Scalable Processing of Moving Flock Patterns

ANDRES CALDERON-ROMERO, University of California, USA
PETKO BALAKOV, Esri, USA
MARCOS VIEIRA, Google, USA
VASSILIS J. TSOTRAS, University of California, USA

This work presents a scalable approach for identifying moving flock patterns in large trajectory databases, addressing the inefficiencies in current techniques for handling large spatio-temporal datasets. A moving flock pattern refers to a group of entities that move closely together within a defined spatial radius for a minimum time interval. We focus on improving the state-of-the-art sequential algorithms, which is capable of detecting such patterns but suffers from high computational costs, particularly with large datasets. By leveraging distributed frameworks and utilizing spatial partitioning, the proposed solution aims to significantly reduce the time required for detecting flock patterns. We highlight the challenges of spatial and temporal joins in massive datasets and offer optimizations like partition-based parallelism and strategies for managing flock patterns that span multiple partitions. The paper presents an experimental evaluation using synthetic trajectory datasets, demonstrating that the proposed methods substantially improve scalability and performance compared to existing sequential algorithms.

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**; **MapReduce algorithms**; • **Information systems** → **Data structures**.

Additional Key Words and Phrases: Mobile patterns

ACM Reference Format:
Andres Calderon-Romero, Petko Balakov, Marcos Vieira, and Vassilis J. Tsotras. 2024. Scalable Processing of Moving Flock Patterns. 1, 1 (November 2024), 24 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Technological advances in the past few decades have triggered an explosion in the collection of spatio-temporal data. The increasing popularity of GPS devices and smartphones, along with the emergence of new disciplines such as the Internet of Things (IoT) and high-resolution Satellite/UAS imagery, has made it possible to collect vast amounts of data with spatial and temporal components.

In tandem, interest in extracting valuable information from such large databases has also grown. Spatio-temporal queries about popular places or frequent events remain useful, but there has been growing interest in more complex patterns. In particular, patterns that describe the group behavior of moving objects over significant periods. Moving cluster [18], convoys [17], flocks [11] and swarm patterns [22] reveal how entities move together over a minimum time interval.

Authors' addresses: Andres Calderon-Romero, acald013@ucr.edu, University of California, USA, Riverside; Petko Balakov, petko@esri.com, Esri, USA, Redlands; Marcos Vieira, marcos@gmail.com, Google, USA, San Francisco; Vassilis J. Tsotras, tsotras@cs.ucr.edu, University of California, USA, Riverside.

Applications for this type of information are both diverse and intriguing, particularly when dealing with trajectory datasets [14, 16]. Case studies span various domains, including transportation system management and urban planning [6], as well as ecology [20]. For example, [26] explores the identification of complex motion patterns to discover similarities in tropical cyclone paths. Similarly, [1] investigates eye movement trajectories to understand the strategies people use during visual searches. Additionally, [13] tracks the behavior of tiger sharks along the coasts of Hawaii to gain insight into their migration patterns.

One particular pattern of interest is the moving flock pattern, which captures how objects move within close proximity for a given time period. Closeness is defined by a disk of a specified radius within which the entities must remain. Since this disk can be positioned anywhere, detecting such patterns is a non-trivial problem. In fact, [11] highlights that finding flock patterns where the same entities stay together over time is an NP-hard problem. To address this, [27] proposed the BFE algorithm, the first approach capable of detecting flock patterns in polynomial time.

Despite the increasing availability of data, current state-of-the-art techniques for mining complex movement patterns still struggle with the performance demands of large-scale spatial data. This work introduces a scalable approach designed to detect moving flock patterns in very large trajectory databases. By leveraging emerging trends in distributed frameworks for spatial operations we aim to significantly improve the speed and efficiency of detecting these patterns.

2 RELATED WORK

The recent increased use of location-aware devices (such as GPS, smartphones, and RFID tags) has enabled the collection of vast amounts of data with spatial and temporal components. Several studies have focused on discovering and analyzing these types of datasets [21, 23]. In this area, trajectory datasets have emerged as an interesting field where diverse kind of patterns can be identified [28, 32]. For instance, researchers have proposed techniques to discover spatial motion patterns such as moving clusters [18], convoys [17] and flocks [3, 11]. Specifically, [27] introduced BFE (Basic Flock Evaluation), an innovative algorithm designed to efficiently identify moving flock patterns in polynomial time across large spatio-temporal datasets.

A flock pattern is defined as a group of entities that move together over a specified time period [3]. The applications of such patterns are broad and diverse. For instance, [5] identifies moving flock patterns in iceberg trajectories to analyze their movement behavior and their relationship with changes in ocean currents.

The BFE algorithm provides an initial approach for detecting flock patterns. It begins by identifying disks with a predefined diameter (ε) where moving entities are sufficiently close at specific time instants. This operation is computationally expensive due to the large number of points and time instances to be analyzed, with a complexity of $O(2n^2)$ per time. Although the algorithm leverages a grid-based index and a stencil to accelerate this process, the overall complexity remains high.

Both [5] and [26] adopt a frequent pattern mining approach to enhance performance when combining disks across time instants. Similarly, [24] utilize plane sweeping techniques, binary signatures, and inverted indexes to further accelerate this process. However, these methods retain the core strategy of BFE for detecting disks at each time instant.

In contrast, [2] and [10] employ depth-first algorithms to analyze the time intervals of individual trajectories and report maximal duration flocks. However, these methods are less effective for dense datasets or those that involve large numbers of entities per time step, as they struggle to scale efficiently in such conditions.

Given the high computational demands of flock pattern detection, it is not surprising that parallelism has been employed to improve performance. For example, [9] use extreme and intersection sets to report maximal, longest, and largest flocks on GPUs, albeit with limitations imposed by the GPU's memory model.

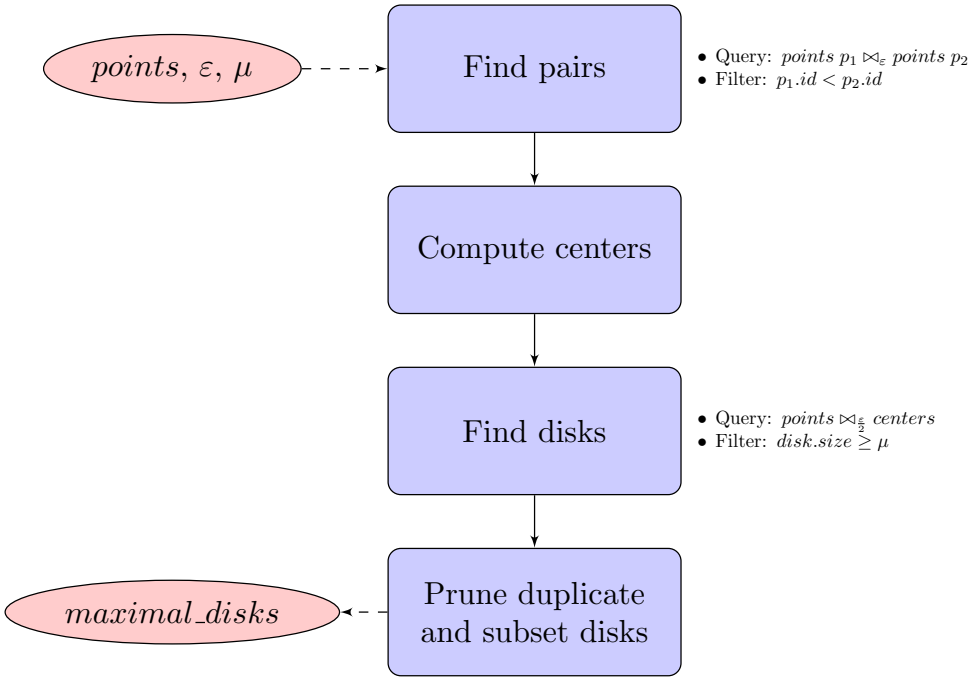


Fig. 1. General steps in phase 1 of the sequential algorithm.

