Real-time Distributed Co-Movement Pattern Detection on Streaming Trajectories

Lu Chen[†], Yunjun Gao[*,♯], Ziquan Fang[*,♯], Xiaoye Miao[‡], Christian S. Jensen[†], Chenjuan Guo[†]

[†]Department of Computer Science, Aalborg University, Aalborg, Denmark
[*]College of Computer Science, Zhejiang University, Hangzhou, China
[‡]Center for Data Science, Zhejiang University, Hangzhou, China
[♯]Alibaba–Zhejiang University Joint Institute of Frontier Technologies, Hangzhou, China

[†]{luchen, csj, cguo}@cs.aau.dk [*]{gaoyj, zqfang}@zju.edu.cn [‡]miaoxy@zju.edu.cn

ABSTRACT

With the widespread deployment of mobile devices with positioning capabilities, increasingly massive volumes of trajectory data are being collected that capture the movements of people and vehicles. This data enables co-movement pattern detection, which is important in applications such as trajectory compression and future-movement prediction. Existing co-movement pattern detection studies generally consider historical data and thus propose off-line algorithms. However, applications such as future movement prediction need real-time processing over streaming trajectories. Thus, we investigate real-time distributed co-movement pattern detection over streaming trajectories.

Existing off-line methods assume that all data is available when the processing starts. Nevertheless, in a streaming setting, unbounded data arrives in real time, making pattern detection challenging. To this end, we propose a framework based on Apache Flink, which is designed for efficient distributed streaming data processing. The framework encompasses two phases: clustering and pattern enumeration. To accelerate the clustering, we use a range join based on two-layer indexing, and provide techniques that eliminate unnecessary verifications. To perform pattern enumeration efficiently, we present two methods FBA and VBA that utilize id-based partitioning. When coupled with bit compression and candidate-based enumeration techniques, we reduce the enumeration cost from exponential to linear. Extensive experiments offer insight into the efficiency of the proposed framework and its constituent techniques compared with existing methods.

PVLDB Reference Format:
Lu Chen, Yunjun Gao, Ziquan Fang, Xiaoye Miao, Chrisitan S. Jensen, Chenjuan Guo. Real-time Co-Movement Pattern Detection on Streaming Trajectories. *PVLDB*, 12(10): 1208-1220, 2019.
DOI: https://doi.org/10.14778/3339490.3339502

1. INTRODUCTION

With the proliferation of GPS-equipped devices, massive and increasing volumes of trajectory data that capture the movements of humans, vehicles, and animals are being generated. Analyzing this data is important in a wide range of applications. One important type of analysis is the discovery of co-moving objects, termed co-movement pattern detection. It can be used in future-movement prediction [16, 21], trajectory compression [21], and location-based services [38], to name but a few applications. Fig. 1 exemplifies future-movement prediction using co-movement patterns. Given seven moving objects o_i ($1 \leq i \leq 7$), it is of interest to discover groups of objects that travel together. Specifically, we find three co-movement patterns, i.e., $P_1 = \{o_1, o_2\}$ with pattern "Home → City center → Shopping mall", $P_2 = \{o_3, o_5\}$ with pattern "Home → City center → Kommune", and $P_3 = \{o_4, o_6\}$ with pattern "Home → Countryside → University". Based on these patterns, we predict the next movement (i.e., "University") for a new object o_8 that moves following "Home → Countryside".

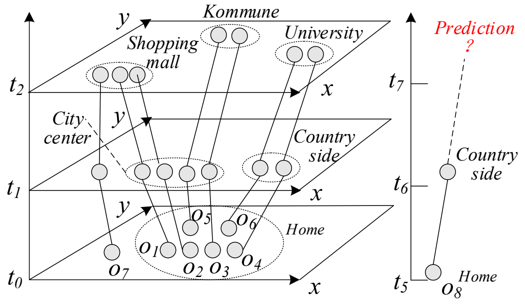


Figure 1: Example of Future Movement Prediction

A co-movement pattern [18, 37] refers to a group of objects traveling together for a certain period of time. Many variants of co-movement patterns (e.g., *flock* [13], *group* [29], *convoy* [17], *swarm* [20], *platoon* [19]) have been developed with different constraints. Fan et al. [10] propose a unified definition, which we adopt for generality. In this definition, a co-movement pattern contains a set O of objects with a time sequence T satisfying five constraints: (i) *closeness*, to control the spatial proximity of objects in O (i.e., the objects in O should belong to the same cluster at each time in T); (ii) *significance* M, to control the minimum number of objects in O; (iii) *duration* K, to control the length of T; (iv) *consecutiveness* L, to control the minimum length of each consecutive segment in T; and (v) *connection* G, to control the length of gaps between consecutive segments in T, where a segment is an consecutive sub time sequence of T. Although an arbitrary clustering method can be employed to capture closeness, we use density-based clustering and DBSCAN [9] that are used widely [17].

Due to the deployment of massive populations of devices with positioning capabilities, it is becoming increasingly relevant to being able to support large-scale settings with streaming trajectories.

Proceedings of the VLDB Endowment, Vol. 12, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3339490.3339502

Functionality such as future movement prediction and trajectory compression needs real-time processing. For example, in future-movement prediction, if an unexpected event such as an accident occurs in the road network, it is important to be able to detect new patterns in real time. **Hence, we investigate the problem of real-time co-movement pattern mining over streaming trajectories.**

Existing off-line batch processing algorithms [10, 13, 17, 19, 20, 29] were not intended for, and are not effective at, real-time co-movement pattern detection on streaming data. In off-line processing, all data is available when the processing starts. In a streaming setting, unbounded data arrives in real time, making pattern detection more difficult. For instance, one partitioning technique [10] used in pattern detection partitions the objects according to their closeness. In Fig. 1, objects o_1 and o_7 are not close at time t_0, but they are close at time t_3. In the off-line setting, we can put o_1 and o_7 in the same partition because all data is available when the processing starts. However, in the online setting, we cannot put o_1 and o_7 in the same partition at t_0. To support efficient co-movement pattern detection over streaming trajectories, three challenges have to be tackled.

Challenge I: How to handle the large scale of streaming trajectories in real time? To address this, we leverage Flink for distributed stream processing. Flink exploits pipelined data transfer to enable low latency and high throughput.

Challenge II: How to efficiently cluster streaming trajectories? To address this, we adopt the two-layered GR-index, which uses a grid index as the global index and R-trees as local indexes for grid cells. Based on the GR-index, we use a range join in an initial clustering step, and we provide provably correct techniques to eliminate so-called unnecessary verifications. The join processing is accelerated by performing range queries when building the GR-index, rather than performing querying after index construction. Again, we prove the correctness of the processing.

Challenge III: How to reduce the exponential cost of pattern enumeration? To address this, a simple yet efficient id-based partitioning method is presented. In addition, fixed-length and variable-length bit compression techniques are developed, which reduce the storage cost from $O(2^n)$ to $O(n)$, where n is the number of trajectories. We also propose candidate-based enumeration approaches, where patterns are generated based on valid candidates; this also reduces the processing cost from exponential to linear.

To sum up, the key contributions of this paper are as follows.

- We offer the first proposal for real-time co-movement pattern detection over streaming trajectories in Flink, adopting an established general co-movement pattern definition and using DBSCAN for clustering.
- We use a two-layered GR-index based range join to accelerate clustering, and we provide provably correct techniques that make it possible to avoid unnecessary verifications.
- We propose two approaches, FBA and VBA, to perform pattern enumeration efficiently. Moreover, we develop bit compression and candidate-based enumeration techniques that reduce the cost from exponential to linear.
- Extensive experiments with both real and synthetic data offer insight into the efficiency and scalability of the presented framework and its constituent techniques.

The rest of the paper is organized as follows. We review related work in Section 2. Then, we present preliminaries in Section 3. Section 4 describes the framework. Sections 5 and 6 detail the algorithms for clustering and pattern enumeration, respectively. Experimental findings are reported in Section 7. Finally, we conclude the paper and provide directions for future work in Section 8.

2. RELATED WORK

We proceed to review related work on co-movement pattern mining and then distributed stream processing.

2.1 Co-movement Pattern Mining

Work on co-movement pattern mining can be classified into two categories according to the constraints on the pattern duration. The first category requires strict consecutiveness, and does not allow any gap between consecutive segments. The second category allows relaxed constraints on the pattern duration. The first category includes the concepts of *flock* [13] and *convoy* [17]. The difference between *flock* and *convoy* lies in the clustering methods used. In *flock*, objects are clustered based on the inter-object distances. Specifically, the objects in a cluster have pairwise distances below a given threshold. In contrast, *convoy* relies on density-based clustering [9]. The second category contains *group* [29], *swarm* [20], and *platoon* [19]. The main idea of these three methods is to grow an object set from an empty set in a depth-first manner. During the growing, different pruning techniques are provided to eliminate unqualified branches. To unify the two categories, Fan et al. [10] offer a more general co-movement pattern definition, which we aim to support. However, we notice that the above methods focus on historical trajectories, while we aim to provide real-time co-movement pattern mining on streaming trajectories. Although proposals for computing *flock* [28], *convoy* [33], and *group* [18] on streaming trajectories exist, they assume centralized settings and only aim to support specific co-movement patterns. In contrast, we aim to support general co-movement pattern mining, and provide a distributed framework capable of supporting real-time mining on large-scale streaming trajectories.

Several mining frameworks for distributed stream processing also exist. Agrawal et al. [2] study pattern matching over event streams. Gu et al. [12] explore ranking in pattern matching for complex event streams. Yu et al. [35] propose two efficient methods for discovering frequent co-occurrence patterns across multiple data streams. Vistream [32] supports interactive visual exploration of neighbor-based patterns in data streams. Further, Yang et al. [31] present a real-time distributed stream processing framework. None of proposals target co-movement pattern mining over streaming trajectories, and thus, they are unable to solve our problem.

Finally, several studies [1, 6, 7, 24, 25, 26, 27, 34] investigate clustering on streaming trajectories. Although clustering is the first step of co-movement pattern mining, these existing efforts provide centralized methods, which are unable to contend with support large-scale steaming trajectories.

2.2 Distributed Stream Processing

The processing of streaming data is gaining in importance, due to the steadily growing number of data sources and the increasing real-time requirements for data analysis [30]. In keeping with this, different distributed stream processing systems have been explored, proposed, including SPADE [11], Naiad [22], Microsoft StreamInsight[1], and IBM Streams[2]. These systems are either simple prototype systems or closed-source systems, which renders them unsuited as an underlying platform for our work.

Recently, several open-source distributed stream processing platforms have also been proposed, which offer two different types of processing. In **tuple-at-a-time processing**, every incoming record is processed as soon as it arrives, without waiting for other records.

[1]https://blogs.msdn.microsoft.com/streaminsight/

[2]https://www.ibm.com/cloud/streaming-analytics/

Table 1: Symbols and Description

Notation	Description
$r = (l, t)$	the GPS record with location $l = (x, y)$ and time t
$o = \langle r_1, r_2, ... \rangle$	the streaming trajectory
T	the time sequence
$T[i]$	the i-th element in T
$\|T\|$	the number of elements in T
$max(T)$	the last time in T
T_l	the last time segment in T
ϵ	the distance threshold
$minPts$	the minimal number of points to form a dense region used in density-based clustering DBSCAN
M	the significance constraint
K	the duration constraint
L	the consecutive constraint
G	the connection constraint
$CP(M, K, L, G)$	the co-movement pattern w.r.t. M, K, L, and G
$RJ(O, \epsilon)$	the range join query for a set O with ϵ
$RQ(o, \epsilon)$	the range query for a query object o with ϵ
S_t	the snapshot at time t
l_g	the grid cell width
key	the key for a grid cell g
$B[o_i]$ or $B[O]$	the bit string for the trajectory o_i or the set O

Storm[3], Samza[4], and Flink[5] support this type of processing. Next, with **mini-batch semantics**, in-coming records that have arrived within the last few seconds are batched and then processed in a single mini-batch. Spark Streaming[6] supports this type of processing. We choose Flink, because it is a typical stream processing platform, and because it offers both efficiency and reliability. Nonetheless, our methods and techniques (e.g., GR-index, bit-compression, and candidate-based enumeration) are generic and hence can be easily adapted to other distributed stream processing platforms.

