Towards Efficient Processing of Neighborhood Analytics in Emerging Applications

QI FAN

(B. Comp. 1st Class Hons. NUS)

NUS Graduate School of Integrative Science and Engineering

National University of Singapore

A thesis submitted for the degree of

Doctor of Philosophy

Jan 2017

Abstract

With the increasing variety and volume of the data produced by today's applications, the adoption of effective analytics becomes remarkably demanding. Window functions, being an important part of SQL family, have proven numerous successes in relational analytics. A window function assigns each tuple a set of related tuples, on which analytics can be applied. However, the window function defines the related tuples based on sorting which limits its usage in the domains where sorting may not be meaningful. In this thesis, we generalize the concept the window function to *neighborhood analytics* which eliminates the stringent sorting requirement. We propose three domain-specific queries tailored for emerging applications on the basis of two simple neighborhood functions. Then, we study how to process these queries efficiently given today's data scale.

In particular, we first propose the *Graph Window Query* (GWQ) in the graph domain. GWQ computes aggregation for each vertex on its graph window. We formally define two instances of such graph windows: k-hop window and topological window. Then, we develop the Dense Block Index (DBIndex) and Inheritance Index (I-Index) to facilitate efficient processing of both queries. These indexes effectively compress the windows of each vertex and reuse the shared components during query processing, which achieve both space and query efficiency.

Second, we propose the k-Sketch query on the sequence data to summarize a subject's history. k-Sketch query utilizes the novel *ranked-streak* which

is formed by a nested neighborhood function. Specifically, a streak is first constructed by grouping temporally nearby events. Subsequently, streaks with the same length are compared to generate their ranks. A k-Sketch query then selects k ranked-streaks which best summarize a subject's history. We study the k-Sketch query processing in both offline and online scenarios. In particular, we design two nontrivial streak-level pruning techniques and a $(1-1/e)$-approximate algorithm to achieve efficient processing in offline. Then we design a $1/8$-approximate algorithm for the online sketch maintenance.

Third, we propose the General Co-Movement Pattern (GCMP) query for trajectory databases. A GCMP is defined as the temporal invariant portion of an object's spatial neighborhood. Our GCMP is versatile to express other moving patterns defined in the literature. Meanwhile, GCMP is also able to eliminate the so-called loose-connection anomaly which has not been addressed before. We design two parallel frameworks for supporting scalable GCMP detection. First, we propose a baseline method named *Temporal Replication and Parallel Mining* (TRPM) which partitions trajectories via replication of object locations and mines GCMPs from each partition in parallel. Then, we design an advanced method named *Star Partition and ApRiori Enumerator* (SPARE) to resolve the limitations of TRPM. We adopt three novel techniques in SPARE to achieve load balance while minimizing data replications. To the best of our knowledge, this is the first work which detects co-moving patterns from trajectories with hundreds of millions of data points.

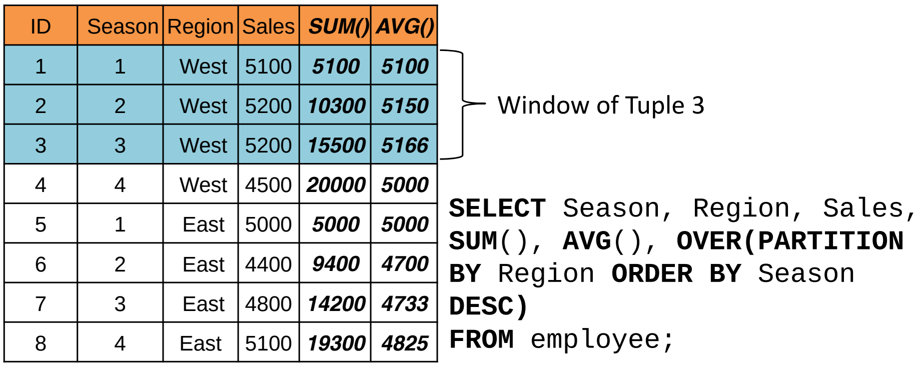
Chapter 1

Introduction

With the maturity of database technologies, many applications today collect and store data from all domains at unprecedented scale. For example, billions of social network users and their activities are collected in the form of *graphs*; thousands of sensor ticks are collected every second in the form of *time series*; hundreds of millions of spatiotemporal points are collected as *trajectories*. Flooded by the tremendous amount of data, it is becoming increasingly critical to efficiently conduct analytics to discover useful insights. However, traditional SQL analytics, which comprises primary operations (e.g., partition, sorting and aggregation), is unable to cope with emerging analytics such as graph traversal and pattern detection. In practice, expressing these domain-specific analytics using SQL queries often involves complex joins which are hard to optimize. In this thesis, we introduce the concept of *neighborhood analytics* which originates from the recognized SQL window functions. We study the usage of *neighborhood analytics* in emerging applications from different data domains and design efficient algorithms to harness today's big data.

1.1 Neighborhood Analytics

Neighborhood analytics aims to provide summaries of each object over its vicinity. In contrast to aggregating the entire collection of data as a whole, neighborhood analytic provides a personalized view for each object from its own perspective. Neighborhood data analytics originates from the window function defined in SQL which is illustrated in Figure 1.1.



ID	Season	Region	Sales	*SUM()*	*AVG()*
1	1	West	5100	***5100***	***5100***
2	2	West	5200	***10300***	***5150***
3	3	West	5200	***15500***	***5166***
4	4	West	4500	***20000***	***5000***
5	1	East	5000	***5000***	***5000***
6	2	East	4400	***9400***	***4700***
7	3	East	4800	***14200***	***4733***
8	4	East	5100	***19300***	***4825***

Window of Tuple 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 `avg` of sales. The window of tuple 3 is highlighted.

As shown in the figure, the sales report contains six columns: “ID”, “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 another tuple o_j if o_i and o_j are in the same “region” and the “season” of o_j is prior to the season of o_i. The window of tuple-3 is highlighted. Apart from this example, there are also many other usages of the window functions in the relational context [14]. Being aware of the success of the window functions, SQL 11 [72] standard incorporates “`LEAD`” and “`LAG`” keywords to offer fine-grained specifications on a tuple’s window.

Despite the usefulness, there are few works reporting the usage of window functions in data domains such as graphs, sequence data and trajectories. This may be due to the requirement of *sorting* in the window functions. For example, in Figure 1.1, objects need to be sorted according to “Season”, and then the window of each object is implicitly formed based on the sorted order. However, in domains like graphs and time series data, sorting may be ambiguous and even undefined.

To broaden the usages of the window functions, we propose the *neighborhood analytics* in a more general context. Given a set of objects (such as tuples in relational tables, vertexes in graphs, moving objects in trajectories), the neighborhood analytics is a composite function ($\mathcal{F} \circ \mathcal{N}$) applied on every object. Here, $\mathcal{N}$ is the *neighborhood function*, which contains the related objects (i.e., vicinity) of an object; $\mathcal{F}$ is the *analytic function*, which could be aggregation, ranking, pattern matching etc. Apparently, the SQL 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 $\mathcal{F} = \texttt{avg}$. By relaxing the sorting constraint, neighborhood analytics gains an enriched semantic and can be applied on many other data domains.

1.2 Thesis Scope

In this thesis, we explore the neighborhood analytics in emerging applications from three prevalent data domains, namely **attributed graph**, **sequence data** and **trajectory**. To provide useful analytics, we define the following two intuitive instances of the neighborhood function:

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 (i.e., $=, <, >, \leq, \neq, \geq$).

In spite of the simplicity of these two neighborhood functions, they can weave many useful analytics as we shall see in the remaining part of the thesis.

1.3 Thesis Contributions

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

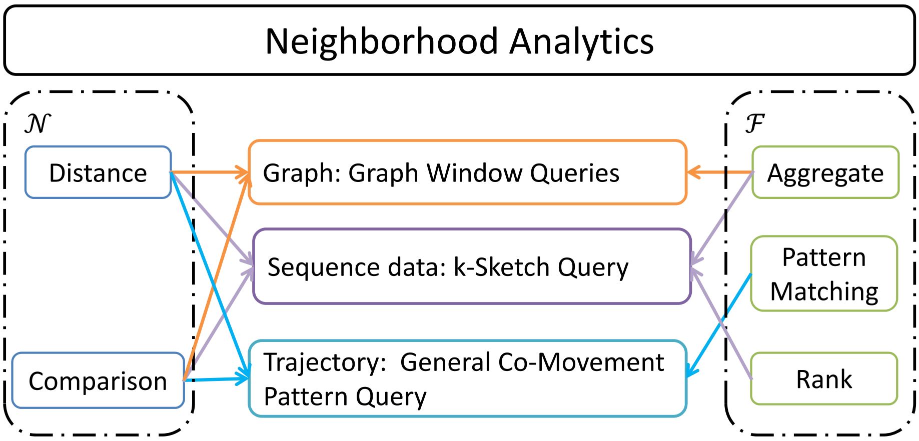

Neighborhood Analytics
$\mathcal{N}$
Distance
Comparison
Graph: Graph Window Queries
Sequence data: k-Sketch Query
Trajectory: General Co-Movement Pattern Query
$\mathcal{F}$
Aggregate
Pattern Matching
Rank

Figure 1.2: The roadmap of this thesis. There are three major contributions as highlighted in the center. Each contribution is an application based on neighborhood analytics with $\mathcal{N}$ and $\mathcal{F}$ as indicated by arrows.

In a nutshell, we propose three neighborhood based queries in respective data domains. In *graph*, we define the Graph Window Query which summarizes the vicinity of each vertex. The query utilizes both distance neighborhood and comparison neighborhood to facilitate both general graphs and direct acyclic graphs. In *sequence data*, we propose a k-Sketch Query to summarize a subject's history. The k-Sketch query builds on a nested *distance* and *comparison* neighborhood based pattern called *rank-aware streak*. In *trajectory*, we propose a General Co-movement Pattern (GCMP) query to discover co-moving behaviors among moving objects. The GCMP query leverages the neighborhood notion to unify existing co-moving patterns in the literature. In the following parts of this section, we present our contributions in detail.

1.3.1 Graph Window Queries

The first piece of the thesis deals with neighborhood analytics 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:

***k*-hop Window:** The k-hop neighbors of a vertex form its k-hop window. Since the k-hop neighbors are the most structurally relevant vertexes to the vertex, analytics on the information from the k-hop window would be beneficial. Typical analytic queries include summarizing the related connections' distribution among different companies, and computing age distribution of the related friends can be useful.

Topological Window: The topological neighbors are defined in the context of Directed Acyclic Graph (DAG). In DAGs, topological neighbors are composed of all the ascendant vertexes of a vertex. The topological neighbors represent the most influential vertexes of a given vertex. Since DAGs are often found in biological networks, topological window would be helpful to analyze the statistics of molecules of each biological protein's pathway.

The two *windows* shown in the above 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 point $a_v \in A$ 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 how Graph Window Query processing can be efficiently supported. 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 the index construction time as compared to the state-of-the-art competitor and up to 10 times speedup in query processing.

1.3.2 k-Sketch Query on Sequence Data

The second piece of this thesis explores the neighborhood analytics on sequence data. As part of the sequence data analysis, summarizing a subject's history with sensational patterns is an important and revenue-generating task in a plethora of applications such as computational journalism [20, 73], automatic fact checking [32, 60], and perturbation analysis [61]. An outstanding example of such patterns is the *streak* [73], which is commonly found in stock and sports reports. For instance:

1. [STOCK]:"Apple Inc. has an *average* price of USD 115.5 in the *last week*"

2. [SPORTS]: "Kobe has scored *at least* 60 points in *three straight games*"

In general, a streak is constructed from two concepts: an *aggregate function* (e.g., `avg`, `min`) applied on *consecutive events* (e.g., seven days, three games). However, the streak itself does not embed the strikingness information, which is limited to represent a sensational pattern. For example, *Streak 2* would not be striking if all NBA players were able to score over 60 points. On the contrary, knowing that most NBA players only score 20 points in a game, *Streak 2* is indeed quite striking. Therefore, the strikingness of a streak should be measured by comparing with other streaks. Based on this observation, we propose a rank-aware pattern named *ranked-streak* which measures the strikingness of a streak by comparing among all streaks under the same condition (i.e., streak length).

Technically, the ranked-streak can be viewed as a joint neighborhood function. Let $t_s(e)$ be the sequence number of the event e in the history of subject s. Then the ranked-streaks of length L are generated using neighborhood functions in a two-step manner:

1. a *distance neighborhood* $\mathcal{N}_1(s, e, L) = \{e_j | t_s(e) - t_s(e_j) \leq L\}$ groups a consecutive L events for each event in the history of subject s. Let $\overline{v}$ be the aggregate value associated with $\mathcal{N}_1$, then the output of this step is a set of *streak*s of the form $sk = \langle s, L, t, \overline{v} \rangle$.

2. a *comparison neighborhood* $\mathcal{N}_2(sk) = \{sk_i | sk.L = sk_i.L \wedge sk_i.\overline{v} \geq sk.\overline{v}\}$ ranks a subject's streak among all other streaks with the same length. Note that the rank information can be simply calculated from a `count` function. The result of this step is a tuple $\langle s, L, t, \overline{v}, r \rangle$, where r is the *rank*.

As the ranked-streak contains the relative position of the streak among its cohort, it provides a quantitative measure of the strikingness. For example, the rank-aware version of *Streak 2* would be "Kobe has scored at least 60 points in three straight games, which is best in the league". This clearly suggests that *Streak 2* is striking.

On the basis of the ranked-streaks, we study the problem of effectively summarizing a subject history. We notice that, for a subject with n events, there are $O\binom{n}{2}$

streaks. In real life, “Kobe” has played $1,000$ games which may produce near half million streaks. Such a large number of streaks is too overwhelming to represent a subject’s history. Hence, we are motivated to propose a k-Sketch query that selects k ranked-streaks which best represent a subject’s history. To find the qualified streaks, we design a novel scoring which considers both the events covered of the streaks and the ranks of the streaks.

In this thesis, we extensively study the technical issues in processing k-Sketch queries in both online and offline scenarios. In the offline scenario, we design two pruning techniques which largely reduce the streaks enumerated in generating ranked-streaks. Then, we adopt a $(1 - 1/e)$-approximate sketch selection algorithm by utilizing the submodularity of the k-Sketch query. In the online scenario, we design an *online-streak bound* to avoid evaluating many unnecessary streaks. Furthermore, we propose a 1/8-approximate algorithm to facilitate efficient sketch maintenance. In the experimental study, we compare our solutions with baselines using four real datasets, and the results demonstrate the efficiency and effectiveness of our proposed algorithms: the running time achieves up to 500 times speedup as compared to the baseline and the quality of the detected sketch is endorsed by the anonymous users from Amazon Mechanical Turk[1].

1.3.3 Co-Movement Pattern Query on Trajectory Data

The third piece of the thesis studies the neighborhood analytics in the trajectory domain. In trajectory analysis, an important mining task is to discover traveling patterns among moving objects. A traveling pattern is often determined by the spatial neighborhoods of moving objects. One of the prominent examples is the *co-movement pattern* [41,76]. A co-movement pattern refers to a group of moving objects traveling together for a certain period of time. We observe that the co-movement pattern can

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

be concisely represented using two neighborhood functions in spatial and temporal domains as follows:

(1) In spatial domain, let $o(t)$ be the spatial location of object o at time t. The co-moving objects of an object o can be determined by a *distance neighborhood* $\mathcal{N}_1$. For example, flock [28] and group [63] patterns use the *disk-based* clustering, which is equivalent to $\mathcal{N}_1(o,t) = \{o_j | \texttt{dist}(o(t), o_j(t)) < r\}$. Convoy [34], swarm [44] and platoon [43] patterns use the *density-based* clustering, which is equivalent to $\mathcal{N}_1(o,t) = \{o_j | \texttt{dist}(o_j(t), o_k(t)) \leq \epsilon \wedge o_k \in \mathcal{N}_1(o,t)\}$.

(2) In temporal domain, objects that co-move with o for a duration T can be determined by a *comparison neighborhood* $\mathcal{N}_2$: $\mathcal{N}_2(o,T) = \{o_j | \forall t \in T, o_j \in \mathcal{N}_1(o,t)\}$.

A pattern is called significant if the group size exceeds M (i.e., $|\mathcal{N}_2(\cdot)| \geq M$ and the length of duration exceeds K (i.e., $T \geq K$). Rooted from the basic movement definition and driven by different mining applications, there are several instances of co-movement patterns that have been developed with more advanced constraints, namely *flock* [28], *convoy* [34], *swarm* [44], *group* [63] and *platoon* [43]. However, these solutions are tailored for each individual pattern and it is cumbersome to deploy and optimize each of the algorithms in real applications. Therefore, there calls for a general framework which provides versatile and efficient support on these pattern discoveries.

Towards this goal, we propose a *General Co-Movement Pattern* (GCMP) query to capture all existing co-movement patterns in one shot. 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., $\mathcal{N}_2$). We relax the parameter settings on the co-moving duration (i.e., T) and by tuning different parameters (as explained in later sections), GCMP query is able to detect any of the existing patterns.

In the technical aspect, we study how to efficiently process GCMP query on the modern parallel processing platform (i.e., Apache Spark) to gain scalability over large-scale trajectories. In particular, we propose two parallel frameworks: (1) TRPM, which partitions trajectories by replicating snapshots in the temporal domain. Within each partition, 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 deploy the two solutions in our in-house cluster with 11 machines. The experiments on three real trajectory datasets up to 170 million data points confirm the scalability and efficiency of our methods.

1.4 Thesis Organization

The rest of the thesis is organized as follows: in Chapter 2, we review the literature related to our proposed queries in different data domains. In Chapter 3, we describe the Graph Window query on graph data. In Chapter 4, k-Sketch query on sequence data is presented. In Chapter 5, we report the General Co-Movement Pattern query on trajectory data. Chapter 6 summarizes this thesis and highlights future directions.

1.5 Published Material

The research in this thesis has led to numerous publications, which are listed as follows:

- The overview of the thesis has been published in SIGMOD Ph.D. Symposium. **Qi Fan**, Kian-Lee Tan. Towards Neighborhood Analytics. Proceedings of the ACM SIGMOD on PhD Symposium, 2015

- The work in Chapter 3 appears in DASFAA 2016. **Qi Fan**, Zhengkui Wang, Chee-Yong Chan, Kian-Lee Tan. Towards Window

Analytics over Large-Scale Graphs. International Conference on Database Systems for Advanced Applications, 2016

- The findings in Chapter 4 have been submitted for publication.

  Qi Fan, Yuchen Li, Dongxiang Zhang, Kian-Lee Tan. Discovering Newsworthy Themes From Sequence Data: A Step Towards Computational Journalism.

- The work in Chapter 5 has been accepted in VLDB 2017.

  Qi Fan, Dongxiang Zhang, Huayu Wu, Kian-Lee Tan. A General and Parallel Platform for Mining Co-Movement Patterns over Large-scale Trajectories. Proceedings of the VLDB Endowment, 2017

Chapter 2

Literature Review

Our proposed neighborhood based queries are inspired by the usefulness of window functions in SQL analytics [72]. A window function in SQL specifies a set of partitioning attributes A and an aggregate function f. Its evaluation first sorts input records based on A to form overlapped partitions for each record. And then, f is evaluated for every partition and the aggregate result is associated with the corresponding records. Several optimization techniques [8,14] have been developed to evaluate complex SQL queries involving multiple window functions.

However, the semantic and evaluation of the window function are restricted. In SQL window functions, tuples need to be sorted in order to form individual partitions (i.e., windows). In fact, 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 data domains. Nevertheless, there are quite a few works that are related to our proposed neighborhood based queries and we review them in this section.

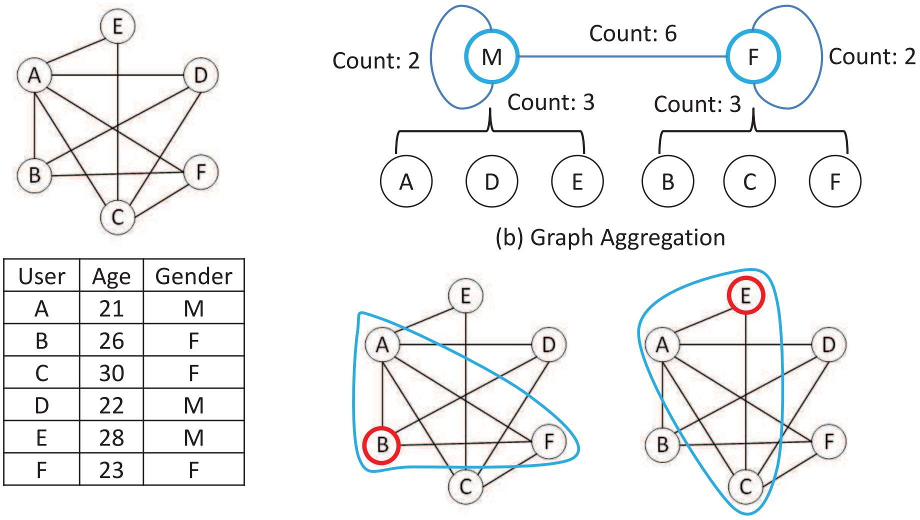
2.1 Graph Window Queries

2.1.1 Graph Aggregation

Works on graph data analytics have focused on graph aggregation [15, 57, 64, 74], which are different from Graph Window Queries (GWQ). In a general model, graph aggregation comprises three steps: (1) partition graph based on attributes of vertex (and/or edges), (2) aggregate each partition to form Aggregated Nodes, and (3) link each aggregated node to form one Aggregated Graph. An illustration of the Graph Aggregation is shown in Figure 2.1 (b). In the first step, the input graph is partitioned on the "Gender" attribute of vertex which results in two partitions. In the second step, two aggregated nodes are formed, i.e., M (stands for Male) containing nodes A, D, E and F (stands for Female) containing nodes B, C, F. In the third step, the links between M and F are added, with the "count" attached on each link. Differently, Graph Window Queries perform graph analytics from the vertex-centric perspective. In GWQ, the neighborhood structure of each vertex form overlapping partitions. Then, analytics are computed over each neighborhood structure. In Figure 2.1 (c), the neighborhood structures of B and E are highlighted. Clearly, the GWQ is different from graph aggregation and they could not model each other.

2.1.2 Reachability Queries and Indexes

Classic reachability queries, which answer whether two vertexes are connected, have been studied extensively in literature. To facilitate fast query processing, many indexes are proposed [17, 18, 65, 70]. 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 [18] which tests 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 $\theta(n^2)$



User	Age	Gender
A	21	M
B	26	F
C	30	F
D	22	M
E	28	M
F	23	F

(a) Mini social network

(b) Graph Aggregation

(c) Graph Window

Figure 2.1: Illustration of *Graph Aggregation* and *Graph Window Queries*. (a) is an example social network, (b) is graph aggregation, (c) are the windows of vertexes B and E.

reachability tests. This would be inefficient on graphs with over millions of vertexes.

2.1.3 Top-k Neighborhoods

In [67], 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 query, while the difference is that they focus on providing pruning techniques to select the k best vertexes and our graph window query aims to compute the analytics for each vertex. Therefore, in our setting, the pruning techniques in [67] does not take effect and would be equivalent to the non-indexed approach as described in Section 3.3.

2.1.4 Egocentric Networks

Egocentric networks [47, 50] have been playing an important role in network study. The egocentric networks refer to the neighborhood structure of each vertex in a graph. Although many works have studied structural analysis on egocentric networks, they do not consider efficient processing of data analytics (e.g., aggregation) within each egocentric network. Recently, Jayanta et.al. [51] proposed an EAGR system to summarize attribute information among each vertex's neighborhoods. Their technique builds an overlay graph to leverage the shared components among vertexes' neighborhood structures to boost query processing. Technically, EAGR runs in iterations and starts with the vertex-neighborhood mapping as the initial overlay graph. During each iteration, it sorts vertexes in an overlay graph according to their neighborhood information. Then an FP-Tree [30] is built to mining the largest shared components based on the sorted vertexes. As the algorithm iterates, the overlay graph evolves to be sparser.

The main drawback of EAGR is its high demands of resources on the overlay construction. In terms of memory cost, EAGR assumes the initial vertex-neighbor mapping can 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 in disk, the performance of EAGR will be largely slow down. In terms of computational cost, EAGR requires to sort all vertexes in a graph and build an FP-Tree in each iteration. When the graph has millions of vertexes, the construction of the index is largely slow down.

We tackle these drawbacks by adopting a hashing based approach that clusters each vertex according to its neighborhood similarity. During the hashing, a vertex's neighborhood information is computed on-the-fly. As compared to the sorting based approach, we do not require vertex's neighborhood to be resided in memory. In order to reduce the repetitive computation, we adopt a Dense Block heuristic to leverage the shared components among vertexes' neighborhoods. We then propose an estimation scheme that further reduces the number of neighborhood accesses. Experiments show

[1] Available at http://snap.stanford.edu/data/index.html, which is used [51]

that our schemes outperform EARG in both query processing and memory usage. Our methods are able to perform well even when EAGR algorithm fails when neighborhood information overwhelms system's memory and our methods takes much shorter index construction time.

2.2 k-Sketch Query

Our proposed k-Sketch query is closely related to the following four areas: *news discovery in computational journalism*, *frequent episode mining on sequence data*, *top-k diversity query* and *event detection in information retrieval.*

2.2.1 Computational Journalism

An important aspect of computational journalism is to leverage computational technology to discover striking news themes. Previous works on automatic news theme generation belong to two categories: *dimension-oriented* approach and *subject-oriented* approach. Dimension-oriented approaches aim to select appropriate dimensions to make an event interesting. Representative works include the *situational facts* [55] and the *one-of-the-few facts* [66]. On the other hand, subject-oriented approaches aim to summarize subjects' histories from their historical events, such as the *prominent streaks* [73]. Our proposed solution falls into the subject-oriented category.

2.2.1.1 Situational facts and one-of-the-few facts

Situational Facts [55]: it finds for a given event, the best constraint-measure pair that makes the event unique (i.e., not dominated by others). An example of the *situational fact* is listed in Table 2.1 and is extracted from the NBA game events where the measure dimensions are "points, assists, and rebounds" and the constraint dimensions are "team, result, and date". The *situational fact* of the event in Table 2.1

is the constraint-measure pair ⟨team=Blazer, points⟩, since under this situation, this event is a skyline among all events (i.e., no other events matching the constraint "team=Blazer" contain an even higher "point").

