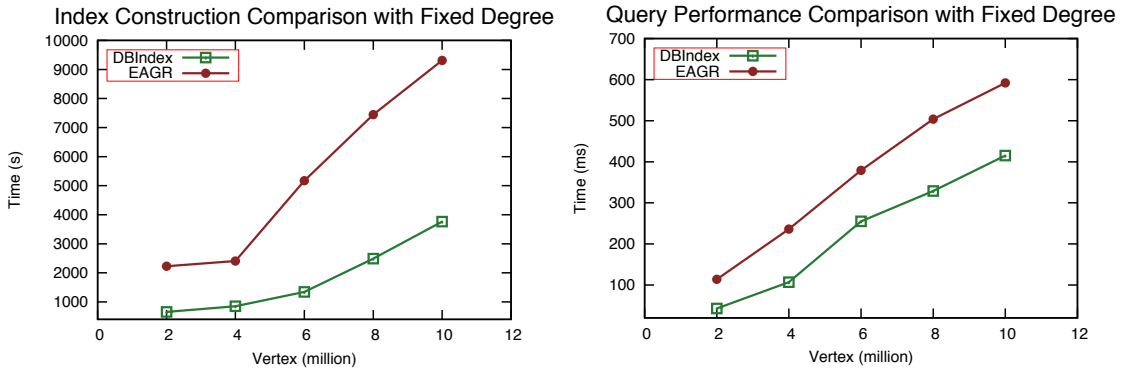
Impact of Number of Vertices. First, we study how the performance changes when we fix the degree [10] at 10 and vary the number of vertices from 2M to 10M. Figs. 3.11 (a) and (b) show the execution time for index construction and query per-

[10]Degree means average degree of the graph. The generated graph is of Erdos-Renyi model



(a) Index Construction (b) Query Performance

Figure 3.10: Comparison between DBIndex and EAGR for 2-hop query

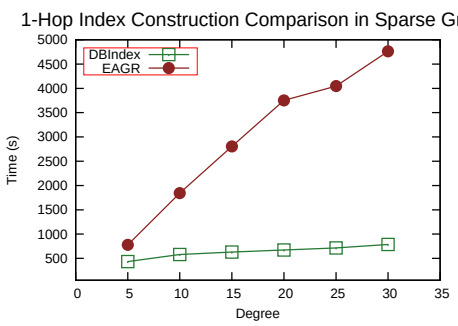


(a) Index Construction (b) Query Performance

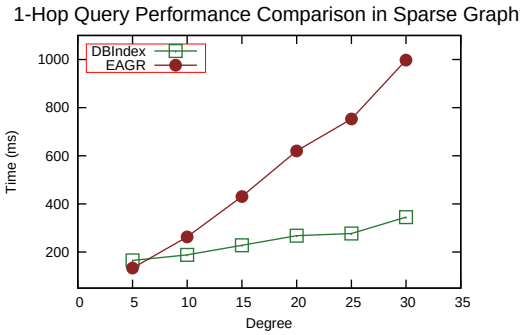
Figure 3.11: Impact of number of vertices

formance respectively. From the results, we can see that DBIndex outperforms EAGR in both index construction and query performance. For the graph with 10M vertices and 100M edges, the DBIndex query time is less than 450 milliseconds. Moroever, when the number of vertices changes from 2M to 10M, the query performance only increases 3 times. This shows that DBIndex is not only scalable, but offers acceptable performance..

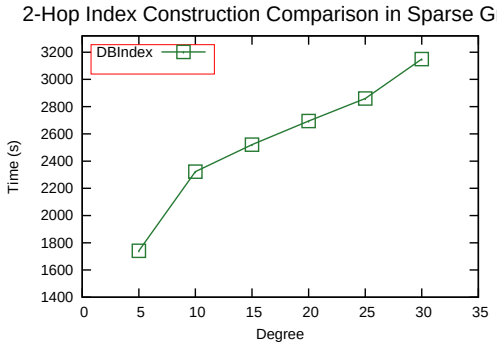
Impact of Degree over Sparse Graphs. Our proposed DBIndex is effective when there is significant overlap between windows of neigboring nodes. As such, it is interesting to study how it performs for sparse graph where the nodes may not share many common neigbors. So, in these experiments, we study the impact of



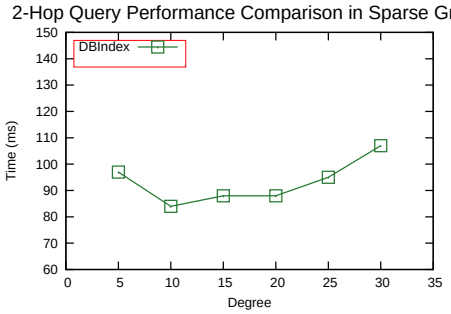
(a) Index Construction



(b) Query Performance



(c) Index Construction



(d) Query Performance

Figure 3.12: Impact of Degree over Sparse Graphs over 2M vertices. (a) and (b) are the results for 1-hop query; (c) and (d) are the results for 2-hop query.

degree when the graph is relatively sparse. We fix the number vertices of 2M and vary the vertex degree from 5 to 30. Figs. 3.12 (a) and (c) present the results on index construction for 1-hop and 2-hop queries respectively. For 1-hop queries, as degree increases, the time for index construction also increases. However, the index creation time of DBIndex increases much slower than EAGR. This is because EAGR incurs relatively more overhead to handle multiple FPT creation and reconstruction. For 2-hop queries, EAGR failed to run. This is because even for a degree 5 sparse graph, the initial vertex-mapping can be as large as 90GB in a linked list manner, which exceeds the available memory. Note that the size becomes even larger when it is stored in a matrix manner. Therefore, we can only show the results of DBIndex. In Fig. 3.12 (c), indexing time of DBIndex increases as the degree increases. This is expected as a bigger degree increases the overhead of graph traversal time to collect

the window.

Fig. 3.12 (b) and (d) show the results on query time for 1-hop and 2-hop queries respectively. We observe a similar pattern for the index construction time: for 1-hop queries, the query time increases with increasing degree but at a much slower rate than EAGR; for 2-hop queries, we observe in Fig. 3.12 (d) that the query performance of DBIndex hovers around 100ms, which is much smaller than that of 1-hop query performance. This is because there are more dense blocks in the 2-hop case, in which case the query time can be faster compared to the 1-hop case.

Impact of Degree over Dense Graphs. We study the impact of degree over very dense graphs with 200k vertices when the degree changes from 80 to 200. Figs. 3.13 (a) and (c) show the execution time for index construction for 1-hop and 2-hop queries respectively. From the results, we can see that DBIndex performs well for dense graphs as well. As the degree increases, EAGR's performance degrades much faster than DBIndex. For 2-hop queries, as shown in Figs. 3.13 (b) and (d), EAGR is only able to work on the dataset with degree 80 due to the memory issue. Even though the number of vertices is relatively small (only 200k), the number of edges is very large when the degree becomes big (e.g. 40M edges with degree of 200).

Figs. 3.13 (b) and (d) show the results on query performance for 1-hop and 2-hop queries respectively. The results are consistent with that for sparse graphs - DBIndex is superior over EAGR.

In summary, the insight we obtain is that the scalability of EAGR is highly limited by its approach to build the index over the vertex-window mapping information. EAGR is limited by two factors: the graph size and the number of hops. DBIndex achieves better scalability as it does not need to create a large amount of intermediate data in memory.

(a) Index Construction

(b) Query Performance

(c) Index Construction

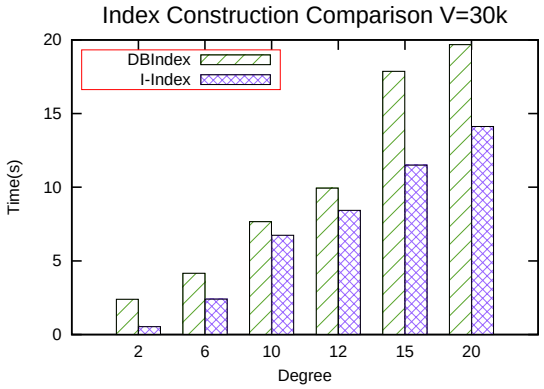
(d) Query Performance

Figure 3.13: Impact of Degree over Dense Graphs over 200K Vertices. (a) and (b) are the results for 1-hop query; (c) and (d) are the results for 2-hop query.

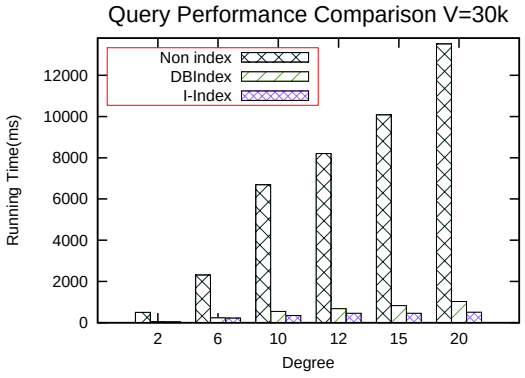
3.6.3 Evaluation of I-Index

In this set of experiments, we evaluate I-Index. All the datasets are generated from the DAGGER generator.

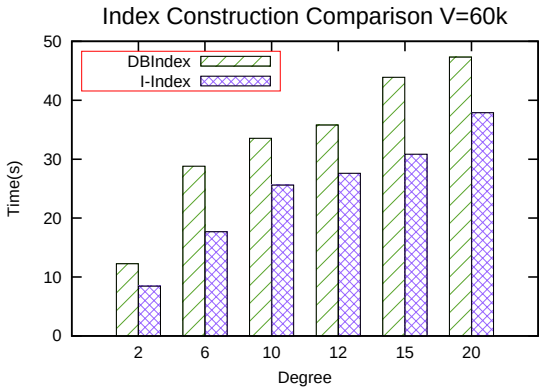
Impact of Degree. First, we evaluate the impact of degree changes when we fix the number of vertex as 30k and 60k. We compare *DBIndex* with I-Index. In the query results, we also implement one non-index algorithm which dynamically calculates the window and then performs the aggregation. For indexing time, as shown in Figures 3.14 (a) and (c), as the index size increases, both the indexing time of DBIndex and I-Index increase. However I-Index is more efficient than DBIndex, this is due to the special containment optimization used. We observe that the index construction time is almost the same as the one time non-index query time. In other



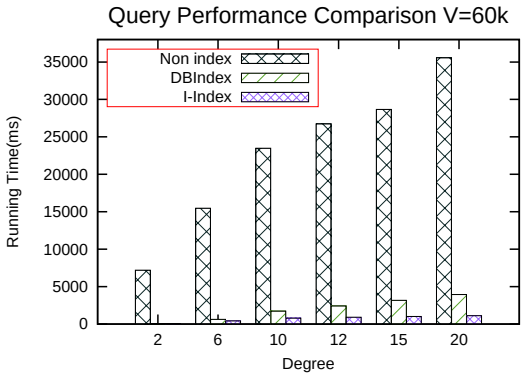
(a) Index Construction



(b) Query Performance



(c) Index Construction



(d) Query Performance

Figure 3.14: Impact of Degree. (a) and (b) are the results for 30K vertices; (c) and (d) are the results for 60K vertices.

words, we can use one query time to create the index which is able to provide much faster query processing for subsequent queries. In terms of query performance, shown in Figs. 3.14 (b) and (d), the non-index approach is, on average, 20 times slower than the index-based schemes. I-Index outperforms DBIndex by 20% to 30%. The results clearly show that I-Index outperforms DBIndex for topological window in both index construction and query performance. Therefore, in the following experiments, we only present the results for I-Index.

Impact of Number of Vertices. Next, we study how the performance of I-Index is affected when we fix the degree and vary the number of vertices from 50k to 350K. Figs. 3.15 (a) and (c) show the index construction time when we fix the degree to 10 and 20 respectively. From the results, we see that the construction time

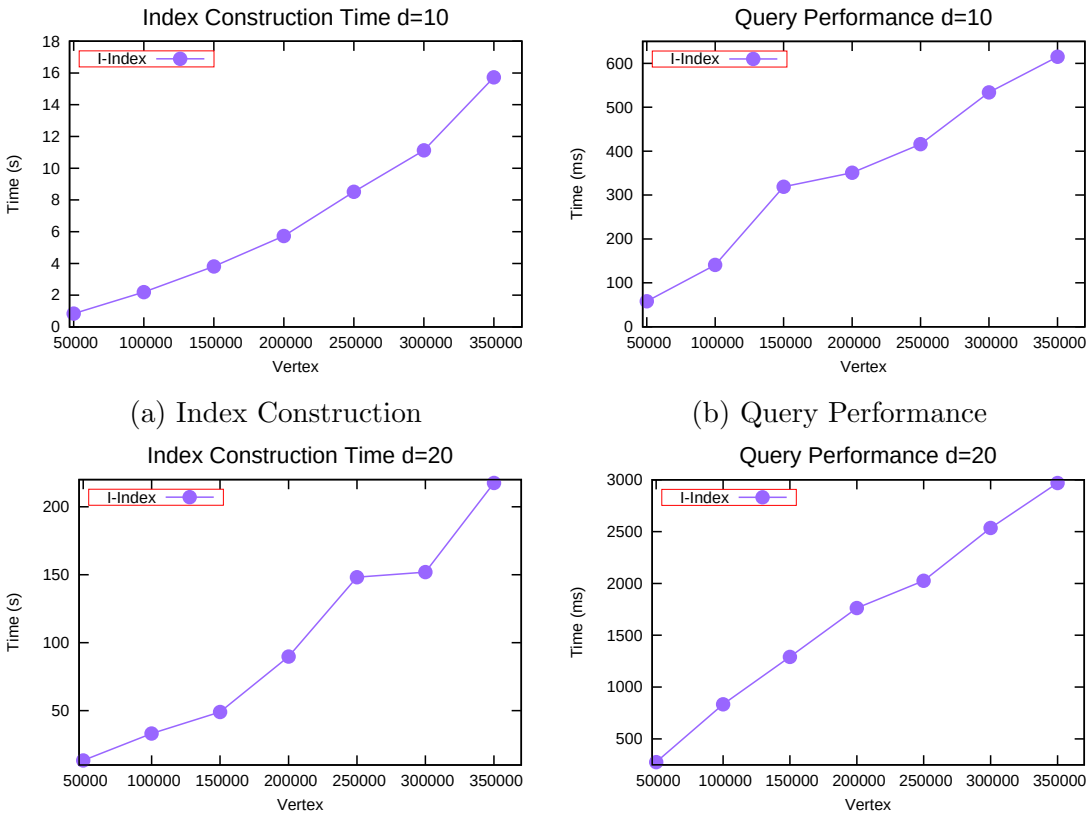
increases while the number of vertex increases and the construction time of a high degree graph is longer than that for low degree graphs. Figs. 3.15 (b) and (d) show the query time when we fix the degree to 10 and 20 respectively. As shown, the degree affects the query processing time - when the degree increases, the query time increases as well. We also observe that the query time is increasing linearly when the number of vertices increases. This shows the I-Index has good scalability.

Index Size. Fig. 3.16 presents the index size ratio (i.e. size of index divided by the size of original graph) under different degrees from 3 to 20. There are four different sizes of data used with 100k, 150k, 200k and 300k vertices. For every vertex setting, the index size maintains the same trend in various degrees. The index size is linear to the input graph size. As a graph gets denser, the difference field of the I-Index effectively shrinks. Thus, the index size in turn becomes smaller, which explains the bends in the figure.

3.7 Conclusion and Future Work

In this paper, we have proposed a new type of graph analytic query, *Graph Window Query*. We formally defined two instantiations of graph windows: k-hop window and topological window. We developed the Dense Block Index (DBIndex) to facilitate efficient processing of both types of graph windows. In addition, we also proposed the Inheritance Index (I-Index) that exploits a containment property of DAG to further improve the query performance of topological window queries. Both indices integrate window aggregation sharing techniques to salvage partial work done, which is both space and query efficient. We conducted extensive experimental evaluations over both large-scale real and synthetic datasets. The experimental results showed the efficiency and scalability of our proposed indices.

There remain many interesting research problems for graph window analytics.



(a) Index Construction (b) Query Performance

(c) Index Construction (d) Query Performance

Figure 3.15: Impact of the number of vertices with a fixed degree. (a) and (b) are the results for the graphs with degree 10; (c) and (d) are the graphs with degree 20.

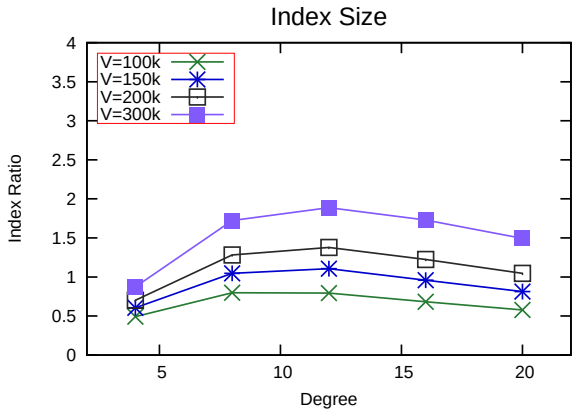


Figure 3.16: Index Ratio of Inheritance-Index

As part of our future work, we plan to explore structure-based window aggregations which are complex than attri- bute-based window aggregations. In structure-based aggregations, $W(v)$ refers to a subgraph of G instead of a set of vertices, and the aggregation function Σ (e.g., centrality, PageRank, and graph aggregation [25, 31]) operates on the structure of the subgraph $W(v)$.

Chapter 4

Efficient k-Sketch Discovery in Sequence Data

4.1 Introduction

Today's journalists must pore over large amounts of data to discover interesting newsworthy 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. To support automatic news theme discovery, early research efforts have focused on discovering newsworthy patterns derived from a single event, such as the *situational facts* [21], *one-of-few facts* [26] and FactsWatcher [**?**]. Recently, Zhang et. al. proposed a new type of news theme pattern, named *prominent streak* [30]. A *streak* corresponds to an event window with a sequence of consecutive events belonging to the same subject. Their goal is to discover all the non-dominated streaks to be news themes. However, the number of non-dominated streaks can be overwhelmingly large.

In this paper, we study the automatic discovery of a novel news theme, named *rank-aware* theme. Compared with previous works, the rank-aware theme is able to

capture the strikingness of an event window. We observe that such a rank-aware theme is an interesting pattern that has been frequently used:

1. (Feb 26, 2003) With 32 points, Kobe Bryant saw his 40+ scoring streak end at **nine** games, tied with Michael Jordan for **fourth** place on the all-time list[1].
2. (April 14, 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**)[2].
3. (May 28, 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[3].
4. (Jun 9, 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[4].
5. (Jul 22, 2011) Pelican Point recorded a maximum rainfall of 0.32 inches for **12 months**, making it the **9th driest** places on earth[5].

In the above news themes, there are a subject (e.g., Kobe Bryant, Stocks, Delhi), an event window (e.g., nine straight games, seventh consecutive days, past month), an aggregate function on an attribute (e.g., minimum points, count of gains, average of degrees), a rank (e.g., fourth, first time, highest), and a historical dataset (e.g., all time list, seven years, since 1952). These news theme indicators are summarized in Table 4.1. To avoid outputting near-duplicate news themes and control the result size, we further propose a novel concept named *Sketch*. A sketch contains k most representative rank-aware news themes under a scoring function that considers both

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

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

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

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

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

E.g.	Aggregate function	Event window	Rank
1	min(points)	9 straight games	4
2	sum(shot attempts)	82 games	10
3	count(gains)	7 consecutive days	1
4	average(degree)	past months (30 days)	1
5	max(raindrops)	12 months	9

Table 4.1: News theme summary

strikingness and diversity. Our objective is to discover sketches for each subject in the domain.

We study the sketch discovery problem under both offline and online scenarios. In the offline scenario, the objective is to efficiently discover the sketches for each subject from historical data. The major challenge lies in generating candidate themes from event windows. Since the number of event windows is quadratic, enumerating all of them is not scalable. By leveraging the subadditivity among the upper bounds of event windows, we design two effective pruning techniques to facilitate efficient candidate theme generation. Furthermore, we note that generating exact sketches from candidate themes is expensive. Thus, we design an efficient $(1 - 1/e)$ approximation algorithm by exploiting submodularity of the sketch discovery problem.

In the online scenario, fresh data is continuously fed into the system and our goal is to maintain the sketches for each subject up-to-date. When a new event about subject s arrives, many event windows of various lengths can be derived. For each derived event window, 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 $1/8$ approximation solution which only examines $2k$ candidate themes for each subject whose sketch is affected.

Our contributions are hereby summarized as follows:

- We study the automatic discovery of a new type of news theme that is common

in real-life reports but has not been addressed in previous works. We formulate it as a sketch discovery problem under a novel scoring function that considers both strikingness and diversity.

- We study the sketch discovery problem in both online and offline scenarios. In the offline scenario, we propose two novel pruning techniques to efficiently generate candidate themes. Then we design a $1 - 1/e$ greedy algorithm to discover the sketches for each subject. In the online scenario, we propose a $1/8$ approximation solution to efficiently support the complex updating pattern as each new event arrives.

- We conduct extensive experiments with four real datasets to evaluate 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 a 500 times speedup. In addition, we also perform an anonymous human study via Amazon Mechanical Turk[6] platform, which shows the effectiveness of sketch in news theme generation.

The rest of the paper is organized as follows. In Section 4.2, we summarize the related work. Section 4.3 formulates the sketch discovery problem. Section 4.4 presents the algorithms for offline sketch discovery. Section 4.5 describes the algorithms for processing and maintaining sketches in the online scenario. In Section 4.6, comprehensive experiments studies on both efficiency and effectiveness of our algorithms are conducted. Section A.1 discusses the extension of our methods. Finally, Section 4.7 concludes our paper.