3. BACKGROUND

In this section, we introduce in turn the notion of co-movement pattern, DBSCAN, and range join. Table 1 summarizes the symbols used frequently throughout the paper.

3.1 Co-Movement Pattern

A GPS record is a pair $r = (l, t)$, where l is a location and t is a time value. A sequence $o = \langle r_1, r_2, ..., r_n \rangle$ of GPS records that capture a particular trip make up a trajectory.

Following an existing trajectory pattern detection approach [10], we first discretize the timestamps in trajectories. The discretization maps the real clock times to indices of the time intervals during which they occurred. For instance, assume that we partition the time line into intervals of duration 5s and that the start time is 13:00:20 UTC. Then the time series ⟨13:00:21 UTC, 13:00:24 UTC, 13:00:28 UTC, 13:00:32 UTC, 13:00:42 UTC⟩ is discretized into ⟨0, 0, 1, 2, 4⟩. This example discretization causes (i) a sequence where 0 appears twice, and (ii) that has a misleading gap. To avoid such problems, it is important to choose the duration used for discretization carefully. The duration cannot be too large or too small. In our experiments, the interval duration is set to 1s or 5s depending on sampling rates of the datasets used. Next, we give the definition of a discretized time sequence.

DEFINITION 1. **(Time Sequence)**. *Let $\mathbb{T} = \{1, 2, ..., \mathbb{N}\}$ be a discretized temporal dimension. A time sequence T is defined as a sequence of elements from $\mathbb{T}$, i.e., $T = \langle t_1, t_2, ..., t_m \rangle$, where (i) t_i $(1 \le i \le m) \in \mathbb{T}$, and (ii) $t_i > t_j$ iff $i > j$ $(t_i \in T, t_j \in T)$.*

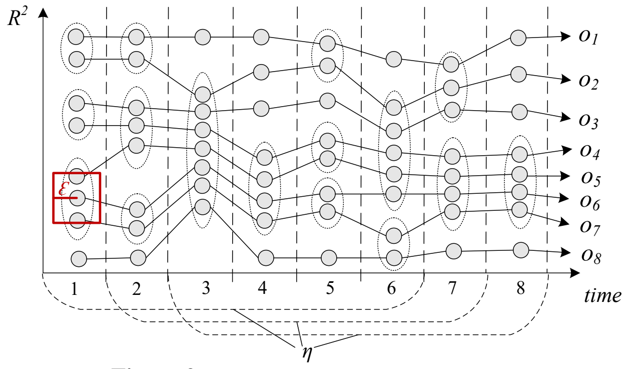


Figure 2: Example of Co-movement Patterns

A time sequence T is consecutive if $\forall 1 \le i < |T|$ $(T[i+1] = T[i] + 1)$. We call a consecutive sequence T a segment. For example, $T_1 = \langle 1, 2, 3, 4 \rangle$ and $T_2 = \langle 1, 2, 4, 5 \rangle$ are two sequences, where T_1 is a segment while T_2 is not, because time 3 is missing. Based on the definition of sequence, we define the notions of L-consecutive and G-connected. Here, L-consecutive is used to control the lengths of segments, while G-connected is employed to control the lengths of gaps between consecutive segments.

DEFINITION 2. **(L-consecutive)**. *Let T_i $(i \le i \le m)$ be segments with $|T_i| \ge L$. Then sequence $T = \cup T_i$ is L-consecutive.*

DEFINITION 3. **(G-connected)**. *A sequence T is G-connected if the gap between any neighboring times is at most G, i.e., $\forall 1 \le i \le |T| - 1$ $(T[i+1] - T[i] \le G)$, where $T[i]$ denotes the i-th element in T.*

For instance, $T = \langle 1, 2, 4, 5, 6 \rangle$ is 2-consecutive and 2-connected. Specifically, there are two segments $T_1 = \langle 1, 2 \rangle$ and $T_2 = \langle 4, 5, 6 \rangle$ in T, and the length of each segment is no smaller than 2. Thus, T is 2-consecutive. Further, according to Definition 3, $\forall 1 \le i \le 4$ $(T[i+1] - T[i] \le 2)$. Hence, T is 2-connected.

Next, we formalize the definition of co-movement pattern. Co-movement pattern mining detects a group of objects that move together while satisfying five constraints: (1) "*closeness*" that defines the concept of "moving together", (2) "*significance*" that controls the number of the objects that move together, (3) "*duration*" that controls the length of time when objects move together, while (4) "*L-consecutive*" and (5) "*G-connected*" that relax the consecutiveness of "duration". Specifically, the entire time period that objects move together is not necessarily strictly consecutive, as gaps are allowed between consecutive segments. Hence, "L-consecutive" and "G-connected" control the length of each consecutive time segment and the gap between two consecutive time segments, respectively.

DEFINITION 4. **(Co-movement Pattern)**. *Given a set ST of discretized trajectories, a subset O of ST is a co-movement pattern $CP(M, K, L, G)$ if a time sequence T exists such that the following five constraints are satisfied: (i)* ***closeness:*** *the locations of trajectories in O belong to the same cluster in every time of T; (ii)* ***significance:*** *$|O| \ge M$; (iii)* ***duration:*** *$|T| \ge K$; (iv)* ***consecutiveness:*** *T is L-consecutive; and (v)* ***connection:*** *T is G-connected.*

To provide a concrete definition of the first constraint (i.e., closeness), we choose to rely on density-based clustering as implemented by the popular clustering method DBSCAN (as also done for *convoy* [17]), which is detailed in the next subsection. Considering

[3]http://storm.apache.org/
[4]http://samza.apache.org/
[5]http://flink.apache.org/
[6]http://spark.incubator.apache.org/

the example in Fig. 2, a dotted circle denotes a cluster. Given $M = 3, K = 4, L = 2$, and $G = 2$, then $O = \{o_4, o_5, o_6\}$ is a co-movement pattern. Specifically, for $T = \langle 3, 4, 6, 7\rangle$, the following hold: (i) o_4, o_5, and o_6 belong to the same cluster at times 3, 4, 6, and 7; (ii) $|O| = 3$; (iii) $|T| = 4$; and (iv) T is 2-consecutive and 2-connected.

In a setting with streaming trajectories, GPS records are produced continuously over time. Therefore, we define the notion of streaming trajectory below.

DEFINITION 5. **(Streaming Trajectory)**. *A streaming trajectory is an unbounded ordered sequence of GPS records, i.e., $o = \langle r_1, r_2, ...\rangle$.*

In Fig. 2, o_1 to o_8 are streaming trajectories. A streaming trajectory is unbounded, i.e., the next GPS record and the total length of the trajectory are unknown in advance, which makes real-time co-movement pattern mining more difficult. Here, "real-time" means being able to process the data, and show it in results as soon as it arrives. For the setting of stream processing, we introduce the notion of a snapshot and the real-time co-movement pattern mining.

DEFINITION 6. **(Snapshot)**. *A snapshot $S_t = \{o_1.l, o_2.l, ..., o_n.l\}$ contains all the locations of a trajectory set $\{o_1, o_2, ..., o_n\}$ at time t.*

For simplicity, we use $\{o_1, o_2, ..., o_n\}$ to represent $\{o_1.l,\ o_2.l,\ ..., o_n.l\}$. In Fig. 2, there exist eight snapshots.

DEFINITION 7. **(Real-time Co-movement Pattern Mining)**. *Given parameters M, K, L, and G that defines a general co-movement pattern, real-time co-movement pattern mining finds all co-movement patterns in the snapshot set $S = \{S_1, S_2, ..., S_t\}$, where t is the current time t.*

As an example, if the current time is 5, $\{o_4, o_5\}$ and $\{o_6, o_7\}$ are *CP*(2, 4, 2, 2) patterns where $T = \langle 2, 3, 4, 5\rangle$. However, no *CP*(3, 4, 2, 2) pattern exists until time 7, where $\{o_4, o_5, o_6\}$ qualifies with $T = \langle 3, 4, 6, 7\rangle$.

3.2 DBSCAN

DBSCAN [9] is a popular density-based clustering method. It relies on two parameters to characterize density or sparsity, i.e., a positive real value ϵ and a positive integer $minPts$. Next, we introduce the definitions of core point and density reachable point.

DEFINITION 8. **(Core Point)** *A location u is a core point if at least $minPts$ locations v satisfy $d(u, v) \leq \epsilon$, where $d(u, v)$ denotes the distance between u and v.*

DEFINITION 9. **(Density Reachable Point)** *A location u is density reachable from location v if there exist a sequence of locations $x_1, x_2, ..., x_t (t \geq 2)$ such that (i) $x_1 = v$ and $x_t = u$; (ii) x_i $(1 \leq i < t)$ are core points; and (iii) $d(x_i, x_{i+1}) \leq \epsilon$ $(1 \leq i < t)$.*

Based on Definitions 8 and 9, a cluster is formed by a set of core points and their density reachable points. At time 3 in Fig. 2, given the ϵ shown in the figure and $minPts = 3$, o_3, o_4, o_5, o_6, and o_7 are core points, while o_2 and o_8 are density reachable points. Thus, a cluster $\{o_i | 2 \leq i \leq 8\}$ is formed. By scanning the whole data set, we can find all clusters.

3.3 Range Join

According to Definition 8, to determine whether u is a core point, we need to find all locations v in each snapshot S_t with their distances to u satisfying $d(u, v) \leq \epsilon$. A range query can be employed to find core points.

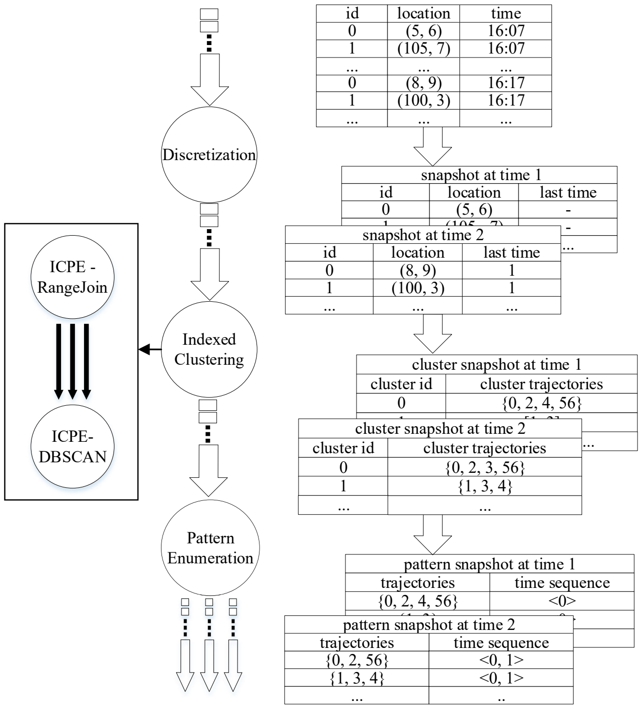


Figure 3: **Indexed Clustering and Pattern Enumeration (ICPE)**

DEFINITION 10. **(Range Query)** *Given a set O of locations, a threshold ϵ, and a query location u, a range query finds all locations v in O for u with their distances to u no larger than ϵ, i.e., $RQ(u, \epsilon) = \{(u, v) | d(u, v) \leq \epsilon, v \in O\}$.*

We use the L_1-norm to measure the distance between two locations, although it is easy to also support other distance functions. A range query $RQ(u, \epsilon)$ retrieves all locations v located in the range region $([u.x - \epsilon, u.x + \epsilon], [u.y - \epsilon, u.y + \epsilon])$, e.g., the red square in Fig. 2. Thus, at time 1, $RQ(o_6, \epsilon) = \{(o_6, o_5), (o_6, o_7)\}$.