One-of-the-few Facts [66]: it finds the dimensions under which no more than τ events are in the k-skyband (i.e., not dominated by k events and k is as small as possible). An example of the *one-of-the-few fact* is listed in Table 2.1. In the example, when $\tau = 5$, two dimensions are selected (i.e., points and rebounds). Under these two dimensions, five players are the skylines (i.e., 1-skyband), thus each of them is a *one-of-the-5* player.

As demonstrated, "situational facts" and "one-of-the-few facts" are dimension-oriented since they attempt to generate news themes by selecting dimensions.

Table 2.1: Examples of different news themes.

Method	**Example news theme**
Situational facts [55]	Damon Stoudamire scored 54 points on January 14, 2005. It is the highest score in history made by any Trail Blazer.
One-of-the-τ facts [66]	Jordan, Chamberlain, James, Baylor and Pettit are the five players with highest points and rebounds in NBA history.
Prominent streaks [73]	1.Kobe scored 40+ in 9 straight games, first in his career! 2.Kobe scored 50+ in 4 straight games, first in his career! 3...
k-Sketch	1.Kobe scored 40+ in 9 straight games, ranked 4th in NBA history! 2.Kobe scored 50+ in 4 straight games, ranked 1st in NBA history. 3....

2.2.1.2 Prominent streaks

Zhang et al. [73] proposed a subject-oriented approach to generate news themes by discovering *prominent streaks.* In [73], a streak is modeled as a pair of the streak duration and the minimum value of all events in the streak. For example, as shown in Table 2.1, a streak of “Kobe” may be $\langle$9 consecutive games, minimum points of 40$\rangle$. The objective of [73] is to discover all the non-dominated streaks (i.e., prominent streaks) where the dominance is defined among streaks of the same subject. Despite being the same subject-oriented approach, our k-Sketch query differs from [73] in two aspects. First, we look at the *global prominence among all* subjects (i.e., rank in the entire NBA history) rather than local prominence within one subject (i.e., non-dominance in one’s career). Second, our model provides the best k ranked-streaks for each subject, whereas [73] returns a set of skylines which potentially could be large.

2.2.2 Frequent Episode Mining

In sequence data mining, an episode [40, 49, 56, 78] 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 containing events. The objective of frequent episode mining is to discover episodes whose occurrences exceed a threshold. Our k-Sketch query differs from the episode mining in two major aspects. First, episodes are associated with categorical values thus they can be grouped to count the occurrences. On the other hand, our ranked-streaks are defined with numerical values, making it inappropriate to be grouped. Second, the episodes are selected based on the occurrences which do not contain the *rank* information, whereas our k-Sketch query explicitly provides the rank of selected streaks. As such, episode mining techniques cannot support the k-Sketch query.

2.2.3 Top-k Diversity Query

Top-k diversity queries [1,9,16,24] aim to find a subset of objects to maximize a scoring function. The scoring function normally penalizes subsets with similar elements. Our k-Sketch query has two important distinctions. First, the inputs to top-k diversity queries are known in advance, whereas in k-Sketch query, the ranks of streaks need to be derived. Second, existing methods for online diversity queries [9, 16, 24] 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.

2.2.4 Event Detection and Tracking

In information retrieval, event detection and tracking aim to extract and organize new events from various media sources such as text streams [3, 10], social media streams [42] and web articles [59]. Despite the usefulness of these works, they differ from our k-Sketch query as they focus on the detection of a single event, whereas k-Sketch aims to summarize a subject's history. Therefore, the abovementioned techniques cannot be directly applied.

2.3 General Co-Movement Pattern Query

Existing works on movement patterns can be grouped into three categories: *co-movement patterns*, *dynamic movement patterns* and *trajectory mining frameworks.*

2.3.1 Co-Movement Patterns

2.3.1.1 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 pairwise 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 [26]. Technically, *flock* utilizes a m^{th}-order Voronoi diagram [39] to detect whether a subset of n ($n \geq m$) objects stay in a disk region. *Convoy* employs a trajectory simplification [23] 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 snapshot. During the scan, the object groups that do not appear in consecutive snapshots 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 dataset are scanned.

2.3.1.2 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 an 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 [63]. *Swarm* designs two more pruning rules called backward pruning and forward pruning [44]. *Platoon* [43] leverages a prefix table structure to steer the depth-first search, which shows efficiency as compared to the other two methods. However, the pruning rules adopted by the three patterns heavily rely on depth-first search which loses efficiency in a parallel scenario.

2.3.2 Other Related Trajectory Patterns

A closely related literature to co-movement patterns is the *dynamic movement* patterns. Instead of requiring the same set of object traveling together, *dynamic movement* patterns allow objects to temporally join or leave a group. Typical works include *moving clusters* [36], *evolving convoy* [5], *gathering* [75] etc. These works cannot model GCMP since they enforce global consecutiveness on the temporal domain.

2.3.3 Trajectory Mining Frameworks

Jinno et al. in [35] 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 similar time. Li et al. proposed a framework of processing online *evolving group* pattern [41], 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

Graph Window Query: Neighborhood Analytics on Attributed Graphs

3.1 Introduction

In this chapter, we study the neighborhood analytics on large-scale attributed graphs. Attributed graphs are prevalently adopted to model real life information networks such social networks, biological networks and phone-call networks. In the attributed graph model, the vertexes correspond to objects and the edges capture the relationships between these objects. For instance, in social networks, every user is represented by a vertex and the friendship between two users is reflected by an edge connecting the vertexes. Besides, a user's profile is maintained as the vertex's attributes. Such graphs contain a wealth of valuable information which can be analyzed to discover interesting patterns [15, 57, 64, 74]. With the increasingly larger network sizes, it is becoming challenging to query, analyze and process these graph data. Therefore, there is an urgent call for effective and efficient mechanisms to draw out information over graph data resources.

Recent advances in graph analytics such as graph aggregation [64, 74] and summarization [15, 57] focus on analyzing the entire graph as a whole. In fact, it is often useful to perform *neighborhood analytics* on graph data to analyze the vicinity of vertexes. That is for each vertex, the analytics is conducted over its *neighborhoods.* For instance, in a social network, it is important to detect a person's social influence among his/her social community. The "social community" of the person is essentially his/her neighborhood vertexes representing his/her friends.

Similar neighborhood concept has been supported by window functions [8, 14] in relational data analytics. Instead of performing analysis (e.g., ranking and aggregate) over the entire data set, a window function returns for each tuple a value derived from its neighboring tuples. For instance, when finding each employee's salary ranking within every department, each tuple's neighbors are basically the tuples from the same department. However, the window definition in the relational context ignores graph structures which makes it unsuitable in the graph context. Therefore, we seek to utilize the general *neighborhood analytics* to formulate the notion of *graph windows.*

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

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

Example 3.1.2. *(Topological window)* In biological networks (such as Argocyc, Ecocyc etc. [37]), genes, enzymes and proteins are vertexes and their dependencies in a pathway are edges. Because these networks are directed and acyclic, **comparison neighborhood** based on the ancestry relationship helps to reveal the influences among molecules. For instance, to find out the statistics of molecule in a protein production pathway, we can traverse the graph to find every molecule that is in its upstream. Then we summarize the number of genes and enzymes among those molecules. To answer such queries, computing the ancestry based comparison neighborhood is necessary.

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

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

For *k*-hop window query, the latest processing algorithm can be adopted from literature is EAGR [51]. EAGR leverages an overlay graph to represent the shared components among different windows. It incrementally constructs the overlay graph through multiple iterations. In each iteration, it builds a Frequent-Pattern Tree [31] to discover the largest shared component among vertex windows. However, to achieve efficient shared component detection, EAGR requires all vertex's k-hop neighbors to be pre-computed and resided in memory; otherwise EAGR incurs high performance overheads due to secondary storage accesses (e.g., disk I/Os). This limits the usage of EAGR in large-scale graphs. For instance, a LiveJournal social network graph[1] (4.8M vertexes, 69M edges) generates over 100GB neighborhood information for $k = 2$ in adjacency list representation. In addition, the overlay graph construction is not a one-time task, but periodically performed after a certain number of structural updates in order to maintain the quality. The high memory consumption renders the scheme impractical when k and the graph size increases.

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

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

[1] Available at http://snap.stanford.edu/data/index.html, which is used in [51]

stand the graphs under these different semantics.

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

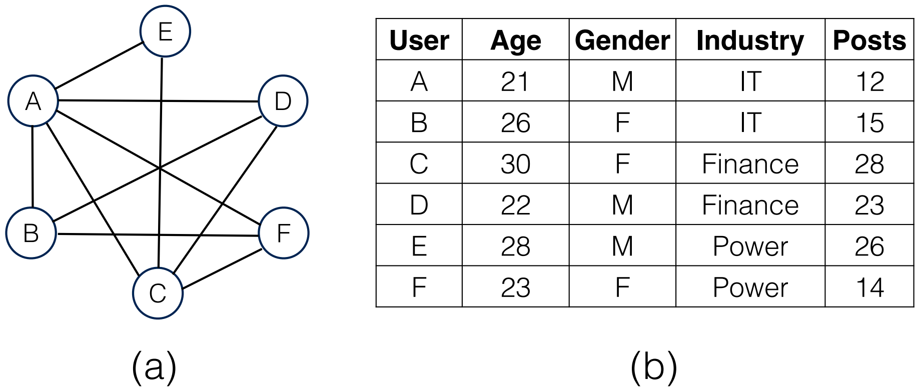
- We perform extensive experiments over both real and synthetic datasets with hundreds of millions of vertexes and edges on a single machine. Our experiments indicate that our proposed index-based algorithms outperform the naive non-index algorithm by up to four orders of magnitude. In addition, our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

The rest of the chapter is organized as follows. In Section 3.2, the graph window query is formulated. In Section 3.3, Dense Block Index is presented to process general window queries. A specialized index to handle topological query is presented in Section 3.4. Section 3.5 demonstrates our experimental findings and Section 3.6 concludes this chapter.

3.2 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.

Figure 3.1 (a) shows an undirected graph representing a social network that will be used as our running example in this chapter. For convenience, each vertex is labeled by its "user" attribute; and there is one edge between vertex X and vertex Y if user X and user Y are connected in the social network. The table in Figure 3.1 (b) shows the values of five attributes (User, Age, Gender, Industry, and Number of posts) associated with each user.



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

(a)

(b)

Figure 3.1: A miniature social graph. (a) the graph structure. (b) the attributes associated with the vertexes in (a).

Given a data graph $G = (V, E)$, a *Graph Window Function (GWF)* over G can be expressed as a pair (W, Σ), where $W(v)$ denotes a *window specification* for a vertex $v \in V$ that determines the set of v's neighborhood nodes[2], Σ denotes an *aggregate function*[3]. The evaluation of a GWF (W, Σ) on G computes for each vertex v in G, the aggregation Σ over all the nodes in $W(v)$, which is denoted by $\Sigma_{v' \in W(v)} v'$. In this chapter, we focus on the distributive or algebraic aggregate functions (e.g., sum, count, average), as these aggregate functions are widely used in practice.

In the following, we introduce two useful types of window specification (i.e., W),

[2] We use "vertex" to refer the vertex in the original graph and "node" to refer to the vertex in the windows.

[3] An aggregate function is associated with an attribute. I.e., `average`(age) and `average`(salary) are considered to be two different aggregate functions.

namely, *k-hop window* and *topological window*.

Definition 3.2.1 (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 nodes in G which can be reached by v within k hops. For an undirected graph G, a node u is in $W_{kh}(v)$ if there is a α-hop path between u and v where $\alpha \leqslant k$. For a directed graph G, a node u is in $W_{kh}(v)$ if there is a α-hop directed path from v to u[4] where $\alpha \leqslant k$.

Intuitively, a k-hop window selects the neighboring nodes within a k-hop distance. These neighboring nodes typically represent the most important entities to a vertex with regard to their structural relationship in a graph. Thus, the k-hop window provides a meaningful specification for many applications, such as customer behavior analysis [11, 21] , digital marketing [47] etc.

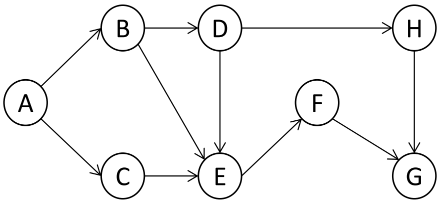
As an example, in Figure 3.1, the 1-hop window of vertex E is $\{A, C, E\}$ and the 2-hop window of vertex E is $\{A, B, C, D, E, F\}$.

Definition 3.2.2 (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 nodes of v in G, i.e., a vertex u is in $W_t(v)$ if 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 if paper X cites paper Y, the topological window of a paper represents the citation impact of that paper [13, 33, 48].

To illustrate, Figure 3.2 shows a small example of a Pathway Graph from a biological network. The topological window of E (i.e., $W_t(E)$) is $\{A, B, C, D, E\}$ and $W_t(H)$ is $\{A, B, D, H\}$.

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



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

Figure 3.2: A miniature pathway DAG. (a) the DAG structure. (b) the attributes associated with the vertexes of (a).

Definition 3.2.3 (Graph Window Query)**.** A graph window query on a data graph G is of the form $GWQ(G, W_1, \Sigma_1, \cdots, W_m, \Sigma_m)$, where $m \geq 1$ and each pair (W_i, Σ_i) is a graph window function on G.

In this chapter, 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 simply processed as a sequence of window functions one after another. We leave the optimization of processing multiple window functions for future studies.

3.3 Dense Block Index

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

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

To efficiently evaluate graph window queries, we propose an index technique named *Dense Block Index* (*DBIndex*), which achieves both space and query efficiency. The main idea of DBIndex is to try to reduce the aggregation 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 Figure 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 $\Sigma_{v \in S} v$ can be computed only once and then reused to compute the aggregation for $\Sigma_{v \in W(B)} v$ and $\Sigma_{v \in W(C)} v$.

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 $v_1 \neq v_2 \in V$, s.t. $B \subseteq W(v_1)$, 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 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 of the aggregate functions (i.e., Σ).

Figure 3.3(a) shows an example of a DBIndex with respect to the social graph in

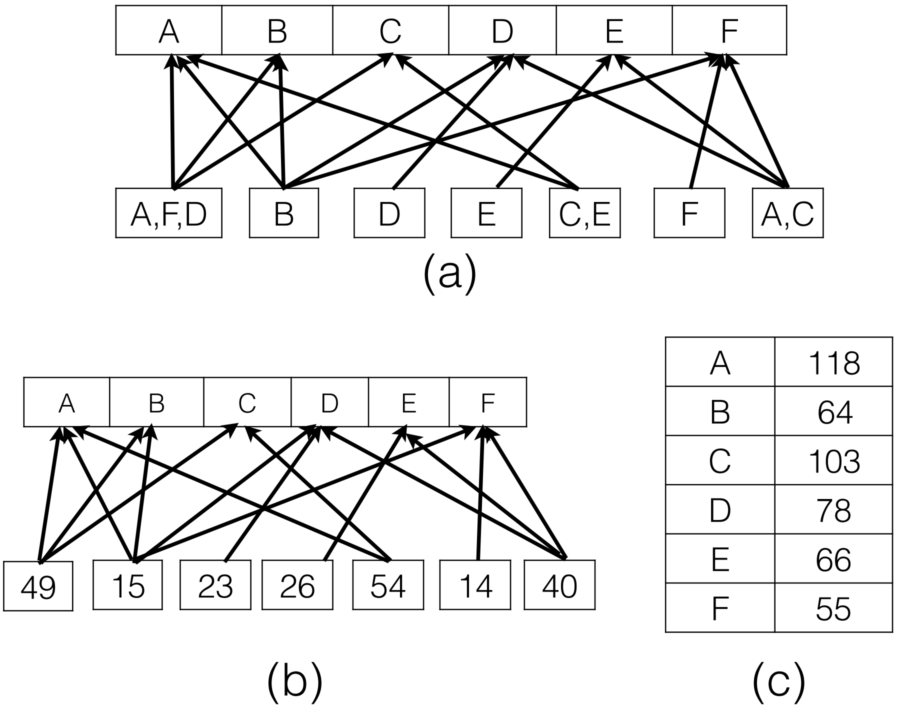


Figure 3.3: Window query processing using DBIndex. (a) provides the DBIndex for 1-hop window query in Figure 3.1; (b) shows the partial aggregate results based on the dense block; (c) provides the final aggregate value of each window.

Figure 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.

3.3.1 Query Processing with DBIndex

Given a DBIndex with respect to a graph G and a window function W, a graph window query $Q = (G, W, \Sigma)$ 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., $T_i = \Sigma_{v \in B_i} v$. 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 $W(v)$. In other words, if $B(v)$ is the collection of blocks linked to $W(v)$, then the aggregation for $W(v)$ is computed by $\Sigma_{B_i \in B(v)} T_i$.

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

3.3.2 DBIndex Construction

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

The first challenge is the time complexity of the index construction. From the discussion of query processing using DBIndex, we note that the number of aggregations 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 aggregations 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[5]. Therefore, effective 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 the query processing, a straightforward approach is to first derive the window $W(v)$ for each vertex $v \in V$ 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 large graphs.

In this section, we present two heuristic approaches, namely *Min-hash Clustering (MC)* and *Estimated Min-hash Clustering (EMC)*, to construct DBIndex. The first

[5] Note that a simpler variation of our problem has been proven to be NP-hard [58].

approach MC is to construct a DBIndex for general window functions; While the second approach EMC is to construct a DBIndex specifically for k-hop window function. Compared to MC, EMC adopts a heuristic to speed up the index construction at the expense of sacrificing the "quality" of the dense blocks (in terms of their sizes).

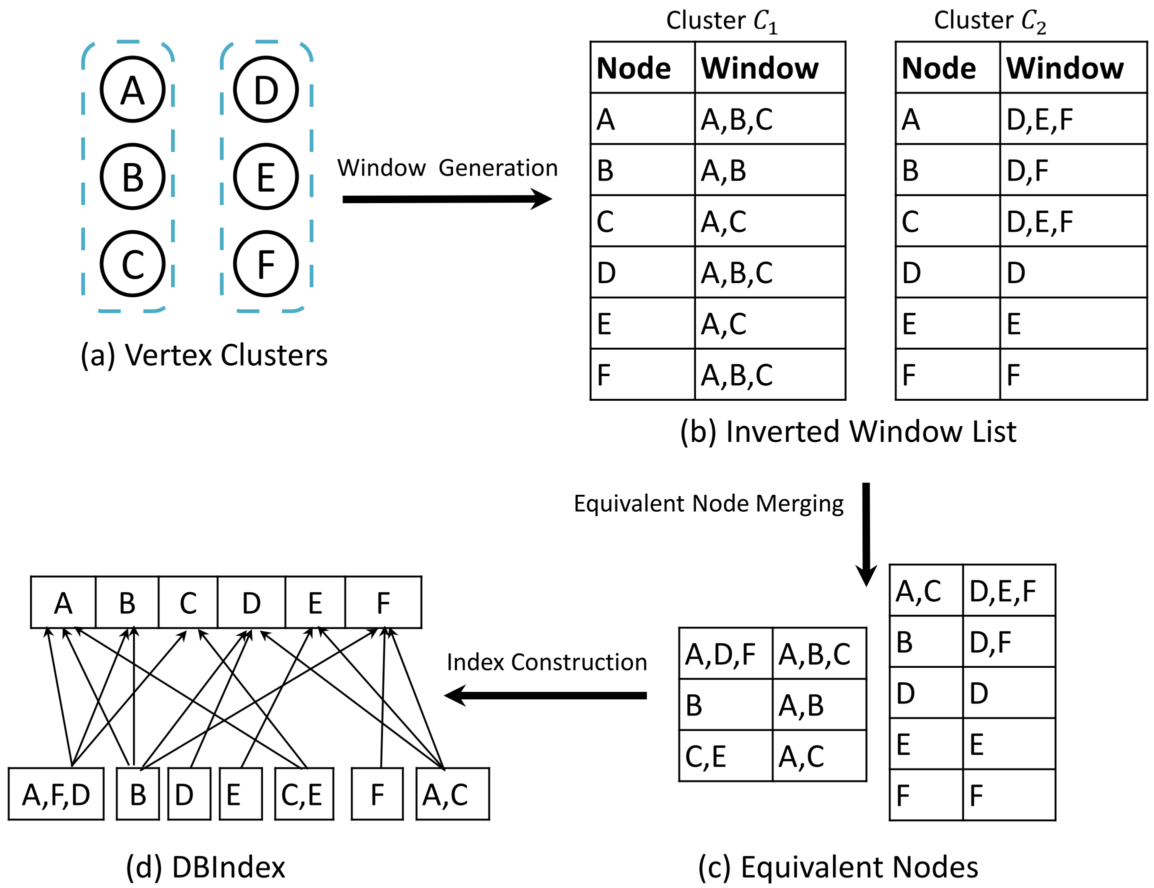
3.3.2.1 Min-hash clustering (MC)

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 window contents and then identify large dense blocks from each of the smaller clusters. Intuitively, two windows are considered to be similar if they share a large subset of nodes. We apply the well-known *Min-hash based Clustering (MC)* algorithm [12] to partition the windows into clusters of similar windows. The Min-hash clustering algorithm is based on the *Jaccard Coefficient* which measures 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 indicates a higher similarity.

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 vertexes in V into clusters using Min-hash algorithm such that vertexes with similar windows belong to the same cluster. For each vertex $v \in V$, we first derive its window $W(v)$ by an appropriate traversal of the graph G. Next, we compute a hash signature (denoted by $H(v)$) for $W(v)$ based on applying m hash



(a) Vertex Clusters

Cluster C_1

Node	Window
A	A,B,C
B	A,B
C	A,C
D	A,B,C
E	A,C
F	A,B,C

Cluster C_2

Node	Window
A	D,E,F
B	D,F
C	D,E,F
D	D
E	E
F	F

(b) Inverted Window List

A,D,F	A,B,C
B	A,B
C,E	A,C

A,C	D,E,F
B	D,F
D	D
E	E
F	F

(c) Equivalent Nodes

(d) DBIndex

Figure 3.4: DBIndex construction over social graph in Figure 3.1. (a) shows two clusters after MinHash clustering. (b) shows inverted window list for each node. (c) shows the dense blocks via equivalent node merging. (d) provides the final DBIndex.

functions on the set $W(v)$. Vertexes 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, 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 vertexes.

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 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 $W(x), x \in C_i$, $u \in W(x)$ if and only if $v \in W(x)$. 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 vertex $v \in C_i$ to determine its window $W(v)$[6].

However, since C_i is now a smaller cluster of nodes, we can now materialize all the windows for the vertexes in C_i in memory. 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 re-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 $links(I)$ 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 the remaining part of C_i following the above two-step procedure.

[6] Note that although we could have avoided deriving $W(v)$ a second time if we had materialized all the derived windows the first time, our approach is designed to avoid such a high space complexity at the cost of computing each $W(v)$ twice. We present an optimization in Section 3.3.2.2 to avoid the recomputation cost.

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 1 CreateDBIndex

```
Input: Graph $G = (V, E)$, window function $W$
Initialize DBIndex $I$: $nodes(I) = V$, $blocks(I) = \emptyset$, $links(I) = \emptyset$
for all $v \in V$ do
    Traverse $G$ to determine $W(v)$
    Compute the hash signature $H(v)$ for $W(v)$
end for
Partition $V$ into clusters $\mathcal{C} = \{C_1, C_2, \cdots\}$ based on hash signatures $H(v)$
for all $C_i \in \mathcal{C}$ do
    for all $v \in C_i$ do
        Traverse $G$ to determine $W(v)$
    end for
    IdentifyDenseBlocks $(I, G, W, C_i)$
end for
return $I$
```

Algorithm 2 IdentifyDenseBlocks

```
Input: DBIndex $I$, Graph $G = (V, E)$, window function $W$, a cluster $C_i \subseteq V$
Partition $C_i$ into blocks based on node equivalence
Initialize $DenseNodes = \emptyset$
for all dense block $B$ do
    Insert $B$ into $blocks(I)$ if $B \notin blocks(I)$
    Insert $(B, v)$ into $links(I)$ for each $v \in C_i$ where $B \subseteq W(v)$
    $DenseNodes = DenseNodes \cup B$
end for
if $(DenseNodes = \emptyset)$ then
    for all block $B$ do
        Insert $B$ into $blocks(I)$ if $B \notin blocks(I)$
        Insert $(B, v)$ into $links(I)$ for each $v \in C_i$ where $B \subseteq W(v)$
    end for
else if $(C_i - DenseNodes \neq \emptyset)$ then
    if $(C_i \neq DenseNodes)$ then
        RefineCluster $(I, G, W, C_i - DenseNodes)$
    end if
end if
```

Figure 3.4 illustrates the construction of the DBIndex with respect to the social graph in Figure 3.1(a) and 1-hop window using the MC algorithm. First, the set of graph vertexes are partitioned into clusters using Min-hash clustering; Figure 3.4(a) shows that the set of vertexes $V = \{A, B, C, D, E, F\}$ are partitioned into two clusters $C_1 = \{A, B, C\}$ and $C_2 = \{D, E, F\}$.

For example, cluster C_1 in Figure 3.4(b) shows the inverted list representing the node n and the vertexes $v \in C_1$ whose windows contains n, i.e., $\{v|n \in W(v)\}$. Consider the identification of dense blocks in cluster C_1. As shown in Figure 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: $\mathit{blocks}(I) = \{B_1, B_2, B_3\}$ and $\mathit{links}(I) = \{(B_1, \{A, B, C\}), (B_2, \{A, B\}),, (B_3, \{A, C\})\}$. 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 Min-hash cost is H and the total cost for collecting window function for all vertexes 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 window functions. As we demonstrate in Section 3.5, for k-hop window, the H and B are the major factors in the index construction. To reduce the construction time, we design further optimization techniques for the k-hop window.

Algorithm 3 RefineCluster

```
Input: DBIndex $I$, Graph $G = (V, E)$, window function $W$, a cluster $C \subseteq V$
for all $v \in C$ do
    Compute the hash signature $H(v)$ for $W(v) \cap C$
end for
Partition $C$ into clusters $\mathcal{C} = \{C_1, C_2, \cdots\}$ based on hash signatures $H(v)$
for all $C_i \in \mathcal{C}$ do
    IdentifyDenseBlocks $(I, G, W, C_i)$
end for
```

3.3.2.2 Estimated min-hash 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 traversal to compute all k-hop windows could incur a high computation overhead. Moreover, the cost of initial Min-hash 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, Min-hash clustering would incur high computation cost.