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

4.2 Related Work

In this section, we briefly review three closely related areas: automatic news theme detection, frequent episode mining and top-k diversity query.

4.2.1 Automatic News Theme Detection

Earlier works on automatic news theme generation were focused on finding interesting themes from a single event. For example, Sultana et al. [21] proposed the *Situational Fact* pattern, which is modeled as a skyline point under certain dimensions. Wu et al. [26] proposed the *One-of-the-Few* concept to detect news themes with some rarities. Examples of candidate news themes for the above two patterns are illustrated in Table **??**.

Method	**Example news theme**
Situational facts [21]	Ellen's tweet generates 3.3M retweets with 170,000 comments.
One-of-the-few facts [26]	Perry is one of the three candidates who received $600k.
Prominent streak [30]	Kobe scored 40+ in 9 straight games!
Rank-aware theme	Kobe scored 40+ in 9 straight games ranked 4th in NBA history!

Table 4.2: Examples of different news themes

Zhang et al.[30] proposed using prominent streaks to generate interesting news themes. In [30], a *Prominent Streak* is characterized by a 2D point which represents the window length and the minimum value of all events in the window. The objective is to discover the non-dominated event windows for each subject, where the dominance relationship is defined among streaks of the same subject. Our model differs from [30] in two aspects. First, we look at the global prominence (quantified by the rank) among all subjects rather than local prominence (quantified by the dominance) within one subject. Second, our model provides the best k-sketch for each subject whereas

[30] returns a dominating set which could be potentially large.

4.2.2 Frequent Episode Mining

In time sequenced data mining, an episode [19, 33, 22, 15] is defined as a collection of time sequenced events which occur together within a time window. The uniqueness of an episode is determined by the contained events. The objective is to discover episodes whose occurrences exceeding a support threshold. Our sketch discovery differs from the episode mining in three major aspects. First, an episode is associated with a categorical value while our sketch is defined on numerical values. Second, the episodes are selected based on the occurrence, while in sketch, news themes are generated in a rank-aware manner. Finally, episode mining does not restrict its output size, while sketch only outputs the best k news themes. As such, episode mining techniques cannot be straightforwardly applied to sketch discovery.

4.2.3 Top-k Diversity Query

Top-k diversity queries [1, 4, 10, 7] aim to find a subset of objects to maximize a scoring function. The scoring function normally penalizes a subset if it contains similar elements. Our sketch discovery problem has two important distinctions against the top-k diversity queries. First, the inputs of the scoring function are known in advance in top-k diversity queries; whereas in our problem, the ranks of event windows are unknown. Since their calculations are expensive, we need to devise efficient methods to compute the ranks. Second, existing methods for online diversity queries [4, 10, 7] only study the updates on a single result set when a new event arrives. However our online sketch maintenance incurs the problem of multiple sketch updates for each new event. Such a complex update pattern has not been studied yet and hence there is a need to develop efficient update scheme.

4.2.4 Event Detection and Tracking

In the information retrieval field, event detection and tracking aim to extract and organize new events from various media sources. Allen et.al [**?**] first proposed the problem of detecting unseen events from text streams, where they adapted an on-line clustering algorithm to tackle the problem. Subsequent researches extended the problem to facilitate heterogeneous sources. For example, Brants et.al. [**?**] proposed a TF-IDF model to detect unseen events from multiple text streams. Ritter et.al. [**?**] proposed a system named TwiCal to categorize events on Twitter streams. Li et.al. [**?**] proposed a ranking model to detect events on Flickr and Vuurens et.al. [**?**] described a system on tracking events from web articles.

Despite the usefulness of those works, they differ from the rank-aware news themes proposed in this paper. The major difference is that news detection and tracking focuses on detecting a single event from various sources; While the rank-aware news theme aims at providing insightful derived events from a single source. This crucial difference prevents the above-mentioned techniques directly applying to our problem.

4.3 Problem Formulation

Let S denote a set of subjects of potential interests to journalists. For example, S can refer to all the players or teams in the NBA application. Let $e_s(t)$ denote an event about subject s, where t denotes its timestamp or sequence ID. For example, an event can refer to an NBA game a player participated on a certain day. Note that we maintain a sequence ID for each subject that is automatically incremented. It is possible that the events of different subjects may have the same sequence IDs. A sequence of consecutive events of the same subject is grouped as an *event window*:

Definition 4.3.1 (Event Window)**.** Let w be a window length, an event window $W_s(t, w)$ refers to w consecutive events of subject s and ending at sequence t, i.e.,

$W_s(t, w) = \{e_s(t - w + 1), ..., e_s(t)\}$.

In general, if a subject s has $|\mathbb{H}_s|$ events, there are $\binom{|\mathbb{H}_s|}{2}$ possible event windows. Given an aggregate function f, events in an event window can be aggregated to a numerical value $\overline{v}$ as:

$$W_s(t, w).\overline{v} = f(e_s(t - w + 1), ..., e_s(t))$$

We support all the common aggregate functions such as *sum*, *avg*, *count*, *min* and *max*. For simplicity of presentation, we only consider a single aggregate function in our solution. To support multiple aggregation functions, one may simply invoke our solution multiple times to process them independently.

Example 4.3.1. Fig. 4.1 (A) illustrates examples of *event window*s of three NBA players. Each event represents the points scored by a player in a game. In the figure, the event window $W_{s_1}(3, 2)$ refers to two consecutive events about player s_1 ending at $t = 3$, i.e., $W_{s_1}(3, 2) = \{e_{s_1}(2), e_{s_1}(3)\}$. Given an aggregate function $f = avg(points)$, we yield $W_{s_1}(3, 2).\overline{v} = (46 + 10)/2 = 28$.

With the aggregated value $\overline{v}$, we can derive the rank of an event window, which can be used to measure the strikingness of an event window as a news theme. For instance, "*The total points Kobe has scored is* $32,482$" can be transformed into a rank-aware representation: "*Kobe moved into third place on the NBA's all-time scoring list*", in which the rank is 3. Let W be a length-w event window, and $\gamma_w(W.\overline{v})$ be the rank of W by comparing it with all other length-w event windows on their aggregate values. Generally speaking, an event window is newsworthy if its rank is smaller than a predefined threshold. This leads to our definition of *Candidate Theme*:

Definition 4.3.2 (Candidate Theme)**.** An event window $W_s(t, w)$ is said to be a candidate theme, denoted by $N_s(t, w)$, if its rank $\gamma_w(W_s(t, w).\overline{v}) \leq p$. p is a user-defined threshold.

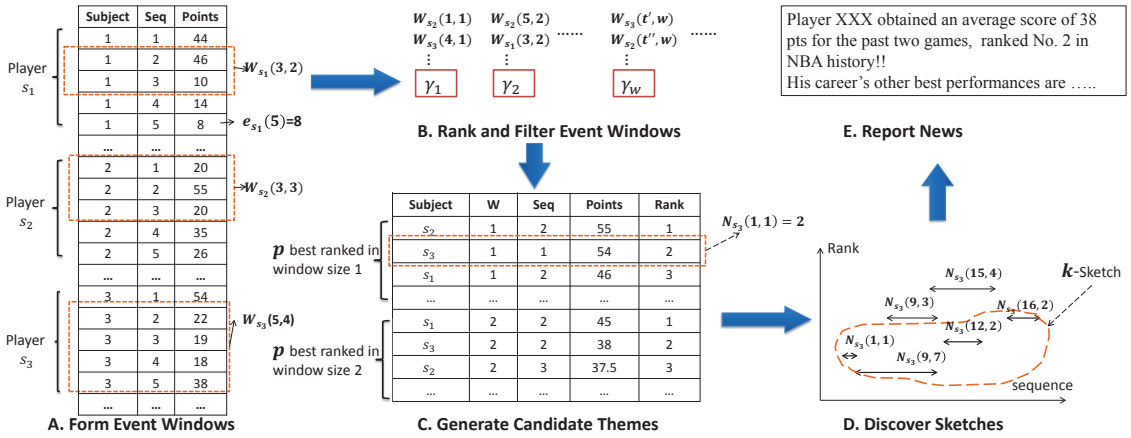


Figure 4.1: An illustration sketch discovery workflow. (A): various event windows are formed based on events' sequence IDs. (B): event windows with rank greater than p are filtered. (C): best ranked event windows form candidate themes. (D): k-sketches are discovered for each subject from their candidate themes. (E): subject related news can be generated from each k-sketch.

Example 4.3.2. In the step (B) of Fig. 4.1, we group the event windows based on their window length. Each window is associated with a rank value γ_w. If the rank value is greater than the threshold p, the associated event window is pruned. Otherwise, it is considered as a candidate. In the step (C) of Fig. 4.1, we present the candidate themes in the tabular format. Each candidate contains the rank computed from step (B). For instance, among all candidate themes with window length 1, $N_{s_3}(1,1)$ (with value 54) is ranked 2nd because its aggregated value is smaller than that of another event window $N_{s_2}(2,1)$ (with value 55).

Let $\mathbb{N}_s$ be the set of candidate themes of subject s. Since for each possible window length, there are at most p candidate themes, the size of $\mathbb{N}_s$ can be as large as $p|\mathbb{H}_s|$, which is still overwhelming for news reporting. To control the output size while maintaining the quality of news themes, we aim to find, for each subject s, a small subset of k candidate themes from $\mathbb{N}_s$. We call the subset a k-**Sketch**.

To measure the quality of a sketch, we define a scoring function $g(\cdot)$ over all sketches. The design of g gives rise to two concerns. First, g needs to assign higher scores to the candidate themes with better ranks. This is because better rank indicates

good strikingness which implies that the news theme would be more eye-catching. For instance, we can expect people to care more about who the top scorer in NBA history is rather than who ranked 20^{th} in the history. Second, g prefers the sketches containing candidate themes with fewer overlapped events and avoids near-duplicate events. This is because near-duplicate themes generally do not add news values. For example, when Kobe Bryant scored 81 points in a single game (2nd highest in NBA history), all event windows containing that event are likely to have good and similar ranks. Thus, reporting all of them is not newsworthy.

To resolve these two concerns, we define g as follows: given any set $\mathbb{X}_s$ of candidate themes about subject s (i.e., $\mathbb{X}_s \subseteq \mathbb{N}_s$), the score of $\mathbb{X}_s$ is:

$$g(\mathbb{X}_s) = \alpha C(\mathbb{X}_s) + (1-\alpha)R(\mathbb{X}_s), \alpha \in [0,1] \tag{4.1}$$

where $C(\mathbb{X}_s)$ is the ratio between the number of distinct events covered by $\mathbb{X}_s$ over the total number of events about subject s. In this way, duplicated candidate themes contribute to a lower score. $R(\mathbb{X}_s) = \frac{1}{|\mathbb{X}_s|}\sum_{X_s \in \mathbb{X}_s} \frac{p - X_s.\gamma}{p}$ is the strikingness of $\mathbb{X}_s$. Any candidate themes in $\mathbb{X}_s$ changing to a better rank increases $R(\mathbb{X}_s)$. The values of $C(\mathbb{X}_s)$ and $R(\mathbb{X}_s)$ are normalized between $[0,1]$. α is an adjustable coefficient to balance the weights between $C(\mathbb{X}_s)$ and $R(\mathbb{X}_s)$. If α is high, it means users are more interested in finding news themes that cover most of the subject's events. If α is low, it indicates users prefer news themes whose ranks are small.

With Equation 4.1, we then define the *Sketch Discovery* problem as follows:

Definition 4.3.3 (Sketch Discovery)**.** Given a parameter k, Sketch Discovery aims to, for each subject s, find a subset SK_s from the candidate themes of s (i.e., $SK_s \subseteq \mathbb{N}_s$), s.t. $|SK|_s = k$ and $g(SK_s)$ is maximized.

Example 4.3.3. In the step (D) of Fig. 4.1, we show a collection of candidate themes and a k-sketch derived from them. The y-axis is the rank and the x-axis represents

the complete sequence of events of a subject. Each candidate theme is represented by a line segment, covering a window of events. When $k = 4$, four of the candidate themes are selected as a 4-sketch as the most representative news for the subject.

Before we move on to the algorithmic part, we first list the notations in Table 4.3. For the ease of presentation, we present our techniques using *avg* as the default aggregate function in the following sections. Extending our techniques to other aggregate functions will be addressed in Section A.1.

Notation	**Meaning**
S	set of subjects
$\mathbb{H}_s$	set of events associated with subject s
$W_s(t, w)$	length-w event window of s ending at t
$\mathbb{N}_s$	set of candidate themes associated with subject s
$N_s(t, w)$	candidate theme derived from $W_s(t, w)$
$WI(w)$	p best ranked candidate themes of window length w
$\beta(w)$	lower bound of $WI(w)$
SK_s	sketch for subject s
J_s	visiting-window bound for subject s
M_s	unseen-window bound for subject s
P_s	online-window bound for subject s

Table 4.3: Notations used in this paper

4.4 Offline Sketch Discovery

In the offline scenario, the input is a set of events of all subjects and the output is a k-sketch for each subject s, denoted by SK_s. The sketch discovery process consists of two major steps: first generating the candidate themes of each subject (i.e., $\mathbb{N}_s$), and then discovering the sketches among those candidate themes.

4.4.1 Candidate Theme Generation

Generating candidate themes for each subject is computationally expensive. In order to retain the candidate themes with rank no greater than p, a brute-force approach

needs to evaluate all the event windows with every possible length to generate accurate ranks. Since there could be $\binom{\mathbb{H}_s}{2}$ event windows (with different lengths and ending sequence ids) associated with subject s, the total time complexity for the brute-force approach is $O(\sum_{s \in S} |\mathbb{H}_s|^2)$. Such a complexity makes the solution infeasible even for moderate-sized datasets.

To improve the efficiency, we observe that it is not necessary to compute all the event windows to generate $\mathbb{N}_s$. The intuition behind is that the upper bound value of event windows with longer lengths can be estimated from those with shorter lengths. This means that we can compute event windows with increasingly lengths, and as the shorter event windows are computed, the longer event windows not in $\mathbb{N}_s$ can be pruned. To realize such an intuition, we design the candidate generation algorithm by adapting two novel window-based pruning methods which exploit the *subadditivity* property among event windows with different window lengths.

4.4.1.1 Overview of window pruning

For each subject, its event windows can be grouped by window lengths. Our candidate generation algorithm gradually evaluates a subject's event window from shorter length to longer length. To support efficient pruning, we define two concepts, namely *visiting-window bound* (J_s) and *unseen-window bound* (M_s). In particular, $J_s(w)$ is the upper bound of all the event windows about subject s and with length w, i.e., $\forall w, J_s(w) \geq \max\{W_s(t,w).\overline{v} | t \in (0,w)\}$. $M_s(w)$ is the upper bound of all the event windows about subject s with length larger than w, i.e., $M_s(w) \geq \max\{J_s(t) | t > w\}$. These two bounds will be used for event window pruning and we will present how to estimate these two upper bounds in Sections 4.4.1.2 and 4.4.1.3.

The overview of news theme generation algorithm is presented in Algorithm 6. We maintain two global structures WI and β (Lines 1-2). For each window length w, $WI(w)$ stores the top p event windows with length w among all the subjects and $\beta(w)$

is the p-th score among the candidates in $WI(w)$. In other words, $\beta(w)$ serves as a lower bound score. An event window with length w is a candidate for the k-sketch only if its aggregate value is larger than $\beta(w)$. A priority queue Q (Line 3) is used to provide an access order among the subjects. Each element in the queue is a triple (s, w, q), where s is a subject, w indicates the next window length of s to evaluate, and q denotes the priority. We use the unseen-window bound (i.e., $M_s(w)$) as the priority during every iteration (Lines 13-14). Intuitively, an event window with higher $M_s(w)$ is more likely to spawn new event windows that can increase β and benefit the subsequent pruning. During initialization, we insert, for each subject, an entry with window length 1 and priority infinity into Q.

In each iteration, we pop the event window with the highest priority (Line 4). Then, we compute the visiting-window bound $J_s(w)$ for the subject and determine whether all the length-w windows about subject s can be pruned (Lines 5-6). If the bound $J(s)$ is no greater than $\beta(w)$, then all these event windows can be ignored. Otherwise, all the length-w event windows of s are computed to update $WI(w)$ and $\beta(w)$ accordingly (Lines 7-9). In the next step, we estimate the unseen-window bound M_s and compare it with the minimum $\beta(w')$ for all $w' > w$. If M_s is smaller, then we would not find a better candidate and all the event windows about subject s and with length larger than w can be pruned (Lines 11-12). Otherwise, we assign M_s to the priority q, and push the triple $(s, w+1, q)$ back into the queue (Lines 13-14). The algorithm terminates when all the candidates have been evaluated and Q becomes empty.

4.4.1.2 Visiting-window pruning

In Algorithm 6, we need to compute the visiting-window bound $J_s(w)$ to facilitate pruning all length-w event windows associated with subject s. Our idea is that suppose we have successfully derived the bounds for windows with smaller lengths,

Algorithm 6 Candidate Theme Generation Overview

```
1: WI() ← {} // p best event windows for each window length
2: β ← {} // smallest scores in WI for each window length
3: Q ← {(s, 1, +∞)|s ∈ S}
4: while (s, w, q) ← Q.pop() do
5:     compute J_s(w)
6:     if J_s(w) > β(w) then
7:         for t ∈ w...|H_s| do
8:             Update WI(w), β(w), and J_s(w)
9:         end for
10:    end if
11:    compute M_s(w) whose value relies on J_s(w)
12:    if M_s(w) > min{β(w')|w < w' ≤ |H_s|} then
13:        q ← M_s(w)
14:        Q.push(s, w + 1, q)
15:    end if
16: end while
17: return WI
```

we can use them to estimate the bounds of larger windows. We formulate it as the subadditivity property and use avg[7] as the aggregate function for illustration:

Theorem 4.4.1 (Subadditivity (for avg))**.**

$$\max_t\{W_s(t,w).\overline{v}\} \leq \frac{w_i J_s(w_i) + (w-w_i)J_s(w-w_i)}{w}, \forall w_i \in (0,w) \tag{4.2}$$

Proof. Given any event window $W = W_s(t,w)$ with length w and a value $w_i \in (0,w)$, we can split the window into two non-overlapping sub-windows with lengths w_i and $w-w_i$, i.e.,$W_1 = W_s(t,w_i)$ and $W_2 = W_s(t-w_i, w-w_i)$. Due to the non-overlapping property of W_1 and W_2, we have $wW.\overline{v} = w_i W_1.\overline{v} + (w-w_i)W_2.\overline{v}$. Since $J_s(w_i)$ and $J_s(w-w_i)$ are two visiting-window bounds, we have $W_1.\overline{v} \leq J_s(w_i)$ and $W_2.\overline{v} \leq J_s(w-w_i)$. Then, we complete the proof and $\frac{w_i J_s(w_i)+(w-w_i)J_s(w-w_i)}{w}$ is a bound for event windows with length w. □

With Theorem 4.4.1, we can estimate $J_s(w)$ by any pair $J_s(w_1)$ and $J_s(w-w_1)$, $\forall w_1 < w$. To derive a tighter bound for $J_s(w)$, we formulate the following optimization

[7]Although here we demonstrate the bound using "avg", the properties and bounds also hold for other aggregate functions. Corresponding properties and bounds for other aggregate functions are listed in Section A.1.

problem:

$$w_1 = \underset{t \in (0,w)}{\operatorname{argmin}} \frac{J_s(t) * t + J_s(w-t) * (w-t)}{w} \tag{4.3}$$

A naive solution would enumerate every possible t to get the tightest bound. However, such a solution has a worst time complexity of $O(|\mathbb{H}_s|)$ for subject s. To quickly compute w_1 without compensating the running time, we apply a continuous relaxation to Eqn. 4.3 as follows: Let $G_s(t) = J_s(t) * t \ \ \forall t$, then Eqn. 4.3 is equivalent to:

$$w_1 = \underset{t \in 1..w-1}{\operatorname{argmin}} \{G_s(t) + G_s(w-t)\} \tag{4.4}$$

Let $L_s(t)$ be a continuous and smooth fitting function of $G_s(t)$ for $t \in [1, w-1]$. Eqn. 4.3 can then be relaxed by replacing $G_s(t)$ with $L_s(t)$ to produce the solution at $w_1 = w/2$ if $L_s(t)$ is convex and $w_1 = 1$ if $L_s(t)$ is concave. We have observed, over all aggregate functions, the convexity and concavity for $L_s(\cdot)$ when approximating $G_s(\cdot)$. For example, we use 800 games of a NBA player and plot the function G_s and L_s under various aggregate functions in Figs. 4.2. In Fig. 4.2(a), G_s and L_s for *min* and *avg* are presented. We can see that the fitting L_s for *min* is convex while that for *avg* is concave. Similarly, in Fig. 4.2(b), we can see that L_s is concave for count, sum and max. In the worst-case scenario, even when $L_s(\cdot)$ is neither convex nor concave, $J_s(w) < \frac{J_s(1)+(w-1)J_s(w-1))}{w}$ still holds due to Theorem 4.4.1. Thus we have the visiting-window bound stated as:

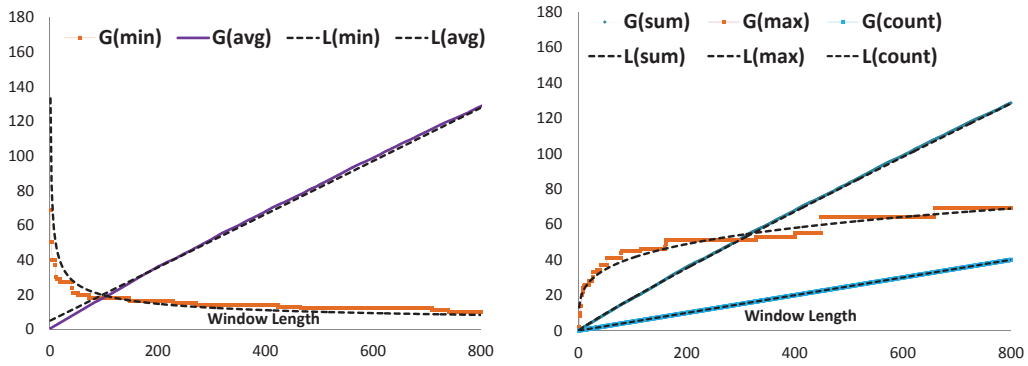
Theorem 4.4.2 (Visiting-Window Bound)**.** Given a window length $w > 1$, let $J_s(w)$ be:

$$J_s(w) = \frac{J_s(1) + (w-1)J_s(w-1)}{w} \tag{4.5}$$

then $J_s(w)$ is a visiting-window bound, i.e. $J_s(w) \geq \max_t W_s(t,w).\overline{v}$

By substitute $w_i = 1$ to the r.h.s. of Theorem 4.4.1, we see this theorem holds.