For clustering, we have to check every object o in S_t to determine whether o is a core point, i.e., we perform a range query for every object o. Therefore, a range join can be used in the first step of DBSCAN in order to improve efficiency.

DEFINITION 11. **(Range Join)** *Given a set O of locations and a threshold ϵ, a range join finds all location pairs in O with their distances no larger than ϵ, i.e., $RJ(O, \epsilon) = \{(u, v) | d(u, v) \leq \epsilon, u \in O, v \in O\}$.*

For example, in Fig. 2, given a set of locations at time 1 (i.e., $O = \{o_1, o_2, ..., o_8\}$) and a threshold ϵ, $RJ(O, \epsilon) = \{(o_1, o_2), (o_3, o_4), (o_5, o_6), (o_6, o_7)\}$.

4. OVERVIEW OF CO-MOVEMENT PATTERN DETECTION

In this section, we present an overview of co-movement pattern detection over streaming trajectories. Fig. 3 shows the framework and the processing flow, termed as Indexed Clustering and Pattern Enumeration (ICPE). ICPE takes streaming trajectories as input.

First, it uses window operations to transform the streaming trajectories into snapshots, as discussed in Section 3.1. For example, in Fig. 3, the streaming trajectories are transformed into snapshots, i.e., a snapshot at time 1, a snapshot at time 2, and so on.

Second, ICPE performs index-based clustering based on RangeJoin and DBSCAN, to be detailed in Section 5. When a new snapshot arrives, ICPE detects the clusters of the trajectories that move together, in order to obtain a cluster snapshot. For instance, in Fig. 3, we get several clusters for the snapshot at time 2, i.e., cluster 0: {0, 2, 3, 56}, cluster 1: {1, 3, 4}, and so forth.

Finally, when each cluster snapshot comes, ICPE utilizes Pattern Enumeration to obtain all co-movement patterns, to be covered in Section 6. For example, in Fig. 3, we get several patterns at time 2, i.e., {0, 2, 56}, {1, 3, 4}, etc.

During the first step, we also need to consider time synchronization for stream processing. Flink cannot ensure that trajectories are processed in time order. However, pattern detection requires that trajectories are processed in ascending time order. Hence, we add "last time" information to each trajectory in every snapshot, to be able to guarantee that trajectories are processed in time order. The "last time" denotes the time of the most recent snapshot for which the trajectory reported a location.

Using the "last time", we can determine whether the system needs to wait for the location of a particular time. As an example, for a stream trajectory $tr = \{r_1, r_2, r_3, r_5, ...\}$, where r_i denotes the GPS record at time i, the "last time" associated with r_3 is time 2, and the "last time" associated with r_5 is time 3. In case the system has only received r_1 and r_3, it must wait for r_2, because "last time" information of r_3 indicates that a location was reported by the trajectory for the snapshot at time 2 that has yet to be received. Next, in case the system has received r_1, r_2, r_3, and r_5, it does not need to wait for r_4, because the "last time" information of r_5 indicates that no position was reported for the snapshot at time 4.

5. INDEXED CLUSTERING

In this section, we first describe the GR-index, and then present the index based RangeJoin and DBSCAN methods.

5.1 GR-index

As stated in Section 4, clustering includes two steps, i.e., we first perform a range join on each snapshot S_t, and then, we use DBSCAN to cluster S_t. To accelerate the range join, a two-layered index, the **GR-index** [36], is built in Flink. GR-index is verified efficient for distributed platforms (e.g., Spark, Storm). It uses a grid index as a global index, and builds an R-tree [3] as a local index for each grid cell.

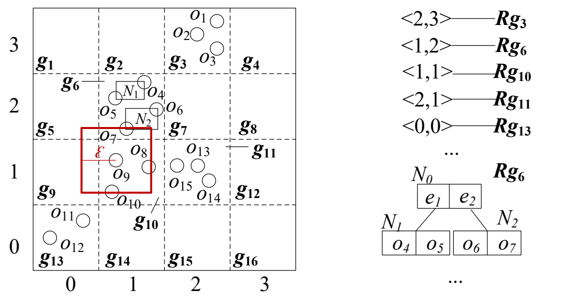
Fig. 4 illustrates a GR-index for a specific snapshot, in which Fig. 4(a) shows the global grid index and Fig. 4(b) shows the local R-trees. The GR-index has 16 grid cells, and an R-tree for grid cell g_6 is shown as an example. The R-tree leaf nodes N_1 and N_2 contain the real locations o_4, o_5, o_7, and o_8, and the minimum bounding rectangles of N_1 and N_2 are shown in g_6.

Key Computation. Each grid cell can be regarded as a partition in Flink. The key of the grid cell that a location $o = (x, y)$ belongs to can be computed as $\langle \lfloor o.x/l_g \rfloor, \lfloor o.y/l_g \rfloor \rangle$, where l_g is the grid cell width. For example, as shown in Fig. 4, given the location $o_5 = (4, 8)$ and the grid cell width $l_g = 3$, the key of g_6 that o_5 belongs to is $\langle 1, 2 \rangle$.

Note that, the GR-index is a primary index. We compute a key for each location, and locations with the same key will be distributed to the same subtask when building local R-trees.

5.2 GR-index Based Range Join

We develop three algorithms to compute the range join based on the GR-tree, i.e., GridAllocate, GridQuery, and GridSync. Fig. 5 depicts the processing of the ICPE-RangeJoin. GridAllocate builds the global grid index (i.e., computes the key for each location) to partition each snapshot into disjoint grid cells, and it transforms locations into data objects (i.e., locations contained in a specific grid cell) and query objects (i.e., locations whose range region intersects



(a) Grid index (b) R-trees for all grid cells

Figure 4: Example of a GR-index

with this grid cell). For each grid cell (key), a GridQuery method builds the local R-tree for the data objects, and performs a range query for each query object. Finally, GridSync collects the results from all grid cells. Here, a GR-index is built for each snapshot, and is deleted after querying. Thus, index maintenance is omitted.

According to the functioning of Flink, after building the grid index, the partitions are independent of each other, i.e., locations in other partitions cannot be accessed. However, during range-join processing, we have to access other partitions. As shown in Fig. 4, although o_9 is not located in grid cell g_6, the range query result *RQ*(o_9, ϵ) includes location o_7 that is located in grid cell g_6. To address this, we replicate each location into multiple GridObjects, and distribute them to the relevant grid cells.

DEFINITION 12. **(GridObject)**. *A GridObject go = (key, flag, location) is a triple, where location is the actual position of go, flag indicates the type of go, and key records the grid cell that go belongs to.*

Specifically, if *flag* is false, then *go* is a data object, meaning that its *location* needs to be inserted into the corresponding R-tree of this grid cell; otherwise, if *key* is true, then *go* is a query object, indicating that the grid cell with *key* might contain the range query result for *go*.

Based on the definition of GridObject, a location can be represented as a data object (*key*(g), false, *location*), where g is the original grid cell that *location* belongs to; and several query objects (*key*(g_i), true, *location*), in which grid cells g_i intersect with the range region of *RQ*(*location*, ϵ).

For example, in Fig. 4, o_9 can be represented as a data object go_1 ($\langle 1, 1 \rangle$, false, o_9), indicating that o_9 needs to be inserted into the R-tree of grid cell g_{10}. In addition, according to the range region (i.e., the red square centered at o_9 with length 2ϵ), o_9 is represented by four query objects $go_2 = (\langle 0, 2 \rangle$, true, $o_9)$, $go_3 = (\langle 1, 2 \rangle$, true, $o_9)$, $go_4 = (\langle 0, 1 \rangle$, true, $o_9)$, and $go_5 = (\langle 1, 1 \rangle$, true, $o_9)$, because the range query result for o_9 is contained in grid cells g_5, g_6, g_9, and g_{10}. To improve the efficiency of the range join, we develop two lemmas below to avoid unnecessary verifications.

According to the definition of a range query, we need to verify all the grid cells that intersect with the range region. Nevertheless, for a range join on a single dataset, verifying all the grid cells might yield duplicated results [4]. For instance, in Fig. 4, o_9 needs to be distributed to grid cell g_6, as the range region of o_9 intersects with g_6, and thus, o_9 is represented by the GridObject ($\langle 1, 2 \rangle$, true, o_9) to find the result pair (o_9, o_7). In addition, o_7 needs to be distributed to grid cell g_{10}, and thus, it is represented by the GridObject ($\langle 1, 1 \rangle$, true, o_7) to find the result pair (o_7, o_9). The outcome is that pair (o_7, o_9) is duplicated in the result. The idea of the first lemma is that, instead of verifying all the grid cells intersected with the range region ($[o.x - \epsilon, o.x + \epsilon]$, $[o.y - \epsilon, o.y + \epsilon]$), we only verify half of those grid cells, i.e., the grid cells that intersect with the upper part

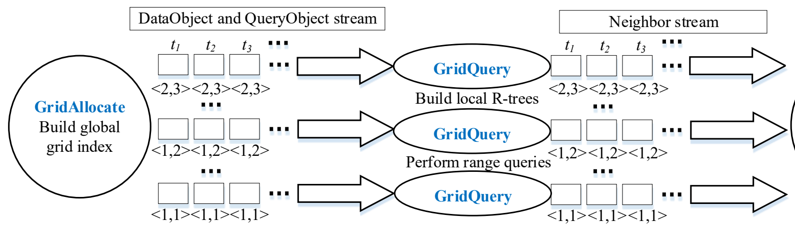
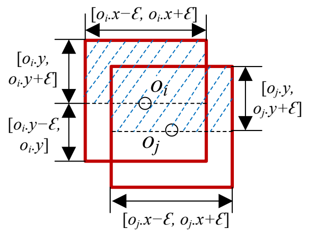


Figure 5: ICPE-RangeJoin



$[o_i.x-\mathcal{E},\ o_i.x+\mathcal{E}]$
$[o_i.y,\ o_i.y+\mathcal{E}]$
$[o_i.y-\mathcal{E},\ o_i.y]$
$[o_j.y,\ o_j.y+\mathcal{E}]$
o_i
o_j
$[o_j.x-\mathcal{E},\ o_j.x+\mathcal{E}]$

Figure 6: Illustration of Lemma 1

Algorithm 1: GridAllocate Algorithm

```
  Input: a snapshot $S_t$, a grid cell width $l_g$, a threshold $\epsilon$
1 for each location $o \in S_t$ do
2     $key \leftarrow \langle \lfloor o.x/l_g \rfloor, \lfloor o.y/l_g \rfloor \rangle$
3     $S_{key} \leftarrow \{\langle x,y \rangle | x \in [\lfloor (o.x-\epsilon)/l_g \rfloor, \lfloor (o.x+\epsilon)/l_g \rfloor], y \in$
          $[\lfloor o.y/l_g \rfloor, \lfloor (o.y+\epsilon)/l_g \rfloor]\} - \{key\}$          ▷ Lemma 1
4     output GridObject($key$, false, $o$)                        ▷ data object
5     for each $key_i \in S_{key}$ do
6         output GridObject($key_i$, true, $o$)                    ▷ query object
```

Algorithm 2: GridQuery Algorithm

```
  Input: a stream of GridObjects $S_q$, a distance threshold $\epsilon$
1 initialize the R-tree $rt \leftarrow \emptyset$
2 for each GridObject $o \in S_q$ and $o.flag = false$ do
3     output $rt$.query($o$, $\epsilon$)
4     $rt$.insert($o$)
5 for each GridObject $o \in S_q$ and $o.flag = true$ do
6     output $rt$.query ($o$, $\epsilon$)
```

of the range region $([o.x-\epsilon, o.x+\epsilon], [o.y, o.y+\epsilon])$, as illustrated in Fig. 6.