To address these issues, we present a more efficient approach, referred to as *Estimated Min-hash 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 for constructing a DBIndex with respect to a k-hop window function by clustering the vertexes 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 to cluster the vertexes 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 hashing on lower hop window is of order of magnitude faster. For the extreme case, adopting 1-hop window of a node v requires only accessing the adjacent nodes of v. The tradeoff for the improved efficiency is the reduced "quality" of the dense blocks (in terms of their sizes). However, our experimental results show that this reduction in quality is only marginal which makes this approximation worthy.

3.3.2.3 Justification of heuristic

In the following, we justify of our heuristic by make an assumption on the independence[7] of the vertexes in a graph. First, we provide the following theorem which links a set of connected vertexes to the newly discovered vertex by one-hop expansion.

Theorem 3.3.1. Let S be a collection of connected vertexes. Let T be the collection of newly discovered vertexes from one-hop expansion of S. Then the ratio of $|T|/|S|$ decreases with respect to $|S|$.

Proof. Consider a random variable Y_i which indicates the newly discovered vertexes 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 vertexes. In total, there are $\binom{|V|-1}{d_i-1}$ combinations with d_i

[7] Although the assumption may not always hold in reality, it makes the analysis feasible. We also conduct an empirical evaluation on real datasets in Section 3.5.1 to indicate the effectiveness of our heuristic.

edges. Therefore, the probability can be written as:

$$P(Y_i = y|v_i \in S) = \frac{\binom{|S|-1}{d_i-y-1}\binom{|V|-|S|}{y}}{\binom{|V|-1}{d_i-1}} \tag{3.2}$$

Thus, the expectation of Y_i is:

$$\begin{aligned} E(Y_i|v_i \in S) &= \Sigma(yP(Y_i = y|v_i \in S)) \\ &= \Sigma_{y=1}^{y=d_i-1}(\frac{\binom{|S|-1}{d_i-y-1}\binom{|V|-|S|}{y}}{\binom{|V|-1}{d_i-1}}y) \\ &= \Sigma_{y=1}^{y=d_i-1}(\frac{\binom{|S|-1}{d_i-y-1}\binom{|V|-|S|-1}{y-1}}{\binom{|V|-1}{d_i-1}}(|V|-|S|)) \\ &= (|V|-|S|)\Sigma_{y=1}^{y=d_i-1}\frac{\binom{|S|-1}{d_i-y-1}\binom{|V|-|S|-1}{y-1}}{\binom{|V|-1}{d_i-1}} \\ &= (|V|-|S|)\frac{\binom{|V|-2}{d_i-2}}{\binom{|V|-1}{d_i-1}} = \frac{(|V|-|S|)(d_i-1)}{|V|-1} \end{aligned} \tag{3.3}$$

Taking the expectation over all vertexes in S, we can find the expectation of $E(Y_i|S) = \frac{(|V|-|S|)(\overline{d}-1)}{|V|-1}$, where $\overline{d}$ is the average degree of the graph. Let T be the number of newly discovered vertexes for one-hop expansion of the entire set S. Since each Y_i is independent, it follows that $E(T) = E(\Sigma_{i=1}^{i=|S|}Y_i|S) = \Sigma_{i=1}^{i=|S|}E(Y_i|S) = \frac{|S|(|V|-|S|)(\overline{d}-1)}{|V|-1}$. Let α be the ration of $\frac{E(T)}{|S|}$. It follows $\alpha = \frac{\overline{d}-1}{|V|-1}(|V|-|S|)$. As $\overline{d}$, V are constants, α decreases linearly with respect to $|S|$. □

Next, we consider the Jaccard coefficient of two vertexes u, v's windows. Let $J_k(u, v)$ be the Jaccard coefficient of u, v at hop k. We use I_k to denote $W_k(u) \cap W_k(v)$, and U_k to denote $W_k(u) \cup W_k(v)$. Therefore, $J_k(u, v)$ can be represented as $\frac{|I_k|}{|U_k|}$. Let $J_{k+1}(u, v) = \frac{|I_k|+\alpha_I|I_k|}{|U_k|+\alpha_U|I_U|}$, where α_I and α_U be the ratio of newly discovered vertexes versus original set of vertexes. Since $|U_k| > |I_k|$, by the above theorem, $1+\alpha_U < 1+\alpha_I$. Therefore, $J_{k+1}(u, v) = \frac{|I_k|+\alpha_I|I_k|}{|U_k|+\alpha_U|I_U|} = \frac{1+\alpha_I}{1+\alpha_U} \cdot \frac{|I_k|}{|I_U|} > J_k(u, v)$. This indicates that the expected Jaccard coefficient of the two vertexes increases with respect to k, which justifies our heuristic. It is notable that, although our analysis of the heuristic relies on the independence of vertexes, our experiments show that the heuristic is effective in many real datasets.

3.4 Inheritance Index

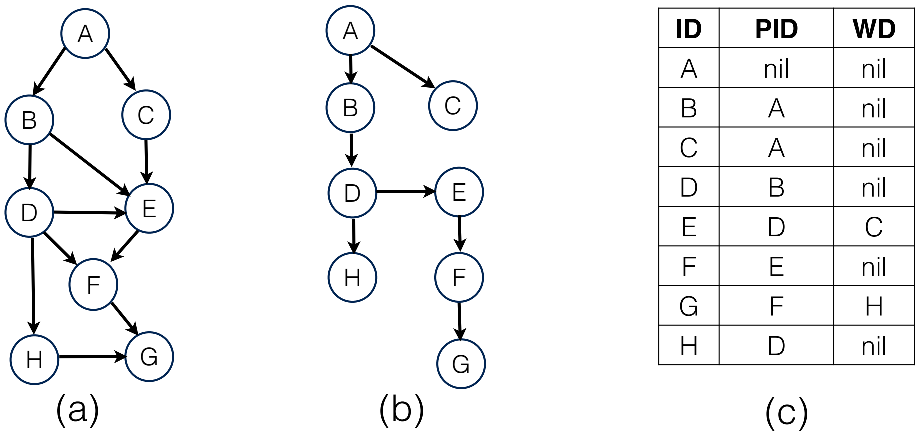
DBIndex is a general index that can support both k-hop as well as topological window queries. However, the evaluation of a topological window function (i.e., W_t) can be further optimized due to its containment feature. In other words, the window of a descendant vertex completely covers that of one of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

Theorem 3.4.1. 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 Figure 3.2 as an example. Figure 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 3.4.1 provides us with opportunities for reducing the space and computation cost in processing topological window queries. First, since the window of a vertex u contains the window of its parent v, there is no need to maintain the full set of nodes of window 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



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

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

closest parent. For instance, in Figure 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 aggregation at a node u can reuse the aggregated 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 $W_t(E)$. Figure 3.5 (b) indicates the inheritance relationship that the values of the father can be inherited to the child in the tree.

Based on our observation, 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 combine with WD(v) to derive $W_t(v)$.

Likewise, we can retrieve the aggregated result of u which can be reused to aggregate v's window. Figure 3.5(c) shows the I-index of our example in Figure 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.

3.4.1 Index Construction

Building an I-Index for a DAG can be done efficiently. This is because the containment relationship can be easily discovered in a topological scan. Algorithm 4 lists the pseudocode for index creation. The scheme iterates through all the vertexes 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 cardinality of v's parent windows, and find the parent with largest cardinality. 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, which makes the scheme space efficient.

We note that the complexity of Algorithm 4 is non-trivial to analyze. This is due 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 index construction complexity is reasonably efficient.

We further note that the size of *I-Index* is hard to be precisely analyzed. This is due to the difficulty of analyzing the window differences. 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 as $O(|V|)$, our experimental results indicate that the index size is always comparable to the graph size. Furthermore, it is possible to reduce the index size (should it be a concern) by employing compression techniques.

Algorithm 4 Create I-Index

```
Input: Input graph: $G$
$I \leftarrow ()$
$p \leftarrow ()$                                  ▷ stores the window for each vertex
$c \leftarrow ()$                                  ▷ stores the cardinality of window for each vertex
for all $v \in$ topological order do
    $diff \leftarrow -\infty$                       ▷ the window difference
    $bestu \leftarrow nil$
    for all $u \in v.parent$ do
        if $c[u] > diff$ then
            $diff \leftarrow c[u]$
            $bestu \leftarrow u$
        end if
    end for
    $I[v].WD \leftarrow diff$
    $I[v].PID \leftarrow bestu$
    $p[v] \leftarrow p[v] \cup v$
    for all $u \in v.child$ do
        $p[u] \leftarrow p[u] \cup p[v]$
    end for
    $c[v] \leftarrow |p[v]|$                        ▷ update window cardinality
    $p[v] \leftarrow ()$                            ▷ release memory
end for
return $I$
```

3.4.2 Query Processing with 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 pseudocode for the query processing. Each vertex v's window aggregation can be calculated as $\Sigma(W_t(v)) = \Sigma(v.PID, \Sigma(v.WD))$, where Σ is the aggregate function[8]. 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