In Algorithm 6, $J_s(w)$ is computed incrementally. Initially, $J_s(1)$ is set as the



(a) (b)

Figure 4.2: Illustration of fitting function L on various aggregation functions; solid lines represent G while dotted lines represent L. (a) fitting on min and average (b)fitting on max,sum and count.

single event of s with highest value. Then, as the subject s is processed, $J_s(w)$ is computed by Theorem 4.4.2. In the case that $J_s(w)$ is not pruned, we may assign $J_s(w)$ with the maximum length-w event window of s to further tighten the bound.

4.4.1.3 Unseen-window pruning

Unseen-window pruning utilizes $M_s(w)$ to check if it is necessary to evaluate any window $w' \in (w, |\mathbb{H}_s|]$. We observe that $M_s(w)$ can be efficiently estimated from the values of $J_s(w')$, where $w' \leq w$. For example, when avg is used as the aggregate function, $J_s(1)$ is obviously an upper bound for $M_s(w)$ because $J_s(1)$ is essentially the maximum event value. However, such an upper bound is very loose. By utilizing the following theorem, we can derive a tighter bound as follows:

Theorem 4.4.3 (Unseen-Window Bound)**.** Given that $J_s(1)$, ..., $J_s(w-1)$ have already been computed, let $M_s(w)$ be:

$$M_s(w) = J_s(w) + \min\{\frac{1}{2}J_s(1), \frac{w-1}{w+1}J_s(w-1)\} \tag{4.6}$$

then $M_s(w)$ is an *unseen-window bound*, i.e. $M_s(w) \geq \max\{J_s(t) |\ t \in [w, |\mathbb{H}_s|]\}$

Proof. First, given any integer $k \geq 1$, we see that $J_s(kt) \leq J_s(t)$ by making use of Lemma 4.4.1 in a simple induction proof. Then, for any integer $x > w$, x can be written as $x = \lfloor \frac{x}{w} \rfloor w + x \bmod w$. Based on the subadditivity of $J_s(\cdot)$, we have:

$$\begin{aligned} J_s(x) &\leq \frac{(\lfloor \frac{x}{w} \rfloor * w) J_s(\lfloor \frac{x}{w} \rfloor * w) + (x \bmod w) J_s(x \bmod w)}{x} \\ &\leq J_s(\lfloor \frac{x}{w} \rfloor * w) + \frac{x \bmod w}{x} J_s(x \bmod w),\ \lfloor \frac{x}{w} \rfloor * w \leq x \\ &\leq J_s(w) + \frac{x \bmod w}{x} J_s(x \bmod w),\ J_s(kt) \leq J_s(t) \end{aligned}$$

On one hand, since $t * J_s(t)$ is a monotone increasing function, it follows $(x \bmod w) J_s(x \bmod w) \leq (w-1) J_s(w-1)$. Moreover, since $x \geq w+1$, we have $(x \bmod w) \frac{J_s(x \bmod w)}{x} \leq (w-1) \frac{J_s(w-1)}{w+1}$. On the other hand, $J_s(x \bmod w) \leq J_s(1)$ and $\frac{x \bmod w}{x} \leq \frac{1}{2}$ for $x > w$, we have $(x \bmod w) \frac{J_s(x \bmod w)}{x} \leq \frac{1}{2} J_s(1)$. Then it naturally leads to Theorem 4.5.1. □

To utilize $M_s(w)$, after the derivation, we check if $M_s(w)$ is no greater than any $\beta(w')$ with $w' > w$, i.e. $M_s(w) \leq \min\{\beta(w') | w' \in (w, |\mathbb{H}_s|]\}$. Whenever the condition holds, it is safe to stop further evaluation on subject s. Note that we may maintain an interval tree [**?**] on β to support efficient checking.

Theorem 4.4.4. Each candidate theme returned by Algorithm 6 has a rank no greater than p.

The proof is quite straightforward according to the descriptions for Algorithm 6, visiting-window and unseen-window pruning. Thus, the details are omitted.

Example 4.4.1. We use Fig. 4.3 to illustrate our pruning techniques on a subject s. Each column in the table represents a window length and the cells contain the average values among different event windows with different lengths. For example, the cell in the second row and third column refers to the second largest average value among the event windows with length 3. Algorithm 6 essentially accesses the event windows in increasing order of the window length. Suppose we are about to estimate

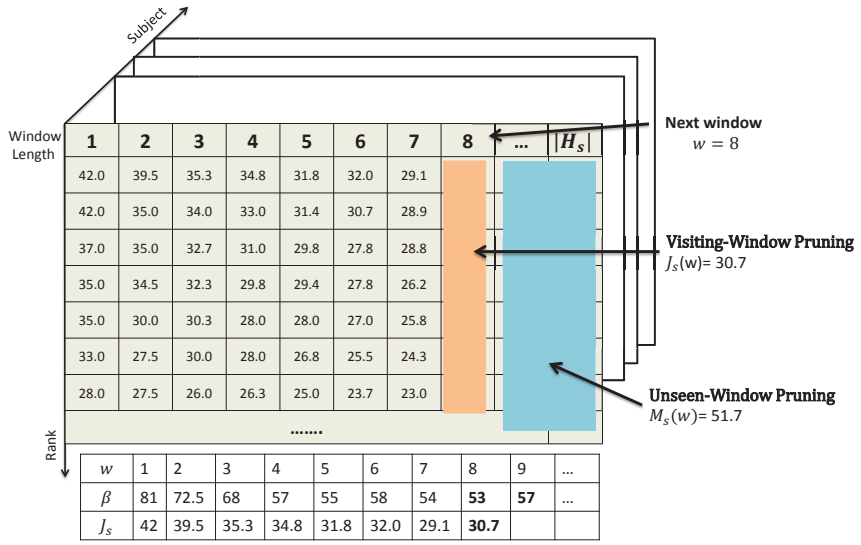


Figure 4.3: An illustration of window-level pruning techniques. Each square slice represents a set of event windows to be computed for a subject, where the column represents window length and the row represents the rank. The value in each cell is the aggregate result (i.e., $\overline{v}$) for the corresponding event window.

the upper bound for event windows with length 8. At this point, the values of β and J_s are depicted in the figure. Then, our first job is to estimate the upper bound for all the average values in the 8-th column. Based on the values of $J_s(w)$ on smaller windows, we estimate $J_s(8) = \frac{42+7*29.1}{8} = 30.7125$. Since $J_s(8) < \beta(8)$, we can safely prune the whole column, as highlighted in the figure. Afterwards, we estimate the upper bound for all the windows with length larger than 8. The value of $M_s(8)$ is then estimated as $M_s(8) = 30.7125 + \min\{21, \frac{7}{9} * 29.1\} = 51.7$. Since all the values of β are greater than $M_s(8)$, it is safe to terminate the event window enumeration.

4.4.2 Sketch Computation

After the candidate themes are obtained, the second step of the sketch discovery process is to, for each subject s, extract k themes to form sketches SK_s to maximize Eqn. 4.1 (i.e., g).

Our goal of optimizing g is related to the Partly Interval Set Cover (PISC) prob-

lem. The goal in PISC is to select a set of intervals which covers at least a certain percentage of elements. If no rank value is considered in g (i.e., $\alpha = 1$), the solution in [**?**] for PISC maximizes g in $O(\mathbb{H}_s^3)$ time for each subject s. However, when the rank is considered (i.e., $0 < \alpha < 1$), optimizing g becomes an open problem as stated in [**?**], where current polynomial time solutions remain unknown. To facilitate scalable sketch discovery, we provide an efficient $(1 - 1/e)$ approximation algorithm by exploiting the submodularity property[8] of the scoring function. Our idea is that since g is not submodular, it is not easy to directly maximize it. However, by applying a relaxation on g, we are able to gain a submodular function g' s.t. maximizing g' would result in the same optimal solution as maximizing g. We design the function of g' as follows:

$$g'(\mathbb{X}_s) = \eta_1 C'(\mathbb{X}_s) + \eta_2 R'(\mathbb{X}_s) \tag{4.7}$$

where $C'(\mathbb{X}_s)$ is the number of distinct events covered by $\mathbb{X}_s$, $R'(\mathbb{X}_s) = \Sigma_{X_s \in \mathbb{X}_s}(p - X_s.r)$, $\eta_1 = \alpha/|\mathbb{H}_s|$ and $\eta_2 = (1 - \alpha)/k$. Given k, s and g, g' is uniquely defined. We have the following theorem which links g' to g:

Theorem 4.4.5. If A^* is the optimal solution wrt. g', then A^* is also the optimal solution wrt. g.

Proof. First observe that, for any set $A \subseteq \mathbb{N}_s$ of size k, (i.e., $|A| = k$), $g(A) = g'(A)$. This can be validated by substituting A into Eq. 4.1 and Eq. 4.7. Then, we prove the theorem by contradiction: If A^* is not optimal wrt. g, then $\exists B^*$ s.t. $g(B^*) > g(A^*) = g'(A^*)$. However, since $|B^*| = k$, $g'(B^*) = g(B^*) > g'(A^*)$, which contradicts with A^*'s optimality w.r.t. g'. □

Henceforth, we are able to compute sketches by maximizing g' instead of g. We then prove the *submodularity* on g' as stated below:

[8] A function I is submodular if and only if given two set $A \subseteq B$ and an element $x \notin B$, then $I(A \cup \{x\}) - I(A) > I(B \cup \{x\}) - I(B))$.

Theorem 4.4.6. Given a candidate theme set $\mathbb{X}_s$, $g'(\mathbb{X}_s)$ is submodular.

Proof. Note that C' is a cover function and R' is a scalar function, thus C' and R' are both submodular. Since g' a linear combination of C' and R', it is also submodular. □

By utilizing the submodularity of g', we apply the greedy algorithm [**?**] to efficiently discover the sketches, which guarantees a $(1 - 1/e)$ approximation ratio. The greedy sketch discovery scheme is presented in Algorithm 7. During each step, the algorithm picks the best candidate theme which maximizes g' among all the candidates which have not been selected yet. The algorithm stops at the k^{th} iteration.

Algorithm 7 Greedy Sketch Discovery

```
for s ∈ S do
    N_s ← candidate themes of subject s filter by Algorithm 6
    SK_s ← {}
    for t ∈ [1, k] do
        x* ← argmax_{x∈N_s} g'(SK_s ∪ {x})
        SK_s ← SK_s ∪ {x*}
    end for
end for
return SK_s  ∀s ∈ S
```

4.5 Online Sketch Maintenance

In the offline scenario, all the events about every subject are assumed to be available at the time of sketch computation. In contrast, in the online scenario, events arrive incrementally. Given an arrival event, our objective is to keep the sketch for each subject up-to-date. Since the event arrival speed could be very high, such step has to been done very efficiently.

Similar to the offline scenario, we maintain a WI to keep track of the p best event windows (candidate themes) for all the possible window lengths. To handle

a newly arrived event $e_s(t)$, a naive solution would generate all the event windows containing $e_s(t)$, (i.e. $W_s(t, w'), w' \in (1, t]$). For each event window, WI is then updated accordingly. Afterwards, Algorithm 7 needs to be invoked to re-compute the sketches. However, there are t associated event windows for each new event $e_s(t)$. Examining all of them is too expensive to support real-time responses. Moreover, Algorithm 7 runs in $O(k|\mathbb{N}_s|)$ time for each affected subject, which imposes further performance challenge.

To achieve instant sketch update, we propose two techniques - *window pruning* and *sketch maintenance*. Window pruning tries to reduce the number of windows evaluated in generating candidate themes. After obtaining the candidate themes, we need to update the sketches affected. As we shall see later, given a candidate theme $N_s(t, w)$, not only the sketch for subject s but also the sketches for other subjects could be affected. Although in the offline scenario we provided a solution with a $(1 - 1/e)$ approximation, to maintain sketches with the same approximation ratio is difficult in the online scenario [**?**, **?**]. Instead, we propose a 1/8-approximation solution for sketch maintenance which only takes $O(k)$ time to perform the sketch update.

Algorithm 8 Online Query

Input: $e_s(t) \leftarrow$ arrival event
1: $WI()$ p best event windows for each window length
2: $\beta()$ smallest score in WI for each window length
3: **for** $w \in 1, ..., t$ **do**
4: **if** $W_s(t, w)$ can be added to $WI(w)$ **then**
5: update $\beta(w)$, $J_s(w)$, compute $N_s(t, w)$
6: UpdateSketches($N_s(t, w)$)
7: **end if**
8: $P_s(w) = \frac{w}{w+1} W_s(t, w).\overline{v} + \frac{t-w}{w+1} J_s(t - w)$
9: **break if** $P_s(w) \leq \max\{\beta(w')|w < w' \leq t\}$
10: **end for**

Before we present *window pruning* and *sketch maintenance*, Algorithm 8 demonstrates the overview of our online solution against a new event $e_s(t)$. We iteratively

examine event windows which contain $e_s(t)$ (i.e.,$W_s(t,w)$ in Line 4). We then update the sketches which are affected by inserting $W_s(t,w)$ into WI (Lines 5-6). Before continuing to examine the next event window w, we compute the maximum score (i.e., $P_s(w)$) of all event windows which have not been evaluated. If $P_s(w)$ is smaller than all $\beta(w'), w' \in (w,t]$, we can safely stop the process since no further event window could cause any sketches to change.

4.5.1 Online Window Pruning

Since there are t event windows associated with each new event $e_s(t)$, we wish to avoid enumerating all the possible cases. We achieve the window pruning by leveraging the online-window bound $P_s(w)$, which is the upper bound score among event windows with length greater than w. The value of $P_s(w)$ is stated as in the following theorem:

Theorem 4.5.1 (Online-Window Bound)**.** Let $W_s(t,1)$,...,$W_s(t,w)$ be the w event windows computed in Algorithm 8 containing event $e_s(t)$. Let $P_s(w)$ be:

$$P_s(w) = \frac{w}{w+1} W_s(t,w).\overline{v} + min\{\frac{t-w}{w+1} J_s(t-w), \frac{t-w}{t} J_s(1)\} \tag{4.8}$$

Where $J_s(\cdot)$ is the visiting-window bound. Then $P_s(w)$ is the online-window bound, i.e.,$P_s(w) \geq \max\{W_s(t,x).\overline{v} | w < x \leq t\}$.

Proof. First, $\forall x \in (w,t]$, we have:

$$\begin{aligned} W_s(t,x).\overline{v} &= \frac{w * W_s(t,w).\overline{v} + (x-w) * W_s(t-w, x-w).\overline{v}}{x} \\ &\leq \frac{w * W_s(t,w).\overline{v}}{w+1} + \frac{(x-w) * W_s(t-w, x-w).\overline{v}}{x} \end{aligned}$$

Note that $J_s(x-w) \geq W_s(t-w, x-w).\overline{v}$, and $yJ_s(y)$ monotonically increases wrt. y. It follows that $(x-w)W_s(t-w, x-w).\overline{v}/x \leq (x-w)J_s(x-w)/x \leq (t-w)J_s(t-w)/(w+1)$.

On the other hand, $J_s(1) \geq W_s(\cdot, y).\overline{v}$, for any y. Therefore, $(x-w)W_s(t-w, x-$

$w).\overline{v}/x \leq (x-w)J_s(1)/x \leq (t-w)J_s(1)/t$. Combining the above deductions, it follows that:

$$\frac{(x-w)W_s(t-w,x-w).\overline{v}}{x} \leq min\{\frac{t-w}{w+1}J_s(t-w), \frac{t-w}{t}J_s(1)\}$$

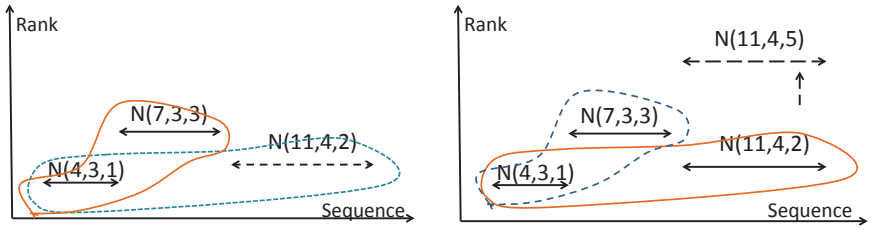
which naturally leads to Theorem 4.5.1. □

When w is small, $\frac{t-w}{w+1}J_s(t-w)$ is too loose as $\frac{t-w}{w+1}$ is large. However, we can leverage $\frac{t-w}{t}J_s(1)$ to obtain a better bound. As w increases, $\frac{t-w}{w+1}J_s(t-w)$ eventually becomes smaller than $\frac{t-w}{t}J_s(1)$. Thus, we leverage $\frac{t-w}{w+1}J_s(t-w)$ to perform efficient pruning.

4.5.2 Sketch Maintenance

Once we obtain an event window $W_s(t,w)$ which causes changes in the $WI(w)$, two kinds of sketch maintenances may occur. The first is to update the sketch for subject s, which we refer to as *active-update*. The second is to update the sketches for other subjects. This happens when some of their candidate themes' ranks become worse due to $W_s(t,w)$. We refer to this case as *passive-update*. If these updates are not properly handled, sketches maintained for those subjects are not able to obtain an approximate ratio on their qualities. We demonstrate these updates in the following example.

Example 4.5.1. Suppose $k = 2$ and we maintain a 2-Sketch for each subject. As shown in Fig. 4.4(a), when the candidate theme $N(11,4)$ is generated, the maintained sketch is no longer the best. This is because replacing $N(7,3)$ would generate a better sketch. This process is referred as *active update*. In Fig. 4.4(b), $N(11,4)$ is pushed up due to the arrival of the event about another subject; as a result, the quality of the sketch drops. We name this process as *passive update*. If passive update is not handled, the rank of $N(11,4)$ may continue to be pushing up and may eventually



(a) Active update of new candidate theme $N(11, 4)$

(b) Passive update due to existing candidate theme $N(11, 4)$

Figure 4.4: The illustration of active updates and passive updates, the solid circle represent the original sketch, the dotted circle represent the updated sketch.

be greater than p, making the entire sketch invalid. Nevertheless, it is evident that when $N(11, 4)$ degrades, replacing it with $N(7, 3)$ would result in a sketch with better quality.

A naive approach to handle online update is to run Algorithm 7 for subjects whose sketches are affected. This maintains a $(1 - 1/e)$ approximation ratio but incurs high computational overheads. In order to support efficient updates, we make a trade-off between the quality of the sketch and the update efficiency by providing a $1/8$ approximation solution with only $O(k)$ candidate themes being accessed for each affecting subject.

In particular, we maintain two sets S_1 and S_2 each of size k. S_1 maintains the k best candidate themes which collectively cover the most events whereas S_2 maintains k candidate themes with best ranks. When performing an active update for a candidate theme $N_s(t, w)$, we check if $N_s(t, w)$ could replace any candidates in S_1 to generate a better cover meanwhile we select the new k best ranked candidate themes into S_2. After S_1 is updated, we perform the greedy selection among candidate themes in $S_1 \cup S_2$. During a passive update, S_1 is not affected. We simply update S_2 to be the new k best ranked candidate themes. After that, the new sketch is obtained by performing a greedy selection on $S_1 \cup S_2$. Algorithm 9 summarizes both the active and passive update.

Given our sketch update strategies, we are now ready to prove the approximation

Algorithm 9 UpdateSketches

Input: $N_s(t, w)$

Active update for the subject s
S_1: k candidate themes with best cover
S_2: k candidate themes with best ranks
$N^* \leftarrow \text{argmax}_{N \in S_1} C(S_1 \cup N_s(t, w) \setminus N)$
if $C(S_1) < C(S_1 \cup N_s(t, w) \setminus N^*)$ **then**
    $S_1 \leftarrow S_1 \cup N_s(t, w) \setminus N^*$
end if
$S_2 \leftarrow$ new k best candidates for rank scores
$greedy(S_1 \cup S_2)$

Passive update for an affecting subject s'
$S_2 \leftarrow$ new k best candidates for rank score s'
$S \leftarrow greedy(S_1 \cup S_2)$

ratio for the maintained sketches.

Theorem 4.5.2 (Approximation Ratio for Sketch Update)**.** Each sketch maintained by Algorithms 8 achieves an at least 1/8 approximation to optimal solution.

Proof. First, we observe that S_2 always keep the candidates with optimal ranks. Second, we note that S_1 maintains the candidates with 1/4 optimality as shown in [**?**].