LEMMA 1. *Given a set O of locations, a grid cell width l_g, and a distance threshold ϵ, if every location $o = (x,y) \in O$ is represented as one go = (key, false, o) with key = $\langle \lfloor o.x/l_g \rfloor, \lfloor o.y/l_g \rfloor \rangle$ and multiple go$_i$ = (key$_i$, true, o) with key$_i$ $\in \{\langle x,y \rangle | x \in [\lfloor (o.x-\epsilon)/l_g \rfloor, \lfloor (o.x+\epsilon)/l_g \rfloor], y \in [\lfloor o.y/l_g \rfloor, \lfloor (o.y+\epsilon)/l_g \rfloor]\} - \{key\}$, no result of RJ$(O,\epsilon)$ is missed.*

PROOF. To prove Lemma 1, we only need to prove that no result is missed if the range query *RQ*(o_i,ϵ) only considers in the upper part of the range region $([o_i.x-\epsilon, o_i.x+\epsilon], [o_i.y, o_i.y+\epsilon])$. As shown in Fig. 6, assume that a location o_j in the lower part of the range region $([o_i.x-\epsilon, o_i.x+\epsilon], [o_i.y-\epsilon, o_i.y])$ is not in the result of *RQ*(o_i,ϵ), i.e., (o_i,o_j) is not included in *RJ*(O,ϵ). For the location o_j, it has to search in the upper part of the range region $([o_j.x-\epsilon, o_j.x+\epsilon], [o_j.y, o_j.y+\epsilon])$, which contains o_i. Hence, (o_j,o_i) is returned by *RQ*(o_j,ϵ). For *RJ*(O,ϵ), $(o_j,o_i) = (o_i,o_j)$ due to the symmetry property, which contradicts the assumption that (o_i,o_j) is not included. □

Based on Lemma 1, we propose the GridAllocate algorithm. The pseudo-code is depicted in Algorithm 1. It takes as inputs a snapshot S_t, a grid cell width l_g, and a distance threshold ϵ. For each location o in S_t, the algorithm first computes the key of the grid cell that o belongs to (line 2). Next, it computes the keys S_{key} of the set of grid cells that might contain locations that can contribute to the range query result of o according to Lemma 1 (line 3). Then, it compacts and outputs o as a data object (*key*,*false*, o) (line 4). Finally, for each *key*$_i$ in S_{key}, the algorithm compacts and outputs o as a query object (*key*$_i$, *true*, o) (lines 5–6).

After performing GridAllocate, a partition (associated with a particular *key*) obtains a set of data objects on which to build the R-tree and a set of query objects for which to compute range queries. Further, we also need to perform a range query for each data object in the R-tee. A traditional approach is to build an R-tree using the data objects and then perform the range query of each data object and query object based on the R-tree. By inspired by [23], according to the symmetry property of the range join on a single dataset, we develop a lemma below to further avoid duplications.

LEMMA 2. *Assume a set O of data objects in a particular partition, a distance threshold ϵ, and an empty R-tree rt. Then, for every data object o in O, if we first perform range query RQ(o,ϵ) on rt and then insert o into rt, no result of RJ(O,ϵ) is missed.*

PROOF. For a data object o_i in O, assume that the result (o_i, o_j) is not in the result of the range query *RQ*(o_i,ϵ) on the current R-tree rt, as o_j comes after o_i, and o_j is not yet inserted into rt. However, when we process the data object o_j, and perform the range query *RQ*(o_j,ϵ) on the current rt, the result (o_j, o_i) will be returned. This is because, o_i is already inserted into rt as o_i comes before o_j. For *RJ*(O,ϵ), we can get that (o_i, o_j) = (o_j, o_i) due to the symmetry property. Therefore, this observation contradicts the assumption that (o_i, o_j) is not reported. □

Based on Lemma 2, we present the GridQuery algorithm, with its pseudo-code shown in Algorithm 2. It takes as inputs a stream of GridObjects S_q and a distance threshold ϵ. The algorithm first initializes an empty R-tree rt (line 1). Next, for each data object $o \in S_q$ (i.e., *o.flag* = *false*), it first performs a range query rt.query(o, ϵ) on rt and outputs the result, and then, it calls the insert function to insert o into rt (lines 2–4). Thereafter, for each query object $o \in S_q$ (i.e., $o.flag = true$), the algorithm performs a range query rt.query(o, ϵ), and outputs the result (lines 5–6).

Having performed the GridQuery algorithm, we obtain a neighbor stream. The GridSync Algorithm that collects all the results is omitted, as it is straightforward.

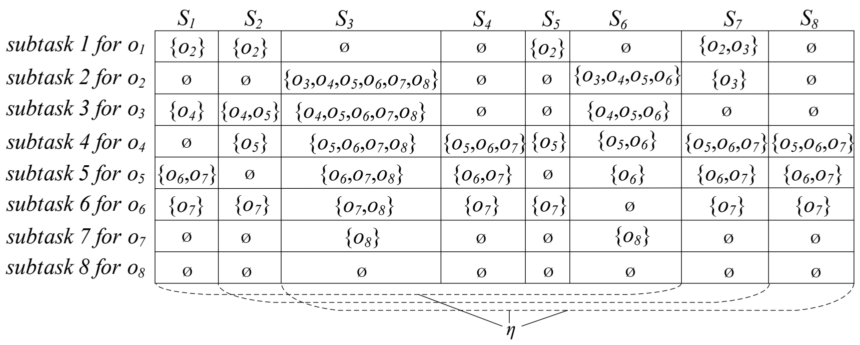
5.3 GR-index Based DBSCAN

Next, we apply our DBSCAN method to the result of a range join, as the core points and the density reachable points can be easily retrieved from the result of range join. The pseudo-code is omitted due to the space limitation.

As described elsewhere [8, 15], we can split the neighbor snapshot into several parts and then perform DBSCAN on each part in parallel, and finally collect the results. However, the time complexity of our DBSCAN method is O(n), where n is the number of the locations contained in each snapshot. This is relatively inexpensive compared with O(n^2) cost of the centralized range join. Thus, we do not need to further split every snapshot to achieve more parallelism. In our ICPE framework, we achieve the parallelism by clustering snapshots separately.

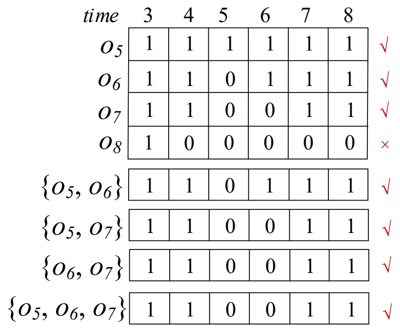
6. PATTERN ENUMERATION

In this section, we present three approaches for pattern enumeration over cluster streams.



	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
subtask 1 for o_1	$\{o_2\}$	$\{o_2\}$	∅	∅	$\{o_2\}$	∅	$\{o_2,o_3\}$	∅
subtask 2 for o_2	∅	∅	$\{o_3,o_4,o_5,o_6,o_7,o_8\}$	∅	∅	$\{o_3,o_4,o_5,o_6\}$	$\{o_3\}$	∅
subtask 3 for o_3	$\{o_4\}$	$\{o_4,o_5\}$	$\{o_4,o_5,o_6,o_7,o_8\}$	∅	∅	$\{o_4,o_5,o_6\}$	∅	∅
subtask 4 for o_4	∅	$\{o_5\}$	$\{o_5,o_6,o_7,o_8\}$	$\{o_5,o_6,o_7\}$	$\{o_5\}$	$\{o_5,o_6\}$	$\{o_5,o_6,o_7\}$	$\{o_5,o_6,o_7\}$
subtask 5 for o_5	$\{o_6,o_7\}$	∅	$\{o_6,o_7,o_8\}$	$\{o_6,o_7\}$	∅	$\{o_6\}$	$\{o_6,o_7\}$	$\{o_6,o_7\}$
subtask 6 for o_6	$\{o_7\}$	$\{o_7\}$	$\{o_7,o_8\}$	$\{o_7\}$	$\{o_7\}$	∅	$\{o_7\}$	$\{o_7\}$
subtask 7 for o_7	∅	∅	$\{o_8\}$	∅	∅	$\{o_8\}$	∅	∅
subtask 8 for o_8	∅	∅	∅	∅	∅	∅	∅	∅

Figure 7: Example of Id-based Partitioning for Fig. 2



time	3	4	5	6	7	8	
o_5	1	1	1	1	1	1	√
o_6	1	1	0	1	1	1	√
o_7	1	1	0	0	1	1	√
o_8	1	0	0	0	0	0	×
$\{o_5, o_6\}$	1	1	0	1	1	1	√
$\{o_5, o_7\}$	1	1	0	0	1	1	√
$\{o_6, o_7\}$	1	1	0	0	1	1	√
$\{o_5, o_6, o_7\}$	1	1	0	0	1	1	√

Figure 8: Bit Compression on $P_3(o_4)$

6.1 Baseline

We adapt the state-of-the-art distributed co-movement pattern detection method (i.e., SPARE [10]) on historical trajectories as the baseline algorithm. We note that SPARE uses a star partitioning scheme for historical trajectories. However, this partitioning cannot be applied to streaming trajectories, because we do not know which trajectories are related in advance, and they thus cannot be distributed to the same partition at the beginning. Instead, we present an id-based partitioning technique.

ID-based Partitioning Technique. A Flink subtask is created for each trajectory o with the *key* equaling to the trajectory id $o.id$. We use partition $P_t(o)$ to denote the set of trajectories distributed to subtask $o.id$ at time t. In order to avoid duplications, $P_t(o)$ contains other trajectories (except for o) in the same cluster with their ids larger than $o.id$. Note that, at different times t_i, partitions $P_{t_i}(o)$ will be sent to the same subtask $o.id$ for processing.

Fig. 7 shows all the partitions for Fig. 2, where a subtask is created for each of 8 trajectories. At time 1, the cluster snapshot is $\{(o_1, o_2), (o_3, o_4), (o_5, o_6, o_7)\}$, which results in partitions: $P_1(o_1) = \{o_2\}$ for subtask 1, $P_1(o_2) = \emptyset$ for subtask 2, $P_1(o_3) = \{o_4\}$ for subtask 3, $P_1(o_4) = \emptyset$ for subtask 4, $P_1(o_5) = \{o_6, o_7\}$ for subtask 5, $P_1(o_6) = \{o_7\}$ for subtask 6, $P_1(o_7) = \emptyset$ for subtask 7, and $P_1(o_8) = \emptyset$ for subtask 8.

Based on the significance constraint M, the lemma below is used to find the valid clusters for a partition.

LEMMA 3. *Given a cluster C and a significance constraint M, if $|C| < M$, then C can be discarded.*

PROOF. The proof is simple due to the significance constraint M, and it is thus omitted. □

As an example, in Fig. 2, if $M = 3$, the clusters $\{o_1, o_2\}$ and $\{o_3, o_4\}$ at time 1 can be discarded.

Pattern Enumeration. For each partition $P_t(o)$ at time t, we first enumerate all possible combinations of trajectories, and then find the valid time sequence for each combination. Specifically, we first initialize all possible patterns $O \subseteq P_t(o) \cup \{o\}$, where $|O| \geq M$. Note that, pattern enumeration on partition $P_t(o)$ should include o. For simplicity, the pattern enumeration on partition $P_t(o)$ removes o because o is a common trajectory. Considering the example in Fig. 7, given a partition $P_1(o_5) = \{o_6, o_7\}$ and $M = 2$, the possible patterns include $\{o_6\}$, $\{o_7\}$, and $\{o_6, o_7\}$, where o_5 is a comment element and is omitted in the patterns.