[8] For the function average, we need to keep both count and sum

re-computation. In general, v's window aggregation is achieved by reusing its parent's aggregated result and corresponding window differences. This avoids repeated aggregation and achieves computation sharing between a vertex and its parent. As a result, the computation overhead is largely reduced. Take the index provided in Figure 3.5 (c) as an example, assume the query wants to calculate the sum. 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 $\Sigma$, inheritance index $IIndex$
$w \leftarrow ()$
for all $v \in$ topological order do
    $u \leftarrow IIndex[v].PID$
    $WD \leftarrow IIndex[w].WD$
    $S \leftarrow v.val$
    $S \leftarrow \Sigma(S, w[u])$
    for all $t \in WD$ do
        $S \leftarrow \Sigma(S, t.val)$
    end for
    $w[v] \leftarrow S$
end for
return $w$
```

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.

3.5 Experimental Evaluation

In this section, we present a comprehensive experimental evaluation of our solutions using several real-world information networks and various synthetic datasets. All experiments are conducted on an Amazon EC2 r3.2xlarge machine[9], with an 8-core 2.5GHz CPU, 60GB memory and 320GB hard drive running with 64-bit Ubuntu

[9] http://aws.amazon.com/ec2/pricing/

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.

Table 3.1: Large-scale real graphs.

Name	Type	Number of Vertexes	Number 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

Datasets. For real datasets, we use 8 information networks which are available at the Stanford *SNAP*[10]: LiveJournal1, Pokec, Orkut, DBLP, YouTube, Google, Amazon and Stanford-web. The detail description of these datasets is provided in Table 3.1.

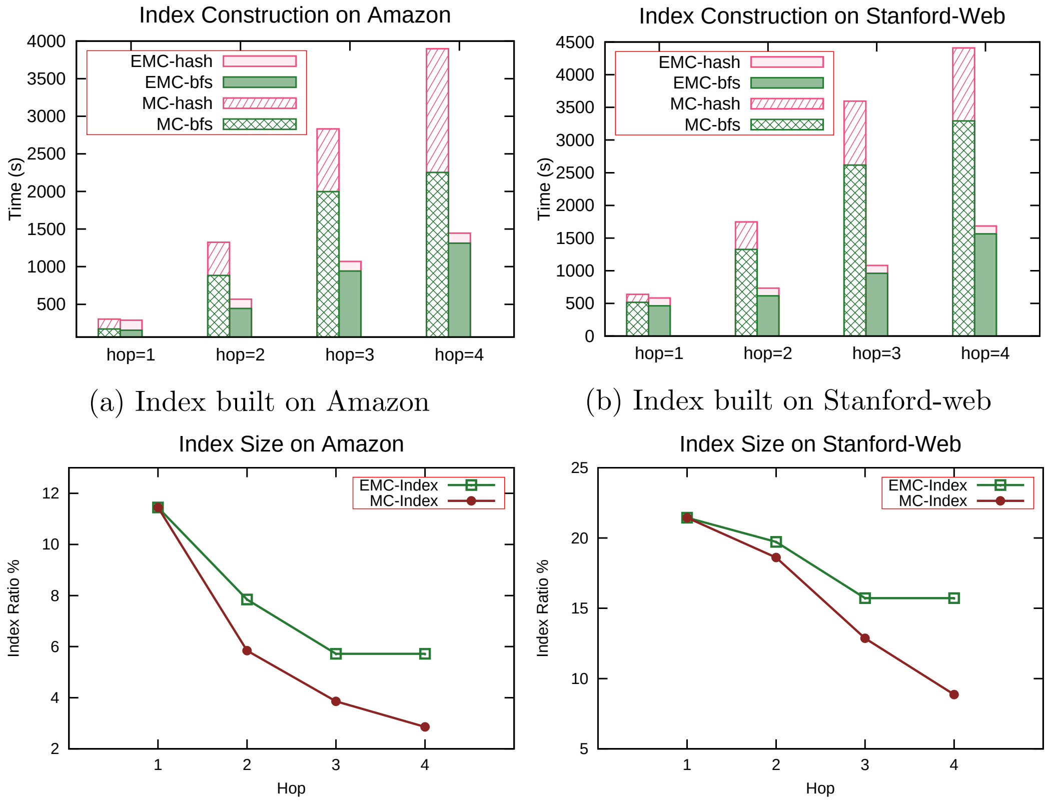
For synthetic datasets, we use two widely used graph data generators. We use the *DAGGER* generator [68] to generate all the synthetic DAGs and the SNAP graph data generator at the Stanford SNAP website to generate non-DAG datasets. For each dataset, each vertex is associated with an integer attribute.

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.

3.5.1 Comparison between MC and EMC

We first compare the effectiveness of the two DBIndex construction algorithms: Min-hash Clustering (MC) and Estimated Min-hash Clustering (EMC). We look at the index construction time, index size and query performance. All these experiments are conducted based on two real datasets: Amazon and Stanford-web. For both datasets,

[10] 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 3.6: Index construction analysis for EMC and MC. (a) and (b) depict the index time for the Amazon and Stanford-web networks. (c) and (d) show the index size for the Amazon and Stanford-web datasets.

we run a series of k-hop queries.[11] For queries with hop count larger than 1, EMC uses 1-hop information for the initial clustering.

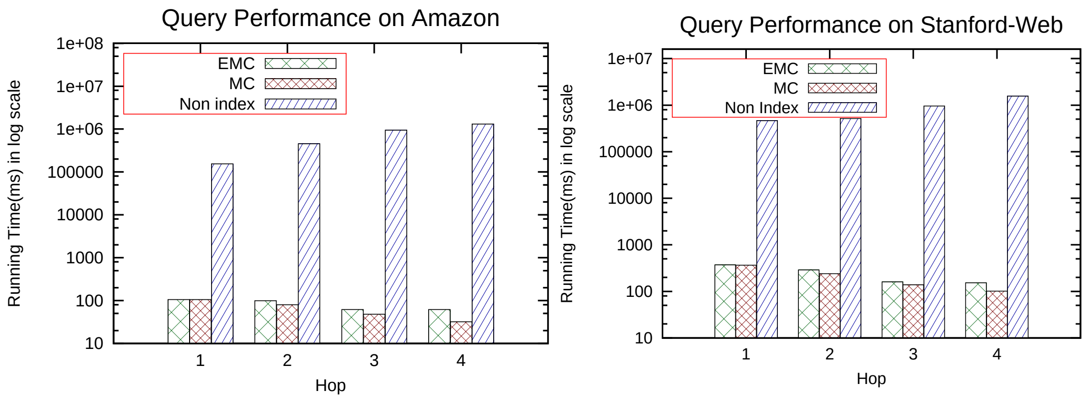
Index Construction. Figures 3.6 (a) and (b) compare the index construction time between MC and EMC when we vary the windows from 1-hop to 4-hop for the Amazon and Stanford-web graphs respectively. To better understand the time differences, the construction time is split into two parts: the Min-hash cost (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. First, as the number of hops increases, the index construction time increases as well.

[11] 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.

This is expected as a larger hop count results in a larger window size and the BFS and the Min-hash time increase correspondingly. Second, as the hop count increases, the difference between the index construction time of EMC and that of MC widens. For instance, as shown in Figures 3.6 (a) and (b), for the 4-hop window 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 Min-hash cost and low BFS cost. We can also see that the Min-hash 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 Min-hash becomes more significant for larger windows. Thus, using 1-hop clustering for larger hop counts reduces the Min-hash 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. Figures 3.6 (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 derived 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. Figures 3.7 (a) and (b) present the query time of MC and



(a) Amazon graph

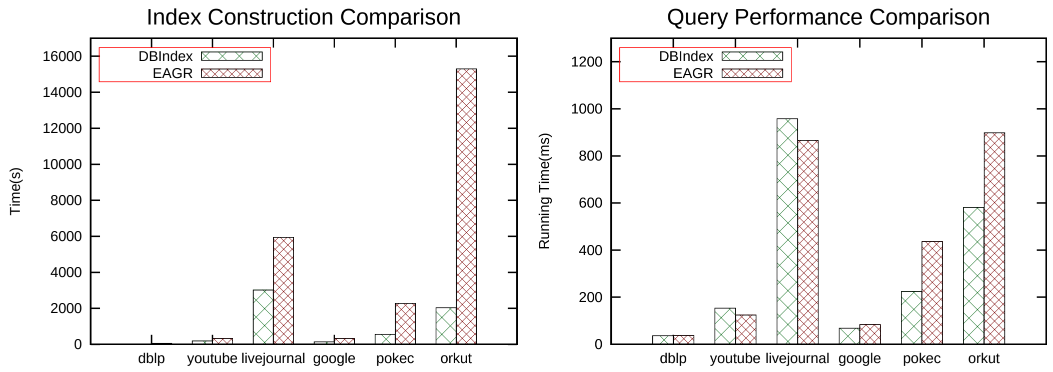
(b) Stanford-web graph

Figure 3.7: Query performance comparison of MC and EMC.

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 run time. In Figures 3.7 (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.

3.5.2 Comparison between DBIndex and EAGR

In this set of experiments, we compare DBIndex and EAGR [51] using both large-scale real and synthetic datasets. As described in [51], for each dataset, EAGR runs for 10 iterations in the index construction.



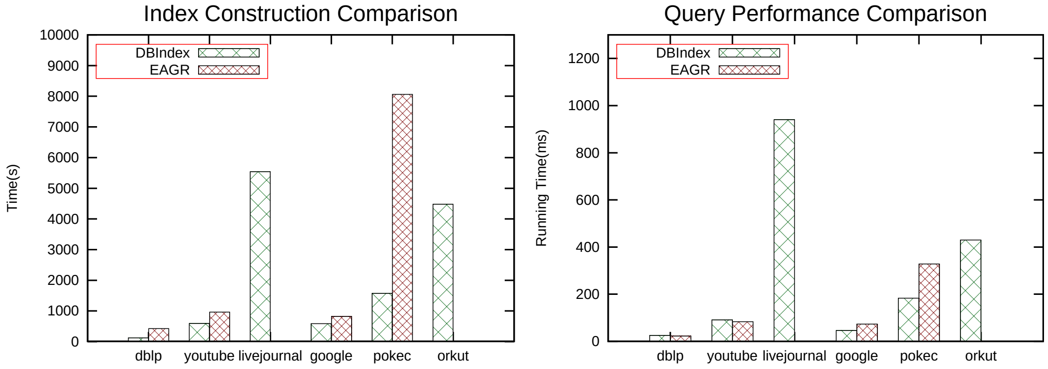
(a) 1-hop index construction

(b) 1-hop query performance

Figure 3.8: Comparison between DBIndex and EAGR for 1-hop query.

3.5.2.1 Real datasets

We first study the index construction and query 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 Figures 3.8 and 3.9 respectively. As shown in Figures 3.8(a) and 3.9(a),



(a) 2-hop index construction

(b) 2-hop query performance

Figure 3.9: Comparison between DBIndex and EAGR for 2-hop query.

both DBIndex and EAGR can build the index for all the real datasets for the 1-hop but EAGR runs out of the memory for the 2-hop query on LiveJournal and Orkut datasets. This further confirms that EAGR incurs high memory usage as it needs to build the FP-Tree 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.

Figure 3.8 (b) and Figure 3.9 (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 the 1-hop query on Youtube and LiveJournal datasets and the 2-hop query 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 worst case of Livejournal, DBIndex is 9% slower than EAGR where the actual time difference remains tens of milliseconds. Another insight is that as expected, compared to Figure 3.8 (b), the 2-hop query runs faster for both algorithms. This is because there is more computation sharing for the 2-hop query.

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

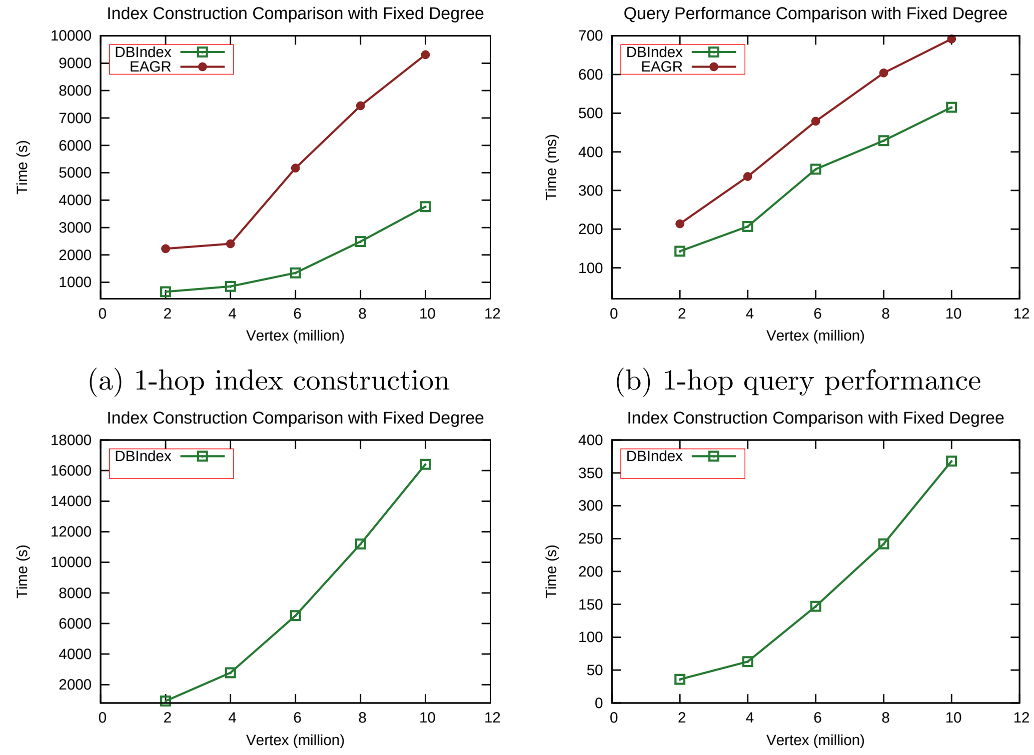
3.5.2.2 Synthetic datasets

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

Impact of Vertexes. First, we study how the performance changes when we fix the degree[12] at 10 and vary the number of vertexes from 2M to 10M. Figures 3.10 (a) and (b) show the execution time for index construction and query performance respectively. From the results, we can see that DBIndex outperforms EAGR in both index construction and query performance. For the graph with 10M vertexes and 100M edges, the DBIndex query time is less than 450 milliseconds. Moreover, when the number of vertexes changes from 2M to 10M, the query performance only increases 3 times. This shows that DBIndex is not only scalable, but offers acceptable performance. Figures 3.10 (c) and (d) show the performance of DBIndex in processing the 2-hop query. We notice that EAGR fails to run due to the high memory requirement. For instance, when vertex equals to 2M, the 2-hop query generates 90GB intermediate data, which exceeds the available memory.

Impact of Sparse Graphs. Our proposed DBIndex is effective when there is significant overlap between windows of neighboring nodes. As such, it is interesting to study how it performs for sparse graph where the vertexes may not share many common neighbors. In these experiments, we study the impact of degree when the graph is relatively sparse. We fix the number vertexes of 2M and vary the vertex degree from 5 to 30. Figures 3.11 (a) and (c) present the results on index construction for the 1-hop and the 2-hop queries respectively. For the 1-hop query, as degree

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



(a) 1-hop index construction

(b) 1-hop query performance

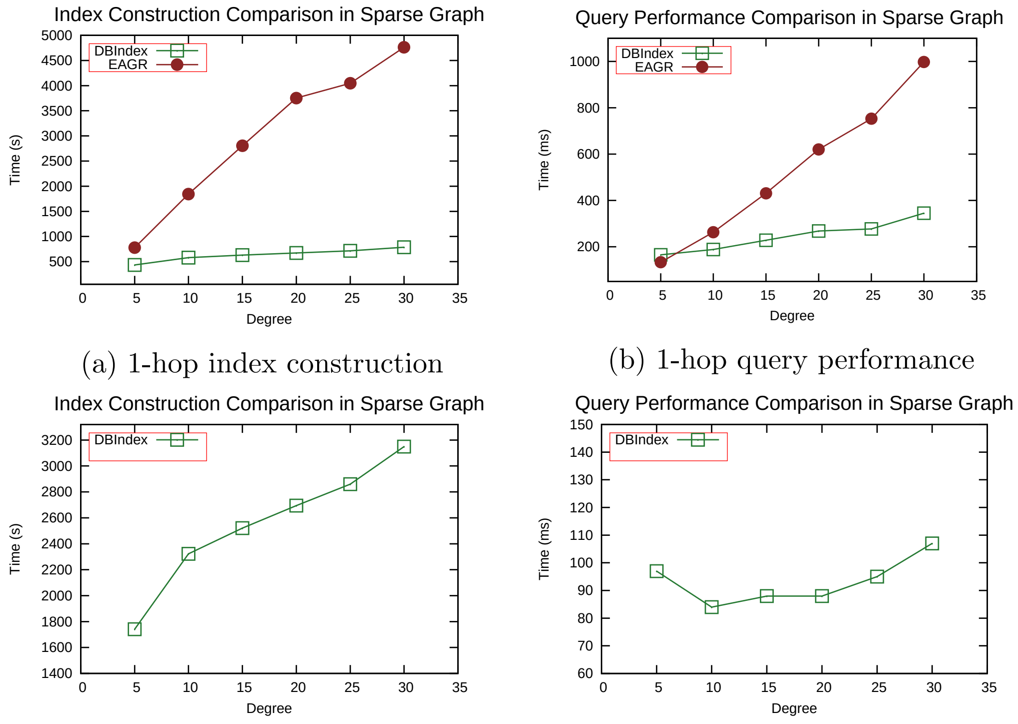
(c) 2-hop index construction

(d) 2-hop query performance

Figure 3.10: Impact of number of vertexes.

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 FP-Tree creation and reconstruction. For the 2-hop query, EAGR again fails to run due to the memory limitation. Therefore, we only show the results of DBIndex. In Figure 3.11 (c), the index construction 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.

Figures 3.11 (b) and (d) show the results on the query performance for 1-hop and 2-hop queries respectively. We observe a similar pattern for the index construction time: for the 1-hop query, the query time increases with increasing degree but at a much slower rate than EAGR. For the 2-hop query, we observe in Figure 3.11 (d) that the query performance of DBIndex hovers around 100ms, which is much smaller than that of the 1-hop query performance. This is because there are more dense blocks



(a) 1-hop index construction

(b) 1-hop query performance

(c) 2-hop index construction

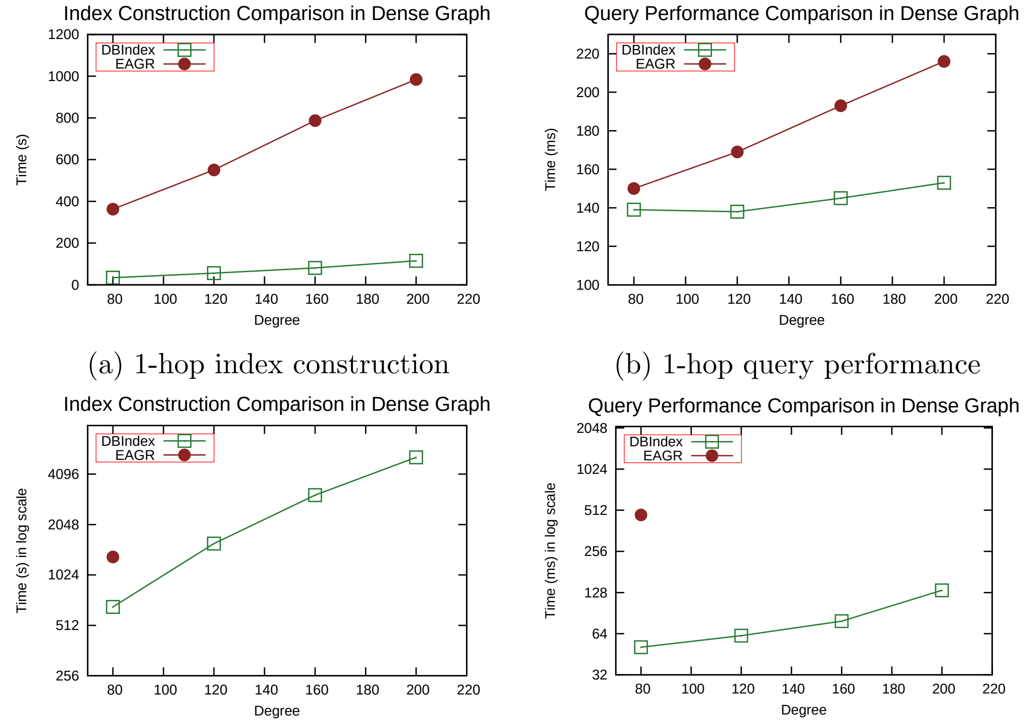
(d) 2-hop query performance

Figure 3.11: Impact of degree in sparse graphs with 2M vertexes.

in the 2-hop case, in which case the query time can be faster compared to the 1-hop case.

Impact of Dense Graphs. We study the impact of degree over very dense graphs with 200k vertexes when the degree changes from 80 to 200. Figures 3.12 (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 also performs well on dense graphs. As the degree increases, EAGR's performance degrades much faster than DBIndex. For the 2-hop query, as shown in Figures 3.12 (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 vertexes 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). Figures 3.12 (b) and (d) show the results on query performance for the 1-hop and the 2-hop queries respectively. The results are consistent with that for sparse graphs: DBIndex is superior over EAGR.

In summary, the insight we obtained is that the scalability of EAGR is highly limited by its large usage of memory. DBIndex achieves better scalability as it does not need to create a large amount of intermediate data in memory.



(a) 1-hop index construction

(b) 1-hop query performance

(c) 2-hop index construction

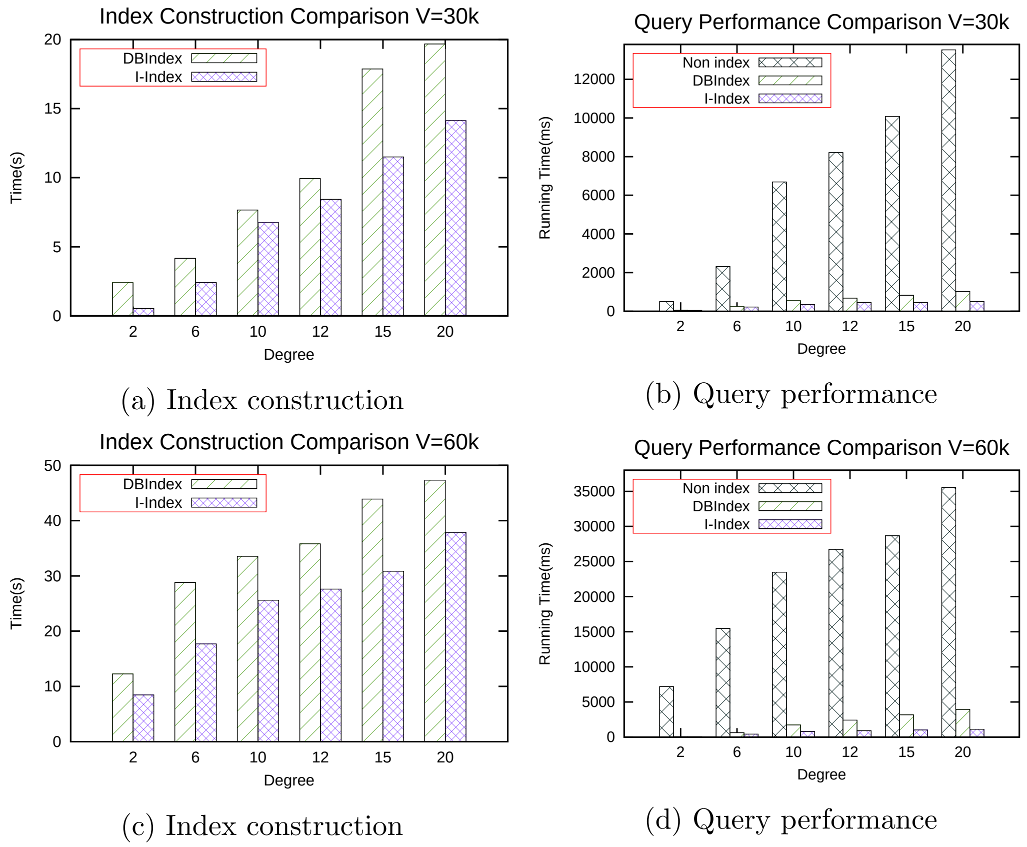
(d) 2-hop query performance

Figure 3.12: Impact of degree in dense graphs with 200K vertexes.

3.5.3 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. We evaluate the impact of degree 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 index construction time, as shown in Figures 3.13 (a) and (c), as the index size increases, both the index construction time of DBIndex



(a) Index construction

(b) Query performance

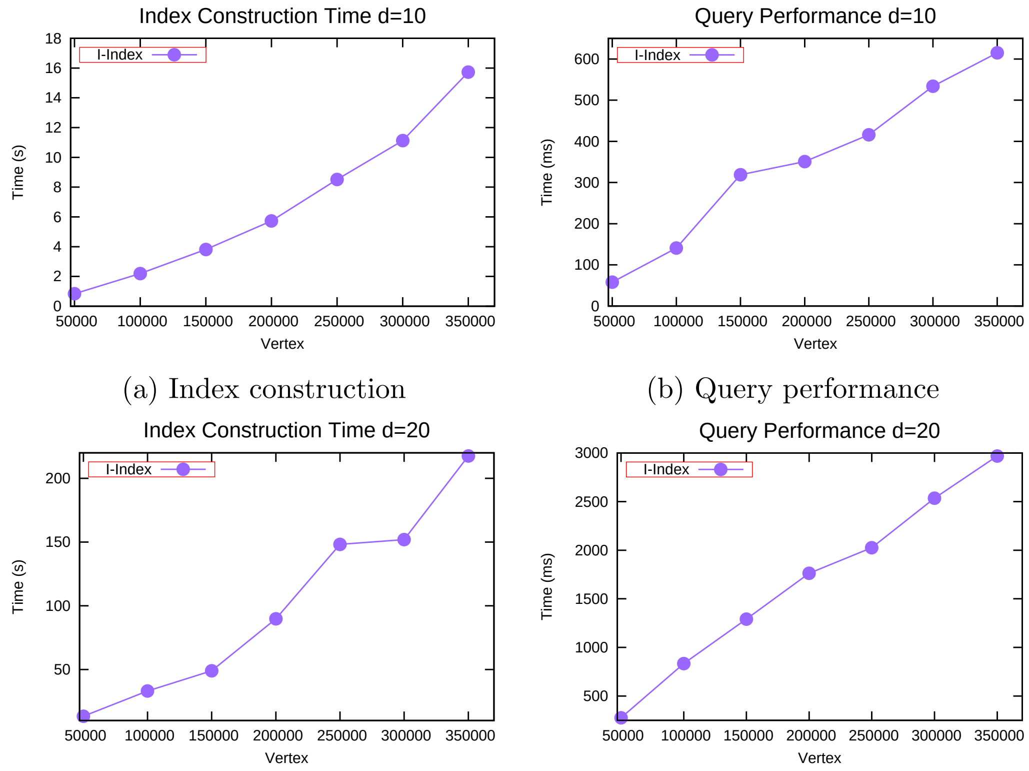
(c) Index construction

(d) Query performance

Figure 3.13: Impact of degree with the fixed number of vertexes.

and I-Index increase. However I-Index is more efficient than DBIndex, this is benefit from the inheritance optimization. We observe that the index construction time is almost the same as the non-indexed query time. In other words, we can use one query time to create the index which is able to subsequently provide much faster query processing. In terms of query performance, shown in Figures 3.13 (b) and (d), the non-index approach is in average 20 times slower than the indexed schemes. Meanwhile, I-Index outperforms DBIndex by 20% to 30%, which confirms the superiority of I-Index.

Impact of Number of Vertexes. Then, we study the effect of graph size by varying the number of vertexes from 50k to 350K. Figures 3.14 (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 as the number of vertexes



(a) Index construction

(b) Query performance

(c) Index construction

(d) Query performance

Figure 3.14: Impact of the number of vertexes 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.

increases and the construction time of a high degree graph is longer than that for low the degree graph. Figures 3.14 (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 vertexes increases. This shows that I-Index has good scalability.

Index Size. Figure 3.15 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 vertexes. 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

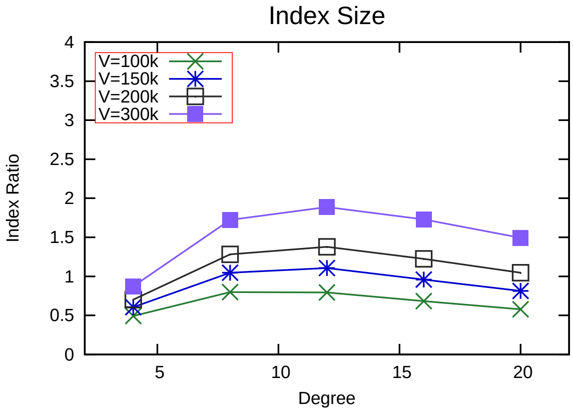


Figure 3.15: Index ratio of I-Index.

bends in the figure.

3.6 Summary

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

Chapter 4

k-Sketch Query: Neighborhood Analytics in Sequence Data

4.1 Introduction

Next, we explore the opportunity of neighborhood analytics in the sequence data. Sequence data is widely adopted to model the history of *subjects* which consists of temporally ordered *events*. For example, in sports application, the subject could be a player and his history would be the games he participated. Likewise, in stock application, the subject could be a stock and its history would be its daily closing prices. An important analytic task in the sequence data domain is to summarize the subject histories with phenomenal patterns. Such a task benefits a variety of applications such as computational journalism [20, 73], automatic fact checking [32, 60], and perturbation analysis [61].

An outstanding example of the neighborhood based pattern in the sequence data is *streaks* [73]. A streak refers to an aggregated period in a subject's history, such as "points scored by a player in the last ten straight games". Technically, a streak is a tuple $sk = \langle s, L, t, \overline{v} \rangle$ which represents the last L events of subject s at time t. $\overline{v}$ is the result of an analytic function (e.g., min, average) on all the events in the streak. A streak is essentially built on a **distance neighborhood** function. Formally, let $t_s(e)$ be the sequence number of the event e in the history of subject s. Then, a length-L streak of event e is defined by the following distance neighborhood function: $\mathcal{N}(s, L, e) = \{e_i | t_s(e) - t_s(e_i) \leq L\}$.

Based on the streaks, Zhang et. al. [73] studied the problem of summarizing a subject's history with *prominent streaks*. Prominent streaks are the skylines among all streaks of a subject in terms of streak size (i.e., L) and analytic value (i.e., $\overline{v}$). Hence, they are outstanding to represent the subject's history. However, there are two major drawbacks that limit the usability of prominent streaks in real applications. First, the prominent streaks generated by [73] may not be striking enough because they are derived from the historical data of a *single* subject without comparing to other subjects. For example, "Steve Nash has scored over 15 points in consecutive 10 games" is a prominent streak for "Steve Nash", but it is not striking given the fact that there are more than 90 players with better performance[1]. Second, the number of the prominent streaks can be overwhelming. Since prominent streaks in [73] are defined as skylines, a subject with n historical events may generate up to n streaks that are not dominated (i.e., prominent streaks). Therefore, there calls for a new method to automatically select a limited number of striking streaks which best summarize a subject's history.

In this chapter, we tackle the problem of effectively and efficiently summarizing a subject's history by applying a novel **comparison neighborhood** function to transform streaks to the *ranked-streaks*. Given a streak $sk = (s, L, t, \overline{v})$, a comparison neighborhood is first defined as $\mathcal{N}(sk) = \{sk_i | sk_i.L = sk.L \wedge sk_i.\overline{v} >= sk.\overline{v}\}$. Then a `count` function is applied on this neighborhood to generate the *rank* of sk. In other words, the neighborhood function groups the streaks with the same length and rank

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

them based on their aggregated values. Compared with streak, ranked-streak is able to capture the strikingness of a streak which is very newsworthy as evidenced in the following news excerpts:

1. (26 Feb 2003) With 32 points, Kobe Bryant saw his 40+ scoring streak end at **nine** games, tied with Michael Jordan for the **fourth** place on the all-time list[2].
2. (14 April 2014) Stephen Curry has made 602 3-pointer attempts from beyond the arc,... are the **10th** most in NBA history in a season (**82 games**)[3].
3. (28 May 2015) Stocks gained for the **seventh consecutive day** on Wednesday as the benchmark moved close to the 5,000 mark for **the first** time in seven years[4].
4. (9 Jun 2014) Delhi has been witnessing a spell of hot weather over the **past month**, with temperature hovering around 45 degrees Celsius, ... **highest** ever since 1952[5].
5. (22 Jul 2011) Pelican Point recorded a maximum rainfall of 0.32 inches for **12 months**, making it the **9th driest** places on earth[6].

In the above examples, each news theme is a *ranked-streak* which consists of five indicators: (I) a subject (e.g., Kobe Bryant, Stocks, Delhi), (II) a streak length (e.g., nine straight games, seventh consecutive days, past month), (III) an aggregate function on an attribute (e.g., minimum points, count of gains, average of degrees), (IV) a rank (e.g., fourth, first time, highest), and (V) a historical dataset (e.g., all time list, seven years, since 1952). The indicators (I)-(IV) are summarized in Table 4.1.

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

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

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

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

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

Table 4.1: Indicators of ranked-streaks.

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

Based on the ranked-streaks, we propose a novel k-Sketch query to effectively summarize a subject's history by leveraging a novel scoring function that chooses the best k ranked-streaks. Our scoring function considers two aspects: (1) we prefer the ranked-streaks that cover as many events as possible to represent a subject's history, and (2) we prefer the ranked-streaks that have better ranks[7] as they indicate more strikingness. Our objective is then to process the k-Sketch query for each subject in the domain.

We study the k-Sketch query processing under both offline and online scenarios. In the offline scenario, our objective is to efficiently discover the sketch for each subject from historical data. The major challenge lies in generating the rank information of streaks. Since the number of streaks is quadratic with respect to the number of events, enumerating all of them is not scalable. By leveraging the subadditivity among the upper bounds of streaks, we design two effective pruning techniques to facilitate efficient ranked-streak generation. Furthermore, we notice that generating exact sketches from ranked-streaks is computationally expensive. Thus, we design an efficient $(1 - 1/e)$-approximate algorithm by exploiting the submodularity of the k-Sketch query.

In the online scenario, fresh events are continuously fed into the system and our goal is to maintain the sketches for each subject uptodate. When a new event about subject s arrives, many ranked-streaks of various lengths can be derived. For each

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

derived ranked-streak, not only the sketch of s but also the sketches of other subjects may be affected. Dealing with such a complex updating pattern is non-trivial. To efficiently support the update while maintaining the quality of sketches, we propose a $1/8$-approximate algorithm which only examines $2k$ ranked-streaks for each subject whose sketch is affected.

Our contributions of this chapter are hereby summarized as follows:

- We study the neighborhood analytics in sequence data to tackle the problem of automatic summarization of a subject's history. We use both the distance and comparison neighborhood functions to model the ranked-streak, which is a common news theme in real-life reports but has not been addressed in previous works. We formulate the summarization problem as a k-Sketch query under a novel scoring function that considers both strikingness and coverage.

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

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

The rest of this chapter is organized as follows. In Section 4.2, we formulate the

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

k-Sketch query. Section 4.3 presents the algorithms for processing the k-Sketch query in the offline scenario. Section 4.4 describes the algorithms for maintaining k-Sketches in the online scenario. In Section 4.5, comprehensive experimental studies on both the efficiency and the effectiveness of our algorithms are conducted. Section A.1 discusses the extension of our methods. Finally, Section 4.6 concludes our paper.

4.2 Problem Formulation

Let S denote a set of subjects which are of potential interests to journalists. For example, S can refer to all the players or teams in the NBA application. We use $e_s(t)$ to denote an event about subject s, where t is 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 ID. Consecutive events of the same subject can be grouped as a *streak*:

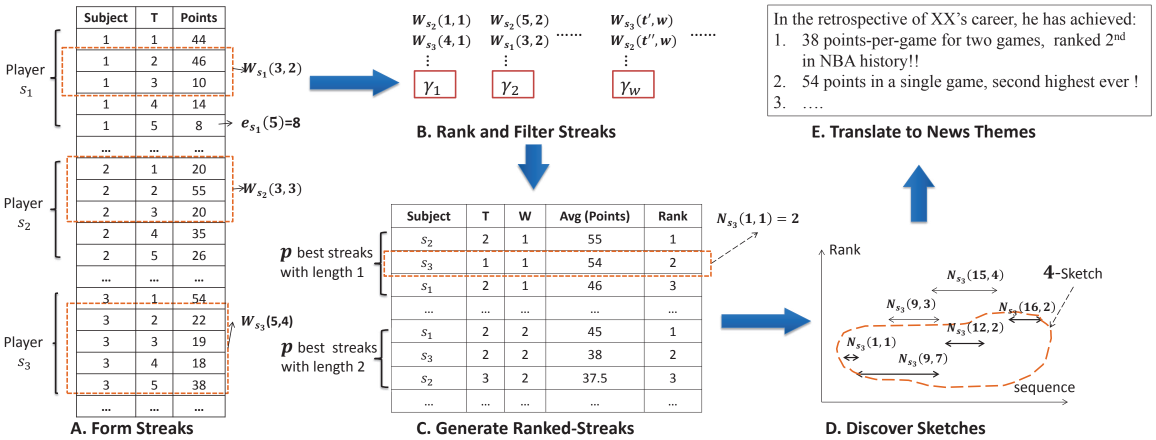
Definition 4.2.1 (Streak [73])**.** Let w be a streak length, a streak $W_s(t, w)$ refers to w consecutive events of subject s ending at sequence t, i.e., $W_s(t, w) = \{e_s(t - w + 1), ..., e_s(t)\}$.

If a subject s has $|\mathbb{H}_s|$ events, then there are $\binom{|\mathbb{H}_s|}{2}$ possible streaks[9]. Given an aggregate function f, events in a streak can be aggregated to a numerical value $\overline{v}$ as:

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

Common aggregate functions include `sum`, `avg`, `count`, `min` and `max`. In this thesis, we only consider a single aggregate function. Multiple aggregate functions can be simply processed independently.

[9] This number is derived by constructing a streak using start and end sequence IDs.



In the retrospective of XX's career, he has achieved:
1. 38 points-per-game for two games, ranked 2nd in NBA history!!
2. 54 points in a single game, second highest ever !
3. ….

Figure 4.1: An illustration of k-Sketch query processing. (A): various streaks are formed based on events' sequence IDs. (B): streaks with rank greater than p are filtered. (C): ranked-streaks for each streak length. (D): k-sketches are discovered for each subject from their ranked-streaks. (E): newsworthy summary of a subject can be generated from its k-sketches.

Example 4.2.1. Figure 4.1 (A) illustrates examples of *streak*s of three NBA players. Each event records the points scored by a player in a game. In the figure, the streak $W_{s_1}(3,2)$ 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 a streak to measure its strikingness. 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*", where the rank is 3. Let W be a length-w streak, and $\gamma_w(W.\overline{v})$ be the rank of W by comparing it with all other length-w streaks on their aggregate values. Let p be a predefined threshold to indicate whether a streak is striking (i.e., top-p). These concepts lead to our definition of *Ranked-streak*:

Definition 4.2.2 (Ranked-streak)**.** A streak $W_s(t,w)$ can be transformed to a ranked-streak, denoted by $N_s(t,w)$, if its rank $\gamma_w(W_s(t,w).\overline{v}) \leq p$, where p is a user-defined threshold.

Example 4.2.2. In step (B) of Figure 4.1, we group the streaks based on their lengths. Each streak is associated with a rank value γ_w. If the rank is greater than the threshold p, the associated streak is pruned. Otherwise, it is considered as a ranked-streak. In step (C) of Figure 4.1, we present the ranked-streaks in the tabular format. Each ranked-streak contains the rank computed from step (B). For instance, among all ranked-streaks with length 1, $N_{s_3}(1,1)$ (with value 54) is ranked 2^{nd} because its aggregated value is smaller than that of another streak $N_{s_2}(2,1)$ (with value 55).

Let $\mathbb{N}_s$ be the set of ranked-streaks of a subject s. Since there are at most p ranked-streaks for each possible streak lengths, the size of $\mathbb{N}_s$ can be as large as $p|\mathbb{H}_s|$ which is still overwhelming for summarizing the subject's history. To control the output size while maintaining the quality of the summary, we aim to find a subset of k ranked-streaks from $\mathbb{N}_s$ which best summarize the history of s. We name such a subset a k-**Sketch**.

To measure the quality of the selected ranked-streaks, we define a scoring function $g(\cdot)$. The design of g gives rise to two concerns. First, g prefers ranked-streaks covering as many of the subject's historical events as possible. This is because a higher coverage indicates fewer missing historical events in the summary. Second, g needs to assign a higher score to the ranked-streaks with better ranks. This is because a better rank indicates higher strikingness which implies that the news themes generated would be more eye-catching. For instance, we often care more about who the top scorer in NBA history is rather than who is ranked 50^{th}.

To address these two concerns, we define g as follows: let $\mathbb{X}_s$ be a set of ranked-streaks 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, ranked-streaks with a poor coverage contribute to a low 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 ranked-streak in $\mathbb{X}_s$ changing to a better rank increases $R(\mathbb{X}_s)$. The value ranges of $C(\mathbb{X}_s)$ and $R(\mathbb{X}_s)$ are guaranteed to be in $[0, 1]$. 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 ranked-streaks that cover most of the subject's history. If α is low, it indicates that users prefer more striking ranked-streaks.

With Equation 4.1, we then define the *k-Sketch Query* as follows:

Definition 4.2.3 (k-Sketch Query)**.** Given a parameter k, k-Sketch Query aims to, for each subject s, find a subset SK_s from the ranked-streaks of s (i.e., $SK_s \subseteq \mathbb{N}_s$), s.t. $|SK|_s = k$ and $g(SK_s)$ is maximized.

Example 4.2.3. In step (D) of Figure 4.1, we show a collection of ranked-streaks 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 ranked-streak is represented by a line segment, covering consecutive events. When $k = 4$, four of the ranked-streaks are selected as the 4-Sketch.

Before we move on to the algorithmic part, we first list the frequently used notations in Table 4.2. For ease of presentation, we present our techniques using *avg* as the default aggregate function. Extending our techniques to other aggregate functions is addressed in Section A.1.

4.3 Offline k-Sketch Query Processing

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 k-Sketch query processing consists of two major steps: first generating the ranked-streaks of each subject (i.e., $\mathbb{N}_s$), and then discovering the sketches among those ranked-streaks.

Table 4.2: Notations used in this chapter.

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

4.3.1 Ranked-Streak Generation

Generating ranked-streak for each subject is computationally expensive. In order to generate accurate ranks for selecting ranked-streaks, a brute-force approach needs to evaluate all the streaks with every possible length. Since there could be $\binom{\mathbb{H}_s}{2}$ streaks 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 streaks to generate $\mathbb{N}_s$. The intuition behind is that the upper bound value of streaks with longer lengths can be estimated from those with shorter lengths. This means that we can compute streaks with increasing lengths, and as the shorter streaks are computed, the longer streaks not in $\mathbb{N}_s$ can be pruned. To realize such an intuition, we design the ranked-streak generation algorithm by adopting two novel streak-based pruning methods which exploit the *subadditivity* property among streaks with different lengths.

4.3.1.1 Overview of streak pruning

For each subject, its streaks can be grouped by lengths. Our ranked-streak generation algorithm gradually evaluates a subject's streak from a shorter length to a longer length. To support efficient pruning, we define two concepts, namely *visiting-streak bound* (J_s) and *unseen-streak bound* (M_s). In particular, $J_s(w)$ is the upper bound of all the streaks about subject s with length w, i.e., $\forall w, J_s(w) \geq \max\{W_s(t, w).\overline{v} | t > w\}$. $M_s(w)$ is the upper bound of all the streaks about subject s with a length larger than w, i.e., $M_s(w) \geq \max\{J_s(w_1) | w_1 > w\}$. These two bounds will be used for streak pruning and we will present how to derive these two bounds in Sections 4.3.1.2 and 4.3.1.3.

The overview of ranked-streak generation algorithm is presented in Algorithm 6. We maintain two global structures WI and β (Lines 1-2). For each length w, $WI(w)$ stores the top-p streaks with length w among all the subjects and $\beta(w)$ is the p^{th} value among the streaks in $WI(w)$. In other words, $\beta(w)$ serves as a lower bound value. A streak with length w is a ranked-streak only if its aggregate value is larger than $\beta(w)$. A priority queue Q (Line 3) is used to provide an access order among subjects. Each element in the queue is a triple (s, w, q), where s is a subject, w indicates the next streak length of s to evaluate, and q denotes the priority. We use the unseen-streak bound (i.e., $M_s(w)$) as the priority during every iteration (Lines 13-14). Intuitively, a subject with higher $M_s(w)$ is more likely to spawn new streaks that can increase β and benefit the subsequent pruning. During initialization, we insert, for each subject, an entry with length 1 and priority infinity into Q.

In each iteration, we pop the streak with the highest priority (Line 4). Then, we compute the visiting-streak bound $J_s(w)$ for the subject and determine whether all the length-w streak about subject s can be pruned (Lines 5-6). If the bound $J_s(w)$ is no greater than $\beta(w)$, then all these streaks can be ignored. Otherwise, all the length-w streaks of s are computed to update $WI(w)$ and $\beta(w)$ accordingly

Algorithm 6 Ranked-streak Generation Overview