Despite the increasing adoption of cluster computing frameworks, particularly those with spatial data capabilities [7, 15, 30, 31], significant advancements in this area remain limited. To the best of our knowledge, this work is the first to explore the detection of moving flock patterns in a scalable approach.

3 BACKGROUND

Before discussing the details of our contributions, we first provide an overview of the current state-of-the-art. This will help highlighting their challenges and limitations in handling large spatio-temporal datasets, as described in the next Section.

3.1 The BFE sequential algorithm

The alternative approach we will discuss closely follows the steps outlined in [27]. In that work, the authors introduced the Basic Flock Evaluation (BFE) algorithm, designed to identify flock patterns in trajectory databases. While the full details of the algorithm can be found in the source, we will provide a general overview of the key aspects. It is important to note that the BFE algorithm operates in two phases: first, it identifies maximal disks at the current time step; second, it extends and reports previous flocks by combining them with the newly discovered disks.

The input for the BFE algorithm consists of a set of points, a minimum distance ε (which defines the diameter of the disks where the moving entities must lie), a minimum number of entities μ per disk, and a minimum duration δ, representing the required time units that entities must remain together to be considered a flock. Based on this input, Figure 1 illustrates the workflow of the process in four general steps. The primary goal of this phase is to identify a set of disks at each time step, enabling the combination with future disks to form flocks.

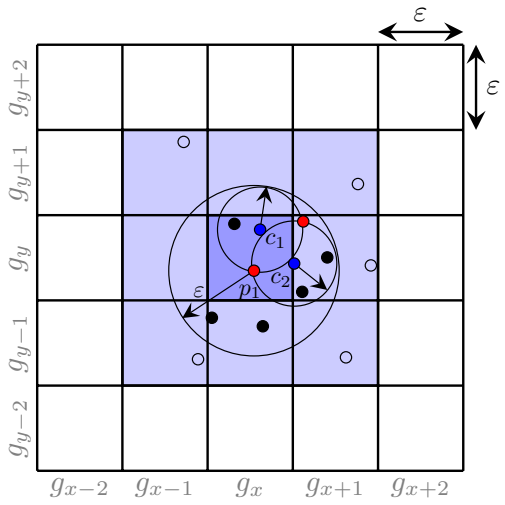


Fig. 2. The grid-based structure proposed in [27].

The main steps in phase 1 follow:

(1) Pair finding: The algorithm uses the parameter ε to identify pairs of points that are within a maximum distance of ε units from each other. This is achieved through a distance self-join operation on the set of points, using ε as the distance threshold. To avoid redundancy, duplicate pairs are eliminated; for example, the pair (p_1, p_2) is considered identical to the pair (p_2, p_1), so only one instance is retained. Point IDs are used to filter out these duplicates efficiently.
(2) Center computation: From the set of pairs obtained, each pair is used to compute the centers of two circles, each with a radius of $\frac{\varepsilon}{2}$, whose circumferences pass through the two points in the pair. The pseudo-code for this procedure is provided in Appendix 1.
(3) Disk finding: Once the centers have been identified, a query is executed to gather the points within a distance of ε units from each center. This is accomplished by performing a distance join between the set of points and the set of centers, using $\frac{\varepsilon}{2}$ as the distance parameter. As a result, each disk is defined by its center and the IDs of the surrounding points. At this stage, a filter is applied to discard any disks that contain fewer than μ entities.
(4) Disk pruning: It is possible for a disk to contain the same set, or a subset, of points as another disk. In such cases, the algorithm reports only that one disk which contains the other(s), referred to as the *maximal* disk. The procedure for identifying maximal disks is explained in Appendix 2.

It is important to note that BFE also employs a grid structure in this phase to optimize spatial operations. The algorithm divides the space into a grid, where each cell has a side length of ε (see Figure 2 from [27]). This structure allows BFE to limit its processing to each grid cell and its eight neighboring cells. There is no need to query cells beyond this neighborhood, as points in more distant cells are too far away to influence the results. Figure 3 shows an example of the Phase 1 steps using a sample dataset.

Figure 4 explains schematically phase 2. This phase performs a recursion using the set of disks found at time i and the set of partial flocks computed at the previous time instant $i-1$. As we do not know where and how far a group of

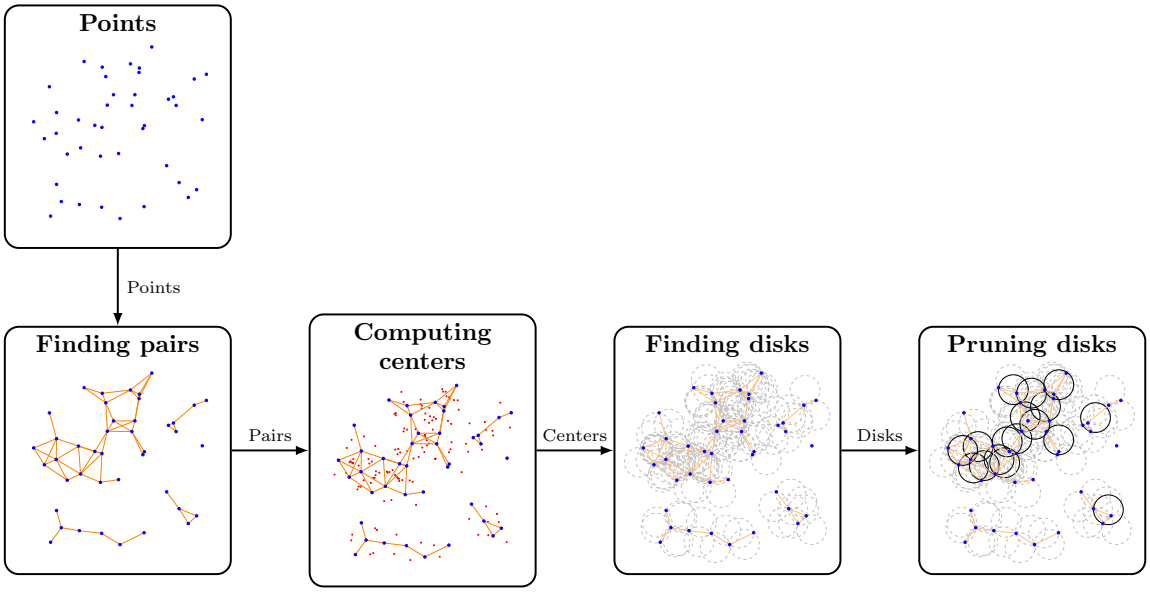


Fig. 3. BFE Phase 1 example execution on a sample dataset.

points can move in the next time instant, this step performs a (temporal) join between both sets (partial flocks computed at time $i-1$ and maximal disks found in time i). When a join is performed, we check that the number of common points remains greater than μ, in which case the partial flock extends in time. A flock is reported in the answer if its duration has reached the minimum duration δ; otherwise, it remains as partial flock and it will be further evaluated during the next iteration at the next time instant.

Similarly, Figure 5 illustrates the recursive process and how the set of partial flocks from previous time instants feeds into the next iteration. The example assumes a δ value of 3, meaning flocks start being reported from time instant t_2. Note that time instants t_0 and t_1 are considered the initial conditions. At the start of the algorithm, maximal disks are identified at t_0, which are immediately transformed into partial flocks with a duration of 1 and then passed on to the next time instant. At t_1, a new set of maximal disks $\mathcal{D}_1$ is found and joined with the partial flocks from t_0, denoted as $\mathcal{F}_0$. The information for each partial flock is updated accordingly, including its duration and the points it contains. From this point onward, subsequent time instants follow the exact steps outlined in Figure 4.