Pattern Verification. Next, we determine whether each pattern O enumerated in $P_t(o)$ is valid, i.e., we try to find the valid time sequence T for O in the partitions $P_i(o)$ $(i \geq t)$. More specifically, for a pattern O, T is first initialized to $\{t\}$. If O also exists in $P_{t'}(o)$ at the next time t', then $T = T \cup \{t'\}$. If T satisfies the K, L, and G constraints in Definition 4, then O is valid. As proved in [10], no valid pattern is missed if every η snapshots are verified.

LEMMA 4. $\eta = (\lceil \frac{K}{L} \rceil - 1) \times (G-1) + K + L - 1$ *guarantees that no valid pattern is missed.*

PROOF. The proof can be found elsewhere [10]. □

Hence, for pattens enumerated in $P_t(o)$, we need to use η snapshots $P_i(o)$ $(t \leq i \leq t+\eta-1)$ to determine whether they are valid. For example, in Fig. 7, if $K = 4$ and $G = L = 2$, then $\eta = 6$, and thus, we use $P_i(o)$ $(1 \leq i \leq 6)$ to verify the patterns enumerated in $P_1(o)$. Although η snapshots are being processed at the same time, this does not mean that our methods are batch methods. This processing is simply necessary because multiple snapshots are needed for verifying the validity according to Definition 4.

Based on the consecutive constraint L and the connection constraint G, two lemmas [10] can be used to avoid unnecessary verifications when finding valid patterns.

LEMMA 5. *Given a pattern O enumerated in partition $P_t(o)$, a consecutive constraint L, and a time sequence T obtained before the current time t', assume that the last time segment T_l of T satisfies $|T_l| < L$. If $O \subseteq P_{t'}(o)$ and $t' - max(T) \neq 1$, then O can be discarded.*

PROOF. The proof is straightforward due to $T = T \cup \{t'\}$ does not satisfy consecutive constraint L. □

For instance, in Fig. 7, considering a pattern $O = \{o_2\}$ enumerated in $P_1(o_1)$, we can get $T = \langle 1, 2, 5 \rangle$ before the current time $t' = 7$. Given $L = 2$, the length of the last time segment $T_l = \{5\}$ in T is smaller than L. In addition, as $O \subseteq P_7(o_1)$ and $t - max(T) = 7 - 5 = 2 > 1$, $O = \{o_2\}$ can be discarded. This holds because $T = \langle 1, 2, 5, 7 \rangle$ does not satisfy the consecutive constraint L.

LEMMA 6. *Given a pattern O enumerated in partition $P_t(o)$, a connection constraint G, and a time sequence T obtained before the current time t', if $O \subseteq P_{t'}(o)$ and $t' - max(T) > G$, then O can be discarded.*

PROOF. For a time sequence T obtained before the current time t', if $O \subseteq P_{t'}(o)$ and $t' - max(T) > G$, then $T \cup \{t'\}$ does not satisfy constraint G, and thus, O can be discarded. □

For example, in Fig. 7, considering a pattern $O = \{o_4\}$ enumerated in $P_1(o_3)$, we can get $T = \langle 1, 2, 3 \rangle$ before the current time $t' = 6$. If $G = 2$, $O \subset P_6(o_3)$ and $t' - max(T) = 6 - 3 = 3 > 2$, then O can be discarded according to Lemma 6.

Based on Lemmas 3 to 6, we present Baseline algorithm, with the pseudo-code shown in Algorithm 3. It takes as inputs a partition $P_t(o) = \{o_i | 1 \leq i \leq |P_t(o)|\}$ and four constraints (M, K, L, G). First, Baseline initializes an empty list H, and computes $\eta = (\lceil \frac{K}{L} \rceil - 1) \times (G-1) + K + L - 1$ (line 1). Then, it enumerates all possible

Algorithm 3: Baseline

```
   Input: Partition $P_t(o) = \{o_i | 1 \le i \le |P_t(o)|\}$, $(M, K, L, G)$
          required for co-movement pattern
1  initialize a list $H \leftarrow \emptyset$ and
   $\eta \leftarrow (\lceil \frac{K}{L} \rceil - 1) \times (G-1) + K + L - 1$
2  for each possible $O \subseteq P_t(o)$ with $|O| \ge (M-1)$ do
3      H.insert($\langle O, T \rangle$)                          ▷ $T = \{t\}$
4  for each candidate pattern $h \in H$ do
5      for each partition $P_i(o)$ $(t+1 \le i \le t+\eta-1)$ do
6          if $(h.O \subseteq P_i \wedge (i - max(h.T)) = 1)$ or $(h.O \subseteq P_i$
             $\wedge |T_l| \ge L \wedge (i - max(h.T)) \le G)$ then
7              $h.T \leftarrow h.T \cup \{i\}$
8              if $|h.T| \ge K \wedge |T_l| \ge L$ then
9                  output $h$
10                 break
11         else
12             H.remove($h$)
```

patterns O ($O \subseteq P$, $|O| \ge (M-1)$), and inserts $\langle O, T \rangle$ ($T = \{t\}$) into H (lines 2–3). In the sequel, for each candidate pattern h in H and each next partition $P_i(o)$ $((t+1) \le i \le (t+\eta-1))$, the algorithm uses Lemmas 5 and 6 to determine whether or not to remove h (lines 4–6). If h satisfies the conditions of Lemmas 5 and 6, it is removed from H (lines 11–12); otherwise, $h.T \leftarrow h.T \cup \{i\}$ (line 7), and the algorithm proceeds to verify the current pattern h (lines 8–10). If the current pattern h is valid, Baseline outputs h (line 9), and breaks to process the next pattern (line 10).

Time Complexity and Storage Cost. Given a partition $P_t(o)$, the number of all possible patterns in $P_t(o)$ is $C^{M-1}_{|P_t(o)|} + C^{M}_{|P_t(o)|} + \ldots + C^{|P_t(o)|}_{|P_t(o)|} \approx O(2^{|P_t(o)|})$. For each possible pattern, we use η snapshots to verify it. Thus, the time complexity of Baseline is $O(\eta \times 2^{|P_t(o)|})$, and the storage cost is $O(2^{|P_t(o)|})$, which is huge. For a large $|P_t(o)|$, Baseline cannot run due to the storage cost.

6.2 Fixed Length Bit Compression Method

To reduce the exponential time and storage costs of Baseline, we present a **fixed length bit compression method**. The fixed length bit string representation of a trajectory's cluster membership is defined below.

DEFINITION 13. **(Fixed Length Bit String)** *Given a trajectory o_i in partition $P_t(o)$, a fixed length bit string $B[o_i]$ is used to represent o_i, where $|B[o_i]| = \eta$, $B[o_i][j] = 1$ $(0 \le j \le (\eta - 1))$ denotes that o and o_i belong to the same cluster at time $t+j$, while $B[o_i][j] = 0$ indicates that o and o_i belong to different clusters.*

For instance, in Figs. 7 and 8, given the partition $P_3(o_4) = \{o_5, o_6, o_7, o_8\}$ at time 3, trajectory o_5 is represented as $B[o_5] = 111111$ (o_4 and o_5 belong to the same cluster at times 3, 4, 5, 6, 7, and 8), and trajectory o_8 is represented as $B[o_8] = 100000$ (o_4 and o_8 belong to the same cluster only at time 3).

Baseline stores every possible combination of trajectories in one partition $P_t(o)$. In contrast, each trajectory is now represented as a bit string of length η. As a result, the storage cost is reduced from $(2^{|P_t(o)|})$ to $O(\eta \times |P_t(o)|)$. However, during verification, we still need to determine whether each combination of trajectories is a valid pattern. For every possible combination O of objects in the partition $P_t(o)$ $(O \subseteq P_t(o))$, we thus also use a fixed length bit string $B[O]$, where bit $B[O][j]$ $(0 \le j \le (\eta-1))$ denotes whether trajectories in $O \cup \{o\}$ belong to the same cluster at time $t + j$.

Bit Operation. To compute the bit string for a set of trajectories $O = \{o_x | 1 \le x \le m\}$, we use the bitwise AND operator on all

Algorithm 4: Fixed Length Bit Compression based Algorithm (FBA)

```
   Input: Partition $P_t(o) = \{o_i | 1 \le i \le |P_t(o)|\}$, $(M, K, L, G)$
          required for co-movement pattern
1  a list $C \leftarrow \emptyset$ and $\eta \leftarrow (\lceil \frac{K}{L} \rceil - 1) \times (G-1) + K + L - 1$
2  for each object $o_i \in P_t(o)$ do
3      initialize a bit string $B[o_i] \leftarrow 0$ with length $\eta$
4      for each partition $P_j(o)$ $(t \le j \le (t+\eta-1))$ do
5          if $o_i \in P_j(o)$ then
6              $B[o_i][j-t] \leftarrow 1$
7      if $B[o_i]$ satisfies $(K, L, G)$ then
8          C.insert($o_i$)
9  $S \leftarrow$ SubSet($C, M-2$)                          ▷ $S$.level = $M-2$
10 while $S \ne \emptyset$ and $S.level \le |C|$ do
11     $S_a \leftarrow \emptyset$
12     for each pattern $O \in S \times C$ do
13         $B[O] \leftarrow \&B[o_j](o_j \in O)$
14         if $B[O]$ is valid then
15             output $O$
16             $S_a \leftarrow S_a \cup \{O\}$
17     $S \leftarrow S_a$
```

$B[o_x]$ $(o_x \in O)$, i.e., $B[O] = \&B[o_x]$ $(o_x \in O)$. This holds because, $B[O][j] = 1$ iff $\forall o_x \in O$ $(B[o_x] = 1)$. In the example in Fig. 8, $B[\{o_5, o_6\}] = B[o_5]$ & $B[o_6] = 110111$, and $B[\{o_5, o_6, o_7\}] = B[o_5]$ & $B[o_6]$ & $B[o_7] = 110011$. To verify whether $B[O]$ satisfies the (M, K, L, G) constraints, Lemmas 3 to 6 can be adapted to use the bit strings similarly.

Candidate based Pattern Enumeration. In order to reduce the exponential cost of enumeration in each partition $P_t(o)$, we propose a two-step candidate set based method.

First, we find a candidate set C of trajectories whose bit strings $B[o_i]$ $(o_i \in P_t(o))$ satisfy the (K, L, G) constraints. For instance, as illustrated in Fig. 8, given $K = 4$, $L = 2$, and $G = 2$, we can get $C = \{o_5, o_6, o_7\}$. This is because $B[o_8]$ does not satisfy the (K, L, G) constraints.

Second, we enumerate all possible patterns $O \subseteq C$ to find valid ones. According to the Apriori Enumerator [10], we first enumerate patterns $O \subseteq C$ with $|O| = 2$, and then incrementally increase the cardinality by one until $|O| > |C|$. In each iteration, for every valid pattern O (i.e., $B[O]$ satisfies the (K, L, G) constraints), we proceed to evaluate patterns $O \times C$, i.e., any candidate in C can be inserted into O to yield a new pattern. However, instead of enumerating from $|O| = 2$, we directly enumerate from $|O| = M-1$, saving the enumeration cost from 2 to $M-1$, obtaining substantial savings when M is large.

For example, as depicted in Fig. 8, $C = \{o_5, o_6, o_7\}$ and $M = 3$. We first enumerate all patterns with cardinality 2, i.e., $\{o_5, o_6\}$, $\{o_5, o_7\}$, and $\{o_6, o_7\}$. In the second step, for valid pattern $\{o_5, o_6\}$, we increase its cardinality from 2 to 3 (i.e., we consider $\{o_5, o_6\} \times C$), resulting in $\{o_5, o_6, o_7\}$. Note that, $\{o_5, o_6, o_5\}$ and $\{o_5, o_6, o_6\}$ are omitted, since duplicated elements are not allowed in patterns. In the third iteration, for valid pattern $\{o_5, o_6, o_7\}$, we can stop enumerating because of the termination condition (i.e., $4 > |C|$).