```
1:  $WI() \leftarrow \{\}$ // top-$p$ streaks for each length
2:  $\beta \leftarrow \{\}$ // smallest scores in $WI$ for each length
3:  $Q \leftarrow \{(s, 1, +\infty) | s \in S\}$
4:  while $(s, w, q) \leftarrow Q.pop()$ do
5:      compute $J_s(w)$
6:      if $J_s(w) > \beta(w)$ then
7:          for $t \in w...|\mathbb{H}_s|$ do
8:              Update $WI(w)$, $\beta(w)$, and $J_s(w)$
9:          end for
10:     end if
11:     compute $M_s(w)$ whose value relies on $J_s(w)$
12:     if $M_s(w) > \min\{\beta(w') | w < w' \leq |\mathbb{H}_s|\}$ then
13:         $q \leftarrow M_s(w)$
14:         $Q.push(s, w + 1, q)$
15:     end if
16: end while
17: return $WI$
```

(Lines 7-9). In the next step, we estimate the unseen-streak 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 streak and all the streaks about subject s with lengths larger than w can be pruned (Lines 11-12). Otherwise, we set the priority of M_s to q, and push the triple $(s, w + 1, q)$ back to Q (Lines 13-14). The algorithm terminates when Q becomes empty.

4.3.1.2 Visiting-streak pruning

In Algorithm 6, we need to compute the visiting-streak bound $J_s(w)$ to facilitate pruning all length-w streaks associated with subject s. Our idea is that suppose we have successfully derived the bounds for streaks with smaller lengths, we can use them to estimate the bounds of larger streaks. We formulate it as the subadditivity property and use *avg*[10] as the aggregate function for illustration:

[10] 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.

Theorem 4.3.1 (Subadditivity (for *avg*)).

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

Proof. Given any streak $W = W_s(t,w)$ and a value $w_i \in (0,w)$, we can split the streak into two non-overlapping sub-streaks 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 $wW.\overline{v} = w_iW_1.\overline{v} + (w-w_i)W_2.\overline{v}$. Since $J_s(w_i)$ and $J_s(w-w_i)$ are two visiting-streak bounds, then $W_1.\overline{v} \leq J_s(w_i)$ and $W_2.\overline{v} \leq J_s(w-w_i)$ must hold for any t. It then follows that, for any t, $wW_s(t,w).\overline{v} \leq w_iJ_s(w_i) + (w-w_i)J_s(w-w_i)$. Therefore, $w\max_t\{W_s(t,w).\overline{v}\} \leq w_iJ_s(w_i)+(w-w_i)J_s(w-w_i)$, which leads to the theorem. □

With Theorem 4.3.1, we can estimate $J_s(w)$ by any pair $J_s(w_i)$ and $J_s(w-w_i)$, $\forall w_i < w$. Let w^* corresponds to the tightest $J_s(w)$, then w^* can be formulated as:

$$w^* = \operatorname*{argmin}_{w_i \in (0,w)} \frac{w_iJ_s(w_i) + (w-w_i)J_s(w-w_i)}{w} \tag{4.3}$$

A naive solution to compute w^* is to enumerate every possible w_i. However, such a solution has a worst time complexity of $O(|\mathbb{H}_s|)$ for subject s. To quickly compute w^* without enumerating all w_i, we apply a continuous relaxation to Equation 4.3 as follows: Let $G_s(w_i) = w_iJ_s(w_i)\ \ \forall w_i$, then Equation 4.3 is equivalent to:

$$w^* = \operatorname*{argmin}_{w_i \in (0,w)}\{G_s(w_i) + G_s(w-w_i)\} \tag{4.4}$$

Let $L_s(w_i)$ be a continuous and smooth fitting function of $G_s(w_i)$ for $w_i \in [1, w-1]$. Equation 4.3 can then be relaxed by replacing $G_s(w_i)$ with $L_s(w_i)$, which produces the solution at $w^* = w/2$ if $L_s(w_2)$ is convex and $w^* = 1$ if $L_s(w_i)$ is concave. We have observed that the convexity and concavity for $L_s(\cdot)$ when approximating $G_s(\cdot)$ hold over all aggregate functions. For example, we use 800 game records of a NBA player and plot the function G_s and L_s under various aggregate functions in Figures 4.2. In Figure 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 Figure 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 4.3.1. Thus we have the visiting-streak bound stated as:

Theorem 4.3.2 (Visiting-Streak Bound)**.** Given a length $w > 1$, let $J_s(w)$ be:

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

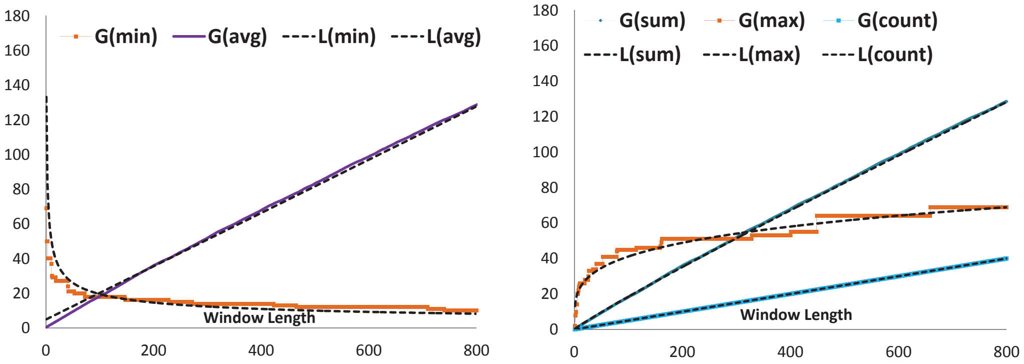
then $J_s(w)$ is a visiting-streak bound, i.e. $J_s(w) \geq \max_t\{W_s(t,w).\overline{v}\}$

Proof. By substitute $w_i = 1$ to the right hand side of Theorem 4.3.1, we see this theorem holds. □

In Algorithm 6, $J_s(w)$ is computed incrementally. Initially, $J_s(1)$ is set to be the single event of s with highest value. Then, as the subject s is processed, $J_s(w)$ is computed by Theorem 4.3.2. In the case when visiting-streak pruning fails, we update $J_s(w)$ to the maximum length-w streak of s to further tighten the bound.

4.3.1.3 Unseen-streak pruning

Unseen-streak pruning utilizes $M_s(w)$ to check if it is necessary to evaluate any streak with length $w' \in (w, |\mathbb{H}_s|]$. 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:



(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.

Theorem 4.3.3 (Unseen-Streak Bound)**.** Given that $J_s(1), \ldots, J_s(w-1)$ have already been computed, let $M_s(w)$ be:

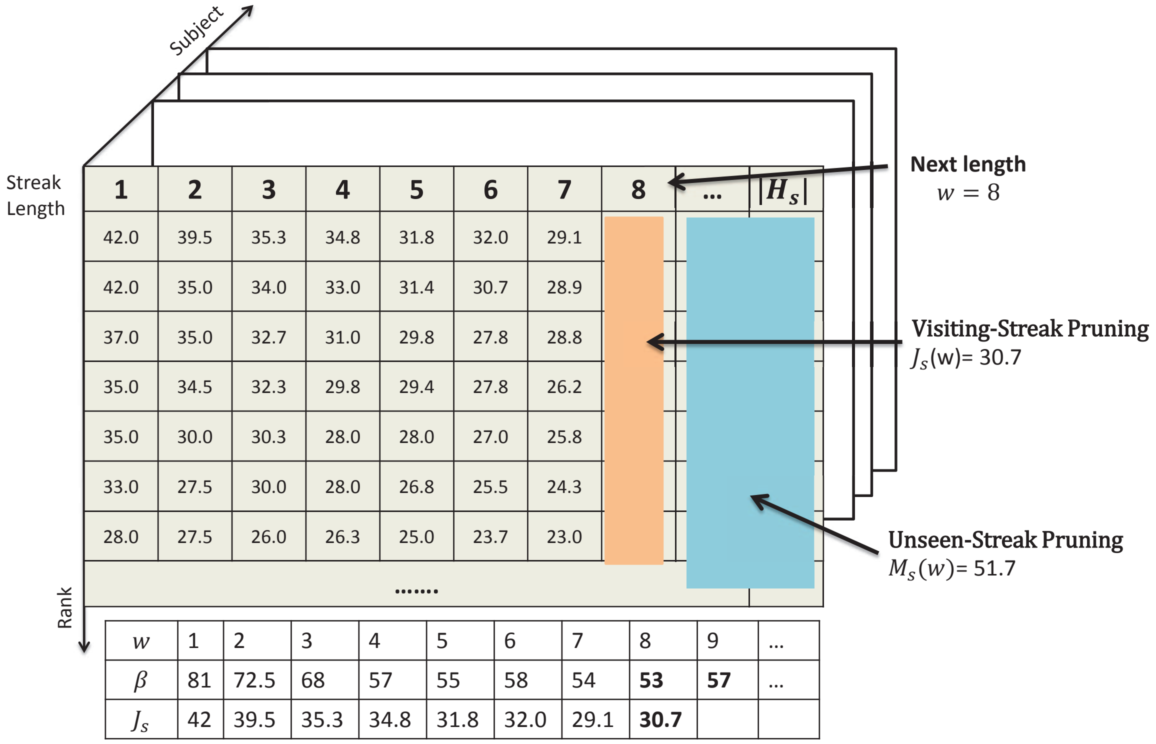
$$M_s(w) = J_s(w) + \min\{\frac{1}{2}J_s(1), \frac{w-1}{w+1}J_s(w-1)\} \tag{4.6}$$

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

Proof. First, given any integer $k \geq 1$, we see that $J_s(kw_i) \leq J_s(w_i)$ by making use of Theorem 4.3.1 in a simple induction. 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:

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

On one hand, since $xJ_s(x)$ is a monotone increasing function with respect to x, 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, since $J_s(x \bmod w) \leq J_s(1)$ and $\frac{x \bmod w}{x} \leq \frac{1}{2}$ for $x > w$, we have $(x \bmod w)\frac{J_s(x \bmod w)}{x} \leq \frac{1}{2}J_s(1)$. Therefore,



1	2	3	4	5	6	7	8	…	$\lvert H_s \rvert$
42.0	39.5	35.3	34.8	31.8	32.0	29.1			
42.0	35.0	34.0	33.0	31.4	30.7	28.9			
37.0	35.0	32.7	31.0	29.8	27.8	28.8			
35.0	34.5	32.3	29.8	29.4	27.8	26.2			
35.0	30.0	30.3	28.0	28.0	27.0	25.8			
33.0	27.5	30.0	28.0	26.8	25.5	24.3			
28.0	27.5	26.0	26.3	25.0	23.7	23.0			
			....						

w	1	2	3	4	5	6	7	8	9	…
β	81	72.5	68	57	55	58	54	**53**	**57**	…
J_s	42	39.5	35.3	34.8	31.8	32.0	29.1	**30.7**		

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

$(x \bmod w)\frac{J_s(x \bmod w)}{x}$ is smaller than both $\frac{w-1}{w+1}J_s(w-1)$ and $\frac{1}{2}J_s(1)$, which implies it is smaller than the minimum of the two values. Then it naturally leads to Theorem 4.3.3.

□

To utilize $M_s(w)$, 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 maintain an interval tree [22] on β to support efficient checking.

Theorem 4.3.4. Each streak returned by Algorithm 6 has a rank no greater than p.

The proof is quite straightforward according to the descriptions of Algorithm 6, visiting-streak bound and unseen-streak bound. Thus, the details are omitted.

Example 4.3.1. We use Figure 4.3 to illustrate our pruning techniques on a subject

s. Each column in the table represents a streak length and the cells contain the average values among different streaks with different lengths. For example, the cell in the second row and third column refers to the second largest average value among the streaks with length 3. Algorithm 6 essentially accesses the streaks in increasing order of the length. Suppose we are about to estimate the upper bound for streak with length 8. At this point, the values of β and J_s are depicted in the figure. Based on the visiting-streak bound, 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 streaks with lengths 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 streak enumeration.

4.3.2 k-Sketch Discovery

After the ranked-streaks are obtained, the second step of the k-Sketch query processing is to, for each subject s, select k ranked-streaks to maximize Equation 4.1 (i.e., g).

Our goal of optimizing g is related to the Partly Interval Set Cover (PISC) problem. 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 [27] 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 [25], where current polynomial time solutions remain unknown. To facilitate scalable k-Sketch discovery, we provide an efficient $(1 - 1/e)$-approximate algorithm by exploiting the submodularity property[11] of the scoring function. Our idea is that since g is not submodular, it is not easy to directly maximize it. However, we are able to transform g to another submodular function g' s.t. maximizing g' would result in

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

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 $\eta_2 = (1-\alpha)/(pk)$. Given k, s, p and g, g' is uniquely defined. We have the following theorem which links g' to g:

Theorem 4.3.5. If A^* is the optimal solution under g', then A^* is also the optimal solution under 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 Equation 4.1 and Equation 4.7. Then, we prove the theorem by contradiction: Let A^* be the optimal solution under g'. If A^* is not optimal under g, then $\exists B^*$ s.t. $g(B^*) > g(A^*)$. Since $|A^*| = k$, then $g(A^*) = g'(A^*)$. Similarly, since $|B^*| = k$, $g'(B^*) = g(B^*)$. As $g(B^*) > g(A^*)$, it follows $g'(B^*) = g(B^*) > g(A^*) = g'(A^*)$, which contradicts with A^*'s optimality under g'. □

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

Theorem 4.3.6. Given a ranked-streak 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 can apply the greedy selection algorithm [52] to efficiently discover the sketches, which guarantees a $(1-1/e)$-approximation ratio. The greedy scheme is presented in Algorithm 7. During each step (Lines 4-7), the algorithm picks the best ranked-streak among the remaining ranked-streaks (i.e., $N_s \setminus SK_s$) to maximizes g'. The algorithm stops at the k^{th} iteration.

Algorithm 7 Greedy Sketch Discovery

```
for $s \in S$ do
    $\mathbb{N}_s \leftarrow$ ranked-streaks of subject $s$
    $SK_s \leftarrow \{\}$
    for $t \in [1, k]$ do
        $x^* \leftarrow \text{argmax}_{x \in (\mathbb{N}_s \setminus SK_s)} g'(SK_s \cup \{x\})$
        $SK_s \leftarrow SK_s \cup \{x^*\}$
    end for
end for
return $SK_s \ \forall s \in S$
```

4.4 Online k-Sketch Maintenance

In the offline scenario, all the events are assumed to be available at the time of k-Sketch query processing. On the contrary, in the online scenario, events arrive incrementally. Given an arrival event, our objective is to maintain the k-Sketch for each subject uptodate. Since events may arrive at a high speed, such a maintenance step has to be done efficiently.

Similar to the offline scenario, we maintain an index WI to keep track of the top-p streaks for all the possible streak lengths. To handle a newly arrived event $e_s(t)$, a naive solution would first generate all the streaks containing $e_s(t)$, (i.e. $W_s(t, w'), w' \in (1, t]$), and then update WI accordingly. Last, Algorithm 7 is invoked to re-compute the sketches. However, there are t associated streaks 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 challenges.

To achieve instant sketch maintenance, we propose two techniques: *online streak pruning* and *sketch update*. *Online streak pruning* tries to reduce the number of streaks evaluated in generating ranked-streaks. After obtaining the ranked-streaks, we need to update the affected sketches. As we shall see later, given a ranked-streak $N_s(t, w)$, not only the sketch of subject s but also the sketches of other subjects could be affected. Although we provide a solution with a $(1 - 1/e)$ approximation in the offline scenario, maintaining sketches to achieve the same approximation ratio is difficult in the online scenario [4, 6]. Therefore, we propose a 1/8-approximate solution which updates a sketch in $O(k)$ time.

Algorithm 8 Online k-Sketch Maintenence

```
Input: $e_s(t) \leftarrow$ arrival event
 1: $WI()$// top-$p$ streaks for each length
 2: $\beta()$// smallest score in $WI$ for each streak 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:         SketchUpdate($N_s(t, w)$)
 7:     end if
 8:     compute $P_s(w)$
 9:     break if $P_s(w) \leq \max\{\beta(w')|w < w' \leq t\}$
10: end for
```

Before we present *online streak pruning* and *sketch update*, Algorithm 8 first depicts the overview of our online solution against a new event $e_s(t)$. We iteratively examine streaks which contain $e_s(t)$ (i.e.,$W_s(t, w)$ in Line 3). Then we update the sketches which are affected by inserting $W_s(t, w)$ into WI (Lines 4-7). Before continuing to examine the next streak length w, we compute the maximum score (i.e., $P_s(w)$) of all streaks which have not been evaluated. If $P_s(w)$ is smaller than all $\beta(w'), w' \in (w, t]$, we can safely stop processing since no further streaks could cause any sketches to change.

4.4.1 Online Streak Pruning

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

Theorem 4.4.1 (Online-Streak Bound)**.** Let $W_s(t,1)$,. . ., $W_s(t,w)$ be the w streaks 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-streak bound. Then $P_s(w)$ is the online-streak bound, i.e., $P_s(w) \geq \max\{W_s(t,x).\overline{v}|x \in (w,t]\}$.

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

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

Note that $J_s(x-w) \geq W_s(t-w,x-w).\overline{v}$, and $yJ_s(y)$ monotonically increases with respect to 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:

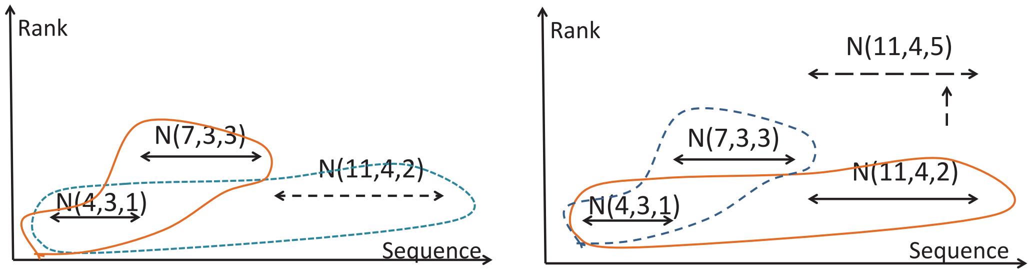
$$\frac{(x-w)W_s(t-w,x-w).\overline{v}}{x} \leq min\{\frac{t-w}{w+1}J_s(t-w), \frac{t-w}{t}J_s(1)\}$$

which leads to Theorem 4.4.1. □

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 can leverage $\frac{t-w}{w+1}J_s(t-w)$ to perform efficient pruning.

4.4.2 Sketch Update

Once we obtain a streak $W_s(t, w)$ which causes changes in the $WI(w)$, two kinds of sketch updates may occur. The first update is directly on the sketch of s, which we refer to as *active update*. The second update is on the sketches for other subjects. This happens when some of their ranked-streaks 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 approximation ratio on their qualities. We demonstrate the two types of updates in the following example.



(a) Active update cased by the new ranked-streak $N(11, 4)$

(b) Passive update cased by the existing ranked-streak $N(11, 4)$

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

Example 4.4.1. Suppose $k = 2$ and we maintain a 2-Sketch for each subject. As shown in Figure 4.4(a), when the ranked-streak $N(11, 4)$ 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 the **active update**. In Figure 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 the **passive update**. If passive update is not handled, the rank of $N(11, 4)$ may continue to be pushing up and may eventually 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 a better quality.

A naive approach to handle these updates is to run Algorithm 7 for each affected subject. This maintains a $(1 - 1/e)$-approximation ratio but incurs a high computational cost. 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-approximate solution with only $O(k)$ ranked-streaks being accessed for each affected subject.

In particular, we maintain two size-k sets S_1 and S_2. S_1 maintains the k best ranked-streaks which collectively cover most events whereas S_2 maintains k ranked-streaks with best ranks. When performing an active update for a streak $N_s(t, w)$, we check if $N_s(t, w)$ could replace any ranked-streak in S_1 to generate a better cover. Meanwhile, we select the new k best ranked-streaks into S_2. After S_1 and S_2 are updated, we perform the greedy selection from $S_1 \cup S_2$. During a passive update, S_1 is not affected. We simply update S_2 to be the new k best ranked-streaks. Afterwards, the new sketch is obtained by performing a greedy selection from $S_1 \cup S_2$. Algorithm 9 presents both the active and passive updates.

Algorithm 9 SketchUpdate

```
Input: $N_s(t, w)$
Active update for the subject $s$
$S_1$: $k$ ranked-streaks with best cover
$S_2$: $k$ ranked-streaks 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 $k$ ranked-streaks with best ranks
$S \leftarrow greedy(S_1 \cup S_2)$