Let OPT_C be the optimal solution for k best covering of s's events; Let $C()$ be the number of events a set of candidate theme covers. Similarly, let OPT_R be the optimal solution for k best ranked candidate themes of s; Let $R()$ be the summation of ranks of all members in a candidate theme set. Let S_s^* be the optimal sketch of subject s. Intuitively, we have the following:

$$g'(S_s^*) \leq \eta_1 C(OPT_C) + \eta_2 R(OPT_R)$$

Since $C(S_1) \geq 1/4C(OPT_C)$ and $R(S_2) = R(OPT_R)$, we have the following:

$$\begin{aligned}\eta_1 C(S_1) + \eta_2 R(S_2) &\geq 1/4 * \eta_1 C(OPT_C) + \eta_2 R(OPT_R) \\ &\geq 1/4 g'(S_s^*)\end{aligned}$$

, which implies $\max\{\eta_1 C(S_1), \eta_2 R(S_2)\} \geq 1/8 g(S_s^*)$. As $g'(S_1) \geq \eta_1 C(S_1)$ and $g'(S_2) > \eta_2 R(S_2)$, it leads to:

$$\max\{g'(S_1), g'(S_2)\} > 1/8 g'(S_s^*)$$

Let SK_s be one of the sketch maintained by Algorithms 9, since the greedy algorithm is run on $S_1 \cup S_2$, $g'(SK_s) \geq max(g'(S_1), g'(S_2)) \geq 1/8 g'(S_s^*)$. As a result, our algorithm always ensures a 1/8 approximation ratio for each sketch. □

4.6 Experiments

In this section, we study our solutions for sketch discovery in both offline and online scenarios using the following four real datasets with statistics being summarized in Table 4.4.

NBA[9] contains the game records for each NBA player from year 1985 to 2013. Among all the records, we pick 1,000 players with at least 200 game records. In total, we obtain a dataset with 569K events.

POWER [**?**] contains the electricity usage for 370 households between Dec. 2006 and Nov. 2010. Each household is treated as a subject with the daily power usage as an event. In total, there are 1.4M events.

PEMS [**?**] contains the occupancy rate of freeway in San Francisco bay area from Jan. 2008 to Mar. 2009. Each freeway is a subject with the daily occupancy rate as

[9]http://www.nba.com

an event. The dataset contains 963 freeways with 5.7M events.

STOCK contains the hourly price tick for 318 stocks from Mar. 2013 to Feb. 2015. The dataset is crawled from Yahoo! Finance[10] and contains 2.3M events.

Table 4.4: Statistics of datasets used in experiments

DataSet	Total Events	Total Subjects	Longest Window
NBA	569,253	1,015	1,476
POWER	1,480,000	370	4,000
PEMS	5,798,918	963	6,149
STOCK	2,326,632	318	10,420

In our efficiency study, we evaluate three parameters: 1) $p \in [20, 200]$, which refers to the worst rank for an event window to become a candidate theme, 2) $k \in [20, 100]$, which refers to the number of candidate themes in a sketch and 3) $h \in [20, 100]$, which refers to the percentage of historical events for scalability test, i.e. $|\mathbb{H}| \cdot h\%$ events are used in the experiments. We use $p = 200$, $h = 100$ and $k = 20$ as the default values.

All the experiments are conducted on a desktop machine equipped with an Intel i7 Dual-Core 3.0GHz CPU, 8GB memory and 160 GB hard drive. All algorithms are developed using Java 7.

4.6.1 Offline Sketch Discovery

Our offline sketch discovery algorithms consist of two functional components, *Candidate Theme Generation* and *Sketch Discovery*. In the candidate theme generation, we report the performance with varying p and h. In the sketch discovery, we report the performance with varying k.

4.6.1.1 Candidate theme generation algorithms

To evaluate the performance, we design the following four methods for comparison:

[10]https://finance.yahoo.com/

Brute-Force (BF): BF exhaustively computes and compares for each subject all the possible window lengths.

Visiting-Window Pruning (V-WP): V-WP only adopts the *visiting-window* bound for pruning.

Unseen-Window Pruning (U-WP): U-WP only adopts the *unseen-window* bound for pruning.

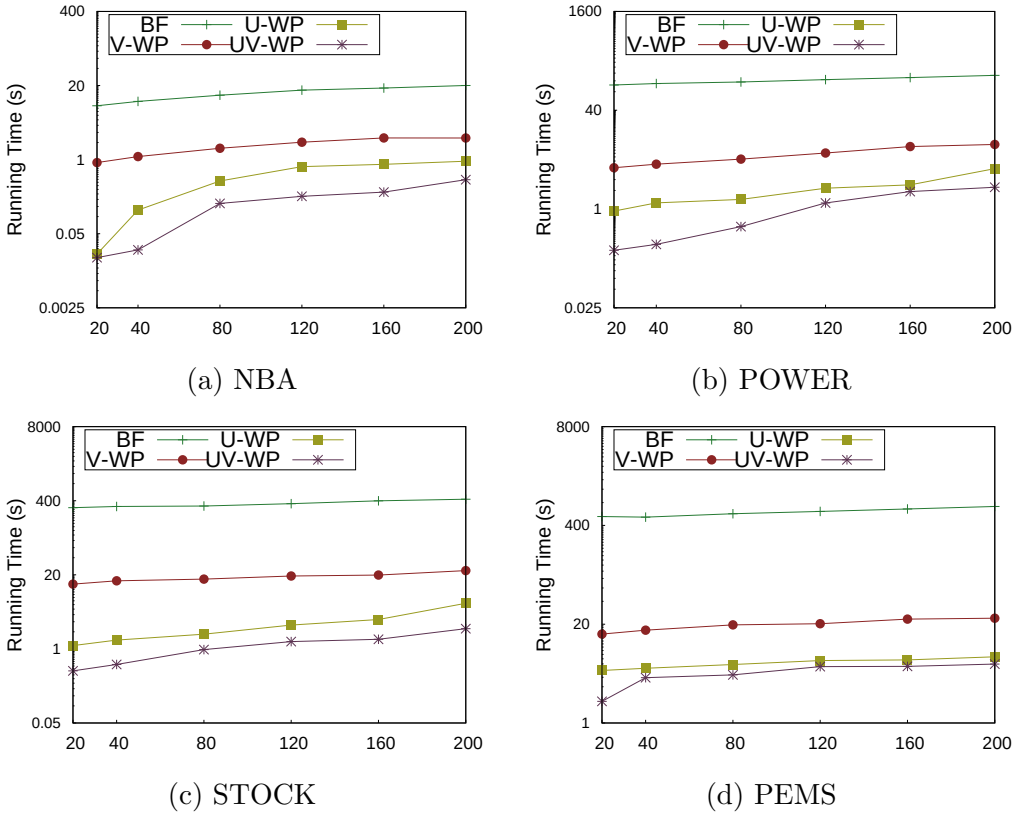
Unseen+Visting Window Pruning (UV-WP): UV-WP adopts both *unseen-window* pruning and *visiting-window* pruning.

4.6.1.2 Candidate themes generation with varying p

The running time of the four algorithms in candidate theme generation wrt. p is shown in Fig. 4.5. It is evident that when p increases, more candidate themes are qualified and thus all four algorithms require more computation time. The effect of the two proposed window-based pruning techniques can also be observed from the figure. The insight is that the unseen-window pruning plays a more important role in reducing the running time. Furthermore, when both pruning techniques are used, our method achieves at least two orders of magnitude of performance improvement.

4.6.1.3 Candidate theme generation with varying h

We then study the performance of four algorithms wrt. the number of events and report the results in Fig. 4.6. As presented in the figures, when h increases, the running time for all the four algorithms also increases. This is because more event windows need to be evaluated. Again, pruning-based methods are much efficient than the baseline method. When both pruning methods are adapted, our method obtains hundreds of times faster than the baseline method.



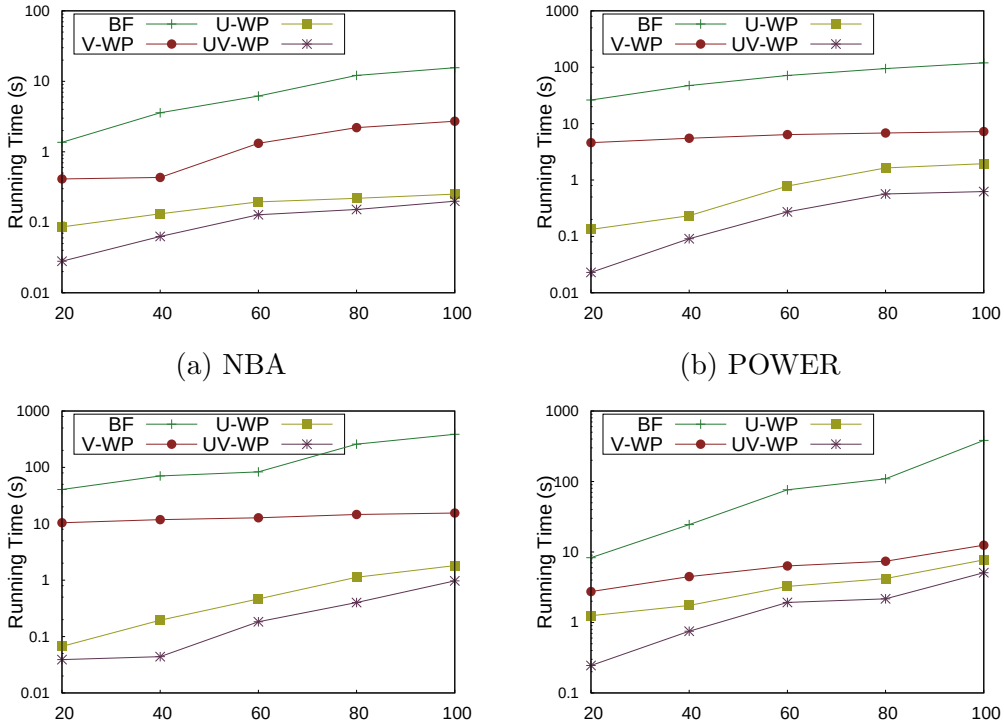
(a) NBA (b) POWER

(c) STOCK (d) PEMS

Figure 4.5: Candidate theme generation in the offline scenario with varying p.

4.6.1.4 Sketch discovery with varying k

After candidate themes are generated, we greedily find the k-sketch for each subject. Here, we study the effect of k on the performance of the greedy algorithm. The results on four datasets are presented in Table 4.5. The table indicates that for all four datasets, as k increases, the running time of the greedy algorithm proportionally increases. This is because the complexity of the greedy algorithm is $O(k\Sigma_s|\mathbb{N}_s|)$. Since PEMS is the largest dataset with highest $\Sigma_s|\mathbb{N}_s|$, greedy algorithm performs worst on PEMS. As we observe that, even greedy algorithm takes upto 400 seconds on PEMS, the performance of the exact solution with quadratic complexity is not acceptable. This confirms the necessity of adapting the approximation algorithm.



(a) NBA (b) POWER

(c) STOCK (d) PEMS

Figure 4.6: Candidate theme generation in the offline scenario with varying h.

Table 4.5: Sketch discovery with varying k in (ms)

k	20	40	60	80	100
NBA	9,097	13,345	17,500	21,597	30,769
POWER	36,297	53,513	69,300	86,603	122,856
STOCK	63,679	93,415	122,500	151,179	215,386
PEMS	138,224	206,820	283,190	353,766	491,000

4.6.2 Online Sketch Maintenance

In the online scenario, we evaluate the following four algorithms to make performance comparisons:

Sketch Computing (SC): SC examines all event windows generated from a fresh event. Whenever there is an update in any subject's sketch, Algorithm 7 will be invoked. To improve efficiency, the updates are processed in a batch manner, i.e., multiple updates on the same subject will be batched and processed by calling Algo-

rithm 7 once.

Sketch with Early Termination (SET): SET adopts "Online Window Bound" in Theorem 4.5.1 for early termination.

Approx. Sketch (AS): AS is similar to *SC* except that it only computes the approximate sketches.

Approx. Sketch with Early Termination (ASET): ASET computes the approximate sketches with early termination, as shown in Algorithm 8.

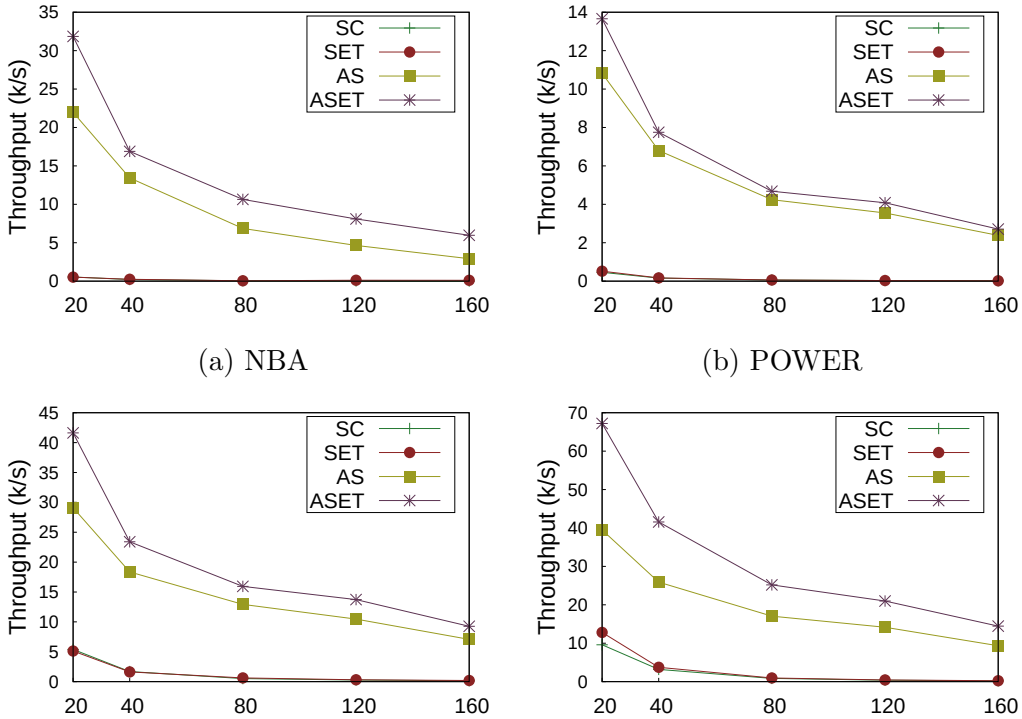
In the online setting, we are more interested in evaluating the throughput of algorithms. We report the performance wrt. p, k and h.

4.6.2.1 Query throughput with varying p

We increase p from 10 to 200 and the throughput results are shown in Fig. 4.7. Figs. 4.7 (a)-(d) show that all four datasets demonstrate similar patterns. As p increases, the throughput of the four algorithms drops. This is because as p increases, the time to maintain the p best candidate themes as well as to update the sketch increases. However, algorithms adapting *online window bound* have higher throughput than their counterparts. This is because with early termination, fewer candidate themes are generated. We can also see that *SC* and *ASC* run very slow in the online setting. This is because they need to invoke Algorithm 7 upon every update. This confirms the necessity of our approximation method as *ASET* achieves up to 500x boosts as compared to *SC*.

4.6.2.2 Query throughput with varying k

We then evaluate how the throughput varies wrt. k. The results are presented in Fig. 4.8. The figures tell similar patterns as Figs. 4.7. First, as k increases, the throughput of all the four algorithms decreases. This is because as k becomes large, more operations are needed for maintaining the sketch. Second, the throughput of



(a) NBA (b) POWER

(c) STOCK (d) PEMS

Figure 4.7: Throughput in online scenario with varying p.

SC and SET are order of magnitudes smaller than AS and $ASET$. This is because of the repetitive calling of Algorithm 7 for each arrival event, which heavily depends on k. We observe that in some datasets (e.g., Fig. 4.8 (a)) there is a hundreds time boost for $ASET$ as compared to SC.

4.6.2.3 Query throughput with varying h

Finally we study the effect of h in affecting algorithm throughput online. We change h from 20 to 100, and the results are in Fig. 4.9. As shown in the figures, when h increases, the throughput of the four algorithms steady drops. This is because as h increases, $|\mathbb{H}_s|$ for each subject increases. Therefore, in Algorithm 8, more time is needed to process each event window. We notice that $ASET$ has a flatter slope than AS; this dues to the benefit from the prunings of *online window bound*.

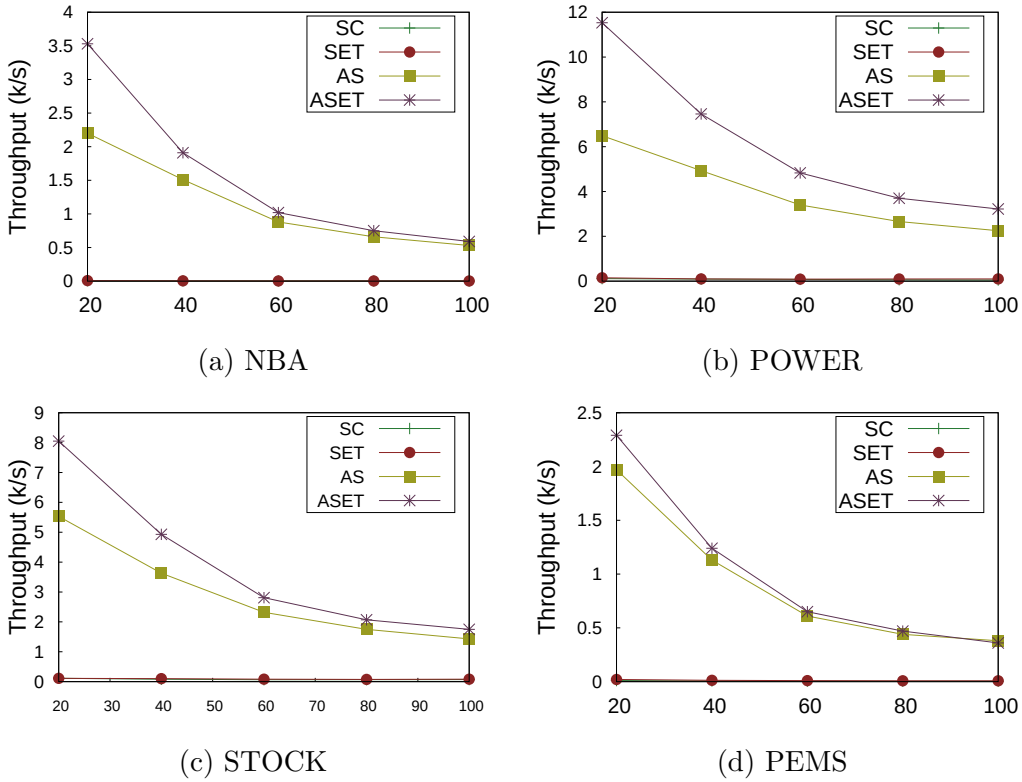
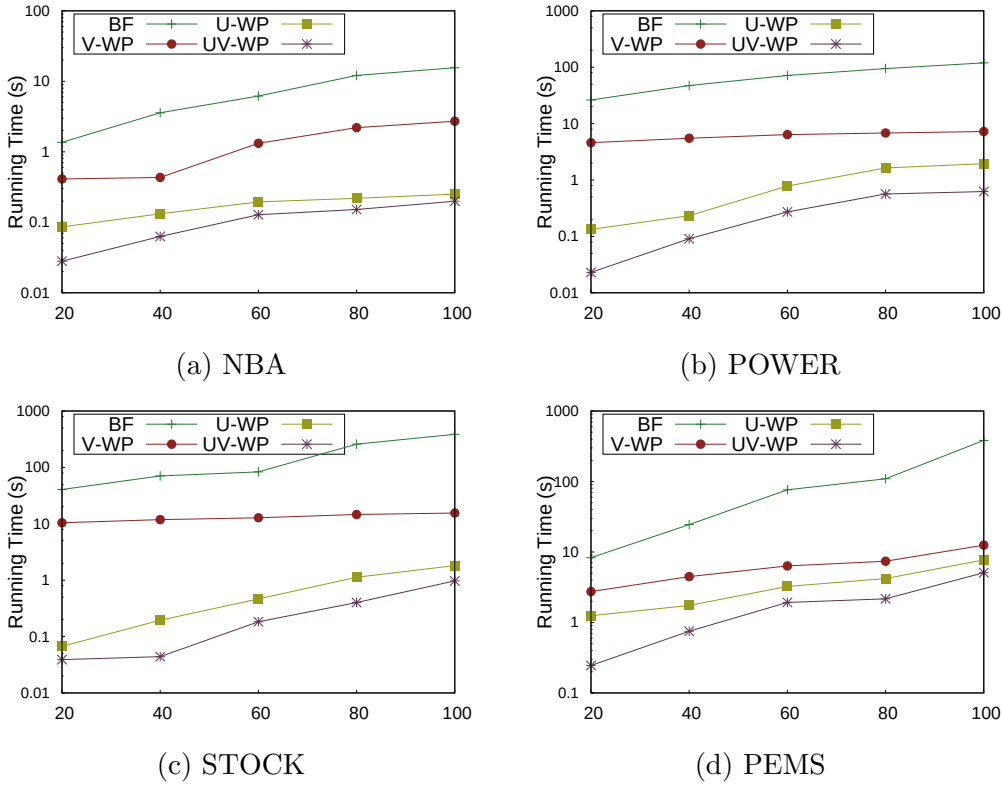


Figure 4.8: Throughput in online scenario with varying k.

4.6.3 Comparison with Other Techniques

We also compare both the performance and effectiveness of our sketch discovery with prominent streak technique [30] (denoted by skyline method). Due to the mismatched goal of sketch and skyline, these experiments only serve as references.