Using the bit compression and candidate based pattern enumeration techniques, we develop the Fixed Length Bit Compression based Algorithm (FBA). The pseudo-code is presented in Algorithm 4. It takes as inputs a partition $P_t(o)$ and four (M, K, L, G) constraints. First, FBA initializes an empty candidate list C, and computes η (line 1). Then, it obtains the bit string $B[o_i]$ for each o_i in $P_t(o)$ using η partitions $P_j(o)$ $(t \le j \le (t+\eta-1))$ (lines

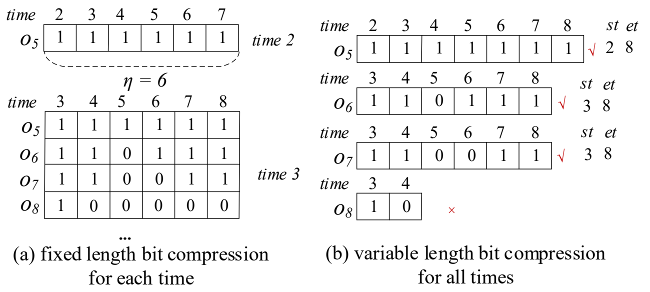


Figure 9: Bit Compression for Subtask of o_4 in Fig. 2

2–6), and it inserts the valid o_i whose $B[o_i]$ satisfies the (K, L, G) constraints into C (lines 7–8). Next, the algorithm enumerates all subsets of C in S with the subset's cardinality (i.e., S.level) equaling $M - 2$ (line 9). In the sequel, a while loop is used to incrementally increase the level of subsets, and output the valid patterns (lines 10–17). If S is empty or S.level $> |C|$, the while loop stops.

Time Complexity and Storage Cost. Given a partition $P_t(o)$, the bit compression reduces the storage cost from $O(2^{|P_t(o)|})$ to $O(\eta \times |P_t(o)|)$. Let R be the final pattern result set in $P_t(o)$. Using the candidate set C, the time complexity of pattern enumeration is reduced from $O(2^{|P_t(o)|})$ to $O(|R| \times |C| + C_{|C|}^{M-1})$. Here, $O(C_{|C|}^{M-1})$ is the cost of enumerating all patterns O from C with $|O| = M - 1$ in the first step, and $O(|R| \times |C|)$ is the total enumeration cost of incrementally increasing the pattern cardinality.

6.3 Variable Length Bit Compression Method

Both Baseline and FBA find valid patterns every η snapshots, resulting in the same snapshot being involved in multiple verifications. Consider the example in Fig. 2, where $[S_1, S_6]$ is verified for snapshot S_1, $[S_2, S_7]$ is verified for snapshot S_2, and so on. In this example, S_2 is verified twice. To address this, we develop a variable length bit compression method that verifies each snapshot only once.

Variable Length Bit Compression. The partitioning method is the same as that in the Baseline and Bit Compression based Algorithm, i.e., a subtask is created for each trajectory. However, instead of using a fixed length bit string for every trajectory in $P_t(o)$ of a specific time t, we use a variable length bit string to represent each trajectory assigned to the subtask of o over all times.

DEFINITION 14. **(Variable Length Bit String)** *Given a trajectory o_i assigned to the subtask of trajectory o, o_i can be represented as a variable length bit string $\langle st_i, et_i, B[o_i]\rangle$, where st_i is the start time, et_i is the end time, and $B[o_i][t - st_i] = 1$ means that o and o_i belong to the same cluster at times $t \in [st_i, et_i]$, while $B[o_i][t - st_i] = 0$ indicates that o and o_i belong to different clusters at times $t \in [st_i, et_i]$.*

Fig. 9 shows the two different bit compression methods, where Fig. 9(a) uses fixed length bit strings for each time, while Fig. 9(b) uses variable length bit strings over all times. More specifically, for the subtask of o_4, o_5 is represented as the variable length bit string $\langle 2, 8, 1111111\rangle$, while o_5 is represented as two fixed length bit strings (i.e., '111111') at both times 2 and 3. Hence, the storage cost is further reduced.

THEOREM 1. *Given a subtask of trajectory o, three (K, L, G) constraints, and the total number n of occurrences for trajectories at any time $t \geq 1$ of this subtask, the total storage cost for the subtask is $O(n\frac{G+L}{L})$ when using variable length bit strings, and the total storage cost of the subtask is $O(n \times (\lceil\frac{K}{L}\rceil - 1) \times (G - 1) + K + L - 1))$ when using fixed length bit strings.*

PROOF. The fixed length bit compression method needs $O(\eta)$ space to store the fixed length bit string for every occurrence of a trajectory at any time $t \geq 1$, where $\eta = (\lceil\frac{K}{L}\rceil - 1) \times (G - 1) + K + L - 1$. Hence, the storage cost is $O(n \times (\lceil\frac{K}{L}\rceil - 1) \times (G - 1) + K + L - 1))$.

With the variable length bit compression method, every occurrence of a trajectory at one particular time $t \geq 1$ is represented by a bit '1' in an variable length bit string. For a variable length bit string B that satisfies the (K, L, G) constraints, we can get that $n_o < n_1\frac{G}{L}$, where n_0 is the number of '0's in B and n_1 is the number of '1's in B. Thus, given n '1's in all the variable length bit strings, the total storage cost is $O(n\frac{G+L}{L})$. □

Pattern Enumeration. To further reduce the cost of enumeration, we use a candidate list C to only store the bit strings $\langle st_i, et_i, B[o_i]\rangle$ with maximal pattern time sequences.

DEFINITION 15. **(Maximal Pattern Time Sequence)** *T is a maximal pattern time sequence for a pattern O, iff (i) T satisfies the (K, L, G) constraints, and (ii) no T' exists such that $T \cup T'$ also satisfies the (L, G, K) constraints.*

Next, we develop a lemma to help obtain bit strings with maximal pattern time sequences.

LEMMA 7. *Given a variable length bit string $\langle st_i, et_i, B[o_i]\rangle$ in the subtask of o that satisfies the (K, L, G) constraints, if $B[o_i][et_i + j] = 0$ $(1 \leq j \leq (G + 1))$, then $T = \{t | t \in [st_i, et_i] \wedge B[t - st_i] = 1\}$) is a maximum pattern time sequence.*

PROOF. The proof is simple due to Definition 15 and the G constraint, and thus, it is omitted. □

For example, in Fig. 9, assume that objects o_5, o_6, and o_7 do not belong to the same cluster as o_4 at future times 9, 10, and 11. Given $L = G = 2$ and $K = 4$, then $\langle 2, 8, B[o_5] = 1111111\rangle$, $\langle 3, 8, B[o_6] = 110111\rangle$, and $\langle 3, 8, B[o_7] = 110011\rangle$ are three maximal pattern time sequences.

After obtaining a new candidate bit string s that has a maximal pattern time sequence, we first enumerate all possible patterns in $s \cup C$ (C is the global candidate set), and then, we insert s into C. The enumeration method is similar to that discussed in Section 6.2. However, since the bit strings have variable lengths, a new lemma is developed for enabling the punning of unqualified combinations.

LEMMA 8. *Given m variable length bit strings $\langle st_i, et_i, B[o_i]\rangle$ $(1 \leq i \leq m)$ and a K constraint, if $min_{i=1}^{m}\{et_i\} - max_{i=1}^{m}\{st_i\} < K$, we can prune the combination $\{o_i | 1 \leq i \leq m\}$.*

PROOF. The proof is straightforward due to the K constraint, and hence, it is omitted. □

Based on Lemmas 7 and 8, we propose Variable Length Bit Compression based Algorithm (VBA). The pseudo-code is shown in Algorithm 5. VBA takes as inputs a partition $P_t(o) = \{o_i | 1 \leq i \leq |P_t(o)|\}$, a global hashmap H, a global candidate list C, and (M, K, L, G) constraints. Initially, it initializes an empty local candidate list C_l (line 1). Then, it updates (lines 2–12) or creates new variable length bit strings (lines 13–14) for trajectories in $P_t(o)$. It first updates the bit strings already in H. More specifically, for each object o_i in the global H, if $o_i \in P_t(o)$, VBA appends a bit '1' to the bit string $H[o_i].B$, and removes o_i from $P_t(o)$ (lines 3–5). Otherwise, it appends a bit '0' to the bit string $H[o_i].B$ (line 7). An isVaild function is called to determine whether the bit string is a maximal pattern time sequence according to Lemma 7

Algorithm 5: Variable Length Bit Compression based Algorithm (VBA)

```
Input: Partition $P_t(o) = \{o_i | 1 \leq i \leq |P_t(o)|\}$, a global
       hashmap $H$, a global candidate list $C$, $(M, K, L, G)$
       required for co-movement pattern
1  a local candidate list $C_l \leftarrow \emptyset$
2  for each object $o_i \in H$ do
3      if $o_i \in P_t(o)$ then
4          append '1' to the end of $H[o_i].B$
5          remove $o_i$ from $P_t(o)$
6      else
7          append '0' to the end of $H[o_i].B$
8          tag $\leftarrow$ isVaild($H[o_i].B$)
9          if $tag = 1$ then
10             $C_l$.insert($o_i$)
11         else if $tag = -1$ then
12             $H$.delete($o_i$)
13 for each object $o_i \in P_t(o)$ do
14     $H$.insert($o_i$, $\langle t$, '1'$\rangle$)
15 for each object $o_i \in C_l$ do
16     $L \leftarrow \emptyset$
17     for each object $o_j \in C$ $(o_i \neq o_j)$ do
18         if $min\{et_i, et_j\} - max\{st_i, st_j\} \geq K$ then
19             $L$.insert($o_j$)
20     find and output valid patterns in $L \cup \{o_i\}$ as in lines 10–18
       of Algorithm 4 by applying Lemma 8
21 $C \leftarrow C \cup C_l$
```

(line 8). If $tag = 1$ (i.e., $H[o_i].B$ is a maximal pattern time sequence), VBA inserts o_i into C_l (lines 9–10). If $tag = -1$ (i.e., $H[o_i].B$ is an invalid pattern), it deletes o_i from H (lines 11–12). Subsequently, VBA processes trajectories that do not exist in the global H. For each trajectory o_i left in $P_t(o)$, VBA inserts a new entry $(o_i, \langle t, \text{'1'}\rangle)$ into H, where t is the start time and '1' is the current bit string of o_i (lines 13–14). Thereafter, for each trajectory o_i in local candidate list C_l, VBA first filters the global candidate list C using Lemma 8 to get a candidate list L (lines 16–19). Then, it finds the valid patterns in $L \cup \{o_i\}$ as in lines 9–17 of Algorithm 4 by applying Lemma 8. Finally, the global candidate list C is updated to $C \cup C_l$ (line 21).

Time Complexity and Storage Cost. According to Theorem 1, variable length bit compression reduces the storage cost from $O(n\eta)$ to $O(n\frac{G+L}{L})$. Here, n denotes the total number of occurrences for trajectories in one subtask, and $\frac{G+L}{L}$ is much smaller than η. Next, since the pattern enumeration methods of VBA and FBA are similar, the two have similar time complexity. The only difference is that, due to the use of maximal pattern time sequences, the candidate size of VBA is much smaller. However, the use of maximum pattern time sequences comes with the cost that, the response time of answering real-time co-movement pattern detection increases. Thus, VBA trades latency for throughput.