3.2 The PSI sequential algorithm

The PSI algorithm, proposed by [24], follows a similar process to the BFE algorithm. However, instead of using a grid structure to index points within the area, PSI employs a sweep-line approach that processes points in order of their x-coordinates. For each visited point p, the algorithm considers a square of side length 2ε centered at p. It only examines the points to the right of p that lie within two half-squares of side length ε, as illustrated by the shaded regions in Figure 6.

While BFE processes points inside a grid cell of side length ε along with its eight neighboring cells, PSI focuses on the points in these two half-squares. As a result, PSI more efficiently identifies the points relevant for detecting candidate pairs, centers, and disks. This indexing method has been shown to outperform BFE in most cases, with BFE offering similar or better performance only when ε values are relatively small. In such cases, the number of points to

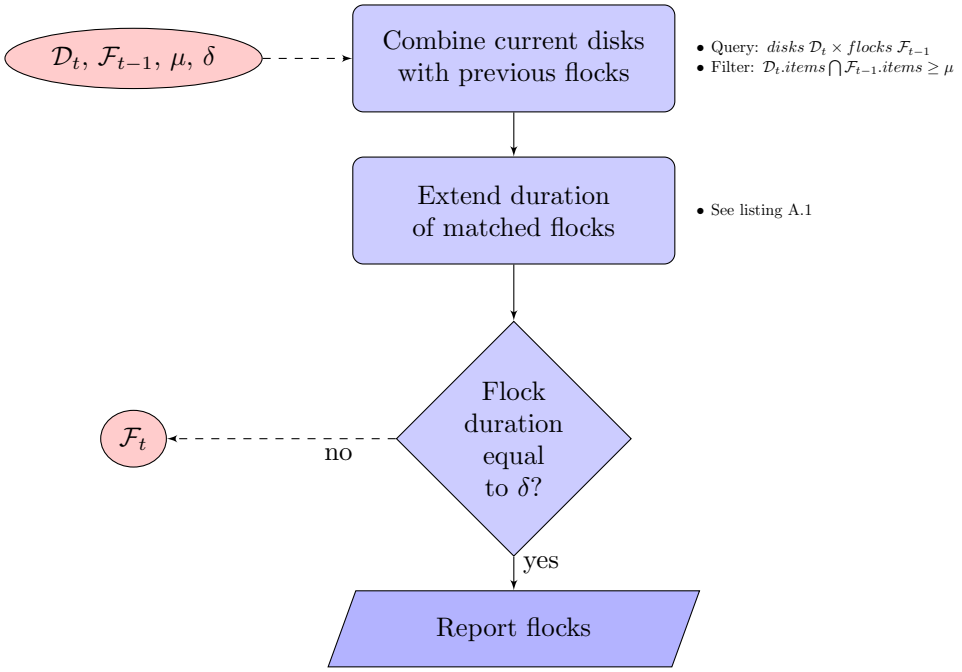


Fig. 4. Steps in BFE phase two. Combination, extension and reporting of flocks.

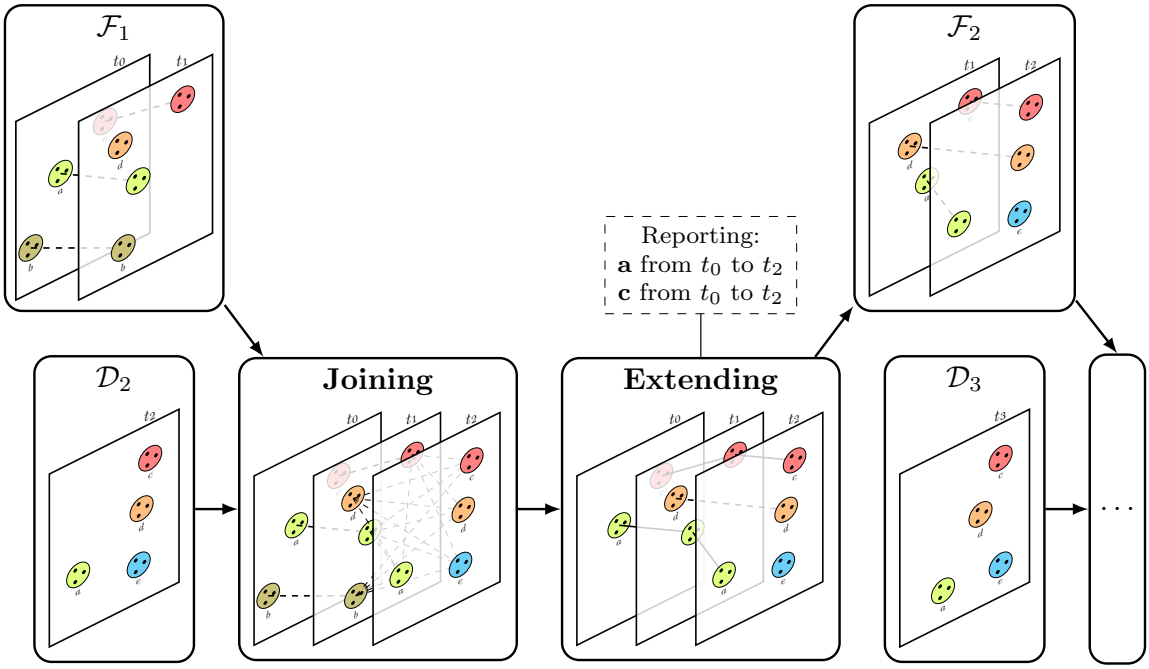


Fig. 5. BFE Phase 2 example explaining the stages along time instants and the initial conditions.

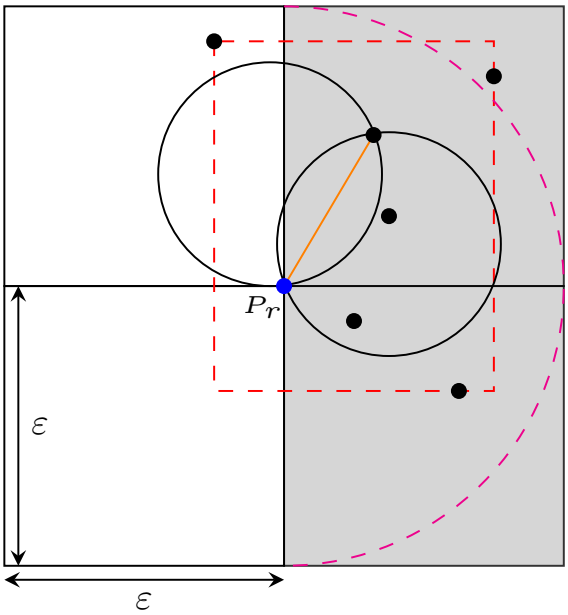


Fig. 6. An example of the two half squares used in PSI algorithm.

consider is smaller, and PSI still requires sorting the points for the sweep-line approach. Therefore, both approaches are considered in the following sections.

4 BOTTLENECKS IN THE SEQUENTIAL APPROACH AND PROPOSED SOLUTIONS

Since both sequential approaches follow the same steps (as shown in Figures 1 and 4), we will focus on discussing their bottlenecks using the BFE as an example. Certain stages in the BFE process are notably impacted when handling very large datasets, which can significantly affect performance.

4.1 Phase 1: Spatial finding of maximal disks.

First, we focus on Phase 1. As illustrated in Figure 3, this phase's steps are demonstrated using a sample dataset. It is important to note that the final set of maximal disks is significantly smaller than the initial number of candidate disks found. Specifically, the number of candidate centers to evaluate is $2|\tau|^2$, where τ represents the number of trajectories [27]. Our experiments reveal that this issue becomes more pronounced not only in very large datasets but also in those containing areas with a high density of moving entities.