To study the efficiency, we first modify the skyline technique in [30] to consider ranks. In the offline scenario, the rank is computed using BF method, while in the online scenario, we directly keep the WI and the rank can be easily maintained. We use the default parameters settings for both scenarios. Figs. 4.10 (a)(b) show the running time comparisons under all four datasets. We can see that, when trying to adapt for rank-awareness, naively computing the rank on skyline method incurs high overheads. For instance, as presented in Fig. 4.10 (a), in the offline scenario, the running time of skyline method is order of magnitude slower than sketch method.



(a) NBA (b) POWER

(c) STOCK (d) PEMS

Figure 4.9: Throughput in online scenario with varying h.

This supports the use of pruning techniques in computing the ranks. Similarly, as shown in Fig. 4.10 (b), in the online scenario, our sketch method still has a much better throughput. Although skyline method is free from passive update, as it is computationally expensive to maintain online skyline points, we still observe an upto 5 times boosts of sketch method.

To study the effectiveness, we conduct a user study over Amazon Mechanical Turk [11] to evaluate the attractiveness of the k-sketch. For our method, we set $p = 200$ to allow more candidate themes and set $\alpha = 0.5$ to pay equal attention to strikingness and diversity. For the skyline method, due to the overwhelming skyline points generated for each subject, we propose three augmented methods to pick k of them. In total, we have the following five algorithms to compare with:

1. SK: selects the k-sketch for each player generated by offline sketch discovery

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

method.

2. SK_a: selects the k-sketch for each player generated by online approximate sketch method;
3. SY_s: randomly selects k streaks for each player from the bunch of skylines generated by [30];
4. SY_m: selects k streaks sorted by freshness of the events;
5. SY_r: selects k rank-attached streaks sorted by the scoring function proposed in this paper;

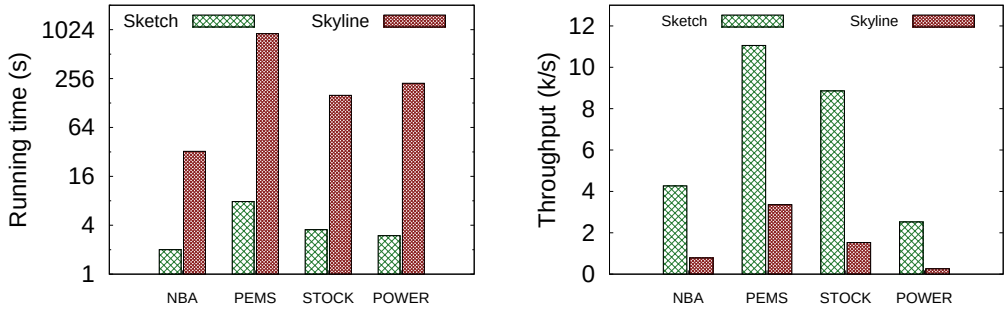
In our experimental setting, we use the NBA dataset and set $k = 20$ to generate 20 news themes for each player. Each news theme is in the following format:

[2003-04-14]: Michael-Jordan scored an average of 30.30 pts for 989 straight games,which is No. 1 in NBA history!!

Thus, each algorithm results in 20 news themes for each player. We design each job in AMT to contain 20 questions. For each question, we randomly pick one unchosen news theme from each algorithm. So each question contains 5 news and we require users to pick the most attractive one in each question. We received responses from 202 participants who have knowledge in NBA[12]. Then, for each algorithm, we count the frequency of being selected as the most attractive and report the percentage results in the pie chart in Fig. 4.11.

The charts clearly shows that SK is the most effective method as 45% of its generated news are considered as the most attractive. Interestingly, we observe that there are respondents who choose other comparing methods as their favorites. Besides human noise, we note that a portion of the news themes, which get votes and are generated by the comparing methods, overlap with news themes produced by our sketch based approach. In fact, SK_a overlaps 52% of event windows with SK. Thus it

[12]In AMT, we are able to request respondents with certain qualifications, i.e. knowledgeable in NBA.



(a) Offline Scenario (b) Online Scenario

Figure 4.10: Efficiency comparison with existing technique

ranked as the second most attractive. Whereas SY_s overlaps only 10% event windows with SK, it is ranked last. The chart also shows that the ranking based on freshness SY_m can slightly improve the attractiveness compared with SY_s. However, when applied with our proposed ranking function(i.e., SY_r), the number of respondents preferring the ranked skyline results increases dramatically, nearly two times of the original number of respondents. This also implies the effectiveness of our scoring function.

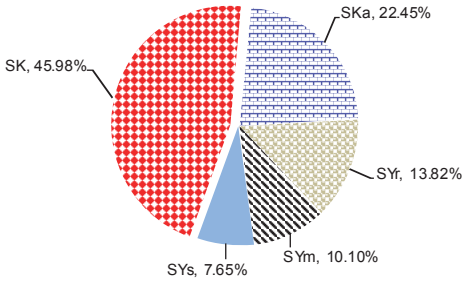


Figure 4.11: Percentage of news considered as the most attractive.

4.7 Conclusion and Future works

In this paper, we look at the problem of automatically discovering striking news themes from sequenced data. We group neighbourhood events into windows and propose a novel idea of *ranking* to represent candidate news themes. We then formulate

the *sketch discovery* problem which aims to select the k best new themes as the sketch for each subject. We study the sketch discovery problem in both offline and online scenarios, and propose efficient solutions to cope each scenario. In particular, we design novel window-level pruning techniques and a (1-1/e) greedy algorithm to achieve efficient sketch discovery in offline. Then we design a 1/8 approximation algorithm for the online sketch discovery. Our comprehensive experiments demonstrate the efficiency of our solutions and a human study confirms the effectiveness of sketch discovery in news reporting.

Our work opens a wide area of research in computational journalism. In our next step, we aim to extend the event sources from sensor data to non-schema data such as tweets in social networks. We also aim to investigate multi-subject models to automatically generate news themes across different subjects.

Chapter 5

A General Framework for Mining General Co-movement Patterns over Large-scale Trajectory Databases

5.1 Introduction

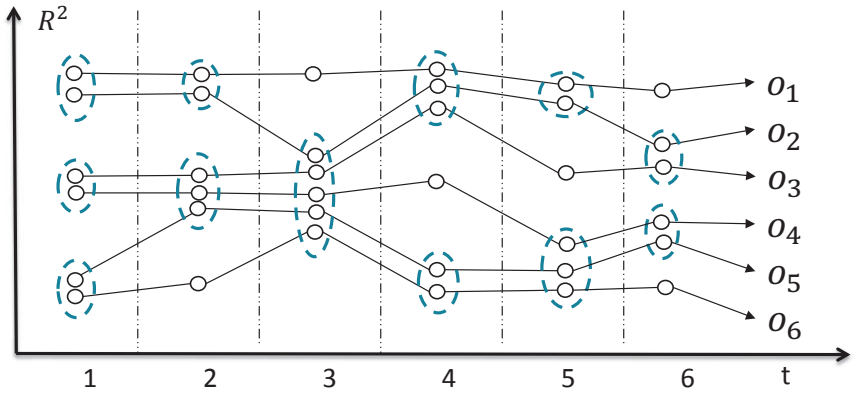
The prevalence of positioning devices has drastically boosted the scale and spectrum of trajectory collection to an unprecedented level. Tremendous amounts of trajectories, in the form of sequenced spatial-temporal records, are continually generated from animal telemetry chips, vehicle GPSs and wearable devices. Data analysis on large-scale trajectories benefits a wide range of applications and services, including traffic planning [**?**], animal analysis [**?**], and social recommendations [**?**], to name just a few.

A crucial task of data analysis 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 and the group is normally determined by their spatial

proximity. 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 a bunch of variants of co-movement patterns that have been developed with more advanced constraints.

Table 5.1 summarizes several popular co-moving patterns with different constraints in the attributes of clustering in spatial proximity, consecutiveness in temporal duration and computational complexity. In particular, the *flock* [**?**] and the *group* [24] patterns require all the objects in a group to be enclosed by a disk with radius r; whereas the *convoy* [**?**], the *swarm* [18] and the *platoon* [17] patterns resort to density-based spatial clustering. In the temporal dimension, 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* [18] does not impose any restriction. The *group* [24] and the *platoon* [17] adopt a compromised manner by allowing arbitrary gaps between the 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 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 black-box 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 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 $\langle o_5, o_6 : 3, 4, 5\rangle$ remains the only two candidates matching the patterns. The *swarm* relaxes the pattern matching by discarding the temporal consecutiveness



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.

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 as an example to illustrate the scenario.

Pattern	Proximity	Consecutiveness	Time Complexity
flock [?]	disk-based	global	$O(\|\mathbb{O}\|\|\mathbb{T}\| \log(\|\mathbb{O}\|))$
convoy [?]	density-based	global	$O(\|\mathbb{O}\|^2 + \|\mathbb{O}\|\|\mathbb{T}\|)$
swarm [18]	density-based	-	$O(2^{\|\mathbb{O}\|}\|\mathbb{O}\|\|\mathbb{T}\|)$
group [24]	disk-based	local	$O(\|\mathbb{O}\|^2\|\mathbb{T}\|)$
platoon [17]	density-based	local	$O(2^{\|\mathbb{O}\|}\|\mathbb{O}\|\|\mathbb{T}\|)$

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.

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 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.

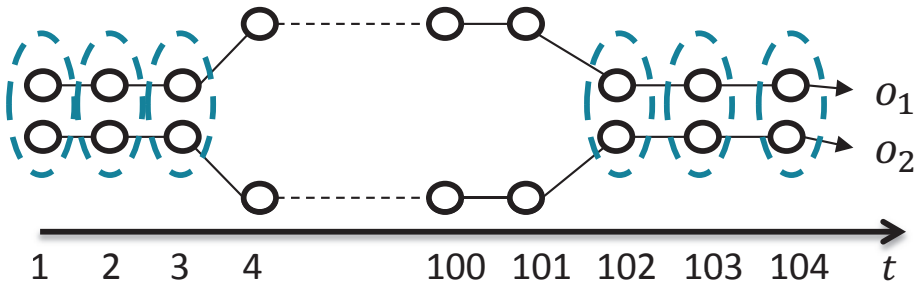
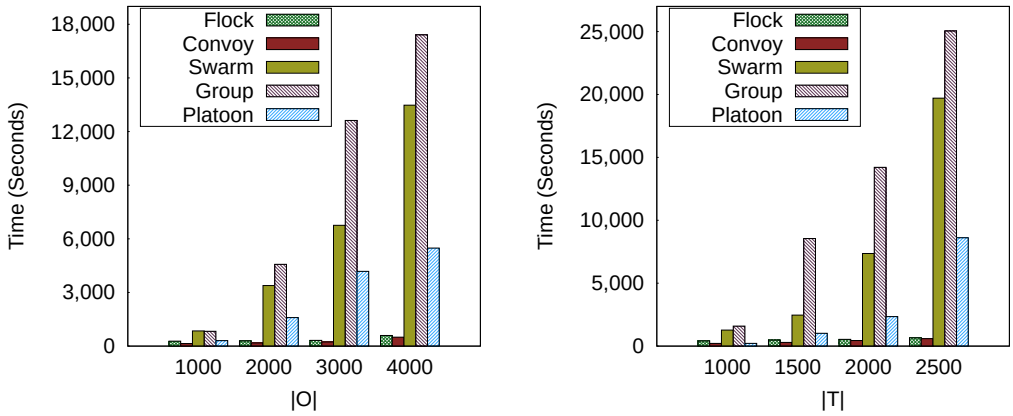


Figure 5.2: *Loose-connection* anomaly. Even though $\langle o_1, o_2 : 1, 2, 3, 102, 103, 104\rangle$ is considered as a valid *platoon* pattern, it is highly probable that these two objects are not related as the two consecutive segments are 98 timestamps apart.

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 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 up two 2.4 million data points. Default parameters are $M = 15$, $K = 180$, $L = 30$.

Therefore, our primary contributions in this paper are to close these two gaps. 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.~~ In GCMP, we adopt the concept of *gap parameter* G in sequential pattern mining field [**?**] to constrain the temporal distance between two consecutive segments. By setting a feasible G, the loose-connection anomaly can be effectively controlled. In addition, our GCMP is both general and expressive. It can be reduced to any of the previous patterns by customizing its parameters.

Second, we investigate deploying our GCMP detector on MapReduce platforms (such as Hadoop and Spark) to tackle the scalability issue. Our technical contributions are three-fold. First, we design a baseline solution by replicating the snapshots in

multiple data chunks 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 star 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-scaled 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 upto 14 times efficiency as compared to the baseline algorithm with almost linear scalability.

The rest of our paper is organized as follows: Section 5.2 summarizes the relevant literature on trajectory pattern mining. Section 5.3 defines the problem of general co-movement pattern mining. Section 5.4 provides a baseline solution. An advanced solution named *Star Partitioning and ApRiori Enumerator* (SPARE) is presented in Section 5.5. Section 5.6 conducts extensive experiments to verify the efficiency of our solutions. Finally, Section 5.7 concludes the paper.

5.2 Related Works

Related works can be grouped into three categories: *co-movement patterns*, *dynamic moving patterns* and *trajectory mining frameworks*. In this section, we distinguish the parallel GCMP mining from these concepts.

5.2.1 Flock and Convoy

The difference between *flock* and *convoy* lies in the object clustering methods. In *flock*, objects are clustered based on their distances. Specifically, the objects in the same cluster need to have a pair-wised distance less than *min_dist*. This essentially requires the objects to be within a disk-region of delimiter less than *min_dist*. In contrast, *convoy* clusters objects using density-based spatial clustering [11]. Technically, *flock* utilizes a m^{th}-order Voronoi diagram [14] to detect whether a subset of object with size greater than m stays in a disk-region. *Convoy* employs a trajectory simplification [9] technique to boost pairwise distance computations in the density-based clustering. After clustering, both *flock* and *convoy* use a sequential scanning method to examine each snapshots. During the scan, object groups that appear in consecutive timestamps are detected. Meanwhile, the object groups that do not match the consecutive constraint are pruned. However, such a method faces high complexity when supporting other patterns. For instance, in *swarm*, the candidate set during the sequential scanning grows exponentially, and many candidates can only be pruned after the entire snapshots are scanned.

5.2.2 Group, Swarm and Platoon

Different from *flock* and *convoy*, all the *group*,*swarm* and *platoon* patterns have more relaxed constraints on the pattern duration. Therefore, their techniques of mining are of the same skeleton. The main idea of mining is to grow object set from an empty set in a depth-first manner. During the growth, various pruning techniques are provided to prune unnecessary branches. *Group* pattern uses a VG-graph to guide the pruning of false candidates [24]. *Swarm* designs two more pruning rules called backward pruning and forward pruning [18]. *Platoon* further leverages a prefix table structure to steer the depth-first search. As shown by Li et.al. [17], *platoon* outperforms other two methods in efficiency. However, the three patterns are not

able to directly discover GCMPs. Furthermore, their pruning rules heavily rely on the depth-first search nature, which lose their efficiencies in the parallel scenario.

5.2.3 Other Related Trajectory Patterns

A closely related literature to co-movement patterns is the *dynamic moving* patterns. Instead of requiring the same set of object traveling together, *dynamic moving* patterns allow objects to temporally join or leave a group. Typical works include *moving clusters* [13], *evolving convoy* [2], *gathering* [32] etc. These works cannot model our GCMP since they enforce the global consecutiveness on the timestamps of a pattern. Mining trajectory related patterns on road networks has also been studied [**?**, **?**]. Due to the bonding with road networks, these models cannot be easily extended to broader scenarios such as visitor trajectories in shopping malls, people movements in cities and user check-in histories in social networks. Nevertheless, GCMP is capable to capture the co-movement patterns in all these applications.

5.2.4 Trajectory Mining Frameworks

Jinno et al. in [12] designed a MapReduce based algorithm to efficiently support T-pattern discovery, where a T-pattern is a set of objects visiting the same place at simliar time. Li et al. proposed a framework of processing online *evolving group* pattern [16], which focuses on supporting efficient updates of arriving objects. As these works essentially differ from co-movement pattern, their techniques cannot be directly applied to discover GCMPs.

5.2.5 Frequent Sequential Pattern Mining

Mining Gap-restricted Sequential Patterns (GSP) [**?**, **?**, **?**] aims to find sequences which satisfy a gap constraint G and occur more than M times. GCMP looks similar

to GSP since each trajectory can be viewed as a sequence. Although it is trivial to extend GSP to consider all temporal constraints in GCMP (i.e., K, L), GSP still fails to support GCMP. First, GSP allows an object to contribute more than once for a sequence's occurrence. Such double counting outputs duplicate objects and thus provides no guarantee to discover at least M distinct objects as required by GCMP. Second, GSP does not enforce output objects to be spatially close at every timestamps which leads to invalid GCMPs. These mismatches motivates us to design the native GCMP for trajectory data.

5.3 Definitions

Let $\mathbb{O} = \{o_1, o_2, ..., o_n\}$ be the set of objects and $\mathbb{T} = (1, 2, ..., N)$ be the discretized temporal dimension. A time sequence T is defined as an ordered subset of $\mathbb{T}$. Given two time sequences T_1 and T_2, we define a bunch of commonly-used operators in this paper in Table 5.2.

Operator	**Definition**
$T[i]$	the i-th element in the sequence T
$\|T\|$	the number of elements in T
$\max(T)$	the maximum element in T
$\min(T)$	the minimum element in T
$\text{range}(T)$	the range of T, i.e., $\max(T) - \min(T) + 1$
$T[i:j]$	subsequence of T from $T[i]$ to $T[j]$ (inclusive)
$T_1 \subseteq T_2$	$\forall T_1[x] \in T_1$, we have $T_1[x] \in T_2$.
$T_3 = T_1 \cup T_2$	$\forall T_3[x] \in T_3$, we have $T_3[x] \in T_1$ or $T_3[x] \in T_2$
$T_3 = T_1 \cap T_2$	$\forall T_3[x] \in T_3$, we have $T_3[x] \in T_1$ and $T_3[x] \in T_2$

Table 5.2: Operators on time sequence.

We say a sequence T is consecutive if $\forall i \in (1, ..., |T| - 1), T[i+1] = T[i] + 1$. We refer each consecutive subsequence of T as a *segment*. It is obvious that any time sequence T can be decomposed into segments and we say T is *L-consecutive* [17] if the length of every segment is no smaller than L. As illustrated in Figure 5.2, patterns

adopting the notion of L-consecutiveness (e.g., *platoon* and *group*) still suffer from the *loose connection* problem. To avoid such an anomaly without losing pattern generality, we introduce a parameter G to control the gaps between timestamps in a pattern. Formally, a G-connected time sequence is defined as follows:

Definition 5.3.1 (G-connected)**.** A time sequence T is G-connected if the gap between any of its neighboring timestamps is no greater than G, i.e., $\forall i \in (1, \ldots, |T| - 1), T[i+1] - T[i] \leq G$.

We take $T = (1, 2, 3, 5, 6)$ as an example, which can be decomposed into two segments $(1, 2, 3)$ and $(5, 6)$. T is not 3-consecutive since the length of $(5, 6)$ is 2. Thus, it is safe to say either T is 1-consecutive or 2-consecutive. On the other hand, T is 2-connected since the maximum gap between its neighboring timestamps is 2. It is worth noting that T is not 1-connected because the gap between $T[3]$ and $T[4]$ is 2 (i.e., $5 - 3 = 2$).

Given a trajectory database that is discretized into snapshots, we can conduct a clustering method, either disk-based or density-based, to identify groups with spatial proximity. Let T be the set of timestamps in which a group of objects O are clustered. We are ready to define a more general co-movement pattern:

Definition 5.3.2 (General Co-Movement Pattern)**.** A general co-movement pattern finds a set of objects O satisfying the following five constraints: (1) **closeness:** the objects in O belong to the same cluster in the timestamps of T; (2) **significance:** $|O| \geq M$; (3) **duration:** $|T| \geq K$; (4) **consecutiveness:** T is L-consecutive; and (5) **connection:** T is G-connected.

There are four parameters in our general co-movement pattern, including object constraint M and temporal constraints K, L, G. These parameters have different effects on the number of patterns discovered. Larger K, L, M put more stringent constraints on patterns such that less patterns could satisfy the requirements. Larger

G relaxes the temporal constraints therefore more patterns could be discovered. Currently, we leave tuning of parameters to users to accommodate various application needs. By customizing these parameters, our pattern can express other patterns proposed in the previous literature, as illustrated in Table 5.3. In particular, by setting $G = |\mathbb{T}|$, we achieve the *platoon* pattern. By setting $G = |\mathbb{T}|, L = 1$, we reach the *swarm* pattern. By setting $G = |\mathbb{T}|$, $M = 2$, $K = 1$, we gain the *group* pattern. Finally by setting $G = 1$, we result in the *convoy* and *flock* patterns. In addition to the flexibility of representing other existing patterns, our GCMP is able to avoid the *loose connection* anomaly by tuning the parameter G. It is notable that GCMP cannot be modeled by existing patterns.

Pattern	M	K	L	G	**Clustering**
Group	2	1	2	$\|\mathbb{T}\|$	Disk-based
Flock	$\cdot$	$\cdot$	K	1	Disk-based
Convoy	$\cdot$	$\cdot$	K	1	Density-based
Swarm	$\cdot$	$\cdot$	1	$\|\mathbb{T}\|$	Density-based
Platoon	$\cdot$	$\cdot$	$\cdot$	$\|\mathbb{T}\|$	Density-based

Table 5.3: Expressing other patterns using GCMP. $\cdot$ indicates a user specified value. M represents the *significance* constraint. K represents the *duration* constraint. L represents the *consecutiveness* constraint. G represents the *connection* constraint.