--------------------------------------------------
Passive update for an affecting subject $s'$
$S_2 \leftarrow$ new $k$ ranked-streaks with best ranks for $s'$
$S \leftarrow greedy(S_1 \cup S_2)$
```

We state the quality of our sketch update strategy in the following theorem:

Theorem 4.4.2 (Approximation Ratio for Sketch Update)**.** Each sketch maintained by Algorithms 8 achieves an at least 1/8-approximation to the optimal solution.

Proof. First, we observe that S_2 always keeps the ranked-streaks with optimal ranks. Second, we note that S_1 maintains the streaks with 1/4-approximate coverage as shown in [54].

Let OPT_C be the optimal k streaks that best covers s's history; Let $C()$ be the number of events a set of streaks cover. Similarly, let OPT_R be the optimal k streaks with highest ranks; Let $R()$ be the summation of ranks of all members in a ranked-streak 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 $S_1 \cup S_2$, $g'(SK_s) \geq max(g'(S_1), g'(S_2)) \geq 1/8g'(S_s^*)$. As a result, our algorithm always ensures at least 1/8-approximation for each sketch. □

4.5 Experiments

In this section, we study our solutions for k-Sketch query processing in both offline and online scenarios using the following four real datasets. The statistics of these datasets are summarized in Table 4.3.

NBA[12] 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 [46] 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 [19] 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 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[13] and contains 2.3M events.

Table 4.3: Statistics of datasets used in the experiments.

DataSet	Total Events	Total Subjects	Longest Sequence
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 threshold of the ranked-streaks, (2) $k \in [20, 100]$, which refers to the size of a sketch and (3) $h \in [20, 100]$, which refers to the percentage of historical events for scalability test, i.e. $|\mathbb{H}|h\%$ events are used in the experiments. We do not evaluate the performance with respect to α since α does not impact the running time. 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 implemented in Java 7.

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

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

4.5.1 Offline k-Sketch Query Processing

Our offline k-Sketch query processing algorithms consist of two functional components, *Ranked-streak Generation* and *Sketch Discovery.* In the ranked-streak generation, we report the performance with varying p and h. In the sketch discovery, we report the performance with varying k.

4.5.1.1 Ranked-streak generation algorithms

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

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

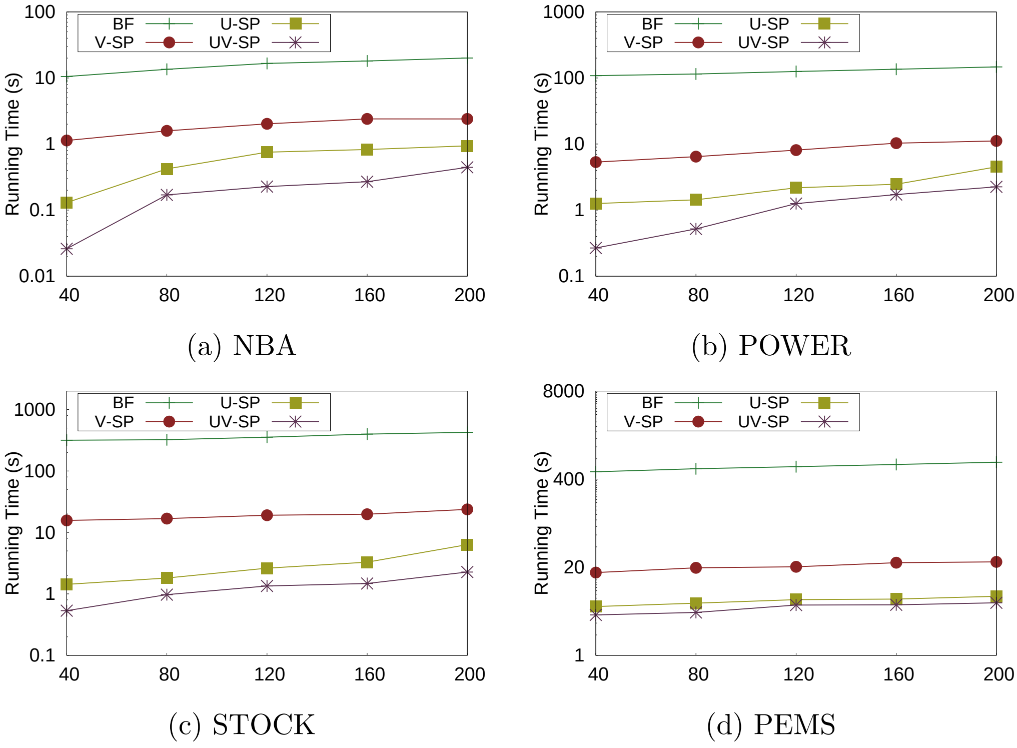
Visiting Streak Pruning (V-SP). V-SP only adopts the *visiting-streak* bound for pruning.

Unseen Streak Pruning (U-SP). U-SP only adopts the *unseen-streak* bound for pruning.

Unseen+Visiting Streak Pruning (UV-SP). UV-SP adopts both *unseen-streak* and *visiting-streak* bounds for pruning.

4.5.1.2 Ranked-streak generation varying p

The running time of the four algorithms in ranked-streak generation with respect to p is shown in Figure 4.5. It is evident that when p increases, more ranked-streaks are qualified and thus all four algorithms require more computation time. The effect of the two proposed streak-based pruning techniques can also be observed from the figure. The insight is that the unseen-streak pruning plays a more important role in reducing the running time. Furthermore, when both pruning techniques are used, our method (UV-SP) achieves at least two orders of magnitude of performance improvement.



(a) NBA

(b) POWER

(c) STOCK

(d) PEMS

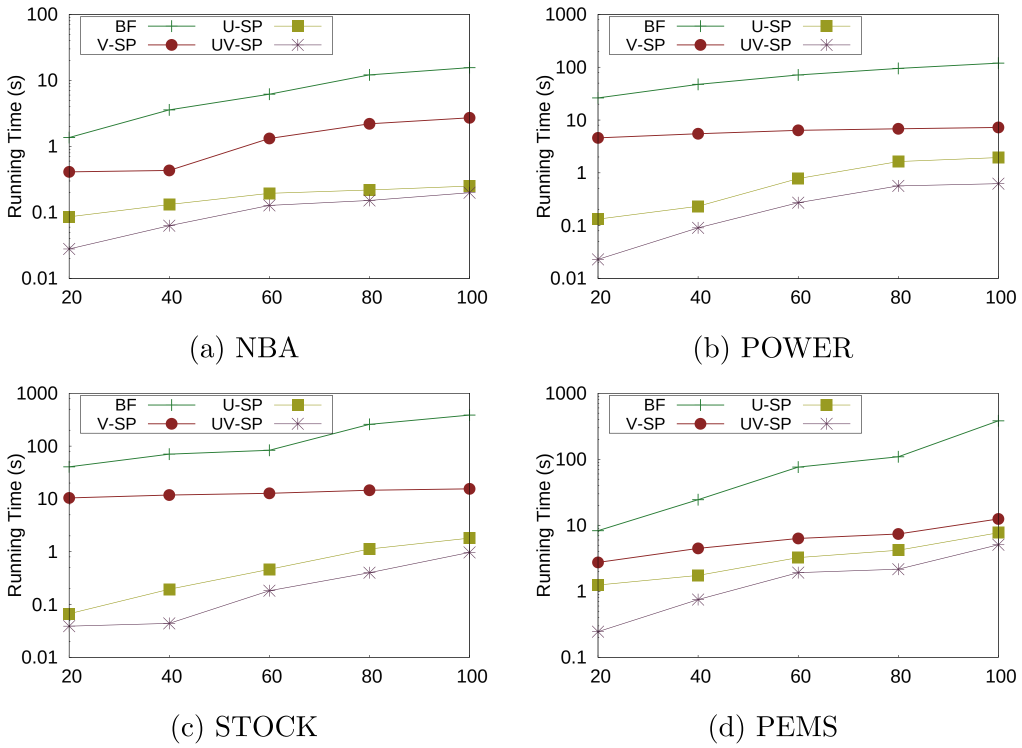
Figure 4.5: Ranked-streak generation in the offline scenario with varying p.

4.5.1.3 Ranked-streak generation varying h

We then study the performance of the four algorithms with respect to the number of events and report the results in Figure 4.6. As the figure shows, when h increases, the running times for all four algorithms increase. This is because more streaks need to be evaluated. Again, pruning-based methods are much more efficient than the baseline method. When both pruning methods are adopted, our method (UV-SP) obtains hundreds of times faster than the baseline method.

4.5.1.4 Sketch discovery varying k

After ranked-streaks 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 the four datasets are presented in Table 4.4. The table indicates that the running time of the greedy algorithm increases proportionally to k. This is because



(a) NBA

(b) POWER

(c) STOCK

(d) PEMS

Figure 4.6: Ranked-streak generation in the offline scenario with varying h.

Table 4.4: 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

the complexity of the greedy algorithm is $O(k\Sigma_s|\mathbb{N}_s|)$. Since PEMS is the largest dataset with highest $\Sigma_s|\mathbb{N}_s|$, the greedy algorithm performs worst on PEMS. We also observe that the greedy algorithm takes near 500 seconds on PEMS, which implies that the performance of exact solutions with cubic complexity is not acceptable. This confirms the necessity of adopting the approximate algorithm.

4.5.2 Online Sketch Maintenance

In the online scenario, we evaluate the following four algorithms in our performance study:

Sketch Computing (SC). SC examines all streaks generated from a fresh event. Then Algorithm 7 is invoked for each affected subject. To improve efficiency, the updates are processed in batches, i.e., multiple updates on the same subject will be batched and processed by calling Algorithm 7 once.

Sketch with Early Termination (SET). SET adopts *online-streak* bound in Theorem 4.4.1 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.

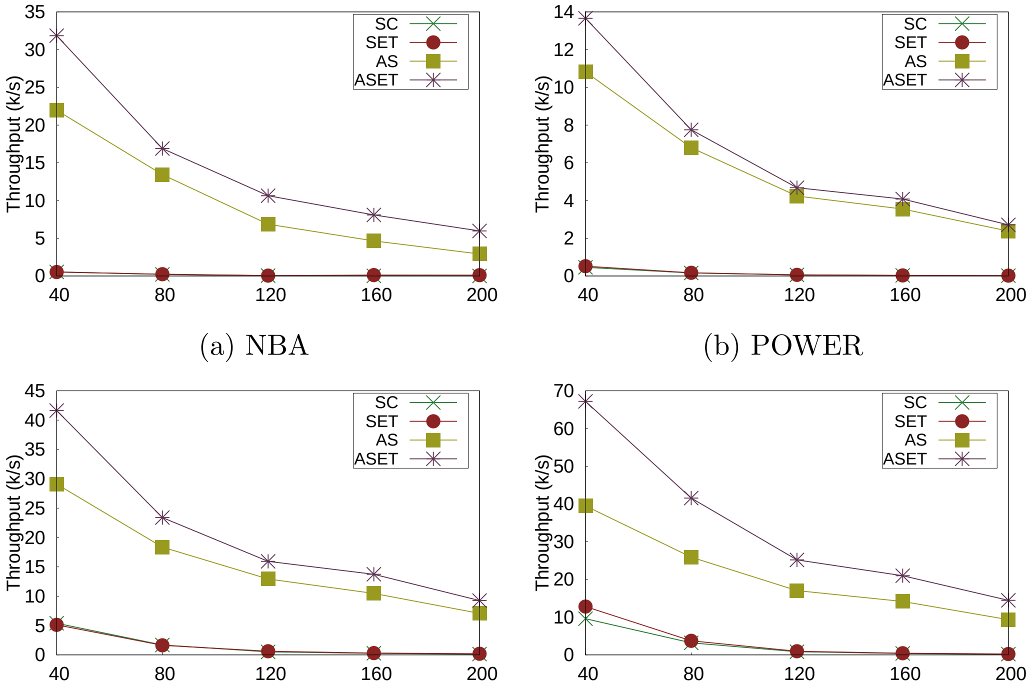
In the online setting, we are more interested in evaluating the throughput of algorithms. We report the performance with respect to p, k and h.

4.5.2.1 Query throughput varying p

We increase p from 10 to 200 and the throughput results are shown in Figure 4.7. The figure demonstrates similar patterns for all four datasets. As p increases, the throughput of the four algorithms drops. This is because as p increases, the time required to maintain the top-p ranked-streaks as well as to update the sketch increases. However, algorithms adopting *online-streak bound* have higher throughput than their counterparts. This is because with early termination, fewer streaks are generated. We can also see that *SC* and *ASC* run very slowly in the online setting. This is because they need to invoke Algorithm 7 upon every update. This confirms the necessity of our approximate method as $ASET$ achieves up to 500x speedup as compared to SC.

4.5.2.2 Query throughput varying k

We then evaluate how the throughput varies with respect to k. The results are presented in Figure 4.8. The figure tells similar patterns as Figure 4.7. First, as



(a) NBA

(b) POWER

(c) STOCK

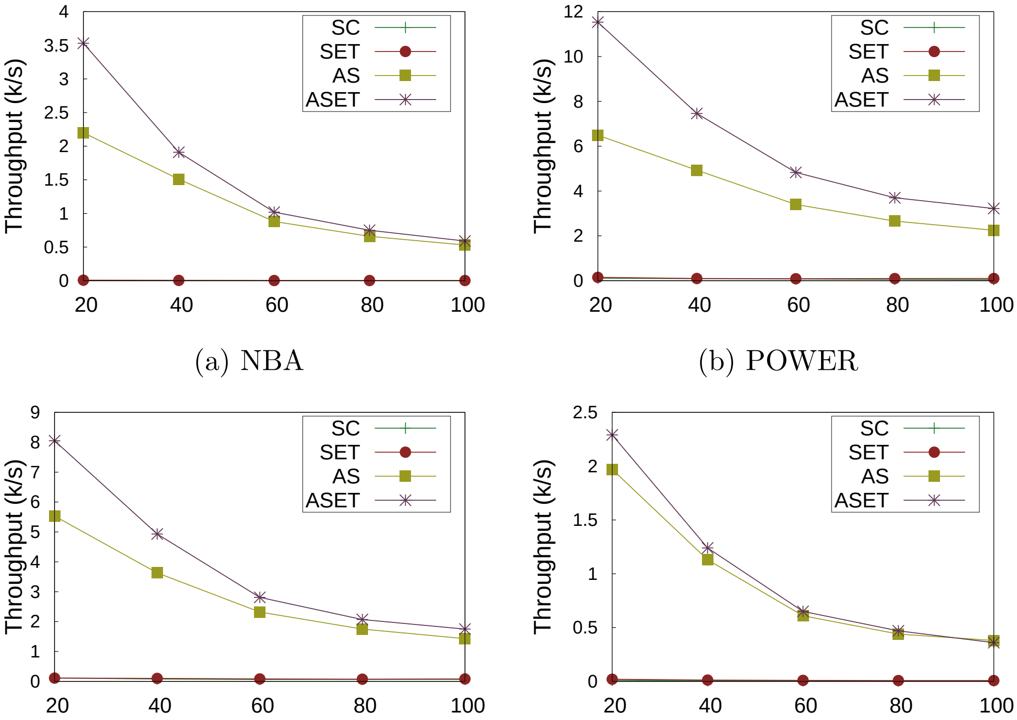
(d) PEMS

Figure 4.7: Throughput in online scenario with varying p.

k increases, the throughput of all four algorithms decreases. This is because as k becomes larger, more operations are needed for maintaining the sketch. Second, the throughput of SC and SET are an order of magnitude smaller than AS and $ASET$. This is because SC and SET repetitively call Algorithm 7 which heavily depends on k. We observe that in some datasets (e.g., Figure 4.8 (a)) there is 100x boost for $ASET$ as compared to SC.

4.5.2.3 Query throughput varying h

Finally, we study the effect of h in affecting the throughput. We change h from 20 to 100, and the results are represented in Figure 4.9. As shown in the figure, when h increases, the throughput of the four algorithms drops steadily. This is because as h increases, $|\mathbb{H}_s|$ for each subject increases. Therefore in Algorithm 8, more time is needed to process each streak. We notice that $ASET$ has a flatter slope than AS;



(a) NBA

(b) POWER

(c) STOCK

(d) PEMS

Figure 4.8: Throughput in the online scenario with varying k.

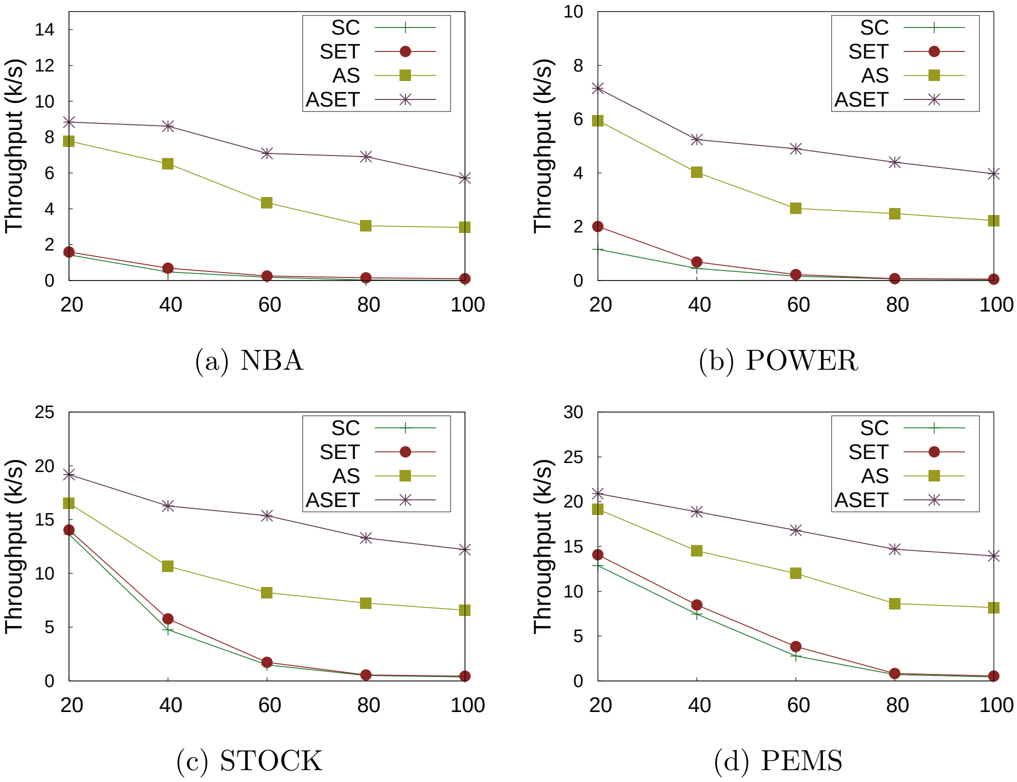
this is benefit from the prunings of *online-streak* bound.

4.5.3 Comparison with Other Techniques

We also compare the efficiency and effectiveness of our k-Sketch query with the state-of-the-art Prominent Streaks query [73] (denoted by the skyline method) in providing newsworthy summaries.

To study the efficiency, we implement the skyline algorithms as described in [73] for both online and offline scenarios. The results under all four datasets are presented in Figure 4.10. The figure demonstrates the superiority of our schemes in both scenarios. Specifically, our offline scheme saves 63% to 75% processing time and our online scheme achieves 2 to 10 time throughput speedups. These results further indicate the efficiency of our schemes.

To study the effectiveness, we conduct a user study over Amazon Mechanical



(a) NBA

(b) POWER

(c) STOCK

(d) PEMS

Figure 4.9: Throughput in the online scenario with varying h.

Turk[14] to evaluate the attractiveness of the summaries generated by different methods. For our method, we set $\alpha = 0.5$ to pay equal attention to the strikingness and the coverage. For the skyline method, due to the overwhelming skylines generated for each subject, we propose two augmented methods to pick k of them. In total, we compare the following four algorithms:

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

2. SK_a: selects the k-sketch for each player generated by the online k-Sketch query.

3. SY_m: randomly selects k streaks for each player from the bunch of skylines generated by [73].

4. SY_r: attaches ranks[15] to the streaks in SY_m.

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

[15] Rank is generated by our offline method

We then apply each method on the NBA dataset and set $k = 5$ to generate 5 streaks for each player. Each streak is then translated into a news theme in the following format:

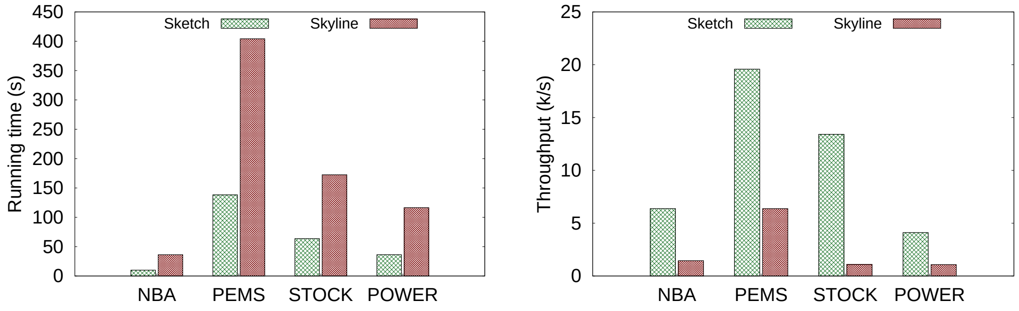
2003/04/14: Jordan obtained 30.3 PPGA[16] *in 989 straight games, ranked* 1^{st} *in NBA history!!*

We design each job in AMT to contain 5 questions and each question presents the four summaries of a player generated by the four methods. A sample question regarding "Kobe Bryant" is listed in Table 4.5. Due to space limitation, we present SY_m and SY_r in one row, since they essentially report the same streak, except that SY_r provides additional rank information. We then ask the respondents to endorse each summary based on the level of attractiveness. We receive responses from 100 participants who have knowledge in NBA[17]. Then, for each algorithm, we count the frequency of it being endorsed as the best and report the percentage results in the pie chart in Figure 4.11.

The chart clearly shows that SK is the most effective method as it takes 51% of the endorsements from the respondents. Overall, Sketch based methods (i.e., SK and SK_a) receive 75% endorsements, which win the skyline methods (i.e., SY_m and SY_r) by three times. The chart also shows that, when applied with the rank information (i.e., SY_r), the number of endorsements increases dramatically, more than two times of the original number of endorsements (i.e., SY_m). This also implies the effectiveness of our ranked-streaks. We can also observe the quality differences of the four methods in Table 4.5. As the table shows, SY_m and SY_r output streaks concentrating on a shorter period (i.e., 2006) as compared with the output of SK_a and SK (i.e., 2003-2008). This is because Kobe unprecedentedly scored 81 points on 2006/01/22, thus most skylines are associated with that event. Moreover, the streaks selected by SY_m

[16] PPGA:Point-per-game-average

[17] 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 prominent streaks.

and SY_r are not ranked well as compared with the results of SK_a and SK. The reason is that skyline based methods only consider the local prominence (i.e., non-dominated in one's career) but k-Sketch considers the global prominence (i.e., rank in history).

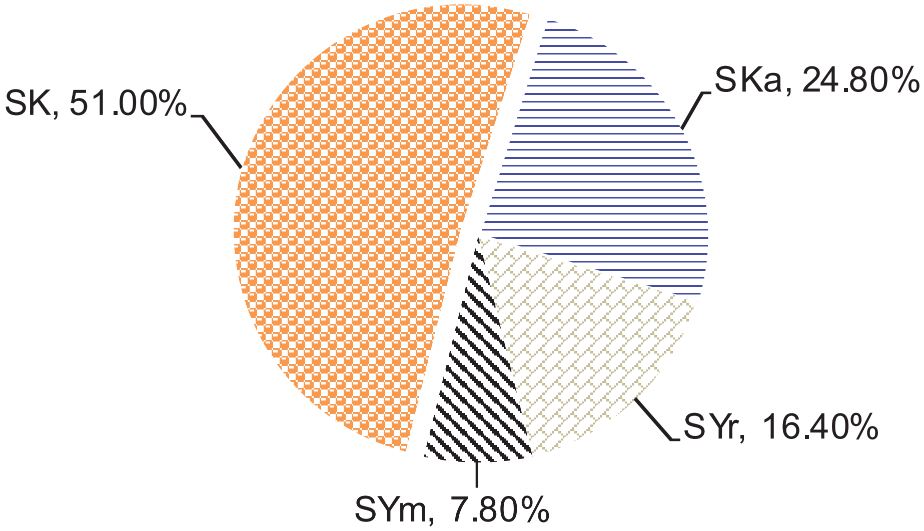


Figure 4.11: Percentage of endorsements received as the most attractive.

4.5.4 Case Study on Real Data

We further study the k-Sketch query in the NBA dataset. We compute a 5-sketch for the player "Dominique-Wilkins" using different α and list the ranked-streaks in Table 4.6. As the table shows, when α is small (i.e., 0.1), the streaks selected tend to have higher ranks. On the other hand, when α is large (i.e., 0.9), the streaks have

Table 4.5: Summaries of "Kobe Bryant" on Point-per-game-average (PPGA) obtained by the four methods. SY_r corresponds to the same SY_m streak with augmented rank information.

Method	**Career Summaries Generated**
SY_m (SY_r)	1. **2006/04/16**, Kobe obtained **37.05** PPGA in **57** straight games (ranked **10** in NBA history)! 2. **2006/04/16**, Kobe obtained **35.09** PPGA in **85** straight games (ranked **111** in NBA history)! 3. **2006/04/19**, Kobe obtained **36.91** PPGA in **61** straight games (ranked **12** in NBA history)! 4. **2006/11/03**, Kobe obtained **35.85** PPGA in **75** straight games (ranked **63** in NBA history)! 5. **2006/11/08**, Kobe obtained **35.29** PPGA in **78** straight games (ranked **85** in NBA history)!
SK_a	1. **2003/03/11**, Kobe obtained **38.05** PPGA in **20** straight games, ranked **25** in NBA history! 2. **2006/02/08**, Kobe obtained **38.67** PPGA in **28** straight games, ranked **1** in NBA history! 3. **2006/02/26**, Kobe obtained **38.10** PPGA in **30** straight games, ranked **1** in NBA history! 4. **2006/04/19**, Kobe obtained **37.17** PPGA in **56** straight games, ranked **5** in NBA history! 5. **2007/10/30**, Kobe obtained **32.27** PPGA in **212** straight games, ranked **194** in NBA history!
SK	1. **2003/02/21**, Kobe obtained **43.20** PPGA in **10** straight games, ranked **3** in NBA history! 2. **2006/01/22**, Kobe obtained **81.00** PPGA in **1** straight games, ranked **1** in NBA history! 3. **2006/02/11**, Kobe obtained **38.24** PPGA in **29** straight games, ranked **1** in NBA history! 4. **2007/03/30**, Kobe obtained **46.12** PPGA in **8** straight games, ranked **1** in NBA history! 5. **2008/02/01**, Kobe obtained **32.24** PPGA in **216** straight games, ranked **195** in NBA history!

larger coverage.

We observe several interesting facts from the sketches. First, the streaks with highest coverage (i.e., $\alpha = 0.9$) concentrate in the period 1986-1993, while the highest ranked-streaks (i.e., $\alpha = 0.1$) locate in the period 1986-1988. Looking up the ground truth[18], we find that "Dominique"'s prime career is in 1985-1993 and he was selected into the All-Star team every year in 1986-1988, which is consistent with our discoveries. Second, our k-Sketch query also discovers a length-1 streak of 57 points (in 1986-04-10, ranked 11th in history), which is in fact the career-highest points scored by "Dominique"[19]. Last, we notice that "Dominique" has a length-2 streak with an average score of 50 points, which ranks 8th in history. Although this streak ranks better than the length-1 streak with 57 points, interestingly, it has not been reported in any news. This indicates that our k-Sketch query is able to discover interesting facts where human experts may miss.

Table 4.6: 5-Sketches for "Dominique-Wilkins" from NBA dataset with respect to α.

α	**Sketches**			
	Date	**Streak Length**	**Avg(points)**	**Rank**
0.1	1986-04-01	2	47.00	14
	1986-04-10	1	57.00	11
	1987-02-10	2	50.00	8
	1988-03-01	2	48.50	11
	1988-03-01	11	40.54	16
0.5	1986-04-01	2	47.00	14
	1987-02-10	2	50.00	8
	1988-03-01	2	48.50	11
	1988-03-11	14	39.35	18
	1991-12-10	3	44.00	38
0.9	1986-11-02	55	32.98	147
	1987-03-26	26	32.46	145
	1987-02-10	14	32.21	117
	1988-04-19	61	33.19	139
	1993-03-29	34	32.88	145

[18] http://www.nba.com/history/players/wilkins_stats.html

[19] http://articles.latimes.com/1986-12-11/sports/sp-2180_1_22-point-deficit

4.6 Summary

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

Chapter 5

GCMP Query: Neighborhood Analytics in Trajectories

5.1 Introduction

Trajectory analysis is another emerging field in data analytics. A trajectory is the spatial trace of a moving object which contains a sequence of spatial-temporal records. Typical trajectories include visitor movements in a shopping mall, taxi flows in a city, animal migration traces in a continent and user action logs in social networks. Data analysis on these trajectories benefits a wide range of applications and services, including traffic planning [77], animal analysis [45], location-aware advertising [29], and social recommendations [7], to name just a few.

An important analytics on top of trajectories is to discover co-moving objects. A *co-movement* pattern [41, 76] refers to a group of objects traveling together for a certain period of time. Such a pattern can be concisely expressed by two neighborhood functions in the spatial and the temporal dimensions respectively. Specifically, in the spatial dimension, let $o(t)$ be the object o's location at time t. Then the objects co-moving with an object o at time t are determined by a *distance neighborhood function*:

$\mathcal{N}_1(o,t) = \{o_i | \texttt{dist}(o(t), o_i(t)) \leq r\}$[1]. Next, in the temporal dimension, the objects co-moving with an object o for a time period T are determined by: $\mathcal{N}_2(o,T) = \{o_i | \forall t \in T, o_i \in \mathcal{N}_1(o,t)\}$. A movement pattern is prominent if the size of the group exceeds M (i.e., $|\mathcal{N}_2(\cdot)| \geq M$) and the length of the duration exceeds K (i.e., $|T| \geq K$), where M and K are parameters specified by users. Rooted from such a basic definition and driven by different mining applications, there are many variants of co-movement patterns that have been developed with additional constraints.

Table 5.1 summarizes several popular co-movement patterns with different constraints with respect to spatial neighborhood, temporal constraints in consecutiveness and computational complexity. In terms of spatial neighborhood, the *flock* [28] and the *group* [63] patterns adopt disk-based clustering which requires all the objects in a group to be enclosed by a disk with radius r[2]; whereas the *convoy* [34], the *swarm* [44] and the *platoon* [43] patterns resort to density-based spatial clustering[3]. In terms of temporal constraints, the *flock* and the *convoy* require all the timestamps of each detected spatial group to be consecutive, which is referred to as *global consecutiveness*; whereas the *swarm* does not impose any restriction. The *group* and the *platoon* adopt a compromised approach by allowing arbitrary gaps between consecutive segments, which is called *local consecutiveness*. They introduce a parameter L to control the minimum length of each local consecutive segment.

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

[1] This refers to as the *disk-based* clustering. Density-based clustering can be expressed similarly as: $\mathcal{N}_1(o,t) = \{o_i | \texttt{dist}(o_j(t), o_i(t)) \leq \epsilon \wedge o_j \in \mathcal{N}_1(o,t)\}$

[2] Disk-based clustering is equivalent to $\mathcal{N}(o_i,t) = \{o_j | \texttt{dist}(o_i(t), o_j(t)) < r\}$.

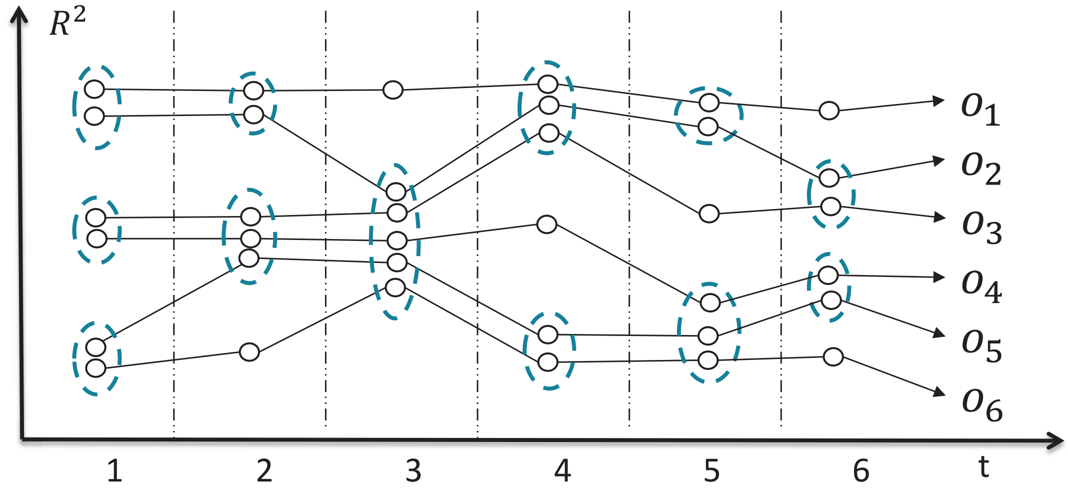
[3] Density-based clustering is equivalent to $\mathcal{N}(o_i,t) = \{o_j | \texttt{dist}(o_j(t), o_k(t)) \leq \epsilon \wedge o_k \in \mathcal{N}(o_i,t)\}$.

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

Pattern	**Spatial Neighborhood**	**Temporal Constraint**	**Time Complexity**
flock [28]	disk based	global consecutive	$O(\|\mathbb{O}\|\|\mathbb{T}\| \log(\|\mathbb{O}\|))$
convoy [34]	density based	global consecutive	$O(\|\mathbb{O}\|^2 + \|\mathbb{O}\|\|\mathbb{T}\|)$
swarm [44]	density based	-	$O(2^{\|\mathbb{O}\|}\|\mathbb{O}\|\|\mathbb{T}\|)$
group [63]	disk based	local consecutive	$O(\|\mathbb{O}\|^2\|\mathbb{T}\|)$
platoon [43]	density based	local consecutive	$O(2^{\|\mathbb{O}\|}\|\mathbb{O}\|\|\mathbb{T}\|)$

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

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



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

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

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

The other issue with existing methods is that they are built on top of centralized indexes. Thus, they may not be scalable to handle real large-scale trajectories collected by today's positioning technologies. 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

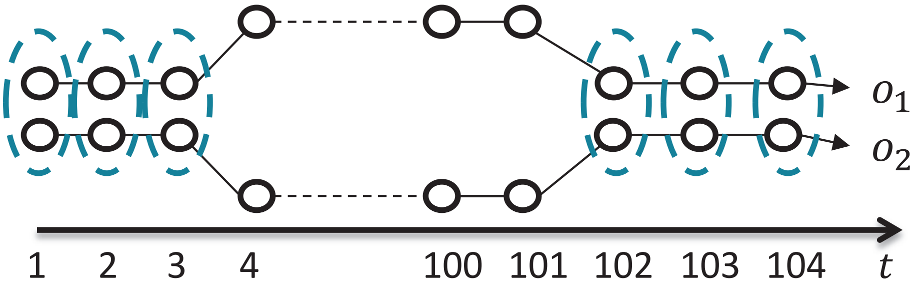
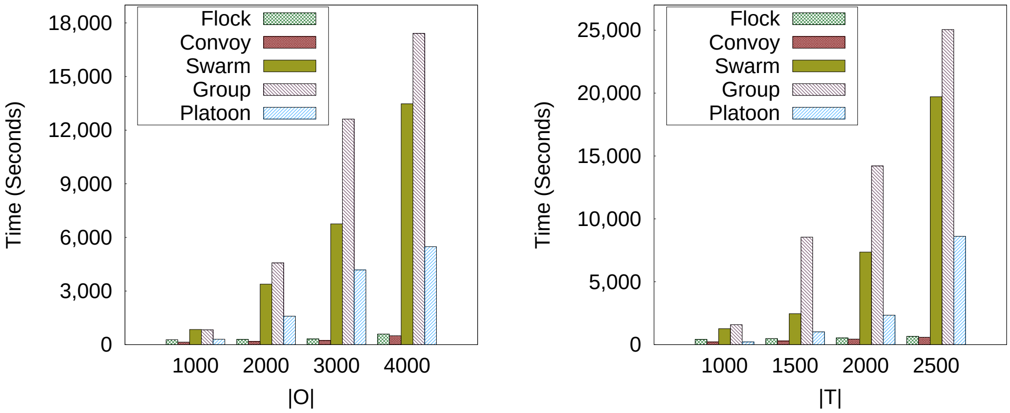


Figure 5.2: *Loose-connection* anomaly. Even though $\langle o_1, o_2 \rangle$ is considered as a valid *platoon* pattern, it is highly probable that these two objects are not related as the two consecutive segments are 98 timestamps apart.

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

In this chapter, we close these two gaps by making the following contributions. First, we propose the *general co-movement pattern* (GCMP) which models various co-moment patterns in a unified way and can avoid the *loose-connection* anomaly. In GCMP, we introduce a new gap parameter G to pose a constraint on the temporal gap between two consecutive segments. By setting a feasible G, the loose-connection anomaly can be effectively controlled. In addition, our GCMP is also general. It can be reduced to any of the previous pattern by customizing its parameters.

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

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

The rest of this chapter is organized as follows: Section 5.2 states the problem of general co-movement pattern mining. Section 5.3 provides a baseline solution. An advanced solution named *Star Partitioning and ApRiori Enumerator* (SPARE) is presented in Section 5.4. Section 5.5 reports our experimental evaluation. Finally, Section 5.6 summarizes this chapter.

5.2 Problem Formulation

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 the commonly-used operators in this chapter in Table 5.2.

Table 5.2: Operators on time sequence.

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
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 $T_3[x] \in T_2$

We say a sequence T is consecutive if $\forall 1 \leq i < |T|, T[i+1] = T[i] + 1$. We refer to each consecutive subsequence of T as a *segment*. It is obvious that any time sequence T can be decomposed into one or more segments and we say T is *L-consecutive* [43] 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* anomaly. To avoid such an anomaly without losing 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 5.2.1 (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., $\forall 1 \leq i < |T|, T[i+1] - T[i] \leq G$.

We take $T = (1, 2, 3, 5, 6)$ as an example. T 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. But 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 5.2.2 (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 every 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. By customizing these parameters, our pattern can express other patterns proposed in the 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.

Table 5.3: Expressing other patterns using GCMP. $\cdot$ indicates a user specified value.

Pattern	M	K	L	G	**Clustering**
Group	2	1	2	$\|\mathbb{T}\|$	disk
Flock	$\cdot$	$\cdot$	K	1	disk
Convoy	$\cdot$	$\cdot$	K	1	density
Swarm	$\cdot$	$\cdot$	1	$\|\mathbb{T}\|$	density
Platoon	$\cdot$	$\cdot$	$\cdot$	$\|\mathbb{T}\|$	density

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 work 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.

Table 5.4: Summary of notations.

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

5.3 Baseline: Temporal Replication and Parallel Mining

In this section, we propose a baseline solution that resorts to MapReduce 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 mapreduce 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,

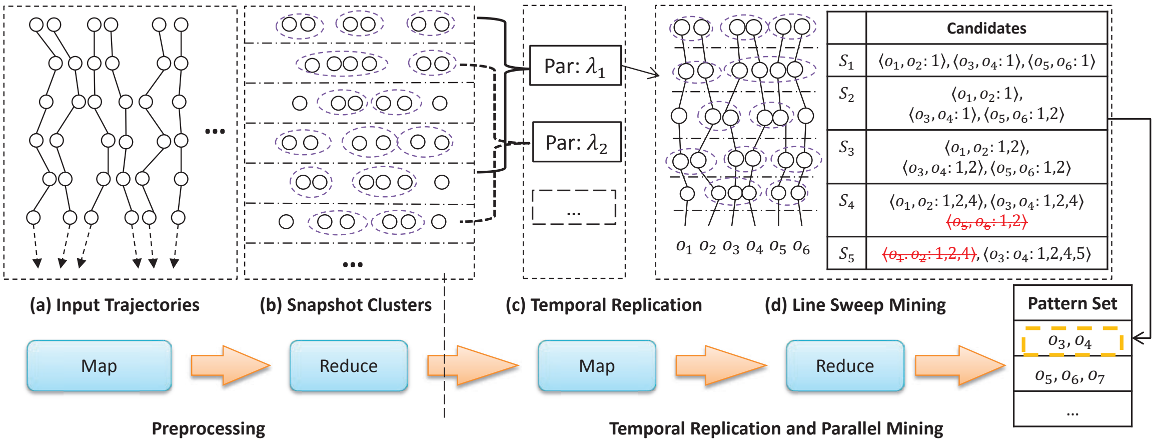

...
...
Par: λ_1
Par: λ_2
...
o_1 o_2 o_3 o_4 o_5 o_6
Candidates
S_1
$\langle o_1, o_2: 1\rangle, \langle o_3, o_4: 1\rangle, \langle o_5, o_6: 1\rangle$
S_2
$\langle o_1, o_2: 1\rangle,$
$\langle o_3, o_4: 1\rangle, \langle o_5, o_6: 1{,}2\rangle$
S_3
$\langle o_1, o_2: 1{,}2\rangle,$
$\langle o_3, o_4: 1{,}2\rangle, \langle o_5, o_6: 1{,}2\rangle$
S_4
$\langle o_1, o_2: 1{,}2{,}4\rangle, \langle o_3, o_4: 1{,}2{,}4\rangle$
~~$\langle o_5, o_6: 1{,}2\rangle$~~
S_5
~~$\langle o_1, o_2: 1{,}2{,}4\rangle$~~, $\langle o_3: o_4: 1{,}2{,}4{,}5\rangle$
(a) Input Trajectories
(b) Snapshot Clusters
(c) Temporal Replication
(d) Line Sweep Mining
Pattern Set
o_3, o_4
o_5, o_6, o_7
...
Map
Reduce
Map
Reduce
Preprocessing
Temporal Replication and Parallel Mining

Figure 5.4: Workflow of Temporal Replication and Parallel Mining (TRPM). (a) and (b) correspond to the first mapreduce stage which clusters objects in each snapshot. (c) and (d) is the second mapreduce stage which uses TRPM to detect GCMPs.

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 chapter is on the second mapreduce 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 5.3.1. $\eta = (\lceil\frac{K}{L}\rceil - 1) * (G-1) + K + L - 1$ guarantees that no valid pattern is missed.

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 range$(T) = \max(T) - \min(T) + 1$ and prove the theorem by showing that 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, 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 range$(T') \leq \eta$. Therefore, for any valid sequence T, there is 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, TRPM forms a partition for every consecutive η snapshots. In other words, each snapshot S_t corresponds to a partition $\lambda_t =$

$\{S_t, ..., S_{t+\eta-1}\}$. Next, we aim to design an efficient pattern mining strategy within each partition. Our solution includes a line sweep algorithm to sequentially scan the η snapshots in a partition and an effective candidate pattern enumeration mechanism.

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 $C$
```