To address this issue, we propose a partition-based strategy that divides the study area into smaller subareas, allowing for independent and parallel evaluation. The strategy consists of three key steps: first, the *partition and replication* stage, followed by the *local flock discovery* within each partition, and finally, the *filtering stage*, where we consolidate and unify the results. Each of these steps is detailed below.

- Partition and Replication: Figure 7 provides a brief example of the partition and replication stage. Different types of spatial indexes, such as grids, R-trees, or quadtrees, can be used to create spatial partitions of the input dataset. In the example shown in Figure 7.b, we use a quadtree, which generates seven partitions. To ensure each

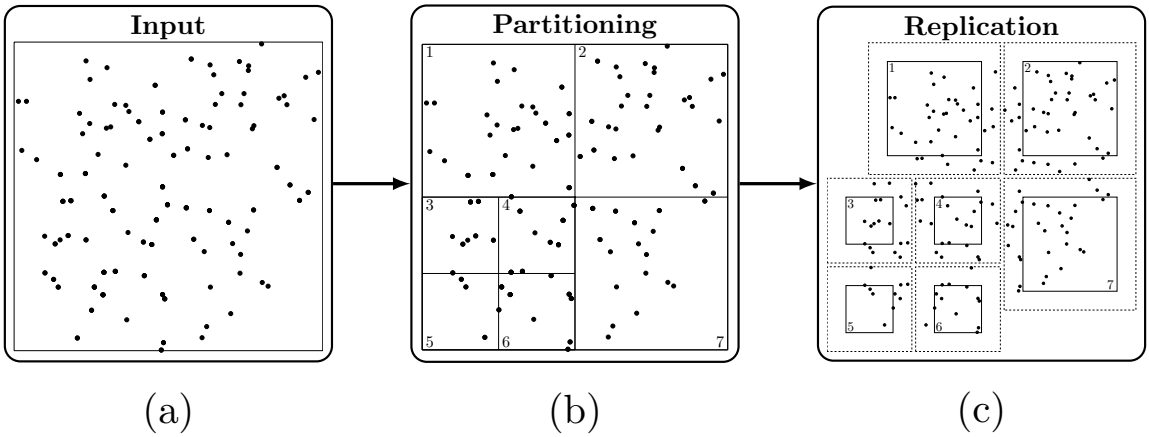


Fig. 7. An example of partitioning and replication on a sample dataset.

partition can locally identify flocks, it must have access to all relevant data. This is achieved by replicating points that are within a distance of ε from the border of each partition, an area referred to as the *expansion zone*, into adjacent partitions. Figure 7.c illustrates each partition, surrounded by a dotted line representing the expansion zone, which includes the points that need to be replicated from neighboring partitions.

- Local flock discovery: At this stage, each partition can be processed independently and in parallel, with partitions assigned to different processing nodes. Within each partition, we can execute the steps of Phase 1 of the BFE algorithm locally, as outlined in Figure 1.
- Filtering: While partitioning and replication facilitate parallelism, they can also lead to result duplication, as different nodes may report the same maximal disk. Specifically, if a disk's center lies within a partition, it will be reported only once by the node processing that partition. However, disks with centers located in an expansion zone will be reported by all partitions that share that zone. To address this, we propose a reporting approach that effectively prevents such duplication, which we detail below.

Disks with centers in an expansion zone are created by points that exist in both partitions due to replication. We assert that each partition should only report disks generated within its own area and not those originating in its expansion zone. Figure 8 illustrates the possible scenarios. Assume partitions 1 and 2 in the figure are contiguous, sharing edge AB. Consider the disks a' and a'' (each with a diameter of ε), which are generated by two points (shown in green) located in the expansion zone of partition 1 but inside partition 2. In this case, both a' and a'' will be reported by partition 2. Similarly, both c' and c'' will be reported by partition 1. However, b' will be reported by partition 1, while b'' will be reported by partition 2.

4.2 Phase 2: Temporal join

At the end of Phase 1, we have computed a set of maximal disks for a given time instant. In Phase 2, we proceed by combining these disks over time instants to form flocks. However, since Phase 1 involved partitioning the spatial domain for parallelism, Phase 2 becomes more complex as flocks can move across spatial partitions over time. Once the maximal disks are identified for a time instant i, a temporal join occurs within each partition to link these disks with

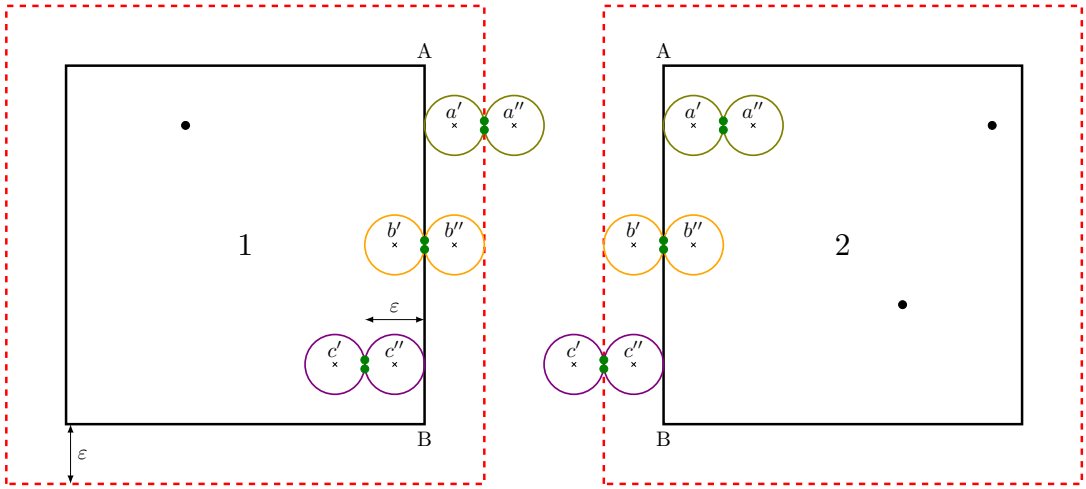


Fig. 8. Ensuring no loss of data in safe zone and expansion area.

partial flocks from the previous time instant $(i-1)$. However, we must account for partial flocks that may appear near the partition borders and potentially move into adjacent partitions (see Figure 9).

To address this issue, we introduce an additional parameter, *maxdist*, which represents the maximum distance a moving object can travel between consecutive time instants. We define the *safe area* of a partition as the internal region that is at least *maxdist* away from the partition's border (illustrated in grey in Figure 10). Any partial or full flocks discovered within a partition's safe area can be directly reported as results. However, flocks that start or end outside the safe area must be collected for post-processing to determine if they correspond with partial flocks from neighboring partitions. These cases, where flocks cross between partitions, are referred to as *crossing partial flocks* (CPFs).

In the post-processing stage, we evaluate four alternatives for collecting and checking crossing partial flocks (CPFs). The simplest approach is to gather all CPFs and process them sequentially on a single node (the master node). However, due to the large number of partitions and the *maxdist* parameter, the volume of CPFs requiring post-processing can become substantial, leading to a bottleneck that negatively impacts overall performance.

We also evaluate an intermediate approach where the CPFs from a given partition are sent to a middle-level node for processing, based on the quadtree structure used to create the partitions. The choice of which middle-level node to send the CPFs to is determined by a user-defined parameter called *step*. A value of $step = 1$ corresponds to sending CPFs to the immediate parent, $step = 2$ to the grandparent, and so on, until the root is reached. For example, with *step=1*, all CPFs from a partition are first sent to its parent node in the quadtree. The parent node processes its CPFs, but some flocks may still cross outside the parent's safe area. These leftover CPFs are then passed to the next parent (since $step = 1$), and this process continues until all CPFs are processed, potentially reaching the root node. This approach allows for parallelism in post-processing, as moving to a parent node increases the partition's area and improves the likelihood that CPFs can be resolved at that level. In the experimental section, we test different values of the *step* parameter, such as $step = 2$, where CPFs are sent to the grandparent at each stage.