Our definition of GCMP is independent of the clustering method. Users can apply different clustering methods to facilitate different application needs. We currently expose both disk-region based clustering and DBSCAN as options to the users. In summary, the goal of this paper is to present a parallel solution for discovering all the valid GCMPs from large-scale trajectory databases. Before we move on to the algorithmic part, we list the notations that are used in the following sections.

Symbol	Meaning
S_t	snapshot of objects at time t
M	significance constraint
K	duration constraint
L	consecutiveness constraint
G	connection constraint
$P = \langle O : T \rangle$	pattern with object set O, time sequence T
S_t	set of clusters at snapshot t
η	replication factor in the TRPM framework
λ_t	partition with snapshots $S_t, .., S_{t+\eta-1}$
G_A	aggregated graph in SPARE framework
Sr_i	star partition for object (vertex) i
Γ	size of the largest star partition

Table 5.4: Summary of the use of notations.

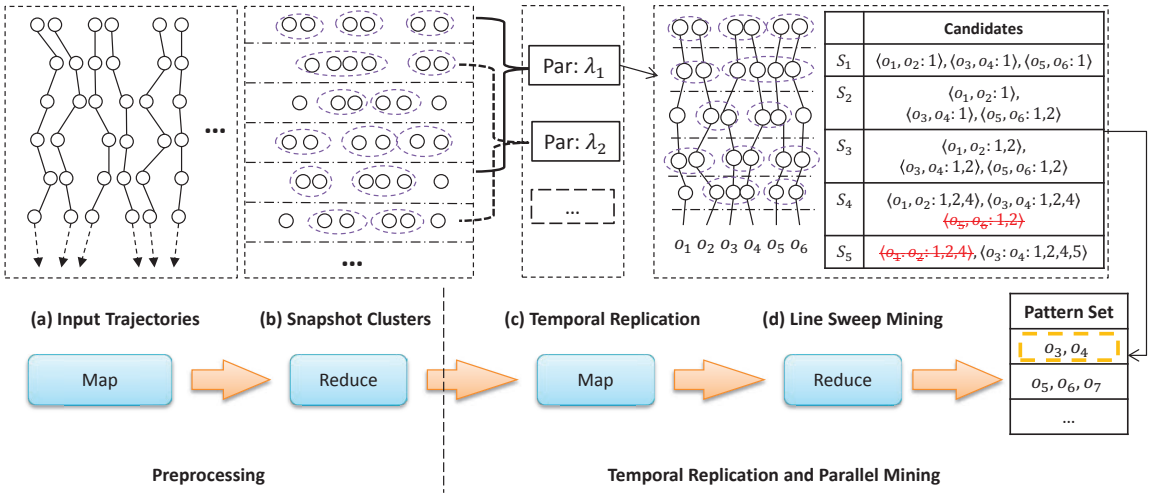


Figure 5.4: Work flow of Temporal Replication and Parallel Mining (TRPM). (a) and (b) correspond to the first map-reduce stage which clusters objects in each snapshot. (c) and (d) correspond to the second map-reduce stage, which uses TRPM to detect GCMP in parallel.

5.4 Baseline: Temporal Replication and Parallel Mining

In this section, we propose a baseline solution that resorts to MapReduce (MR) as a general, parallel and scalable paradigm for GCMP mining. The framework, named *temporal replication and parallel mining* (TRPM), is illustrated in Figure 5.4. There are two stages of map-reduce jobs connected in a pipeline manner. The first stage deals with spatial clustering of objects in each snapshot, which can be seen as a preprocessing step for the subsequent pattern mining phase. In particular, for the

first stage, the timestamp is treated as the key in the map phase and objects within the same snapshot are clustered (DBSCAN or disk-based clustering) in the reduce phase. Finally, the reducers output clusters of objects in each snapshot, represented by a list of key-value pairs $\langle t, S_t \rangle$, where t is the timestamp and S_t is a set of clustered objects at snapshot t.

Our focus in this paper is on the second map-reduce stage of parallel mining, which essentially addresses two key challenges. The first is to ensure effective data partitioning such that the mining on each partition can be conducted independently; and the second is to efficiently mine the valid patterns within each partition.

It is obvious that we cannot simply split the trajectory database into disjoint partitions because a GCMP requires L-consecutiveness and the corresponding segments may span multiple partitions. Our strategy is to use data replication to enable parallel mining. Each snapshot will replicate its clusters to $\eta - 1$ preceding snapshots. In other words, the partition for the snapshot S_t contains clusters in S_t, $S_{t+1} \ldots, S_{t+\eta-1}$. Determining a proper η is critical in ensuring the correctness and efficiency of TRPM. If η is too small, certain cross-partition patterns may be missed. If η is too large, expensive network communication and CPU processing costs would be incurred in the map and reduce phases respectively. Our objective is to find an η that is not large but can guarantee correctness.

In our implementation, we set $\eta = (\lceil \frac{K}{L} \rceil - 1) * (G - 1) + K + L - 1$. Intuitively, with K timestamps, at most $\lceil \frac{K}{L} \rceil - 1$ gaps may be generated as the length of each L-consecutive segment is at least L. Since the gap size is at most $G-1$, $(\lceil \frac{K}{L} \rceil - 1) * (G-1)$ is the upper bound of timestamps allocated to gaps. The remaining part of the expression, $K + L - 1$, is used to capture the upper bound allocated for the L-consecutive segments. We formally prove that η can guarantee correctness.

Theorem 5.4.1. $\eta = (\lceil \frac{K}{L} \rceil - 1) * (G-1) + K + L - 1$ guarantees that no valid pattern is missing.

Proof. Given a valid pattern P, we can always find at least one valid subsequence of $P.T$ that is also valid. Let T' denote the valid subsequence of $P.T$ with the minimum length. In the worst case, $T' = P.T$. We define $\text{range}(T) = \max(T) - \min(T) + 1$ and prove the theorem by showing that $\text{range}(T') \leq \eta$. Since T' can be written as a sequence of L-consecutive segments interleaved by gaps: $l_1, g_1, \ldots, l_{n-1}, g_{n-1}, l_n$ ($n \geq 1$), where l_i is a segment and g_i is a gap. Then, $\text{range}(T')$ is calculated as $\Sigma_{i=1}^{i=n}|l_i| + \Sigma_{i=1}^{i=n-1}|g_i|$. Since T' is valid, then $\Sigma_{i=1}^{i=n}|l_i| \geq K$. As T' is minimum, if we remove the last l_n, the resulting sequence should not be valid. Let $K' = \Sigma_{i=1}^{i=n-1}|l_i|$, which is the size of the first $(n-1)$ segments of T'. Then, $K' \leq K - 1$. Note that every $|l_i| \geq L$, thus $n \leq \lceil \frac{K'}{L} \rceil \leq \lceil \frac{K}{L} \rceil$. By using the fact that every $|g_i| \leq G - 1$, we achieve $\Sigma_{i=1}^{i=n-1}|g_i| \leq (n-1)(G-1) \leq (\lceil \frac{K}{L} \rceil - 1)(G-1)$. Next, we consider the difference between K and K', denoted by $\Delta = K - K'$. To ensure T''s validity, l_n must equal to $\min(L, \Delta)$. Then, $\Sigma_{i=1}^{i=n}|l_i| = K' + l_n = K - \Delta + \min(L, \Delta) \leq K - 1 + L$. We finish showing $\text{range}(T') \leq \eta$. Therefore, for any valid sequence T, it exists at least one valid subsequence with range no greater than η and hence this pattern can be detected in a partition with η snapshots. □

Based on the above theorem, under TRPM, every consecutive η snapshots form a partition. In other words, each snapshot S_t corresponds to a partition $\lambda_t = \{S_t, ..., S_{t+\eta-1}\}$. Our next task is to design an efficient pattern mining strategy within each partition. Our solution includes a line sweep algorithm to sequentially scan the η replicated snapshots in a partition and an effective candidate pattern enumeration mechanism.

Details of the algorithm are presented in Algorithm 10. We keep a candidate set C (Line 1) during the sweeping process. It is initialized using the clusters with size no smaller than M in the first snapshot. Then, we sequentially scan each snapshot (Lines 7-27) and generate new candidates by extending the original ones in C. In particular, we join candidates in C with all the clusters in S_j to form new candidates

Algorithm 10 Line Sweep Mining

Input: $\lambda_t = \{S_t, ..., S_{t+\eta-1}\}$

1: $C \leftarrow \{\}$ ▷ Candidate set
2: **for all** clusters s in snapshot S_t **do**
3: **if** $|s| \geq M$ **then**
4: $C \leftarrow C \cup \{\langle s, t\rangle\}$
5: **end if**
6: **end for**
7: **for all** $S_j \in \{S_{t+1}, \ldots, S_{t+\eta-1}\}$ **do**
8: $N \leftarrow \{\}$
9: **for all** $(c, s) \in C \times S_j$ **do**
10: $c' \leftarrow \langle c.O \cap s.O, c.T \cup \{j\}\rangle$
11: **if** $c'.T$ is valid **then**
12: output c'
13: **else if** $|c'.O| \geq M$ **then**
14: $N \leftarrow N \cup \{c'\}$
15: **end if**
16: **end for**
17: **for all** $c \in C$ **do**
18: **if** $j - \max(c.T) \geq G$ **then**
19: $C \leftarrow C - \{c\}$
20: output c, if c is a valid pattern
21: **end if**
22: **if** c's first segment is less than L **then**
23: $C \leftarrow C - \{c\}$
24: **end if**
25: **end for**
26: $C \leftarrow C \cup N$
27: **end for**
28: output valid patterns in C

(Lines 9-16).

After sweeping all the snapshots, all the valid patterns are stored in C (Line 28). It is worth noting that C continues to grow during the whole sweeping process. We can use three pruning rules to early remove false candidates from C. Since there is a partition λ_t for each S_t, only patterns that start from timestamp t need to be discovered. Therefore, those patterns that do not appear in the S_t are false candidates. In particular, our three pruning rules are as follows: First, when sweeping snapshot S_j, new candidates with objects set smaller than M are pruned (Line 14). Second, after joining with all clusters in S_j, candidates in C with the maximum timestamp

no smaller than $j - G$ are pruned (Lines 18-21). Third, candidates in C with the size of the first segment smaller than L are pruned (Lines 22-24). With the three pruning rules, the size of C can be significantly reduced.

The complete picture of temporal replication and parallel mining is summarized in Algorithm 11. We illustrate the workflow of the TRPM method using Figure 5.4 (c) and (d) with pattern parameters $M = 2, K = 3, L = 2, G = 2$. By Theorem 5.4.1, η is calculated as $(\lceil \frac{K}{L} \rceil - 1) * (G - 1) + K + L - 1 = 5$. Therefore, in Figure 5.4 (c), every 5 consecutive snapshots are combined into a partition in the map phase. In Figure 5.4 (d), a line sweep method is illustrated for partition λ_1. Let C_i be the candidate set when sweeping snapshot S_i. Initially, C_1 contains patterns with object sets in snapshot S_1. As we sweep the snapshots, the patterns in C_i grow. At snapshot S_4, the candidate $\langle o_5, o_6 \rangle$ is removed. This is because the gap between its latest timestamp (i.e., 2) and the next sweeping timestamp (i.e., 5) is 3, which violates the G-connected constraint. Next, at snapshot S_5, the candidate $\langle o_1, o_2 \rangle$ is removed. This is because its local consecutive segment (4) has only 1 element, which violates the L-consecutive constraint. Finally, $\langle o_3, o_4 \rangle$ is the valid pattern and is returned. Note that in this example, $\eta = 5$ is the minimum setting that can guarantee correctness. If η is set to 4, the pattern $\langle o_3, o_4 \rangle$ would be missed.

5.5 SPARE: Star Partitioning and Apriori Enumerator

The aforementioned TRPM scheme replicates snapshots based on the temporal dimension which suffers from two drawbacks. First, the replication factor η can be large. Second, the same valid pattern may be discovered from different partitions which results in redundant works. To resolve these limitations, we propose a new Star Partitioning and ApRiori Enumerator, named SPARE, to replace the second

Algorithm 11 Temporal Replication and Parallel Mining

Input: list of $\langle t, S_t \rangle$ pairs
1: $\eta \leftarrow (\lceil \frac{K}{L} \rceil - 1) * (G - 1) + K + L - 1$
2: *—Map Phase—*
3: **for all** snapshots S_t **do**
4: **for all** $i \in 1...\eta - 1$ **do**
5: emit key-value pair $\langle \max(t - i, 0), S_t \rangle$
6: **end for**
7: **end for**
8: *—Partition and Shuffle Phase—*
9: **for all** key-value pairs $\langle t, S \rangle$ **do**
10: group-by t and emit a key-value pair $\langle t, \lambda_t \rangle$, where $\lambda_t = \{S_t, S_{t+1}, ..S_{t+\eta-1}\}$
11: **end for**
12: *—Reduce Phase—*
13: **for all** key-value pairs $\langle t, \lambda_t \rangle$ **do**
14: call line sweep mining for partition λ_t
15: **end for**

stage of the map-reduce jobs in Figure 5.4. Our new parallel mining framework is shown in Figure 5.5. Its input is the set of clusters generated in each snapshot and the output contains all the valid GCMPs. In the following, we explain the two major components: star partitioning and apriori enumerator.

5.5.1 Star Partitioning

Let G_t be a graph for snapshot S_t, in which each node is a moving object and two objects are connected if they appear in the same cluster. It is obvious that G_t consists of a set of small cliques. Based on G_t, we define an aggregated graph G_A to summarize the cluster relationship among all the snapshots. In G_A, two objects form an edge if they are connected in any G_ts. Furthermore, we attach an inverted list for each edge, storing the associated timestamps in which the two objects are connected. An example of G_A, built on the trajectory database in Figure 5.1, is shown in Figure 5.5 (a). As long as two objects are clustered in any timestamps, they are connected in G_A. The object pair $\langle o_1, o_2 \rangle$ appears in two clusters at timestamps 2 and 3 and is thus associated with an inverted list $(2, 3)$.

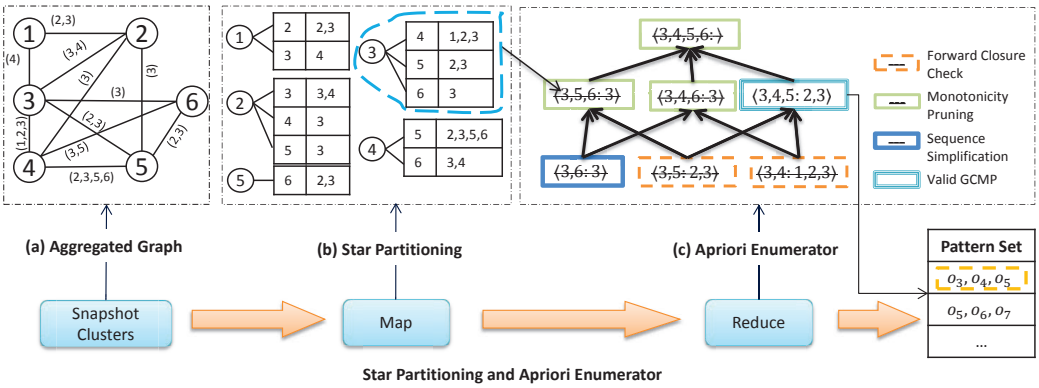


Figure 5.5: Star Partitioning and ApRiori Enumerator (SPARE). (a) Aggregated graph G_A generated from Figure 1. (b) Five star partitions are generated from G_A. Star IDs are circled, vertexes and inverted lists are in the connected tables. (c) Apriori Enumerator with various pruning techniques.

We use *star* as the data structure to capture the pair relationships. To avoid duplication, as G_t is an undirected graph and an edge may appear in multiple stars, we enforce a global vertex ordering among the objects and propose a concept named *directed star*.

Definition 5.5.1 (Directed Star)**.** Given a vertex with global ID s, its directed star Sr_s is defined as the set of neighboring vertexes with global ID $t > s$. We call s the star ID.

With the global vertex ordering, we can guarantee that each edge is contained in a unique star partition. Given the aggregated graph G_A in Figure 5.5 (a), we enumerate all the possible directed stars in Figure 5.5 (b). These stars are emitted from mappers to different reducers. The key is the star ID and the value is the neighbors in the star as well as the associated inverted lists. The reducer will then call the Apriori-based algorithm to enumerate all the valid GCMPs.

Before we introduce the Apriori Enumerator, we are interested to examine the issue of global vertex ordering on the moving objects. This is because assigning different IDs to the objects will result in different star partitioning results, which will eventually affect the workload balance among reducers. The job with the performance

bottleneck is often known as a *straggler* [**?**, **?**]. In the context of star partitioning, a straggler refers to the job assigned with the maximum star partition. We use Γ to denote the size of such straggler partition and Γ is set to the number of edges in a directed star[1]. Clearly, a star partitioning with small Γ is preferred. For example, Figure 5.6 gives two star partitioning results under different vertex ordering on the same graph. The top one has $\Gamma = 5$ while the bottom one has $\Gamma = 3$. Obviously, the bottom one with smaller Γ is much more balanced.

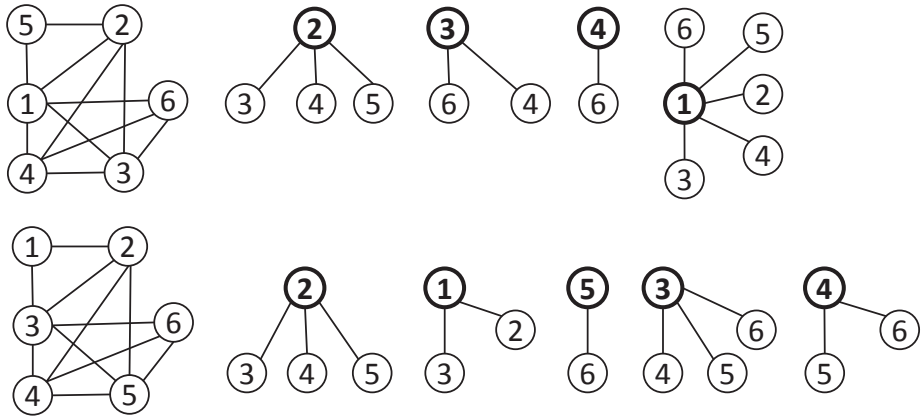


Figure 5.6: Examples of star partitioning with different vertex orderings.

Although it is very challenging to find the optimal vertex ordering from the $n!$ possibilities, we observe that a random order can actually achieve satisfactory performance based on the following theorem.

Theorem 5.5.1. Let Γ^* be the value derived from the optimal vertex ordering and Γ be the value derived from a random vertex ordering. With probability $1 - 1/n$, we have $\Gamma = \Gamma^* + O(\sqrt{n \log n})$.

Proof. See Appendix A.2.1. □

If G_A is a dense graph, we can get a tighter bound for $(\Gamma - \Gamma^*)$.

[1]A star is essentially a tree structure and the number of nodes equals the number of edges minus one.

Theorem 5.5.2. Let d be the average degree in G_A. If $d \geq \sqrt{12 \log n}$, with high probability $1 - 1/n$, $\Gamma = \Gamma^* + O(\sqrt{d \log n})$.

Proof. See Appendix A.2.1. □

Hence, we can simply use object ID to determine the vertex ordering in our implementation.

5.5.2 Apriori Enumerator

Intuitively, given a GCMP with an object set $\{o_1, o_2, \ldots, o_m\}$, all the pairs of $\langle o_i, o_j \rangle$ with $1 \leq i < j \leq m$ must be connected in the associated temporal graphs $\{G_t\}$. This inspires us to leverage the classic Apriori algorithm to enumerate all the valid GCMPs starting from pairs of objects. However, we observe that the monotonicity property does not hold between an object set and its supersets.

Example 5.5.1. In this example, we show that if an object set is not a valid pattern, we cannot prune all its super sets. Consider two candidates $P_1 = \langle o_1, o_2 : 1, 2, 3, 6 \rangle$ and $P_2 = \langle o_1, o_3 : 1, 2, 3, 7 \rangle$. Let $L = 2, K = 3$ and $G = 2$. Both candidates are not valid patterns because the constraint on L is not satisfied. However, when considering their object superset $\langle o_1, o_2, o_3 \rangle$, we can infer that their co-clustering timestamps are in $(1, 2, 3)$. This is a valid pattern conforming to the constraints of L, K, G. Thus, we need a new type of monotonicity to facilitate pruning.

5.5.2.1 Monotonicity

To ensure monotonicity, we first introduce a procedure named *sequence simplification*, to reduce the number of edges as well as unnecessary timestamps in the inverted lists. For instance, if the size of the inverted list for an edge e is smaller than K, then the edge can be safely removed because the number of timestamps in which its supersets are clustered must also be smaller than K. To generalize the idea, we propose three

concepts: *maximal G-connected subsequence*, *decomposable sequence* and *sequence simplification*.

Definition 5.5.2 (Maximal G-connected Subsequence)**.** A sequence T' is said to be a maximal G-connected subsequence of T if (1) T' is the subsequence of T, (2) T' is G-connected, and (3) there exists no other subsequence T'' of T such that T' is the subsequence of T'' and T'' is G-connected.

Example 5.5.2. Suppose $G = 2$ and consider two sequences $T_1 = (1, 2, 4, 5, 6, 9, 10, 11, 13)$ and $T_2 = (1, 2, 4, 5, 6, 8, 9)$. T_1 has two maximal 2-connected subsequences:$T_1^A = (1, 2, 4, 5, 6)$ and $T_1^B = (9, 10, 11, 13)$. This is because the gap between T_1^A and T_1^B is 3 and it is impossible for the timestamps from T_1^A and T_1^B to form a new subsequence with $G \leq 2$. Since T_2 is 2-connected, T_2 has only one maximal 2-connected subsequence which is itself.