7. EXPERIMENTAL EVALUATION

In this section, we evaluate the efficiency and scalability of our proposed framework and methods, and also include comparisons with alternative methods. All experiments were conducted on a cluster consisting of 11 nodes, where one node serves as the master node, and the remaining nodes serve as slave nodes. Each node is equipped with two 12-core processors (Intel Xeon E5-2620 v3 2.40GHz), 64GB RAM, and a Gigabit Ethernet. Each cluster node runs Ubuntu 14.04.3 LTS and Flink 1.3.2. All system algorithms were implemented in Java.

Table 2: Datasets Used in our Experiments

Attributes	GeoLife	Taxi	Brinkhoff
# trajectories	18,670	20,151	10,000
# locations	24,876,978	189,419,934	23,906,131
# snapshots	92,645	502,559	97,241
Storage Size	1.5G	14G	1.7G

Table 3: Parameter Ranges and Default Values

Parameter	Range
grid cell width l_g	0.2%, 0.4%, **0.8%**, 1.6%, 3.2%, 6.4%
distance threshold ϵ	0.02%, **0.04%**, 0.06%, 0.08%, 0.10%, 0.12%
min objects M	5, 10, **15**, 20, 25
min duration K	120, 150, **180**, 210, 240
min local duration L	10, 20, **30**, 40, 50
max gap G	10, 20, **30**, 40, 50
ratio of objects O_r	10%, 20%, 40%, 60%, 80%, **100%**
machine number N	1, 2, 4, 6, 8, **10**

Datasets: We use two real-life datasets and one synthetic dataset to model streaming trajectories, as summarized in Table 2.

- GeoLife[7]: This dataset records the travel records of users during a period of more than three years. The GPS records are collected periodically, and 91% of the trajectories are sampled every 1 to 5 seconds.
- Taxi[8]: This is a real trajectory dataset generated by taxis in Hangzhou. Trajectories are segmented into trips, and each resulting trajectory represents the trace of a taxi during a month. The trajectories are sampled every 5 seconds.
- Brinkhoff[9]: This dataset is generated via the Brinkhoff generator [5]. The trajectories are generated on the real road network of Las Vegas. An object position is generated every second while an object moves through the road network with random but reasonable direction and speed.

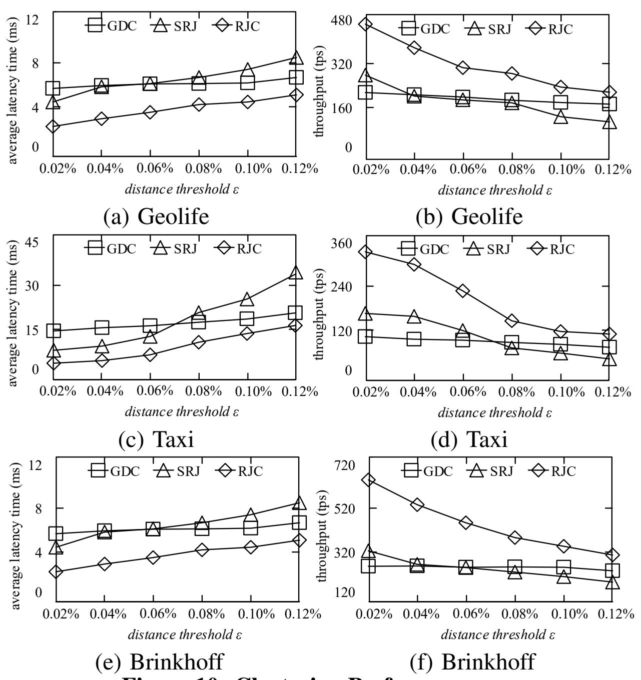
Comparison Methods: As this is the first study of real-time distributed co-movement pattern detection over streaming trajectories, no competitors are available. Instead, we adapt the state-of-the-art distributed co-movement pattern detection method [10] over historical trajectories, so that it can serve as a baseline method. In addition, we include comparisons between our clustering method RJC (discussed in Section 5) and two existing clustering methods.

- SRJ [36] is the state-of-the-art distributed range join method for streaming trajectories. We extend it to support DBSCAN clustering similar as our method RJC.
- GDC [14] is a grid-based DBSCAN clustering approach for a centralized environment. We extend it to work with Flink.

Parameters: In the experiments, we study the effect on performance of several factors, as summarized in Table 3, where the default values are shown in bold. In Table 3, l_g and ϵ are set to a percentage of the maximal distance of the whole dataset. In each set of experiments, we vary one parameter while fixing the others at their default values. Recall that, both ϵ and $minPts$ are used to control the density-based clustering, we vary only ϵ because similar performance is observed for different values of $minPts$. We fix $minPts$ at 10.

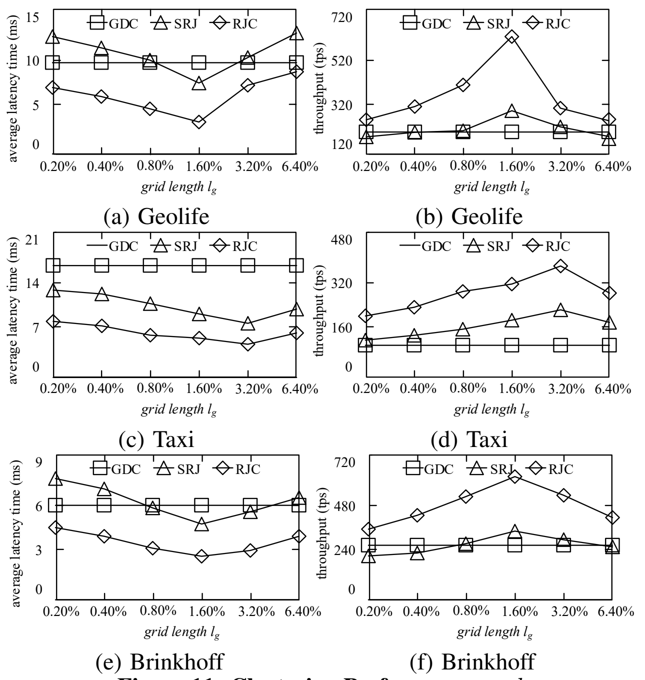
Performance Metrics: We study both latency and throughput. According to the definition of real-time co-movement pattern detection, we need to find all the current co-movement patterns in the snapshot set $\{S_1, S_2, \cdots, S_t\}$ at each timestamp t. Hence,

[7]https://research.microsoft.com/en-us/projects
[8]This is a proprietary dataset.
[9]https://iapg.jade-hs.de/personen/brinkhoff/generator/



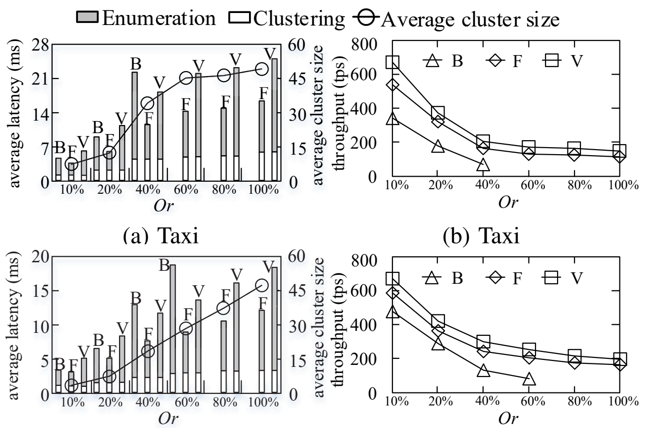
(a) Geolife (b) Geolife
(c) Taxi (d) Taxi
(e) Brinkhoff (f) Brinkhoff

Figure 10: Clustering Performance vs. ϵ



(a) Geolife (b) Geolife
(c) Taxi (d) Taxi
(e) Brinkhoff (f) Brinkhoff

Figure 11: Clustering Performance vs. l_g



(a) Taxi (b) Taxi
(c) Brinkhoff (d) Brinkhoff

Figure 12: Pattern Detection Performance vs. O_r

the average response time for each snapshot is reported as the latency, while the throughput is defined as the number of snapshots processed per second.

7.1 Clustering Performance

We first explore the effect of the distance threshold ϵ and the grid cell width l_g on the range join based clustering algorithm RJC, compared with the (adapted) existing methods SRJ and GDC.

Effect of ϵ. Fig. 10 depicts latency and throughput when increasing ϵ from 0.02% to 0.12%. As expected, RJC achieves better latency and throughput than SRJ, since Lemmas 1 and 2 make it possible to avoid unnecessary verifications. In addition, RJC performs better than GDC. This is because GDC uses ϵ (i.e., a small value) to divide the data space, resulting in too many partitions. Finally, the latency increases and the throughput decreases with the growth of ϵ due to the resulting larger search space.

Effect of l_g. Fig. 11 plots the latency and throughput when increasing l_g from 0.1% to 6.4%. As observed, the clustering performance (including latency and throughput) of RJC and SRJ first improves and then drops as l_g grows. The reason is that, if l_g is too small, the overhead of managing the many partitions is too high; and if l_g is too large, the punning ability decreases due to too many locations in each partition. However, the clustering performance of GDC stays stable as it does not depends on l_g.

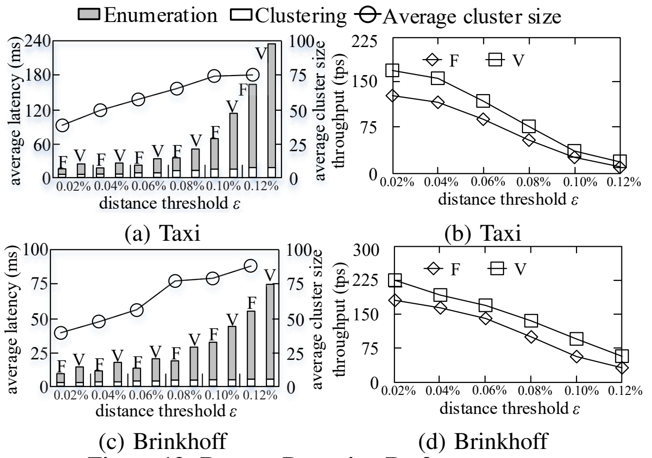
7.2 Scalability

Next, we investigate the scalability of our pattern detection framework, where RJC is used for clustering, and three algorithms BA, FBA, and VBA (covered in Section 6) are used for pattern enumeration. This yields three corresponding methods B, F and V for pattern detection. In this set of and remaining experiments, only Taxi and Brinkhoff are employed due to similar performance on Geolife and the space limitation.

Effect of O_r. Fig. 12 shows the latency and throughput when O_r ranges from 10% to 100%. Here, the bars indicate the average pattern detection latency (including clustering and enumeration), while the curve denotes the average cluster size. The first observation is that, B can only run on small datasets, i.e., when $O_r \leq 60\%$. This is because, the cost of its enumeration method BA is $O(2^n)$, where n is the average cluster size that increases with O_r. Second, V and F can run on large datasets, and F achieves the best latency, while V achieves the best throughput. The reason is that, the costs of their enumeration methods VBA and FBA are linear w.r.t. the average cluster size, and VBA utilizes the variable bit compression and maximal pattern time sequence techniques to trade latency for throughput. As expected, the performance drops as O_r grows, due to the larger search space.

Effect of ϵ. Fig. 13 illustrates the latency and throughput when ϵ ranges from 0.02% to 0.12%. As expected, the performance drops when ϵ grows. This is because, as ϵ ascends, the range join cost also increases due to the larger search space, and the enumeration cost grows due to the increasing average cluster size. Note that, the average cluster size is omitted in the remaining experiments, because it is not affected by other parameters.