Unlike the previous two approaches, which assign all CPFs from a given partition in the same way (partition-based), the third alternative assigns each CPF individually (CPF-based). For a given CPF f, we extend its most recent disk by a ring with a size of *maxdist*, identifying all overlapping partitions for this extended disk —essentially determining

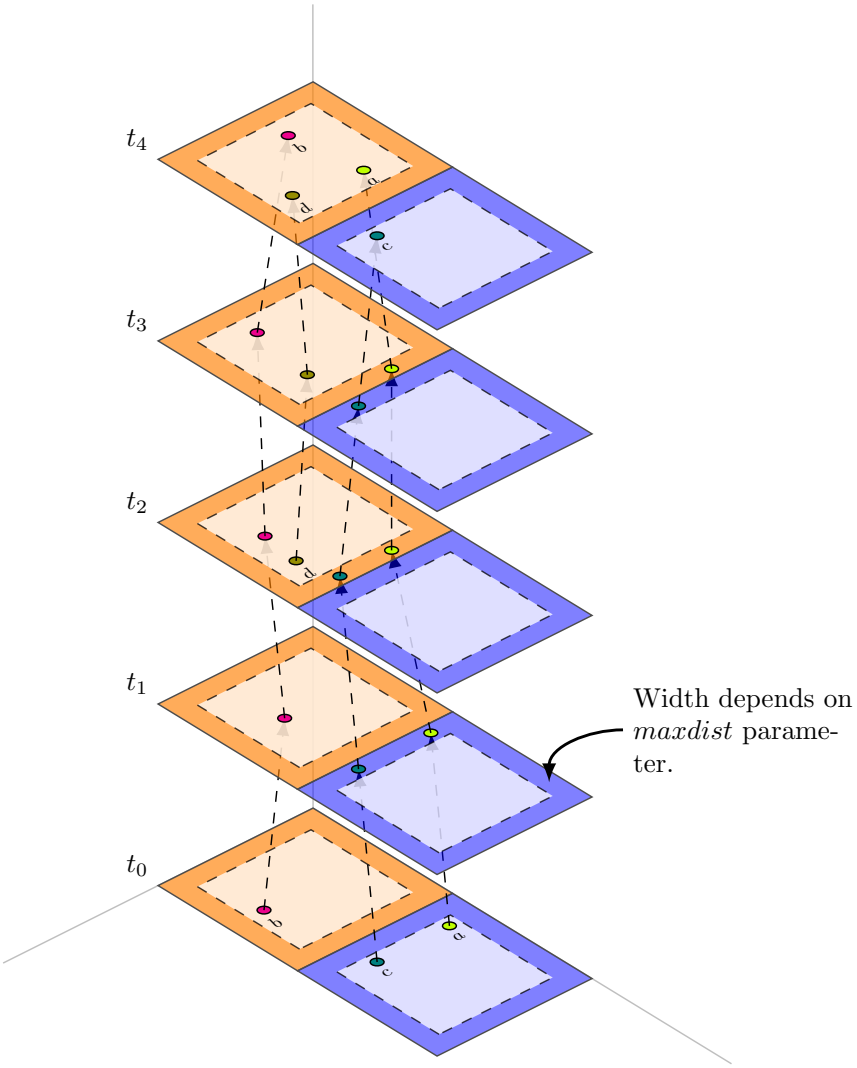


Fig. 9. A flock that moves in different partitions along the time.

which neighboring partitions the objects in f could move to in the next time instant. For each overlapping partition, we retrieve the Least Common Ancestor (LCA) between that partition and f's original partition. CPF f is then sent to the node(s) corresponding to these LCAs. The benefit of this approach is that the LCA can efficiently complete the processing for f, as it exploits proximity using *mindist*. However, the downside is the increased copying overhead, as f may need to be sent to multiple nodes.

A limitation of the previous alternatives is that each spatial partition is processed by a single node, which incrementally evaluates all time instances for that partition. The fourth alternative introduces fixed divisions in the temporal domain, based on a user-defined parameter (number of divisions), as illustrated in Figure 11. In this approach, the spatio-temporal

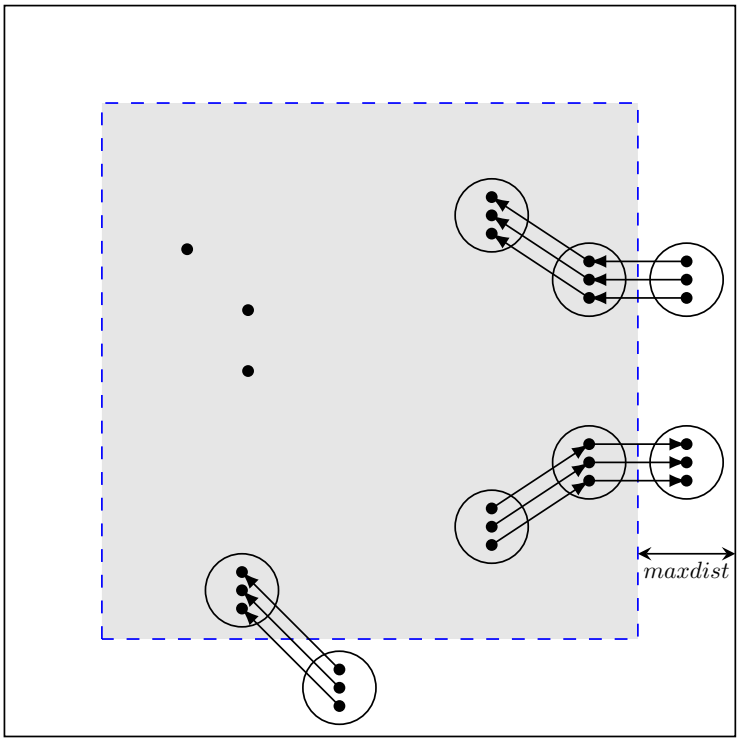


Fig. 10. Examples of CPFs that that start or end in the border area of a partition.

space is divided into temporal 'cubes,' each of which can be processed by different nodes. For simplicity, we assume that each division spans the same length of time. However, an additional validation step is required to ensure continuity of flocks across temporal divisions.

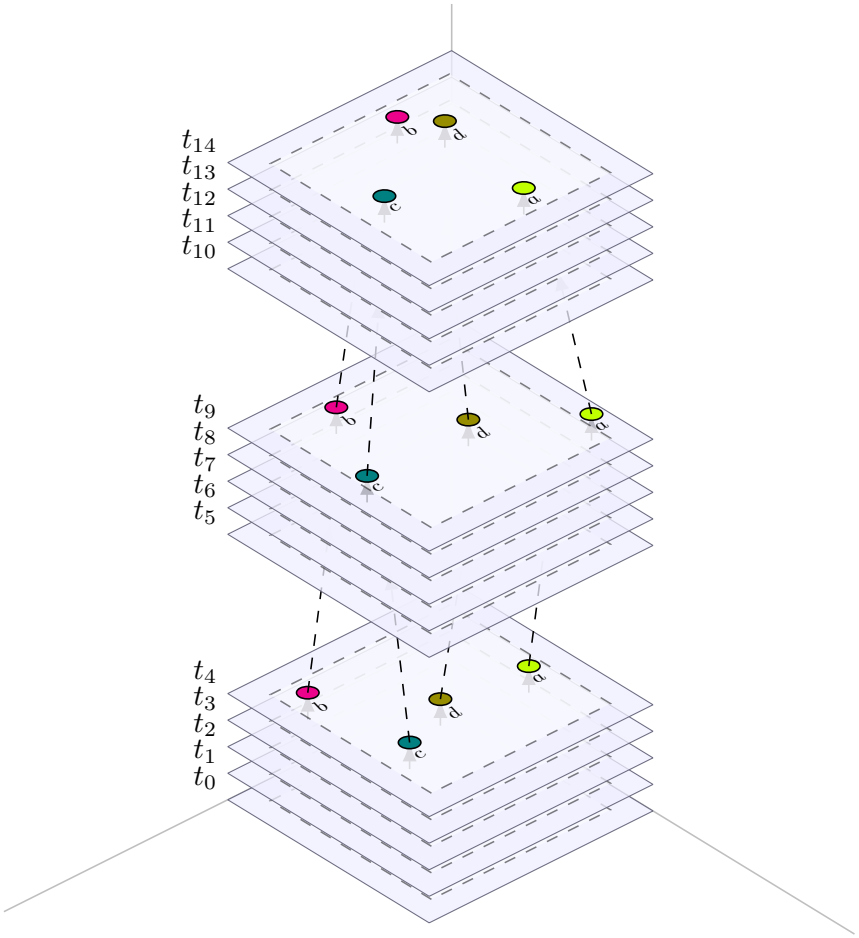
5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