The maximal G-connected subsequence has the following two properties:

Lemma 5.5.3. Suppose $\{T_1, T_2, \ldots, T_m\}$ is the set of all maximal G-connected subsequences of T, we have (1) $T_i \cap T_j = \emptyset$ for $i \neq j$ and (2) $T_1 \cup T_2 \cup \ldots \cup T_m = T$.

Proof. We assume $T_i \cap T_j \neq \emptyset$. Let $T_i = (T_i[1], T_i[2], \ldots, T_i[p])$ and $T_j = (T_j[1], T_j[2], \ldots, T_j[n])$. Suppose $T[x]$ is a timestamp occurring in both T_i and T_j. Let $T[y] = \min\{T_i[1], T_j[1]\}$, i.e., the minimum timestamp of $T_i[1]$ and $T_j[1]$ occurs at the y-th position of sequence T. Similarly, we assume $T[z] = \max\{T_i[p], T_j[n]\}$. Apparently, the two subsequences $T[y : x]$ and $T[x : z]$ are G-connected because T_i and T_j are both G-connected. Then, sequence $(T_y, \ldots, T_x, \ldots, T_z)$, the superset of T_i and T_j, is also G-connected. This contradicts with the assumptions that T_i and T_j are maximal G-connected subsequences.

To prove (2), we assume $\cup_{i=1}^{i=m} T_i$ does not cover all the timestamps in T. Then, we can find a subsequence $T' = T[x : x + t]$ such that $T[x - 1] \in T_a$ $(1 \leq a \leq m)$, $T[x + t + 1] \in T_b$ $(1 \leq b \leq m)$ and all the timestamps in T' is not included in any T_i.

Let $g' = \min\{T[x] - T[x-1], T[x+t+1] - T[x+t]\}$. If $g' \leq G$, then it is easy to infer that T_a or T_b is not a maximal G-connected subsequence because we can combine it with $T[x]$ or $T[x+t]$ to a form superset which is also G-connected. If $g' > G$, T' itself is a maximal G-connected subsequence which is missed in$\cup_{i=1}^{i=m} T_i$. Both cases lead to contradictions. □

Lemma 5.5.4. If T_1 is a subset of T_2, then for any maximal G-connected subsequence T_1' of T_1, we can find a maximal G-connected subsequence T_2' of T_2 such that T_1' is a subset of T_2'.

Proof. Since $T_1' \subseteq T_1 \subseteq T_2$, we know T_1' is a G-connected subsequence of T_2. Based on Lemma 5.5.3, we can find a maximal G-connected subsequence of T_2, denoted by T_2', such that $T_1' \cap T_2' \neq \emptyset$. If there exists a timestamp $T_1'[x]$ such that $T_1'[x] \notin T_2'$, similar to the proof of case (1) in Lemma 5.5.3, we can obtain a contradiction. Thus, all the timestamps in T_1' must occur in T_2'. □

Definition 5.5.3 (Decomposable Sequence)**.** T is decomposable if for any of its maximal G-connected subsequence T', we have (1) T' is L-consecutive; and (2) $|T'| \geq K$.

Example 5.5.3. Let $L = 2, K = 4$ and we follow the above example. T_1 is not a decomposable sequence because one of its maximal 2-connected subsequence (i.e., T_1^B) is not 2-consecutive. In contrast, T_2 is a decomposable sequence because the sequence itself is the maximal 2-connected subsequence, which is also 2-consecutive and with size no smaller than than 4.

Definition 5.5.4 (Sequence Simplification)**.** Given a sequence T, the simplification procedure $\texttt{sim}(T) = g_{G,K} \cdot f_L(T)$ can be seen as a composite function with two steps:

1. f-step: remove segments of T that are not L-consecutive;
2. g-step: among the maximal G-connected subsequences of $f_L(T)$, remove those with size smaller than K.

Example 5.5.4. Take $T = (1, 2, 4, 5, 6, 9, 10, 11, 13)$ as an example for sequence simplification. Let $L = 2, K = 4$ and $G = 2$. In the f-step, T is reduced to $f_2(T) = (1, 2, 4, 5, 6, 9, 10, 11)$. The segment (13) is removed due to the constraint of $L = 2$. $f_2(T)$ has two maximal 2-consecutive subsequences: $(1, 2, 4, 5, 6)$ and $(9, 10, 11)$. Since $K = 4$, we will remove $(9, 10, 11)$ in the g-step. Finally, the output is $\texttt{sim}(T) = (1, 2, 4, 5, 6)$.

It is possible that the simplified sequence $\texttt{sim}(T) = \emptyset$. For example, Let $T = (1, 2, 5, 6)$ and $L = 3$. All the segments will be removed in the f-step and the output is $\emptyset$. We define $\emptyset$ to be not decomposable. We then link *sequence simplification* and *decomposable sequence* in the following lemma:

Lemma 5.5.5. If sequence T is a superset of any decomposable sequence, then $\texttt{sim}(T) \neq \emptyset$.

Proof. It is obvious that $\texttt{sim}(T)$ is a one-to-one function. Given an input sequence T, there is a unique $\texttt{sim}(T)$. Let T_p be a decomposable subset of T and we prove the lemma by showing that $\texttt{sim}(T)$ is a superset of T_p.

Suppose T_p can be decomposed into a set of maximal G-connected subsequences $T_p^1, \ldots, T_p^m$ $(m \geq 1)$. Since T_p is a subset of T, all the T_p^i are also subsets of T. By definition, each T_p^i is L-consecutive. Thus, in the f-step of $\texttt{sim}(T)$, none of T_p^i will be removed. In the g-step, based on Lemma 5.5.4, we know that each T_p^i has a superset in the maximal G-connected subsequences of $f_L(T)$. Since $|T_p^i| \geq K$, none of T_p^i will be removed in the g-step. Therefore, all the T_p^i will be retained after the simplification process and $\texttt{sim}(T) \neq \emptyset$. □

With Lemma 5.5.5, we are ready to define the *monotonicity* concept based on the simplified sequences to facilitate the pruning in the Apriori algorithm.

Theorem 5.5.6 (Monotonicity)**.** Given a candidate pattern $P = \{O : T\}$, if $\texttt{sim}(P.T) = \emptyset$, then any pattern candidate P' with $P.O \subseteq P'.O$ can be pruned.

Proof. We prove by contradiction. Suppose there exists a valid pattern P_2 such that $P_2.O \supseteq P.O$. It is obvious that $P_2.T \subseteq P.T$. Based on Definition 2, the following conditions hold: (1) $P_2.T$ is G-connected. (2) $|P_2.T| \geq K$ and (3) $P_2.T$ is L-consecutive. Note that the entire $P_2.T$ is G-connected. Thus, $P_2.T$ itself is the only maximal G-connected subsequence. Based on conditions (1),(2),(3) and Definition 6, $P_2.T$ is decomposable. Then, based on Lemma 5.5.5, we know $\texttt{sim}(T) \neq \emptyset$ because $P_2.T \subseteq P.T$ and $P_2.T$ is decomposable. This leads to a contradiction with $\texttt{sim}(P.T) = \emptyset$. □

5.5.2.2 Apriori Enumerator

We design an Apriori based enumeration algorithm to efficiently discover all the valid patterns in a star partition. The principle of the Apriori algorithm is to construct a lattice structure and enumerate all the possible candidate sets in a bottom-up manner. Its merit lies in the monotonic property such that if a candidate set is not valid, then all its supersets can be pruned. Thus, it works well in practice in spite of the exponential search space.

Our customized Apriori Enumerator is presented in Algorithm 12. Initially, the edges (pairs of objects) in the star constitute the bottom level (Lines 2-6) and invalid candidates are excluded (Line 4). An indicator *level* is used to control the result size for candidate joins. During each iteration (Lines 8-28), only candidates with object size equals to *level* are generated (Line 10). When two candidates c_1 and c_2 are joined, the new candidate becomes $c' = \langle c_1.O \cup c_2.O, c_1.T \cap c_2.T \rangle$ (Lines 11). To check the validity of the candidate, we calculate $\texttt{sim}(c'.T)$. If its simplified sequence is empty, c' is excluded from the next level (Line 12). This ensures that all the candidates with $P.O \supseteq c'.O$ are pruned. If a candidate cannot generate any new candidate, the it is directly reported (Lines 16-20). To further improve the performance, we adopt the idea of *forward closure* [**?**, **?**] and aggressively check if the union of all the current

candidates form a valid pattern (Lines 22-26). If yes, we can early terminate the algorithm and output the results.

Algorithm 12 Apriori Enumerator

Input: Sr_s

1: $C \leftarrow \emptyset$
2: **for all** edges $c = \langle o_i \cup o_j, T_{o_i} \cap T_{o_j} \rangle$ in Sr_s **do**
3: **if** $\texttt{sim}(T_{o_i} \cap T_{o_j}) \neq \emptyset$ **then**
4: $C \leftarrow C \cup \{c\}$
5: **end if**
6: **end for**
7: level $\leftarrow 2$
8: **while** $C \neq \emptyset$ **do**
9: **for all** $c_1 \in C$ **do**
10: **for all** $c_2 \in C$ and $|c_2.O \cup c_2.O|$ = level **do**
11: $c' \leftarrow \langle c_1.O \cup c_2.O : (c_1.T \cap c_2.T) \rangle$
12: **if** $\texttt{sim}(c'.T) \neq \emptyset$ **then**
13: $C' \leftarrow C' \cup \{c'\}$
14: **end if**
15: **end for**
16: **if** no c' is added to C' **then**
17: **if** c_1 is a valid pattern **then**
18: output c_1
19: **end if**
20: **end if**
21: **end for**
22: $O_u \leftarrow$ union of $c.O$ in C
23: $T_u \leftarrow$ intersection of $c.T$ in C
24: **if** $\langle O_u, T_u \rangle$ is a valid pattern **then**
25: output $\langle O_u, T_u \rangle$, **break**
26: **end if**
27: $C \leftarrow C'; C' \leftarrow \emptyset;$ level $\leftarrow$ level $+ 1$
28: **end while**
29: output C

Example 5.5.5. As shown in Figure 5.5 (c), in the bottom level of the lattice structure, candidate $\langle 3, 6 : 3 \rangle$ is pruned because its simplified sequence is empty. Thus, all the object sets containing $\langle 3, 6 \rangle$ can be pruned. The remaining two candidates (i.e., $\langle 3, 4 : 1, 2, 3 \rangle$ and $\langle 3, 5 : 2, 3 \rangle$) derive a new $\langle 3, 4, 5 : 2, 3 \rangle$ which is valid. By the forward closure checking, the algorithm can terminate and output $\langle 3, 4, 5 : 2, 3 \rangle$ as

the final pattern.

5.5.3 Put Everything Together

We summarize the workflow of SPARE in Figure 5.5 as follows. After the parallel clustering in each snapshot, for ease of presentation, we used an aggregated graph G_A to capture the clustering relationship. However, in the implementation of the map phase, there is no need to create G_A in advance. Instead, we simply need to emit the edges within a star to the same reducer. Before sending stars to reducers, we use a simple best-fit strategy to calculate the star allocation. Under the best-fit strategy, the most costly unallocated star is assigned to the most lightly loaded reducers, where we use the edges in the star as a cost estimation. Each reducer is an Apriori Enumerator. When receiving a star Sr_i, the reducer creates initial candidate patterns. Specifically, for each $o \in Sr_i$, a candidate pattern $\langle o, i : e(o,i)\rangle$ is created. Then it enumerates all the valid patterns from the candidate patterns. The pseudocode of SPARE is presented in Algorithm 13.

Algorithm 13 Star Partitioning and ApRiori Enumerator

Input: list of $\langle t, S_t\rangle$ pairs
1: —*Map phase*—
2: **for all** $C \in S_t$ **do**
3: **for all** $o_1 \in C, o_2 \in C, o_1 < o_2$ **do**
4: emit a $\langle o_1, o_2, \{t\}\rangle$ triplet
5: **end for**
6: **end for**
7: —*Partition and Shuffle phase*—
8: **for all** $\langle o_1, o_2, \{t\}\rangle$ triplets **do**
9: group-by o_1, emit $\langle o_1, Sr_{o_1}\rangle$
10: **end for**
11: —*Reduce phase*—
12: **for all** $\langle o, Sr_o\rangle$ **do**
13: call Apriori Enumerator for star Sr_o
14: **end for**

Compared with TRPM, the SPARE framework does not rely on snapshot replication to guarantee correctness. In addition, we can show that the patterns derived

from a star partition are unique and there would not be duplicate patterns mined from different star partitions.

Theorem 5.5.7 (Pattern Uniqueness)**.** Let Sr_i and Sr_j $(i \neq j)$ be two star partitions. Let P_i (resp. P_j) be the patterns discovered from Sr_i (resp. Sr_j). Then, $\forall p_i \in P_i, \forall p_j \in P_j$, we have $p_i.O \neq p_j.O$.

Proof. We prove by contradiction. Suppose there exist $p_i \in P_i$ and $p_j \in P_j$ with the same object set. Note that the center vertex of the star is associated with the minimum id. Let o_i and o_j be the center vertexes of the two partitions and we have $o_i = o_j$. However, P_i and P_j are different stars, meaning their center vertexes are different (i.e., $o_i \neq o_j$), leading to a contradiction. □

Theorem 5.5.7 implies that no mining efforts are wasted in discovering redundant patterns in the SPARE framework, which is superior to the TRPM baseline. Finally, we show the correctness of the SPARE framework.

Theorem 5.5.8. The SPARE framework guarantees completeness and soundness.

Proof. See Appendix A.2.2. □

5.6 Experimental Study

In this section, we evaluate the efficiency and scalability of our proposed parallel GCMP detectors on real trajectory datasets. All the experiments are carried out in a cluster with 12 nodes, each equipped with four quad-core 2.2GHz Intel processors, 32GB memory and gigabit Ethernet.

Environment Setup: We use Yarn[2] to manage our cluster. We pick one machine as Yarn's master node, and for each of the remaining machines, we reserve one core and 2GB memory for Yarn processes. We deploy our GCMP detector on Apache

[2]`http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`

Spark 1.5.5 [**?**] with the remaining 11 nodes as the computing nodes. To fully utilize the computing resources, we configure each node to run five executors, each taking three cores and 5GB memory. In Spark, one of the 55 executors is taken as the Application Master for coordination, therefore our setting results in 54 executors. All our implementations as well as cluster setups are open-sourced in Github[3].

Datasets: We use three real trajectory datasets in different application scenarios:

- Shopping[4]: The dataset contains trajectories of visitors in the ATC shopping center in Osaka. To better capture the indoor activities, the visitor locations are sampled every half second, resulting in $13,183$ long trajectories.

- GeoLife [5]: The dataset essentially keeps all the travel records of 182 users for a period of over three years, including multiple kinds of transportation modes (walking, driving and taking public transportation). For each user, the GPS information is collected periodically and 91 percent of the trajectories are sampled every 1 to 5 seconds.

- Taxi [6]: The dataset tracks the trajectories of $15,054$ taxies in Singapore. For each taxi, the GPS information are continually collected for one entire month with the sampling rate around 30 seconds.

Preprocessing: We replace timestamps with global sequences (starting from 1) for each dataset. We set a fixed sampling rate for each dataset (i.e., GeoLife = 5 seconds, Shopping=0.5 seconds, Taxi = 30 seconds) and use linear interpolation to fill missing values. For the clustering method, we use DBSCAN [11] and customize its two parameters ϵ (proximity threshold) and $minPt$ (the minimum number of points required to form a dense region). We set $\epsilon = 5$, $minPt = 10$ for GeoLife

[3] `https://github.com/fanqi1909/TrajectoryMining/`
[4] `http://www.irc.atr.jp/crest2010_HRI/ATC_dataset/`
[5] `http://research.microsoft.com/en-us/projects/geolife/`
[6] Taxi is our proprietary dataset

and Shopping datasets; and $\epsilon = 20$, $minPt = 10$ for Taxi dataset. Note that other clustering methods or settings can also be applied. After preprocessing, the statistics of the three datasets are listed in Table 5.5.

Attributes	**Shopping**	**GeoLife**	**SingTaxi**
# objects	13,183	18,670	15,054
# data points	41,052,242	54,594,696	296,075,837
# snapshots	16,931	10,699	44,364
# clusters	211,403	206,704	536,804
avg. cluster size	171	223	484

Table 5.5: Statistics of datasets.

Parameters: To systematically study the performance of our algorithms, we conduct experiments on various parameter settings. The parameters to be evaluated are listed in Table 5.6, with default settings in bold.

Variables	**Meaning**	**Values**
M	min size of object set	5, 10, **15**, 20, 25
K	min duration	120, 150, **180**, 210, 240
L	min local duration	10, 20, **30**, 40,50
G	max gap	10, 15, **20**, 25, 30
O_r	ratio of objects	20%,40%, 60%, 80%, **100%**
T_r	ratio of snapshots	20%,40%, 60%, 80%, **100%**
N	number of machines	1, 3, 5, 7, 9, **11**

Table 5.6: Variables and their default values.

5.6.1 Performance Evaluation

Varying M: Figures 5.7 (a),(g),(m) present the performance with increasing M. The SPARE framework demonstrates a clear superiority over the TRPM framework, with a performance gain by a factor of 2.7 times in Shopping, 3.1 times in GeoLife and 7 times in Taxi. As M increases, the running time of both frameworks slightly improve because the number of clusters in each snapshot drops, generating fewer valid candidates.

Varying K: The performance with increasing K is shown in Figure 5.7 (b),(h),(n). SPARE tends to run faster, whereas the performance of TRPM degrades dramatically. This is caused by the *sequence simplification* procedure in SPARE, which can prune many candidates with large K. However, the line sweep algorithm in TRPM does not utilize such property for pruning. It takes longer time because more replicated data has to be handled in each partition.

Varying L: Figures 5.7 (c),(i),(o) present the performances with increasing L. When $L = 10$, SPARE can outperform TRPM by around 10 times. We also observe that there is a significant performance improvement for TPRM when L increases from 10 to 20 and later the running time drops smoothly. This is because η is proportional to $O(K * G/L + L)$. When L is small (i.e., from 10 to 20), η decreases drastically. As L increases, η varies less significantly.

Varying G: Figures 5.7 (d),(j),(p) present the performances with increasing G. TRPM is rather sensitive to G. When G is relaxed to larger values, more valid patterns would be generated. TPRM has to set a higher replication factor and its running time degrades drastically when G increases from 20 to 30. In contrast, with much more effective pruning strategy, our SPARE scales well with G. Particularly, SPARE is 14 times faster than TRPM when $G = 20$ in GeoLife dataset.

Varying O_r: Figures 5.7 (e),(k),(q) present the performances with increasing number of moving objects. Both TRPM and SPARE take longer time to find patterns in a larger database. We can see that the performance gap between SPARE and TRPM is widened as more objects are involved, which shows SPARE is more scalable.

Varying T_r: Figures 5.7 (f),(l),(r) present the performances with increasing number of snapshots. As T_r increases, SPARE scales much better than TRPM due to its effective pruning in the temporal dimension.

5.6.2 Analysis of SPARE framework

In this part, we extensively evaluate SPARE from three aspects: (1) the advantages brought by the sequence simplification, (2) the effectiveness of load balance, and (3) the scalability with increasing computing resources.

5.6.2.1 Power of sequence simplification

To study the power of sequence simplification, we collect two types of statistics: (1) the number of pairs that are shuffled to the reducers and (2) the number of pairs that are fed to the Apirori Enumerator. The difference between these two values is the number of size-2 candidates pruned by the sequence simplification. The results in Table 5.7 show that the *sequence simplification* is very powerful and eliminates nearly 90 percent of the object pairs, which significantly reduces the overhead of subsequent Apriori enumerator.

Dataset	**Shopping**	**GeoLife**	**Taxi**
Before pruning	878,309	1,134,228	2,210,101
After pruning	76,672	123,410	270,921
Prune ratio	91.2%	89.1%	87.7%

Table 5.7: Pruning power of SPARE.

Dataset	**SPARE-RD**		**SPARE**	
	Straggler	Std. Dev.	Straggler	Std. Dev.
Shopping	295	41	237	21
GeoLife	484	108	341	56
Taxi	681	147	580	96

Table 5.8: Statistics of execution time (in seconds) among executors.

5.6.2.2 Load balance

To study the effect of load balance in the SPARE framework, we use random task allocation (the default setting of Spark) as a baseline, denoted by SPARE-RD, and

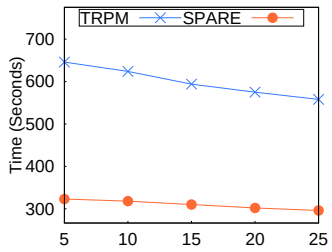
compare it with our best-fit method. In best-fit, the largest unassigned star is allocated to the currently most lightly loaded reducer. Figure 5.8 (a) shows the breakdown of the costs in the map-reduce stages for SPARE and SPARE-RD. We observe that the map and shuffle time of SPARE and SPARE-RD are identical. The difference is that SPARE incurs an additional overhead to generate an allocation plan for load balance (around 4% of the total cost), resulting in significant savings in the reduce stage (around 20% of the total cost). We also report the cost of *straggler*, i.e., the longest job, and the standard deviation (Std. Dev.) for all jobs in Table 5.8, whose results clearly verify the effectiveness of our allocation strategy for load balance.