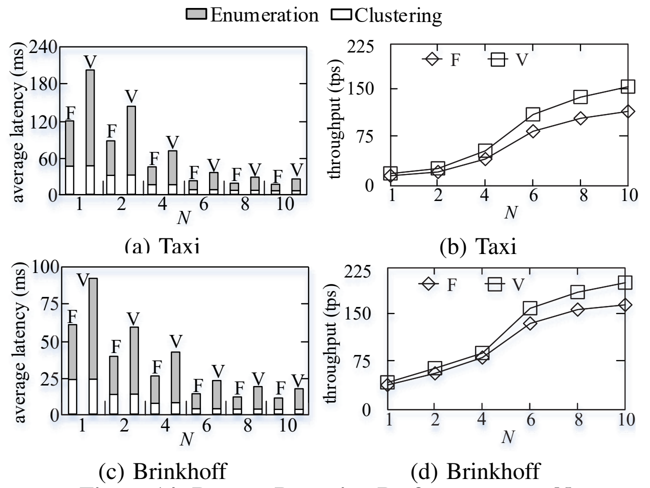
Effect of N. Fig. 14 plots the latency and throughput when N ranges from 1 to 10. As expected, the average latency drops and the throughput increases as the number of nodes grows.



(a) Taxi (b) Taxi

(c) Brinkhoff (d) Brinkhoff

Figure 13: Pattern Detection Performance vs. ϵ



(a) Taxi (b) Taxi

(c) Brinkhoff (d) Brinkhoff

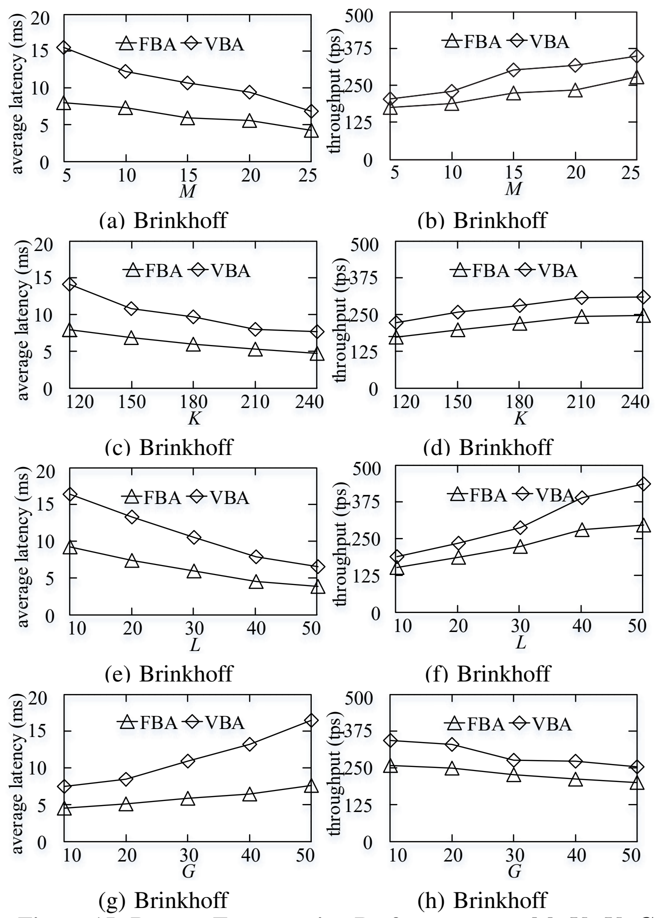
Figure 14: Pattern Detection Performance vs. N

7.3 Pattern Enumeration Performance

Next, we consider the effects of the constraints M, K, L, and G on the performance of pattern enumeration methods VBA and FBA. Here, BA is omitted as it cannot run on large datasets, and clustering is omitted as its performance is not affected by the constraints. Fig. 15 shows the latency and throughput when increasing M, K, L, and G on Brinkhoff. As expected, VBA has better throughput than FBA, while FBA has better latency. In addition, the average latency decreases (while the throughput increases) as M and K or L grow, as fewer valid candidates are generated or the pruning ability of Lemma 5 increases. However, the average latency increases (while the throughput decreases) as G grows. The reason is that, as G grows, more valid patterns are generated.

7.4 Summary

We can conclude that for clustering, our method RJC is more efficient on both latency and throughput when compared with the existing methods SRJ and GDC. When considering pattern enumeration, the scalability of our methods FBA and VBA are better than that of BA. In addition, FBA has the best latency, while VBA has the best throughput. The overall finding is that the ICPE framework and its corresponding methods are scalable, and are able to achieve low latency and high throughput. In addition, we recommend FBA if the throughput achieved is able to keep up with the incoming workload, and we recommend VBA if this is necessary to keep up with the incoming workload or if the higher latency is not critical.



(a) Brinkhoff (b) Brinkhoff

(c) Brinkhoff (d) Brinkhoff

(e) Brinkhoff (f) Brinkhoff

(g) Brinkhoff (h) Brinkhoff

Figure 15: Pattern Enumeration Performance vs. M, K, K, G

8. CONCLUSIONS

In this paper, we investigate real-time distributed co-movement pattern detection over streaming trajectories, which is useful in future movement prediction, trajectory compression, and a variety of location-based services. We develop a Flink-based framework, called ICPE. Flink is used because of its suitability for stream processing and its high efficiency and reliability. The framework encompasses two phases, i.e., clustering and pattern enumeration. To accelerate the clustering, we utilize a GR-index based range join, together with effective pruning techniques. To support efficient pattern enumeration, an id-based partitioning method, two bit compression techniques, and candidate based enumeration are utilized to reduce the storage and processing costs from exponential to linear. Extensive experiments using both real and synthetic datasets suggest that the proposed framework and its constituent data structures and algorithms are efficient and scalable. It is observed that, ICPE can achieve low latency and high throughput, and thus, it can support real-time co-movement pattern detection over streaming trajectories. In future research, it is of interest to extend ICPE to support the detection of additional types of advanced patterns.

9. ACKNOWLEDGMENTS

This work was supported in part by the National Key R&D Program of China under Grant No. 2018YFB1004003, the 973 Program under Grant No. 2015CB352502, the NSFC under Grant No. 61522208, the NSFC-Zhejiang Joint Fund under Grant No. U1609217, and the ZJU-Hikvision Joint Project. Yunjun Gao is the corresponding author of the work.

10. REFERENCES

[1] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. *PVLDB*, 29:81–92, 2003.

[2] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, 2008.

[3] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[4] C. Bohm, B. Braunmuller, F. Krebs, and H. P. Kriegel. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. *SIGMOD Record*, 30(2):379–388, 2001.

[5] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[6] B. Chen, Z. Lv, X. Yu, and Y. Liu. Sliding window top-*k* monitoring over distributed data streams. *DSE*, 2(4):289–300, 2017.

[7] Y. Chen and L. Tu. Density-based clustering for real-time stream data. In *SIGKDD*, pages 133–142, 2009.

[8] I. Cordova and T. S. Moh. Dbscan on resilient distributed datasets. In *HPCS*, pages 531–540, 2015.

[9] M. Ester, H. P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[10] Q. Fan, D. Zhang, H. Wu, and K. L. Tan. A general and parallel platform for mining co-movement patterns over large-scale trajectories. *PVLDB*, 10(4):313–324, 2016.

[11] B. Gedik, H. Andrade, K. L. Wu, P. S. Yu, and M. Doo. SPADE: the System S declarative stream processing engine. In *SIGMOD*, pages 1123–1134, 2008.

[12] J. Gu, J. Wang, and C. Zaniolo. Ranking support for matched patterns over complex event streams: The CEPR system. In *ICDE*, pages 1354–1357, 2016.

[13] J. Gudmundsson and M. van Kreveld. Computing longest duration flocks in trajectory data. In *GIS*, pages 35–42, 2006.

[14] C. H., N. Mamoulis, and D. W. Cheung. Discover of periodic patterns in spatiotemporal sequences. *TKDE*, 19(4):453–467, 2007.

[15] Y. He, H. Tan, W. Luo, S. Feng, and J. Fan. Mr-dbscan: A scalable mapreduce-based dbscan algorithm for heavily skewed data. *Frontiers of Computer Science*, 8(1):83–99, 2014.

[16] H. Jeung, Q. Liu, H. Shen, and Z. X. A hybrid prediction model for moving objects. In *ICDE*, pages 70–79, 2008.

[17] H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. *PVLDB*, 1(1):1068–1080, 2008.

[18] X. Li, V. Ceikute, C. S. Jensen, and K. L. Tan. Effective online group discovery in trajectory databases. *TKDE*, 12:2752–2766, 2013.

[19] Y. Li, J. Bailey, and L. Kulik. Efficient mining of platoon patterns in trajectory databases. *DKE*, 100:167–187, 2015.

[20] Z. Li, B. Ding, J. Han, and R. Kays. Swarm: Mining relaxed temporal moving object clusters. *PVLDB*, 3(1–2):723–734, 2010.

[21] Z. Li, B. Ding, J. Han, R. Kays, and P. Nye. Mining periodic behaviors for moving objects. In *KDD*, pages 1099–1108, 2010.

[22] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455, 2013.

[23] M. Nikos and D. Papadias. Slot index spatial join. *TKDE*, 15(1):211–231, 2003.

[24] S. Puntheeranurak, T. T. Shein, and M. Imamura. Efficient discovery of traveling companion from evolving trajectory data stream. In *COMPSAC*, pages 448–453, 2018.

[25] M. Riyadh, N. Mustapha, M. Sulaiman, and N. B. M. Sharef. CC-TRS: Continuous clustering of trajectory stream data based on micro cluster life. *Mathematical Problems in Engineering*, pages 1–10, 2017.

[26] T. L. C. Silva, K. Zeitouni, and J. A. de Macedo. Online clustering of trajectory data stream. In *MDM*, pages 112–121, 2016.

[27] L. A. Tang, Y. Zheng, J. Yuan, J. Han, A. Leung, C. C. Hung, and W. C. Peng. On discovery of traveling companions from streaming trajectories. In *ICDE*, pages 186–197, 2012.

[28] M. R. Vieira, P. Bakalov, and V. J. Tsotras. On-line discovery of flock patterns in spatio-temporal data. In *SIGSPATIAL*, pages 286–295, 2009.

[29] Y. Wang, E. Lim, and S. Y. Hwang. Efficient mining of group patterns from user movement data. *PVLDB*, 57(3):240–282, 2006.

[30] L. Wu, Y. Ge, E. Chen, R. Hong, J. Du, and M. Wang. Modeling the evolution of users preferences and social links in social networking services. *TKDE*, 29(6):1240–1253, 2017.

[31] D. Yang, J. Guo, Z. J. Wang, Y. Wang, J. Zhang, L. Hu, J. Yin, and J. Cao. Fastpm: An approach to pattern matching via distributed stream processing. *Information Sciences*, 453:263–280, 2018.

[32] D. Yang, Z. Guo, Z. Xie, E. A. Rundensteiner, and M. O. Ward. Interactive visual exploration of neighbor-based patterns in data streams. In *SIGMOD*, pages 1151–1154, 2010.

[33] H. Yoon and C. Shahabi. Accurate discovery of valid convoys from moving object trajectories. In *ICDM workshops*, pages 636–643, 2009.

[34] Y. Yu, Q. Wang, X. Wang, H. Wang, and J. He. Online clustering for trajectory data stream of moving objects. *Computer science and information systems*, 10(3):1293–1317, 2013.

[35] Z. Yu, X. Yu, Y. Liu, W. Li, and J. Pei. Mining frequent co-occurrence patterns across multiple data streams. In *EDBT*, pages 73–84, 2015.

[36] F. Zhang, Y. Zheng, D. Xu, Z. Du, Y. Wang, R. Liu, and X. Ye. Real-time spatial queries for moving objects using storm topology. *ISPRS International Journal of Geo-Information*, 5(10):178, 2016.

[37] Y. Zheng. Trajectory data mining: an overview. *TITS*, 6(3):29:1–29:41, 2015.

[38] Y. Zheng, L. Capra, O. Wolfson, and H. Yang. Urban computing: Concepts, methodologies, and applications. *ACM TIST*, 5(3):38:1–38:55, 2014.