For our experiments, we utilized a 12-node cluster, each running Linux (kernel version 3.10) and Apache Spark 2.4. Each node was equipped with 8 cores, providing a total of 96 cores across the cluster. Each core operated with an Intel Xeon CPU at 1.70 GHz, and each node had 4 GB of main memory.

To evaluate the different approaches, we generated three synthetic datasets with varying characteristics, as detailed in Table 1. These datasets were created using the SUMO simulator [19], by importing traffic networks of Berlin and Los Angeles from OpenStreetMap [12]. We configured SUMO for pedestrian traffic and generated datasets of 10K, 25K, and 50K pedestrian trajectories. The total duration of the trajectories was set to 10, 30, and 60 minutes, respectively, with positions of pedestrians recorded at one-minute intervals.

For the partitioning phase, we employed a quadtree structure, though other indexing methods could also be used. The advantage of using a quadtree is its ability to create nodes that tend to have a similar number of objects. The input to this phase is a set of points in the format *(traj-id, x, y, t)*. To construct the quadtree, we begin by sampling 1% of the input data and inserting this subset into an initially empty quadtree.



*a,b,c and d are flocks moving along time.

Fig. 11. An alternative division on the time dimension to partition the data into cubes.

Table 1. Description of datasets.

Dataset	Number of Trajectories	Total number of points	Maximum Duration (min)
Berlin10K	10000	97526	10
LA25K	25000	1495637	30
LA50K	50000	2993517	60

A key parameter for the quadtree is the node capacity, denoted as c. When the number of points in a node exceeds this capacity, the node splits. After all the sampled points are inserted, we use the Minimum Bounding Rectangles (MBRs) of the leaf nodes as the partitions for our approach. The remaining points are then inserted into these fixed partitions, with no further splits occurring. Each partition is assigned to a different cluster node, where a sequential version of either BFE or PSI is executed locally on the points within that partition.

5.2 Optimizing the number of partitions for Phase 1.

The capacity parameter c directly influences the number of partitions in the quadtree. A smaller value of c results in a higher number of partitions, which leads to many smaller tasks that can be distributed across the cluster. However, this can increase the overhead associated with data transmission and, potentially, replication, which may become a bottleneck. Conversely, a larger value of c reduces the number of partitions, resulting in fewer but larger tasks. This increases the workload of the sequential algorithm within each partition, potentially extending the response time for individual jobs.

Figure 12 presents the execution time (in seconds) for computing maximal disks (Phase 1) at a specific time instant, using different values of c and ε. The experiments were conducted using the LA25K dataset. For the case where $\varepsilon = 20m$, we observe that there is an optimal value of c that minimizes the execution time for finding maximal disks, which occurs at $c = 100$ (corresponding to approximately 1300 partitions). Additionally, the optimal value of c varies based on the value of ε. For instance, with a smaller $\varepsilon = 2m$, the execution time is minimized at a larger capacity $c = 500$ (around 250 partitions). When ε is large, more pairs of points need to be processed, resulting in a higher number of maximal disks to compute. In such cases, using a smaller value of c creates more partitions within the same spatial area, thereby distributing the workload more evenly across partitions and reducing the amount of work per partition.

After determining the optimal value of c for a given ε, we further analyzed the behavior of BFE and PSI on the most 'demanding' partitions, those that required the longest time to complete Phase 1. Since the partitions are processed in parallel across different cores, these demanding partitions have the greatest impact on the overall performance. By focusing on these partitions, we can better understand potential bottlenecks and further optimize the system's efficiency.

5.3 Analyzing most costly partitions.

We began by identifying the top 10 partitions that required the most time to execute the BFE algorithm with $\varepsilon = 20$ meters. For these specific partitions, we ran both BFE and PSI while varying ε from 10 to 20 meters. The Phase 1 execution times are shown in Figure 13, where it is evident that PSI consistently outperformed BFE across all values of ε.

We further investigated the reasons behind some partitions taking longer to compute. Figure 14 shows the Phase 1 execution times per partition while varying ε from 10m to 20m, with partitions ordered by the number of pairs they contain. One key observation is that as ε increases, the number of pairs also increases, since a larger ε allows for more maximal disks. For instance, with $\varepsilon = 10m$, the maximum number of pairs in a partition is around 1800, whereas for $\varepsilon = 20m$, some partitions contain nearly 4000 pairs.

Another notable observation is that BFE is more sensitive to the density of pairs within a partition than PSI, a difference that becomes more pronounced at higher values of ε (e.g., 18m or 20m). As mentioned earlier, the flexible bounding boxes used by PSI (illustrated in Figure 6) more effectively isolate the relevant points for computing pairs, whereas BFE relies on a fixed grid cell, which makes it less efficient in denser partitions.

A final observation is that a few partitions take significantly more time than others, particularly those with a higher density of pairs. This is directly related to the number of maximal disks that need to be computed and subsequently pruned. For example, the partition that takes the longest time when $\varepsilon = 20m$ is the one with the highest number of pairs, which corresponds to partition 187 in Figure 13.

We further analyzed how Phase 1 processing is distributed within the most demanding partition. Figure 15.a (for BFE) and Figure 15.b (for PSI) display the time taken by each Phase 1 stage (refer to Figure 3) for partition 187. The most