5.6.2.3 Scalability

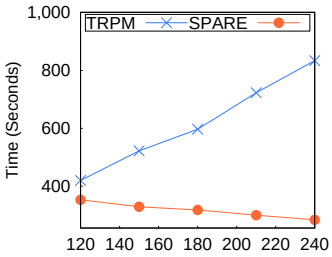
When examining SPARE with increasing computing resources (number of machines), we also compare SPARE with the state-of-the-art solutions for *swarm* and *platoon* in the single-node setting. Since the original *swarm* and *platoon* detectors cannot handle very large-scale datasets, we only use 60% of each dataset for evaluation. To make fair comparisons, we customize two variants of SPARE to mine *swarm*s and *platoon*s, which are denoted as SPARE-S and SPARE-P respectively. The customization is according to the settings in Table 5.3 and the results are reported in Figure 5.9. First, the centralized schemes are not suitable to discover patterns in large-scale trajectory databases. It takes nearly 30 hours to detect *swarm*s and 11 hours to detect *platoon*s in the Taxi dataset in a single machine. In contrast, when utilizing the multi-core environment, SPARE-P achieves 7 times speedup and SPARE-S achieves 10 times speedup. Second, we see that SPARE schemes demonstrate promising scalability in terms of the number of machines available. The running times decrease almost inversely as more machines are used. When all the 11 nodes (162 cores) are available, SPARE-P is upto 65 times and SPARE-S is upto 112 times better than the state-of-the-art centralized schemes.

5.7 Conclusions and Future Work

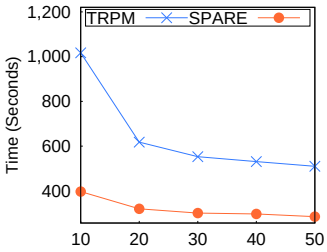
In this paper, we proposed a generalized co-movement pattern to unify those proposed in the past literature. We also devised two types of parallel frameworks in Spark that can scale to support pattern detection in trajectory databases with hundreds of millions of points. The efficiency and scalability were verified by extensive experiments on three real datasets. In the future, we intend to examine co-movement pattern detection in streaming data for real-time monitoring. How to extend the current parallel frameworks to support other types of advanced patterns is also of our interest.



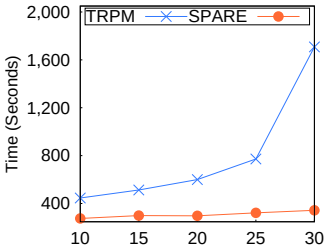
(a) Shopping vary M



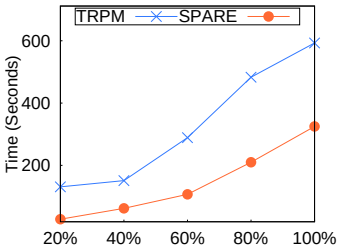
(b) Shopping vary K



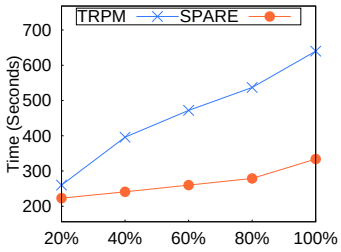
(c) Shopping vary L



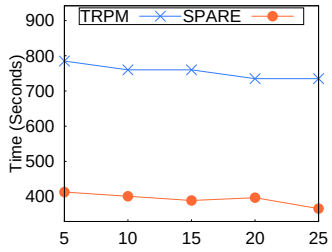
(d) Shopping vary G



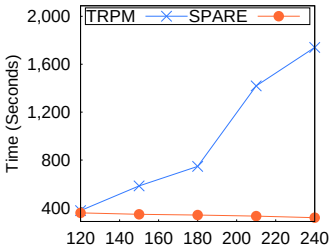
(e) Shopping vary O_r



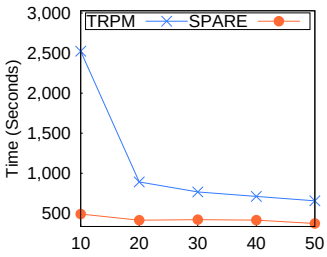
(f) Shopping vary T_r



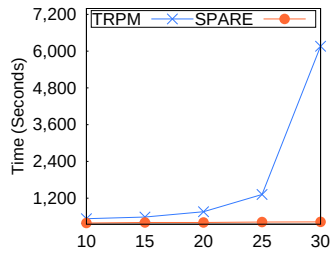
(g) GeoLife vary M



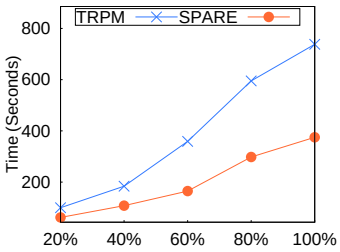
(h) GeoLife vary K



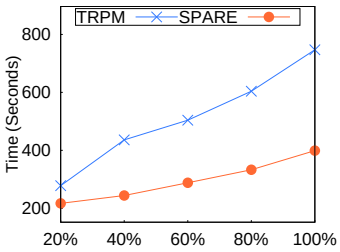
(i) GeoLife vary L



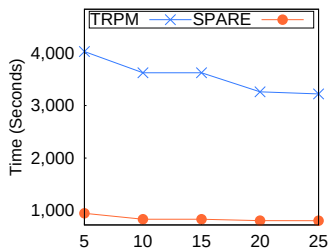
(j) GeoLife vary G



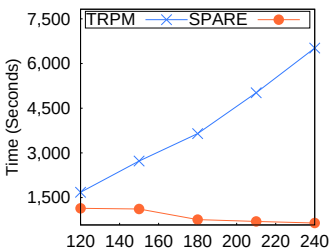
(k) GeoLife vary O_r



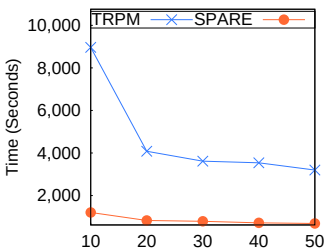
(l) GeoLife vary T_r



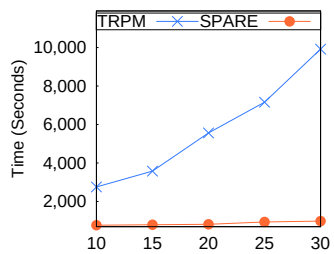
(m) Taxi vary M



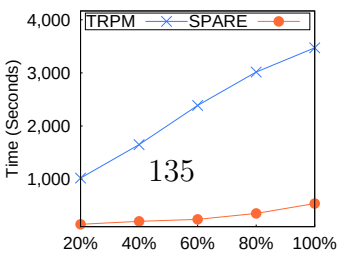
(n) Taxi vary K



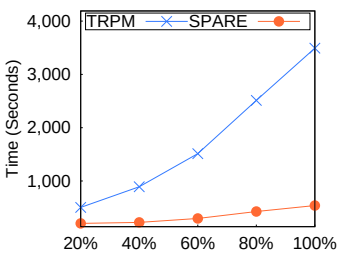
(o) Taxi vary L



(p) Taxi vary G



(q) Taxi vary O_r



(r) Taxi vary T_r

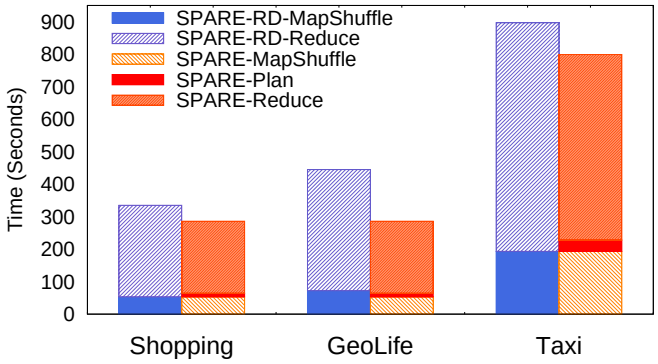
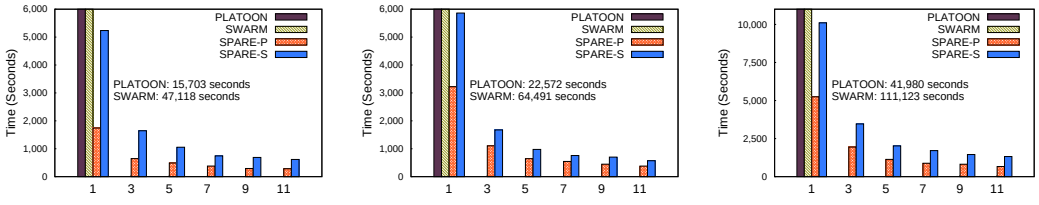


Figure 5.8: Breakdown of cost of SPARE and SPARE-RD.



(a) Shopping vary N (b) GeoLife vary N (c) Taxi vary N

Figure 5.9: Comparisons among TRMP, SPARE, PLATOON and SWARM.

Appendix A

Appendix

A.1 Discussions on Other Aggregate Functions

First, we shall see that supporting *sum* is equivalent to supporting *avg*. A candidate theme which has a rank under *avg* will have the same rank under *sum* as the ranking is derived by comparing all candidates with the same window length. Second, supporting *count* is equivalent to supporting *sum*. By assigning each event with value of either 1 or 0, we can apply the same pruning bounds for *sum* to support *count*. Third, supporting *max* is equivalent to supporting *min*. This is because when *max* is used as the aggregate function, we are more interested to find news themes which have smaller aggregation values. For example, "XXX stock has a maximum of $0.2 price in consecutive 10 days, which is the lowest ever". Then finding the sketches according to *max* can be derived from *min* directly by negating the event values. Therefore, we only provide bounds for *sum* and *min*, which are shown as in Table A.1:

Table A.1: Bounds for other aggregate functions

Aggregate Function	**Subadditivity**
sum	$J_s(w) \leq J_s(w_1) + J_s(w - w_1)$
min	$J_s(w) \leq max(J_s(w_1), J_s(w - w_1))$
Aggregate Function	**Visting-Window Bound**
sum	$J_s(w) = J_s(w-1) + J_s(1)$
min	$J_s(w) = J_s(w/2)$
Aggregate Function	**Unseen-Window Bound**
sum	$M_s(w) = W_s(t,w).\overline{v} + J_s(t-w)$
min	$M_s(w) = max\{W_s(t,w).\overline{v}, J(1)\}$
Aggregate Function	**Online-Window Bound**
sum	$M_s(w) = W_s(t,w).\overline{v} + J_s(t-w)$
min	$M_s(w) = max\{W_s(t,w).\overline{v}, J(1)\}$

A.2 Proofs of Theorems in Chapter 5

A.2.1 Proofs of Theorem 5.5.1 and 5.5.2

Proof. Γ can be formalized using linear algebra: Let G_A be an aggregated graph, with a $n \times n$ adjacent matrix J. Since a vertex order is a permutation of J, the adjacent matrices of any reordered graphs can be represented as PJP^T where $P \in \mathbb{P}$ is a $n \times n$ *permutation matrix* [1]. In star partitioning, we assign each edge $e(i,j)$ in G_A to the lower vertex, then the matrix $B = \text{triu}(PJP^T)$ [2] represents the assignment matrix wrt. P (i.e., $b_{i,j} = 1$ if vertex j is in star Sr_i). Let vector $\vec{b}$ be the *one*[3] vector with size n. Let $\vec{c} = B\vec{b}$, then each c_i denotes the number of edges in star Sr_i. Thus, Γ can be represented as the infinity norm of $B\vec{b}$. Let Γ^* be the minimum Γ among all vertex orderings, that is

$$\Gamma^* = \min_{P \in \mathbb{P}} ||B\vec{b}||_\infty \text{ ,where } ||B\vec{b}||_\infty = \max_{1 \leq j \leq n} (c_j) \tag{A.1}$$

[1] an identity matrix with rows shuffled
[2] triu is the upper triangle part of a matrix
[3] every element in $\vec{b}$ is 1

Let B^* be the assignment matrix wrt. the optimal vertex ordering. Since we have a star for each object, by the degree-sum formula and pigeon-hole theorem, $\Gamma^* = ||B^*\vec{b}||_\infty \geq d/2$. Next, for a vertex ordering P, let $e_{i,j}$ be an entry in PAP^T. Since edges in graph G are independent, then $e_{i,j}$s are independent. Let d_i denote the degree of vertex i, since a vertex ordering does not affect the average degree, then $E[d_i] = E[\Sigma_{1\leq j\leq n} e_{i,j}] = d$. Therefore, entries in B can be written as :

$$b_{i,j} = \begin{cases} e_{i,j}, i > j \\ 0, otherwise \end{cases}$$

There are two observations. First, since $e_{i,j}$s are independent, $b_{i,j}$s are independent. Second, since $i > j$ and $e_{i,j}$s are independent. $E[b_{i,j}] = E[e_{i,j}]E[i > j] = E[e_{i,j}]/2$. As c_i is a sum of n independent 0-1 variables ($b_{i.j}$s). By linearity of expectations, we get: $E[c_i] = E[\Sigma_{1\leq j\leq n} b_{i,j}] = E[\Sigma_{1\leq j\leq n} e_{i,j}]/2 = d/2$. Let $\mu = E[c_i] = d/2$, $t = \sqrt{n \log n}$, by Hoeffding's Inequality, the following holds:

$$Pr(c_i \geq \mu + t) \leq \exp(\frac{-2t^2}{n}) = \exp(-2\log n) = n^{-2}$$

The first step holds since all $b_{i,j}$ are 0-1 variables. Next, the event $(\max_{1\leq j\leq n}(c_j) \geq \mu + t)$ can be viewed as $\cup_{c_i}(c_i \geq \mu + t)$, by Union Bound, the following holds:

$$\begin{aligned} Pr(\Gamma \geq \mu + t) &= Pr(\max_{1\leq j\leq n}(c_j) \geq \mu + t) \\ &= Pr(\cup_{c_i}(c_i \geq \mu + t)) \\ &\leq \Sigma_{1\leq i\leq n} Pr(c_i \geq \mu + t) = n^{-1} = 1/n \end{aligned}$$

Substitute back t and μ, we achieve the following concise form:

$$Pr(\Gamma \geq (d/2 + \sqrt{n \log n})) \leq 1/n$$

This indicates the probability of $(\Gamma - d/2)$ being no greater than $O(\sqrt{n \log n})$ is $(1 - 1/n)$. Since $\Gamma^* \geq d/2$, it follows with probability greater than $(1 - 1/n)$, the $\Gamma - \Gamma^*$ is no greater than $O(\sqrt{n \log n})$. When the aggregated graph is *dense* (i.e., $d \geq \sqrt{12 \log n}$), the Chernoff Bound can be used to derive a tighter bound of $O(\sqrt{\log n})$ following the similar reasoning. □

A.2.2 Proof of Theorem 5.5.8

Proof. For soundness, let P be a pattern enumerated by SPARE. For any two objects $o_1, o_2 \in P.O$, the edge $e(o_1, o_2)$ is a superset of $P.T$. By the definition of star, o_1 o_2 belong to the same cluster at every timestamps in $P.T$. As $P.T$ is a valid sequence, by the definition of GCMP, P is a true pattern. For completeness, let P be a true pattern. Let s be the object with smallest ID in $P.O$. We prove that P must be output by Algorithm 12 form Sr_s. First, based on the definition of star, every object in $P.O$ appears in Sr_s. Since $P.T$ is decomposable, then by Lemma 3 $\forall O' \subseteq O$, the time sequence of O' would not be eliminated by any `sim` operations. Next, we prove at every iteration $level \leq |P.O|$, $P.O \subset O_u$, where O_u is the forward closure. We prove by induction. When $level = 2$, it obviously holds. If $P.O \subset O_u$ at *level* i, then any subsets of $P.O$ with size i are in the candidate set. In *level* $i+1$, these subsets are able to grow to a bigger subset (in last iteration, they grow to $P.O$). This suggests that no subsets are removed by Lines 16-29. Then, $P.O \subset U_{i+1}$ holds. In summary, $P.O$ does not pruned by simplification, monotonicity and forward closure, therefore P must be returned by SPARE. □

Bibliography

[1] Rakesh Agrawal, Sreenivas Gollapudi, Alan Halverson, and Samuel Ieong. Diversifying search results. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pages 5–14. ACM, 2009.

[2] Htoo Htet Aung and Kian-Lee Tan. Discovery of evolving convoys. In *International Conference on Scientific and Statistical Database Management*, pages 196–213. Springer, 2010.

[3] Srikanth Bellamkonda, Hua-Gang Li, Unmesh Jagtap, Yali Zhu, Vince Liang, and Thierry Cruanes. Adaptive and big data scale parallel execution in oracle. *Proceedings of the VLDB Endowment*, 6(11):1102–1113, 2013.

[4] Allan Borodin, Hyun Chul Lee, and Yuli Ye. Max-sum diversification, monotone submodular functions and dynamic updates. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 155–166. ACM, 2012.

[5] Yu Cao, Chee-Yong Chan, Jie Li, and Kian-Lee Tan. Optimization of analytic window functions. *Proceedings of the VLDB Endowment*, 5(11):1244–1255, 2012.

[6] Chen Chen, Xifeng Yan, Feida Zhu, Jiawei Han, and S Yu Philip. Graph olap: Towards online analytical processing on graphs. In *2008 Eighth IEEE International Conference on Data Mining*, pages 103–112. IEEE, 2008.

[7] Lisi Chen and Gao Cong. Diversity-aware top-k publish/subscribe for text stream. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 347–362. ACM, 2015.

[8] James Cheng, Zechao Shang, Hong Cheng, Haixun Wang, and Jeffrey Xu Yu. K-reach: who is in your small world. *Proceedings of the VLDB Endowment*, 5(11):1292–1303, 2012.

[9] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973.

[10] Marina Drosou and Evaggelia Pitoura. Diverse set selection over dynamic data. *IEEE Transactions on Knowledge and Data Engineering*, 26(5):1102–1116, 2014.

[11] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[12] Ryota Jinno, Kazuhiro Seki, and Kuniaki Uehara. Parallel distributed trajectory pattern mining using mapreduce. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 269–273. IEEE, 2012.

[13] Panos Kalnis, Nikos Mamoulis, and Spiridon Bakiras. On discovering moving clusters in spatio-temporal data. In *International Symposium on Spatial and Temporal Databases*, pages 364–381. Springer, 2005.

[14] Patrick Laube, Marc van Kreveld, and Stephan Imfeld. Finding remodetecting relative motion patterns in geospatial lifelines. In *Developments in spatial data handling*, pages 201–215. Springer, 2005.

[15] Srivatsan Laxman, PS Sastry, and KP Unnikrishnan. A fast algorithm for finding frequent episodes in event streams. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 410–419. ACM, 2007.

[16] Xiaohui Li, Vaida Ceikute, Christian S Jensen, and Kian-Lee Tan. Effective online group discovery in trajectory databases. *IEEE Transactions on Knowledge and Data Engineering*, 25(12):2752–2766, 2013.

[17] Yuxuan Li, James Bailey, and Lars Kulik. Efficient mining of platoon patterns in trajectory databases. *Data & Knowledge Engineering*, 100:167–187, 2015.

[18] Zhenhui Li, Bolin Ding, Jiawei Han, and Roland Kays. Swarm: Mining relaxed temporal moving object clusters. *Proceedings of the VLDB Endowment*, 3(1-2):723–734, 2010.

[19] Heikki Mannila, Hannu Toivonen, and A Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data mining and knowledge discovery*, 1(3):259–289, 1997.

[20] Jayanta Mondal and Amol Deshpande. Eagr: Supporting continuous ego-centric aggregate queries over large dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1335–1346. ACM, 2014.

[21] Afroza Sultana, Naeemul Hassan, Chengkai Li, Jun Yang, and Cong Yu. Incremental discovery of prominent situational facts. In *2014 IEEE 30th International Conference on Data Engineering*, pages 112–123. IEEE, 2014.

[22] Nikolaj Tatti and Boris Cule. Mining closed strict episodes. *Data Mining and Knowledge Discovery*, 25(1):34–66, 2012.

[23] Yuanyuan Tian, Richard A Hankins, and Jignesh M Patel. Efficient aggregation for graph summarization. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 567–580. ACM, 2008.

[24] Yida Wang, Ee-Peng Lim, and San-Yih Hwang. Efficient mining of group patterns from user movement data. *Data & Knowledge Engineering*, 57(3):240–282, 2006.

[25] Zhengkui Wang, Qi Fan, Huiju Wang, Kian-Lee Tan, Divyakant Agrawal, and Amr El Abbadi. Pagrol: parallel graph olap over large-scale attributed graphs. In *2014 IEEE 30th International Conference on Data Engineering*, pages 496–507. IEEE, 2014.

[26] You Wu, Pankaj K Agarwal, Chengkai Li, Jun Yang, and Cong Yu. On one of the few objects. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1487–1495. ACM, 2012.

[27] Xifeng Yan, Bin He, Feida Zhu, and Jiawei Han. Top-k aggregation queries over large networks. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 377–380. IEEE, 2010.

[28] Jeffrey Xu Yu and Jiefeng Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*, pages 181–215. Springer, 2010.

[29] Fred Zemke. What. s new in sql: 2011. *ACM SIGMOD Record*, 41(1):67–73, 2012.

[30] Gensheng Zhang, Xiao Jiang, Ping Luo, Min Wang, and Chengkai Li. Discovering general prominent streaks in sequence data. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 8(2):9, 2014.

[31] Peixiang Zhao, Xiaolei Li, Dong Xin, and Jiawei Han. Graph cube: on warehousing and olap multidimensional networks. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 853–864. ACM, 2011.

[32] Kai Zheng, Yu Zheng, Nicholas Jing Yuan, and Shuo Shang. On discovery of gathering patterns from trajectories. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 242–253. IEEE, 2013.

[33] Wenzhi Zhou, Hongyan Liu, and Hong Cheng. Mining closed episodes from event sequences efficiently. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 310–318. Springer, 2010.