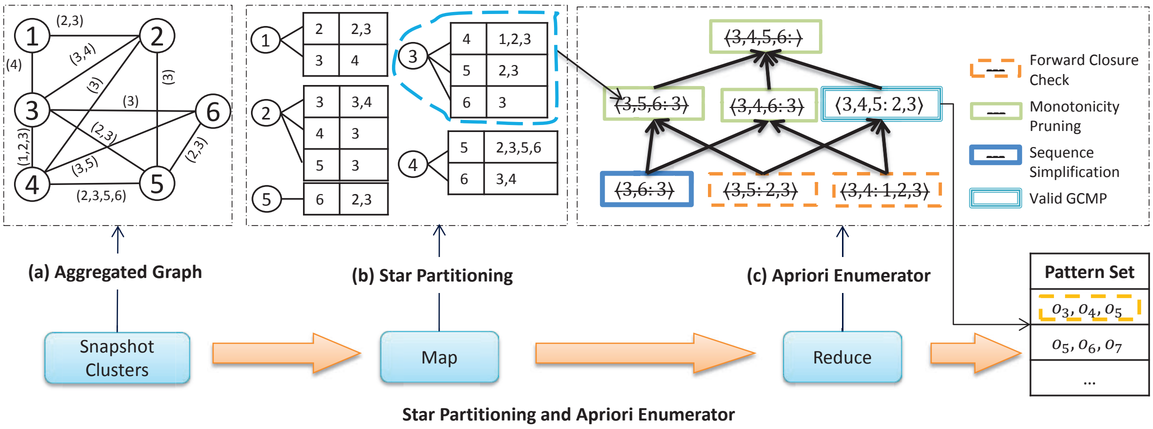
Details of the algorithm are presented in Algorithm 10. We keep a candidate set C (Line 1) during the sweeping. It is initialized using the clusters with size no smaller than M in the first snapshot. Then, we sequentially scan each snapshot (Lines 7-27) and generate new candidates by extending the original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

Algorithm 11 Temporal Replication and Parallel Mining

```
Input: list of $\langle t, S_t \rangle$ pairs
$\eta \leftarrow (\lceil \frac{K}{L} \rceil - 1) * (G - 1) + K + L - 1$
—Map Phase—
for all snapshots $S_t$ do
    for all $i \in 1...\eta - 1$ do
        emit key-value pair $\langle \max(t - i, 0), S_t \rangle$
    end for
end for
—Partition and Shuffle Phase—
for all key-value pairs $\langle t, S \rangle$ do
    group-by $t$ and emit a key-value pair $\langle t, \lambda_t \rangle$, where $\lambda_t = \{S_t, S_{t+1}, ..S_{t+\eta-1}\}$
end for
—Reduce Phase—
for all key-value pairs $\langle t, \lambda_t \rangle$ do
    call line sweep mining for partition $\lambda_t$
end for
```

The complete picture of TRPM is shown in Algorithm 11. We illustrate the workflow of TRPM using Figures 5.4 (c) and (d) with pattern parameters $M = 2, K = 3, L = 2, G = 2$. By Theorem 5.3.1, 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), the line sweep method is illustrated for



1	
2	2,3
3	4

2	
3	3,4
4	3
5	3

3	
4	1,2,3
5	2,3
6	3

4	
5	2,3,5,6
6	3,4

5	
6	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.

partition λ_1. Let C_i be the candidate set when sweeping snapshot S_i. Initially, C_1 contains patterns with all 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 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 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 only 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 be 4, the pattern $\langle o_3, o_4 \rangle$ would be missed.

5.4 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 redundantly discovered from different partitions. To resolve these limitations, we propose a new Star Partitioning and ApRiori

Enumerator, named SPARE, to replace the second stage of the mapreduce 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.

5.4.1 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 $(2, 3)$.

We use *star* [69] 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 5.4.1 (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. 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* [38]. 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[4]. 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 has a smaller Γ and is much more balanced.

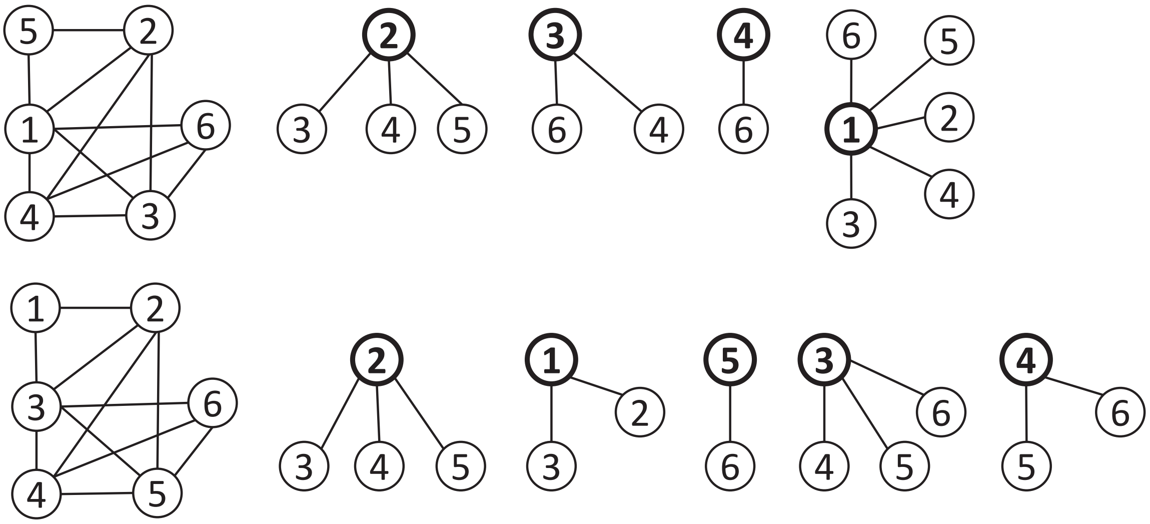


Figure 5.6: Star partitioning with different vertex orderings.

Although it is very challenging to find the optimal vertex ordering from the $n!$

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

possibilities, we observe that a random order can actually achieve satisfactory performance based on the following theorem.

Theorem 5.4.1. 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 dense, we are able to obtain a tighter bound for $(\Gamma - \Gamma^*)$.

Theorem 5.4.2. Let d be the average degree in G_A. If $d \geq \sqrt{12 \log n}$, with 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.

5.4.2 Apriori Enumerator

Intuitively, given a GCMP with an object set $\{o_1, \ldots, o_m\}$, 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 [2] 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 5.4.1. 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.

5.4.2.1 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 5.4.2 (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 5.4.2. 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 5.4.3. Suppose $\{T_1, T_2, \cdots, T_m\}$ 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) $T_1 \cup T_2 \cup \cdots \cup T_m = T$.

Proof. We assume $T_i \cap T_j \neq \emptyset$ and prove (1) by contradiction. Let $T_i = (T_i[1], \cdots, T_i[p])$ and $T_j = (T_j[1], \cdots, T_j[n])$. 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 $(T_y, \cdots, T_x, \cdots, T_z)$, 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 form a 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 5.4.4. If $T_1 \subseteq T_2$, 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 $T_1' \subseteq T_2'$.

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 5.4.3, 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 5.4.3, we can obtain a contradiction. Thus, all the timestamps in T_1' must occur in T_2'. □

Definition 5.4.3 (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 5.4.3. 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 ≥ 4.

Definition 5.4.4 (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 5.4.4. 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 5.4.5. 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 5.4.4, 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 5.4.5, we are ready to define the *monotonicity* concept based on the simplified sequences to facilitate the pruning in the Apriori algorithm.

Theorem 5.4.6 (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 5.4.5, we know $\texttt{sim}(T) \neq \emptyset$ because $P_2.T \subseteq P.T$ and $P_2.T$ is decomposable. This contradicts with $\texttt{sim}(P.T) = \emptyset$. □

5.4.2.2 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 $c' = \langle c_1.O \cup c_2.O, c_1.T \cap c_2.T \rangle$ (Line 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, then it is directly reported (Lines 16-20). To further improve the performance, we adopt the idea of *forward closure* [53, 62] and aggressively check if the union of all the current candidates form a valid pattern (Lines 22-26). If yes, we can terminate the algorithm early and output the results.

Example 5.4.5. 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.

5.4.3 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 use 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. 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

Algorithm 12 Apriori Enumerator

```
Input: $Sr_s$
$C \leftarrow \emptyset$
for all edges $c = \langle o_i \cup o_j, T_{o_i} \cap T_{o_j} \rangle$ in $Sr_s$ do
    if $\texttt{sim}(T_{o_i} \cap T_{o_j}) \neq \emptyset$ then
        $C \leftarrow C \cup \{c\}$
    end if
end for
level $\leftarrow 2$
while $C \neq \emptyset$ do
    for all $c_1 \in C$ do
        for all $c_2 \in C$ and $|c_2.O \cup c_2.O| =$ level do
            $c' \leftarrow \langle c_1.O \cup c_2.O : (c_1.T \cap c_2.T) \rangle$
            if $\texttt{sim}(c'.T) \neq \emptyset$ then
                $C' \leftarrow C' \cup \{c'\}$
            end if
        end for
        if no $c'$ is added to $C'$ then
            if $c_1$ is a valid pattern then
                output $c_1$
            end if
        end if
    end for
    $O_u \leftarrow$ union of $c.O$ in $C$
    $T_u \leftarrow$ intersection of $c.T$ in $C$
    if $\langle O_u, T_u \rangle$ is a valid pattern then
        output $\langle O_u, T_u \rangle$, break
    end if
    $C \leftarrow C'; C' \leftarrow \emptyset;$ level $\leftarrow$ level $+ 1$
end while
output $C$
```

presented in Algorithm 13. In our implementation of SPARE on Spark [71], we take advantage of Spark features to achieve better workload balance. In particular, we utilize Spark DAG execution engine to inject a planning phase between map and reduce phases. By knowing all map results (i.e., star sizes), a simple best-fit strategy is adopted which assigns the most costly unallocated star to the most lightly loaded reducer, where the edges in a star are used as cost estimations. We also leverage Spark in-memory cache to avoid recomputing all stars after the planning phase.