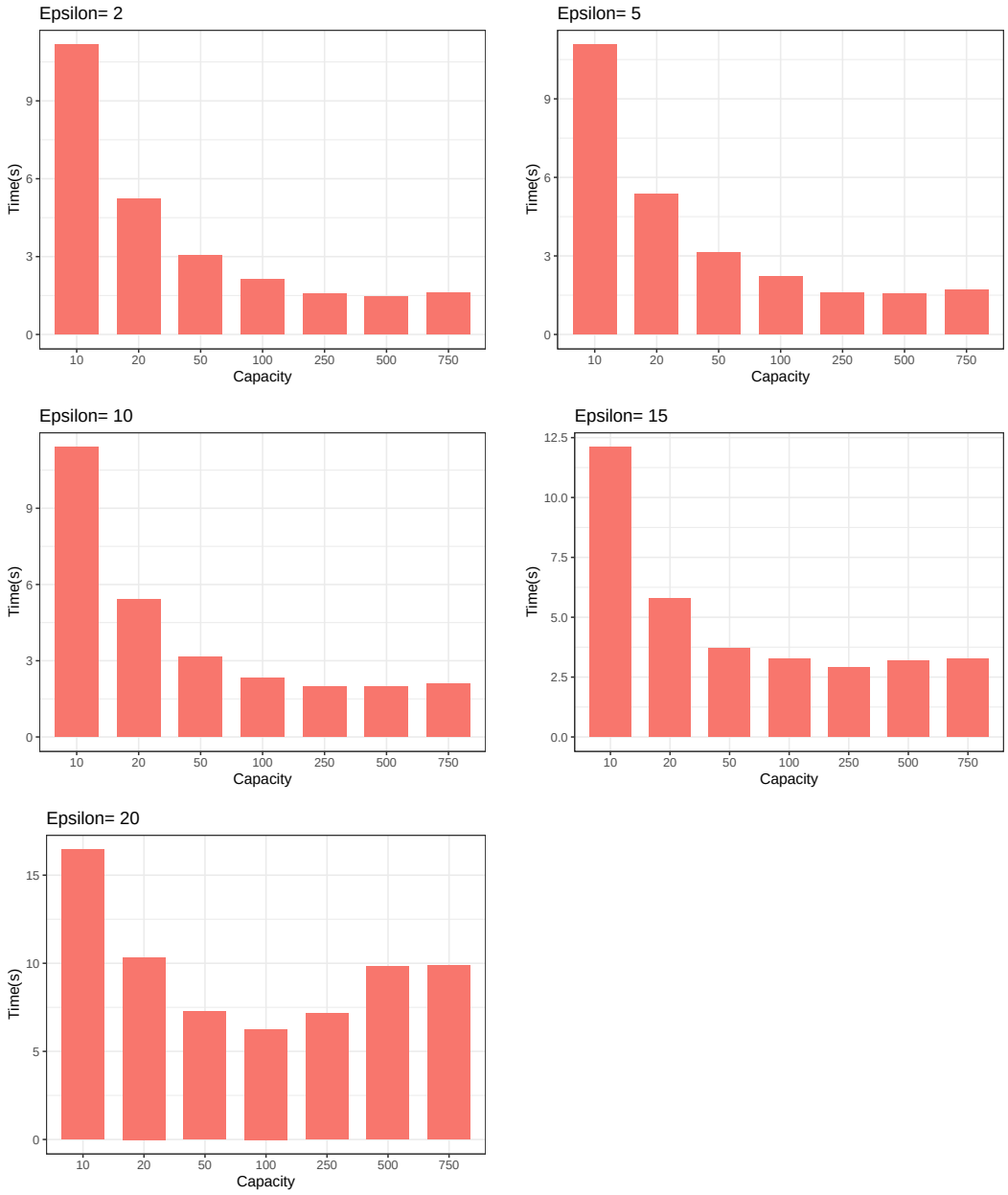


Fig. 12. Execution time testing different values for Capacity (c) and Epsilon (ε).

resource-intensive stage in both cases is the final step of filtering the disks, where disks whose points are contained within others are removed—this stage identifies the *maximal* disks (labeled as 'Maximals' in the figure).

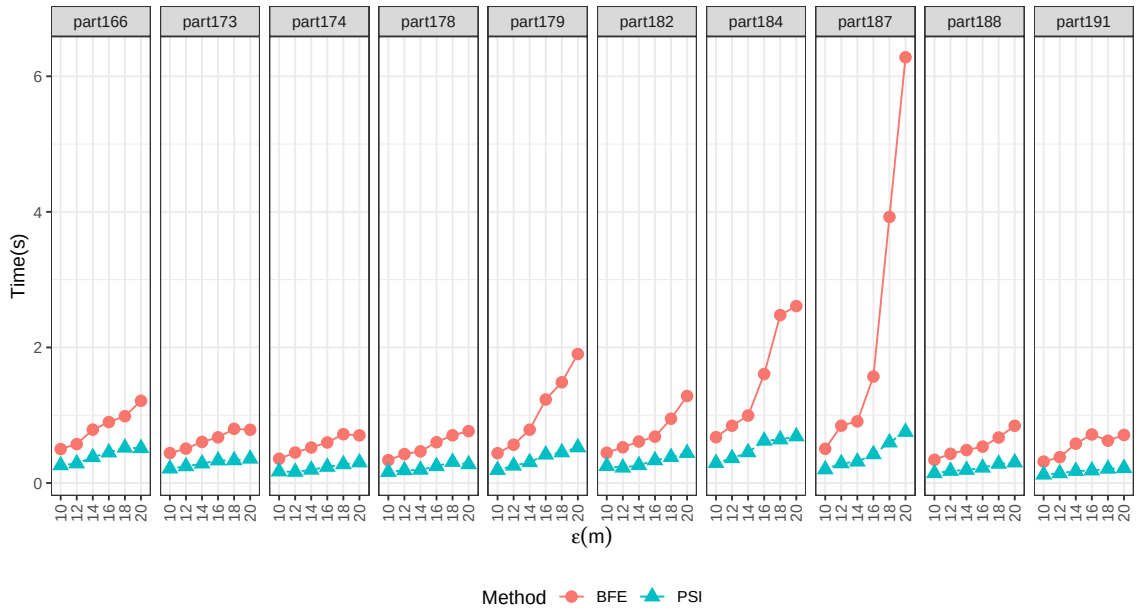


Fig. 13. Comparing the performance of PSI and BFE for time consuming partitions.

This stage is particularly costly because both BFE and PSI must scan a large set of candidate disks, identifying and removing those that are redundant. As ε increases, this processing becomes even more time-consuming, as the number of pairs and candidate disks grows along with ε.

5.4 Can we reduce pruning time?

Dense areas pose challenges for pruning, as they are highly sensitive to increases in the value of ε, leading to an exponential growth in the number of pairs. To address this, we explored alternative strategies that could enable more effective grouping of points. It is important to note that density-based spatial clustering methods, such as DBSCAN [8], are not suitable for this problem. In very dense regions, these approaches often produce a single large cluster, which does not resolve the issue. Additionally, clustering algorithms do not enforce the strict relationships required for a flock, where all points must be within a distance of ε from each other.

Instead, we explored graph-oriented clustering, focusing on the concept of *maximal cliques*. In an undirected graph, a maximal clique is a subset of vertices where each vertex is directly connected to every other vertex in the subset. Additionally, the clique is maximal in the sense that it cannot be extended by adding more vertices [4, 25].

In this context, the points within a partition can be treated as the vertices of an undirected graph, where edges are created between pairs of points that are within a distance of ε. By finding the set of maximal cliques in this graph, we identify subsets of points where each point is connected to all others in the subset. This means that all points in the clique are at most ε apart, and no additional points can be added to the subset. However, not every maximal clique qualifies as a maximal disk. A maximal clique becomes a maximal disk only if it contains at least μ points and can be enclosed by a disk with a radius of $\frac{\varepsilon}{2}$.