Algorithm 13 Star Partitioning and ApRiori Enumerator

```
Input: list of $\langle t, S_t \rangle$ pairs
—Map phase—
for all $C \in S_t$ do
    for all $o_1 \in C, o_2 \in C, o_1 < o_2$ do
        emit a $\langle o_1, o_2, \{t\} \rangle$ triplet
    end for
end for
—Partition and Shuffle phase—
for all $\langle o_1, o_2, \{t\} \rangle$ triplets do
    group-by $o_1$, emit $\langle o_1, Sr_{o_1} \rangle$
end for
—Reduce phase—
for all $\langle o, Sr_o \rangle$ do
    call Apriori Enumerator for star $Sr_o$
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 5.4.7 (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 from different stars, meaning their center vertexes are different (i.e., $o_i \neq o_j$), leading to a contradiction. □

Theorem 5.4.7 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 5.4.8. The SPARE framework guarantees completeness and soundness.

Proof. See Appendix A.2.2. □

5.5 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[5] 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 Spark 1.5.2[6] 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. We set the number of partitions to be 486 to fully utilize the multi-threading feature

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

[6] We have experimented with a query-based TRPM using Spark-SQL 2.0.0 window function. We find that Spark-SQL fails to execute the query-based TRPM in parallel, which results in a 120x performance slowdown compared to mapreduce-based TRPM. Thus we only report the performance of mapreduce-based TRPM in this work.

of every core. All our implementations as well as cluster setups are publicly available[7].

Datasets. We use three real trajectory datasets that are collected from different applications:

- Shopping[8]: 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[9]: 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[10]: 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 [26] 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 and Shopping datasets; and $\epsilon = 20$, $minPt = 10$ for Taxi dataset. After preprocessing, the statistics of the three datasets are listed in Table 5.5.

Parameters. To systematically study the performance of our algorithms, we

[7] `https://github.com/fanqi1909/TrajectoryMining/`.

[8] `http://www.irc.atr.jp/crest2010_HRI/ATC_dataset/`

[9] `http://research.microsoft.com/en-us/projects/geolife/`

[10] Taxi is our proprietary dataset

Table 5.5: Statistics of datasets.

Attributes	**Shopping**	**GeoLife**	**Taxi**
# 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

conduct experiments on various parameter settings. The parameters to be evaluated are listed in Table 5.6, with default settings in bold.

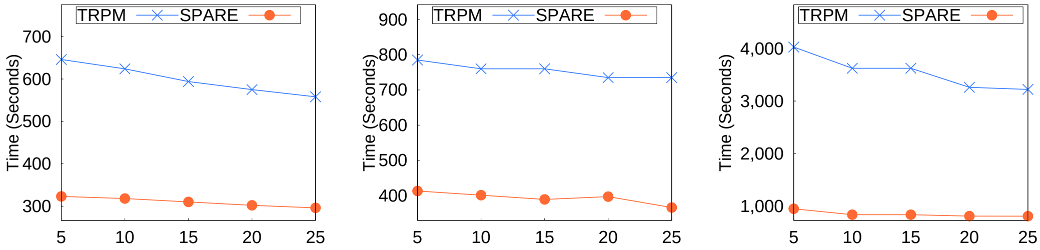
Table 5.6: Parameters and their default values.

Param.	**Meaning**	**Values**
M	min objects	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**

5.5.1 Performance Evaluation

Varying M. Figures 5.7 (a), (b), (c) 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.

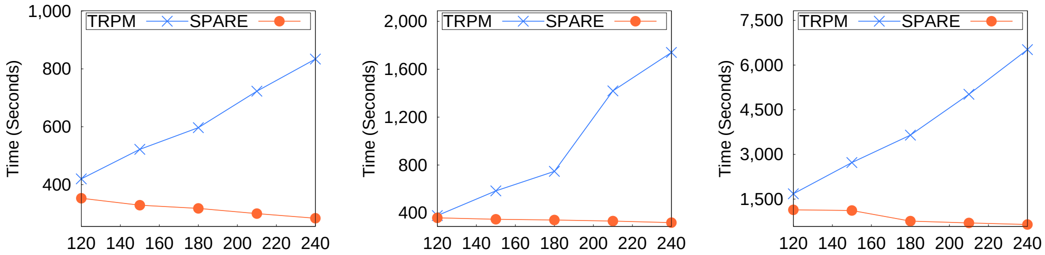
Varying K. The performance with increasing K is shown in Figures 5.8 (a), (b), (c). SPARE tends to run faster, whereas the performance of TRPM degrades dramatically. This is caused by the *sequence simplification* procedure in SPARE, which can prune many candidates with large K. However, the line sweep algorithm in TRPM does not utilize such property for pruning. It takes longer time because



(a) Shopping (b) Shopping (c) Shopping

Figure 5.7: Performance of SPARE and TRPM on real datasets with varying M

more replicated data has to be handled in each partition.

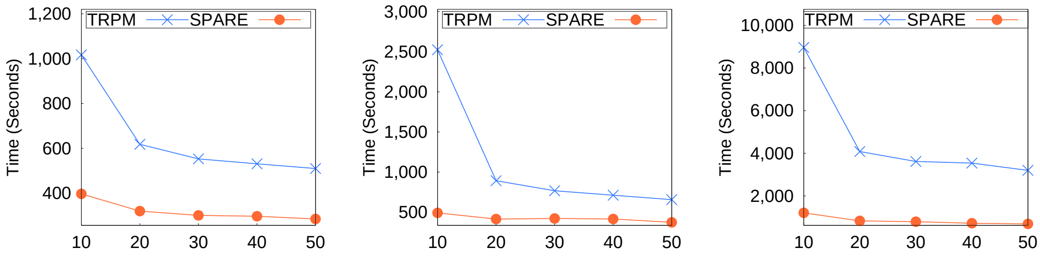


(a) Shopping (b) Shopping (c) Shopping

Figure 5.8: Performance of SPARE and TRPM on real datasets with varying K

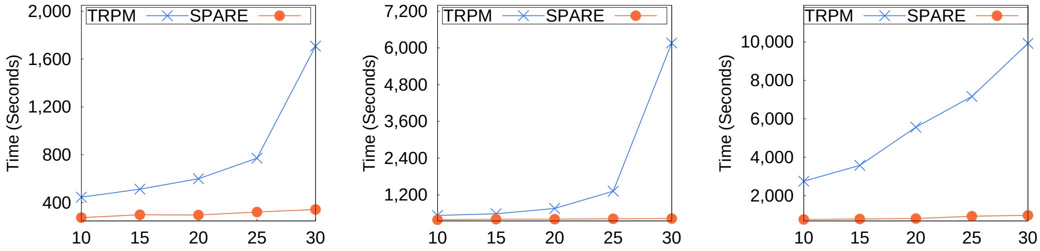
Varying L. Figures 5.9 (a),(b),(c) present the performances with increasing L. When $L = 10$, SPARE can outperform TRPM by around 10 times. We also observe that there is a significant performance improvement for TPRM when L increases from 10 to 20 and later the running time drops smoothly. This is because 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 5.10 (a), (b), (c) 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, SPARE scales well with G. Particularly, SPARE is 14 times faster than TRPM when $G = 20$ in GeoLife dataset.



(a) Shopping (b) Shopping (c) Shopping

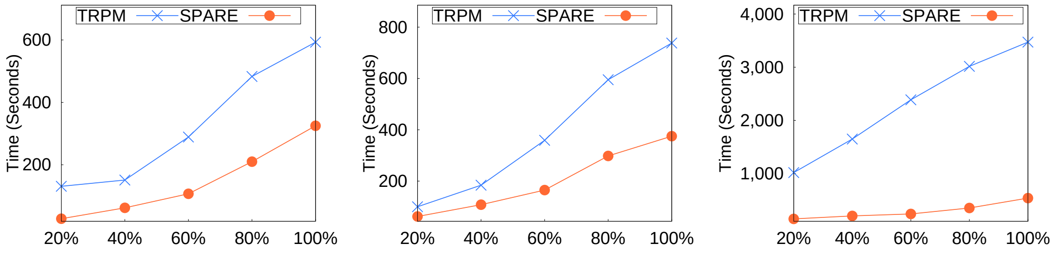
Figure 5.9: Performance of SPARE and TRPM on real datasets with varying L



(a) Shopping (b) Shopping (c) Shopping

Figure 5.10: Performance of SPARE and TRPM on real datasets with varying G

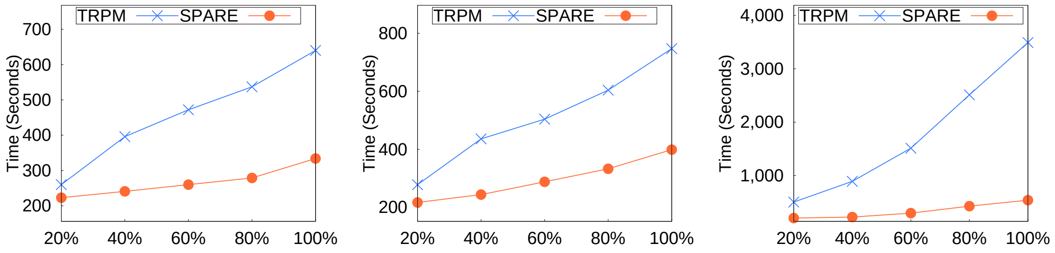
Varying O_r. Figures 5.11 (a), (b), (c) present the performances with increasing number of moving objects. Both TRPM and SPARE take longer time to find patterns in a larger database. We can see that the performance gap between SPARE and TRPM is widened as more objects are involved, which shows SPARE is more scalable.



(a) Shopping (b) Shopping (c) Shopping

Figure 5.11: Performance of SPARE and TRPM on real datasets with varying O_r

Varying T_r. Figures 5.12 (a), (b), (c) 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.



(a) Shopping (b) Shopping (c) Shopping

Figure 5.12: Performance of SPARE and TRPM on real datasets with varying T

Resources. Table 5.7 lists the system resources taken by TRPM and SPARE under the default setting. Both TRPM and SPARE are resource efficient as they only occupy less than 20% of the available memory (i.e., 270GB) . Again, SPARE outperforms TRPM in both the execution time and the memory usage.

Table 5.7: Resources taken for TRPM and SPARE. Vcore-seconds is the aggregate of time spent in each core. Memory is the actual size (in MB) of RDDs.

Dataset	**Method**	**Vcore-seconds**	**Memory**
Shopping	TRPM	90,859	10,019
	SPARE	33,638	8,613
Geolife	TRPM	106,428	18,454
	SPARE	35,343	14,369
Taxi	TRPM	503,460	51,691
	SPARE	68,580	35,912

5.5.2 Analysis on the 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.

5.5.2.1 Power of sequence simplification

To study the power of *Sequence Simplification* (SS), 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. Their difference is the number of size-2 candidates pruned by SS. The results in Table 5.8 show that SS is very powerful and eliminates nearly 90 percent of the object pairs, which significantly reduces the overhead of the Apriori enumerator. In fact, without SS Apriori cannot finish in five hours.

Table 5.8: Pruning power of SPARE.

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%

5.5.2.2 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 5.13 shows the breakdown of the costs in the mapreduce 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). Meanwhile, both SPARE and SPARE-RD outperform TRPM in each phase. This shows the efficiency of the star partition and apriori enumeration. We also report the cost of the longest job (Max) and the standard deviation (Std. Dev.) for all jobs in Table 5.9, whose results clearly verify the effectiveness of our allocation strategy for load balance.

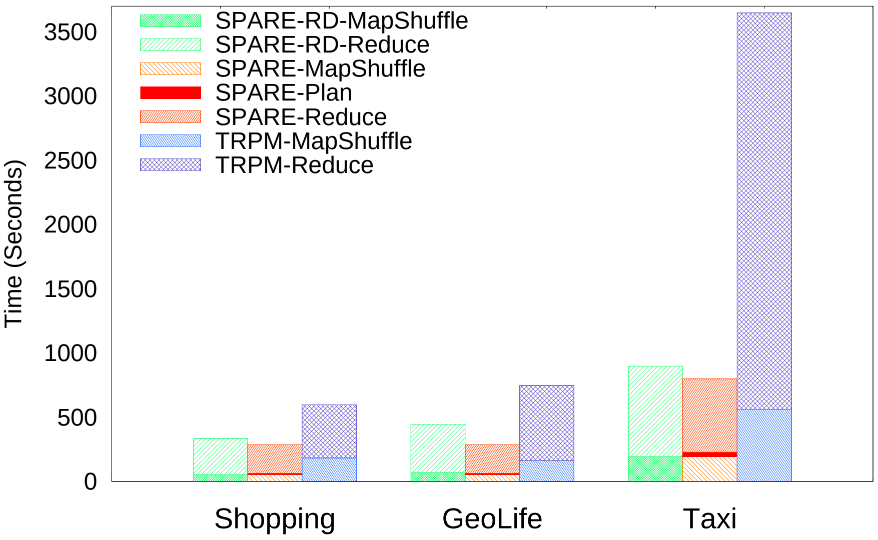


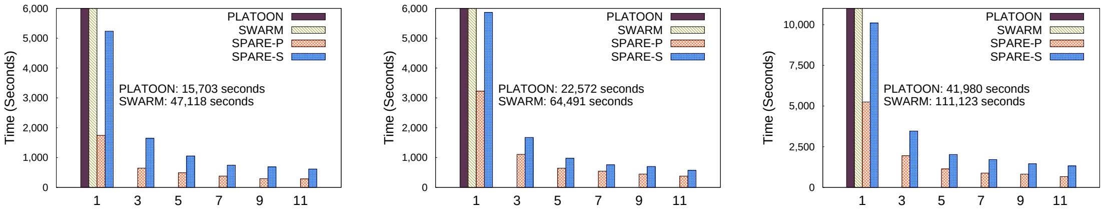
Figure 5.13: Cost breakdown of TRPM, SPARE-RD and SPARE.

Table 5.9: Statistics of execution time (seconds) on all jobs.

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

5.5.2.3 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. For a fair comparisons, we customize two variants of SPARE to mine *swarms* and *platoons*, 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 5.14. First, the centralized schemes are not suitable to discover patterns in large-scale trajectory databases. It takes nearly 30 hours to detect *swarms* and 11 hours to detect *platoons* in the Taxi dataset in a single machine. In contrast, when utilizing the



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

Figure 5.14: Comparisons among TRMP, SPARE, PLATOON and SWARM.

multi-core (i.e., a single node with four executors) 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 up to 112 times better than the state-of-the-art centralized schemes.

5.6 Summary

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

Chapter 6

Conclusion and Future Work

With the increasing variety and volume of the data managed by the nowadays database systems, the adoption of effective analytics becomes remarkably demanding. Window analytics, being an important part of SQL analytics, has proven successes in many relational applications. However, window analytics requires a strict ordering among objects which may not be meaningful in other data domains. In this thesis, we proposed an analogous analytics named *neighborhood analytics*, which generalizes the window analytics by eliminating the ordering requirement. Followed by the concept of neighborhood analytics, we then systematically studied three instances of such analytics in supporting emerging applications in three data domains. We proposed domain-tailored neighborhood queries and demonstrated their usefulness. To support large-scale data, we further designed various optimization techniques which achieved efficient query processing.

6.1 Thesis Contributions

We hereby revisit our contributions of this thesis. Our first contribution is the *Graph Window Query* (GWQ) on data graphs. GWQ computes aggregations for each vertex on its windows. We formally defined two instances of graph windows: k-hop window and topological window. Then, we developed the Dense Block Index (DBIndex) to facilitate efficient processing of both types of graph windows. In addition, we proposed the Inheritance Index (I-Index) that exploits a containment property of DAG to further improve the query performance of topological window queries. Both indexes 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 indexes.

Our second contribution is the *k-Sketch* query on sequence data. k-Sketch query utilizes the *ranked-streaks* to summarize a subject's history. The ranked-streak is formed by a nested neighborhood function: the neighborhood events were grouped to form a streak; then streaks with the same size were ranked to indicate their strikingness. We formulated the *k-Sketch* query to select k ranked-streaks which best summarize a subject's history. We studied the k-Sketch query processing in both offline and online scenarios, and proposed efficient solutions to cope each scenario. Specifically, we designed novel streak-level pruning techniques and a $(1 - 1/e)$-approximate algorithm for offline processing. Then we designed a $1/8$-approximate algorithm for online maintenance. Our comprehensive experiments demonstrated the efficiency of our solutions and a human study confirms the effectiveness of the k-Sketch query.

Our third contribution is the *General Co-movement Pattern* (GCMP) query on trajectory. We modeled the GCMP using the spatial neighborhoods among objects: the invariant portion of an object's neighborhood across certain timestamps forms a pattern. By adjusting temporal constraints, our GCMP is able to express all co-movement patterns proposed in the past literature. On the technical side, we devised two parallel frameworks on Spark platform which can be scaled to support query processing in trajectories with hundreds of millions of points. The efficiency and scalability were verified by extensive experiments on three real datasets.

6.2 Future Research Directions

This thesis describes the neighborhood analytics in three data domains, which induces many interesting problems to follow up with. We would like to highlight them to inspire future explorations.

In Chapter 3, we proposed the graph window query, which leads to at least the following directions. First, we wish to empower the graph window queries to support more general aggregate functions such as median, centrality and user-defined aggregate functions. Second, we would like to study how to support graph window queries to dynamic graph and graph streams. This boils down to the challenging problem of handling structural updates (i.e., edge insertion and deletion) on our indexes. Last but not least, we aim to leverage modern parallel systems to facilitate scalable graph window query processing on graphs with multi-billion vertexes and edges.

In Chapter 4, we introduced k-Sketch query to summarize sequence data. There are several further directions worthy exploring using the neighborhood based *ranked-streak*. First, we would like to generalize the rank-streak to non-schema data such as tweets and replies in social networks. This requires a more sophisticated ranking criteria. Second, we plan to study the problem of summarizing a subject's history in the sliding window model. This is particular helpful in generating news themes that are emerging recently. Last, leveraging big data technology to support fast-growing event data is also important and of our interests.

In Chapter 5, we utilized the neighborhood concept to design the general co-movement pattern mining framework. In the next stage, we would like to explore the real-time movement pattern detection. Meanwhile, we also wish to leverage the co-movement patterns to facilitate advanced trajectory analysis. For example, it is of great interest to discover the latent social network from drivers based on their co-moving behaviors.

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] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[3] James Allan, Ron Papka, and Victor Lavrenko. On-line new event detection and tracking. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 37–45. ACM, 1998.

[4] Noga Alon, Baruch Awerbuch, and Yossi Azar. The online set cover problem. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 100–105. ACM, 2003.

[5] 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.

[6] Baruch Awerbuch, Yossi Azar, Amos Fiat, and Tom Leighton. Making commitments in the face of uncertainty: How to pick a winner almost every time. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 519–530. ACM, 1996.

[7] Jie Bao, Yu Zheng, David Wilkie, and Mohamed F Mokbel. A survey on recommendations in location-based social networks. *ACM Transaction on Intelligent Systems and Technology*, 2013.

[8] 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.

[9] 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.

[10] Thorsten Brants, Francine Chen, and Ayman Farahat. A system for new event detection. In *Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Informaion Retrieval*, SIGIR '03, pages 330–337, New York, NY, USA, 2003. ACM.

[11] Erica J Briscoe, D Scott Appling, Rudolph L Mappus IV, and Heather Hayes. Determining credibility from social network structure. In *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 1418–1424. ACM, 2013.

[12] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8):1157–1166, 1997.

[13] Juan Miguel Campanario. Empirical study of journal impact factors obtained using the classical two-year citation window versus a five-year citation window. *Scientometrics*, 87(1):189–204, 2011.

[14] 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.

[15] 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.

[16] 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.

[17] James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 193–204. ACM, 2013.

[18] 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.

[19] Tom Choe, Alexander Skabardonis, and Pravin Varaiya. Freeway performance measurement system: operational analysis tool. *Transportation Research Record: Journal of the Transportation Research Board*, (1811):67–75, 2002.

[20] Sarah Cohen, Chengkai Li, Jun Yang, and Cong Yu. Computational journalism: A call to arms to database researchers. In *CIDR*, volume 2011, pages 148–151, 2011.

[21] Lianghao Dai, Jar-der Luo, Xiaoming Fu, and Zhichao Li. Predicting offline behaviors from online features: an ego-centric dynamical network approach. In *Proceedings of the First ACM International Workshop on Hot Topics on Interdisciplinary Social Networks Research*, pages 17–24. ACM, 2012.

[22] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 2000.

[23] 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.

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

[25] Katherine Edwards, Simon Griffiths, and William Sean Kennedy. Partial interval set cover–trade-offs between scalability and optimality. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 110–125. Springer, 2013.

[26] 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.

[27] Lukasz Golab, Howard Karloff, Flip Korn, Avishek Saha, and Divesh Srivastava. Sequential dependencies. *Proceedings of the VLDB Endowment*, 2(1):574–585, 2009.

[28] Joachim Gudmundsson and Marc van Kreveld. Computing longest duration flocks in trajectory data. In *Proceedings of the 14th annual ACM international symposium on Advances in geographic information systems*, pages 35–42. ACM, 2006.

[29] Long Guo, Dongxiang Zhang, Gao Cong, Wei Wu, and Kian-Lee Tan. Influence maximization in trajectory databases. In *TKDE*, page 1, 2016.

[30] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *ACM Sigmod Record*, volume 29, pages 1–12. ACM, 2000.

[31] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data mining and knowledge discovery*, 8(1):53–87, 2004.

[32] Naeemul Hassan, Afroza Sultana, You Wu, Gensheng Zhang, Chengkai Li, Jun Yang, and Cong Yu. Data in, fact out: automated monitoring of facts by factwatcher. *Proceedings of the VLDB Endowment*, 7(13):1557–1560, 2014.

[33] Clyde W Holsapple and Wenhong Luo. A citation analysis of influences on collaborative computing research. *Computer Supported Cooperative Work (CSCW)*, 12(3):351–366, 2003.

[34] Hoyoung Jeung, Man Lung Yiu, Xiaofang Zhou, Christian S. Jensen, and Heng Tao Shen. Discovery of convoys in trajectory databases. *Proceedings of the VLDB Endowment*, 1(1):1068–1080, 2008.

[35] 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.

[36] 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.

[37] Ingrid M Keseler, Julio Collado-Vides, Socorro Gama-Castro, John Ingraham, Suzanne Paley, Ian T Paulsen, Martín Peralta-Gil, and Peter D Karp. Ecocyc: a comprehensive database resource for escherichia coli. *Nucleic acids research*, 33(suppl 1):D334–D337, 2005.

[38] Yong-Chul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.

[39] 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.

[40] 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.

[41] 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.

[42] Xuefei Li, Hongyun Cai, Zi Huang, Yang Yang, and Xiaofang Zhou. Social event identification and ranking on flickr. *World Wide Web*, 18(5):1219–1245, 2015.

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

[44] 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.

[45] Zhenhui Li, Bolin Ding, Jiawei Han, Roland Kays, and Peter Nye. Mining periodic behaviors for moving objects. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1099–1108. ACM, 2010.

[46] M. Lichman. UCI machine learning repository, 2013.

[47] Huaiyu Harry Ma, Steven Gustafson, Abha Moitra, and David Bracewell. Egocentric network sampling in viral marketing applications. In *Mining and Analyzing Social Networks*, pages 35–51. Springer, 2010.

[48] Nan Ma, Jiancheng Guan, and Yi Zhao. Bringing pagerank to the citation analysis. *Information Processing & Management*, 44(2):800–810, 2008.

[49] 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.

[50] Peter V. Marsden. Egocentric and sociocentric measures of network centrality. *Social networks*, 24(4):407–422, 2002.

[51] 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.

[52] George L Nemhauser, Laurence A Wolsey, and Marshall L Fisher. An analysis of approximations for maximizing submodular set functionsi. *Mathematical Programming*, 14(1):265–294, 1978.

[53] Jian Pei, Jiawei Han, and Runying Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD workshop on research issues in data mining and knowledge discovery*, volume 4, pages 21–30, 2000.

[54] Barna Saha and Lise Getoor. On maximum coverage in the streaming model & application to multi-topic blog-watch. 9:697–708, 2009.

[55] 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.

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

[57] 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.

[58] Virginia Vassilevska and Ali Pinar. Finding nonoverlapping dense blocks of a sparse matrix. *Lawrence Berkeley National Laboratory*, 2004.

[59] Jeroen B.P. Vuurens, Arjen P. de Vries, Roi Blanco, and Peter Mika. Online news tracking for ad-hoc information needs. In *Proceedings of the 2015 International Conference on The Theory of Information Retrieval*, ICTIR '15, pages 221–230, New York, NY, USA, 2015. ACM.

[60] Brett Walenz, You Will Wu, Seokhyun Alex Song, Emre Sonmez, Eric Wu, Kevin Wu, Pankaj K Agarwal, Jun Yang, Naeemul Hassan, Afroza Sultana, et al. Finding, monitoring, and checking claims computationally based on structured data. In *Computation+ Journalism Symposium*, 2014.

[61] Brett Walenz and Jun Yang. Perturbation analysis of database queries. *Proc. VLDB Endow.*, 9(14):1635–1646, October 2016.

[62] Jianyong Wang, Jiawei Han, and Jian Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 236–245. ACM, 2003.

[63] 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.

[64] 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.

[65] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. Reachability querying: An independent permutation labeling approach. *Proceedings of the VLDB Endowment*, 7(12):1191–1202, 2014.

[66] 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.

[67] 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.

[68] Hilmi Yildirim, Vineet Chaoji, and Mohammed J Zaki. Dagger: A scalable index for reachability queries in large dynamic graphs. *arXiv preprint arXiv:1301.0977*, 2013.

[69] Jin Soung Yoo and Shashi Shekhar. A joinless approach for mining spatial colocation patterns. *IEEE Transactions on Knowledge and Data Engineering*, 18(10):1323–1337, 2006.

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

[71] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

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

[73] 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.

[74] 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.

[75] 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.

[76] Yu Zheng. Trajectory data mining: an overview. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(3):29, 2015.

[77] Yu Zheng, Yanchi Liu, Jing Yuan, and Xing Xie. Urban computing with taxicabs. In *Proceedings of the 13th international conference on Ubiquitous computing*, pages 89–98. ACM, 2011.

[78] 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.

Appendix A

Appendix

A.1 Discussions on Other Aggregate Functions in Chapter 4

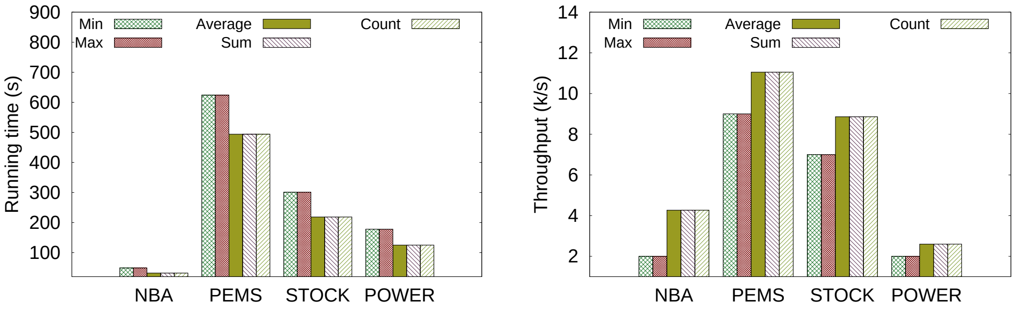
First, we shall see that supporting *sum* is equivalent to supporting *avg*. A ranked-streak which has a rank under *avg* will have the same rank under *sum* as the ranking is derived by comparing all candidates with the same length. Second, supporting *count* is equivalent to supporting *sum*. By assigning each event with a 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 streaks 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.

We present the performance variations of our k-Sketch query under different aggregate functions in Figure A.1. We can see from the figures that when adopting *min* (*max*) the performance in both online and offline scenarios drops (20% to 30%). This

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

indicates that the pruning bounds in *min* (*max*) is weaker than *avg* (*sum*, *count*).



(a) Offline Scenario

(b) Online Scenario

Figure A.1: Performance under different aggregate functions

A.2 Proofs of Theorems in Chapter 5

A.2.1 Proofs of Theorem 5.4.1 and 5.4.2

Proof. Γ can be formalized in linear algebra as follows: 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

[1] An identity matrix with rows shuffled

$e(i, j)$ in G_A to the lower vertex, then the matrix $B = \text{triu}(PJP^T)$[2] represents the assignment matrix with respect to 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 as follows:

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

Let B^* be the assignment matrix with respect to 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 all $e_{i,j}$ 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 $e_{i,j}$ are independent, then $b_{i,j}$ are also independent. Second, since $i > j$ and $e_{i,j}$ are independent. $E[b_{i,j}] = E[e_{i,j}|i > j] = E[e_{i,j}]E[i > j] = E[e_{i,j}]/2$. As c_i is a sum of n independent 0-1 variables (i.e, $b_{i.j}$). 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, it follows:

[2] `triu` is the upper triangle part of a matrix

[3] every element in $\vec{b}$ is 1

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

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 5.4.8

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 the smallest ID in $P.O$. We prove that P must be outputted 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. □