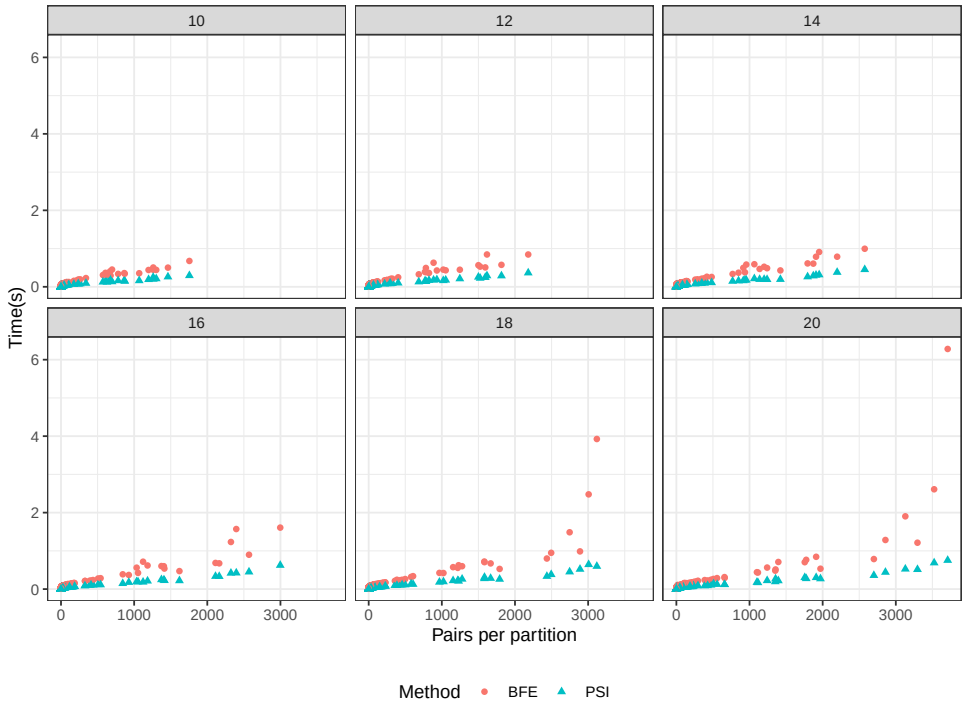


Fig. 14. Execution time for pairs/disks finding in the dense partition.

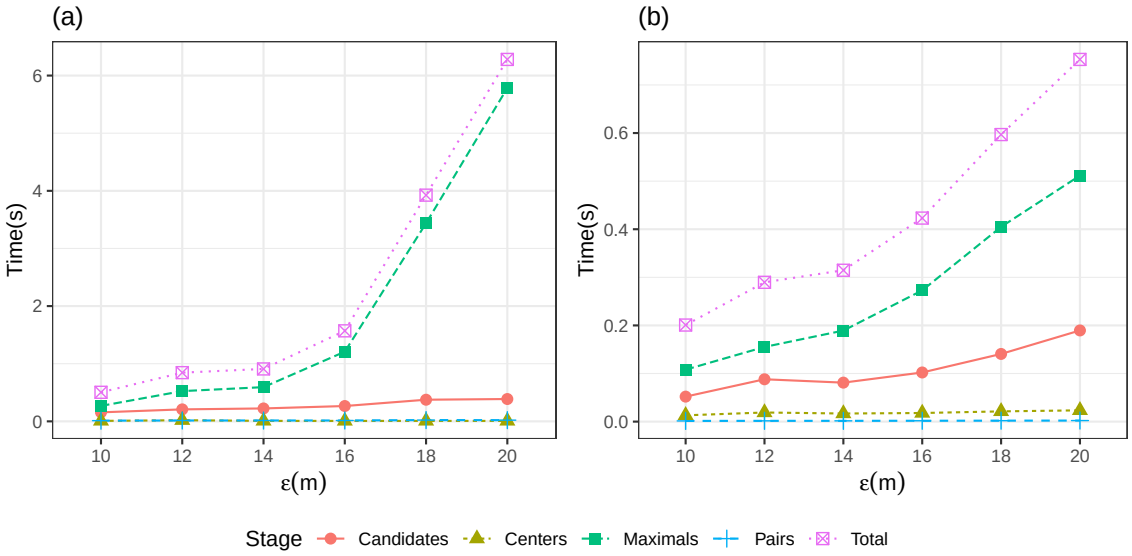


Fig. 15. Processing time for the stages of Phase 1, in (a) standard BFE and (b) standard PSI.

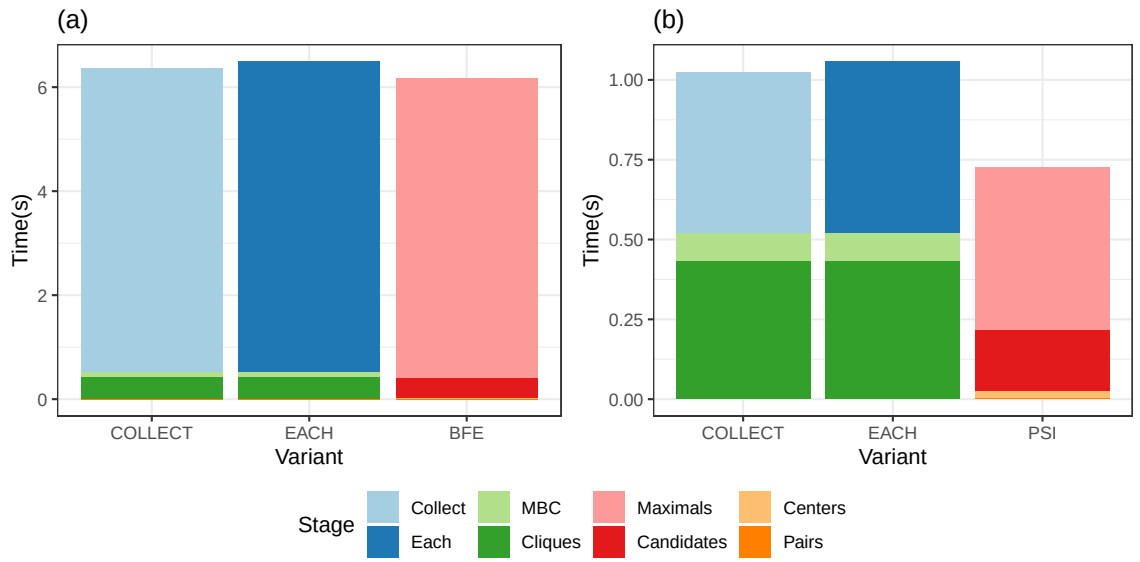


Fig. 16. Execution time of the Cliques approach compared to (a) standard BFE and (b) standard PSI.

To verify whether a maximal clique qualifies as a maximal disk, we introduce the concept of the *Minimum Bounding Circle* (MBC) [29]. Given a set of points in Euclidean space, the MBC is the smallest circle that can enclose all the points. For each maximal clique identified within a partition, we can quickly check if all points in the clique fit within an MBC with a diameter of ε. If they do, we can immediately report the set of points and their MBC as a maximal disk. However, cliques that do not satisfy this condition must be evaluated using the traditional method. This involves computing the potential disk centers, identifying candidate disks, and pruning them, as outlined in Figure 1.

To evaluate the cliques that do not meet the above condition, we implemented two variants. The first variant, termed *COLLECT*, gathers the points from all cliques that are not reported as maximal disks, removes duplicates (since points may appear in multiple cliques), and then applies the traditional pruning method to the entire set. In the second variant, *EACH*, we apply the pruning procedure independently for each clique that does not qualify as a maximal disk.

Figure 16 compares the performance of these variants against the time taken by BFE (a) and PSI (b) for the same stage. Surprisingly, neither variant improves execution time. A closer examination reveals that while identifying the cliques and their MBCs is relatively fast, few cliques actually qualify as maximal disks. As a result, the overhead involved in processing the remaining cliques is significant, making the original approach more efficient for both BFE and PSI.

5.5 Relative performance of BFE and PSI Phase 1 using synthetic datasets.

To further examine the relative performance of the scalable BFE and PSI approaches for Phase 1, we also conducted experiments using a synthetic dataset where we could control the values of c, ε, and point density. We used a fixed square area of 1000m x 1000m, within which we uniformly distributed 25K, 50K, 75K, and 100K points.

We experimented with different quadtree capacities (c values of 100, 200, and 300), which resulted in varying numbers of partitions (as shown in Table 2). Both BFE and PSI were tested for phase 1, where maximal disks are identified, using ε values ranging from 1m to 5m. The results are presented in Figure 17.

Table 2. Number of partitions by capacity and number of points in synthetic uniform datasets.

	25K	50K	75K	100K
c=100	544	1024	1024	2185
c=200	256	514	1024	1024
c=300	256	514	481	1024

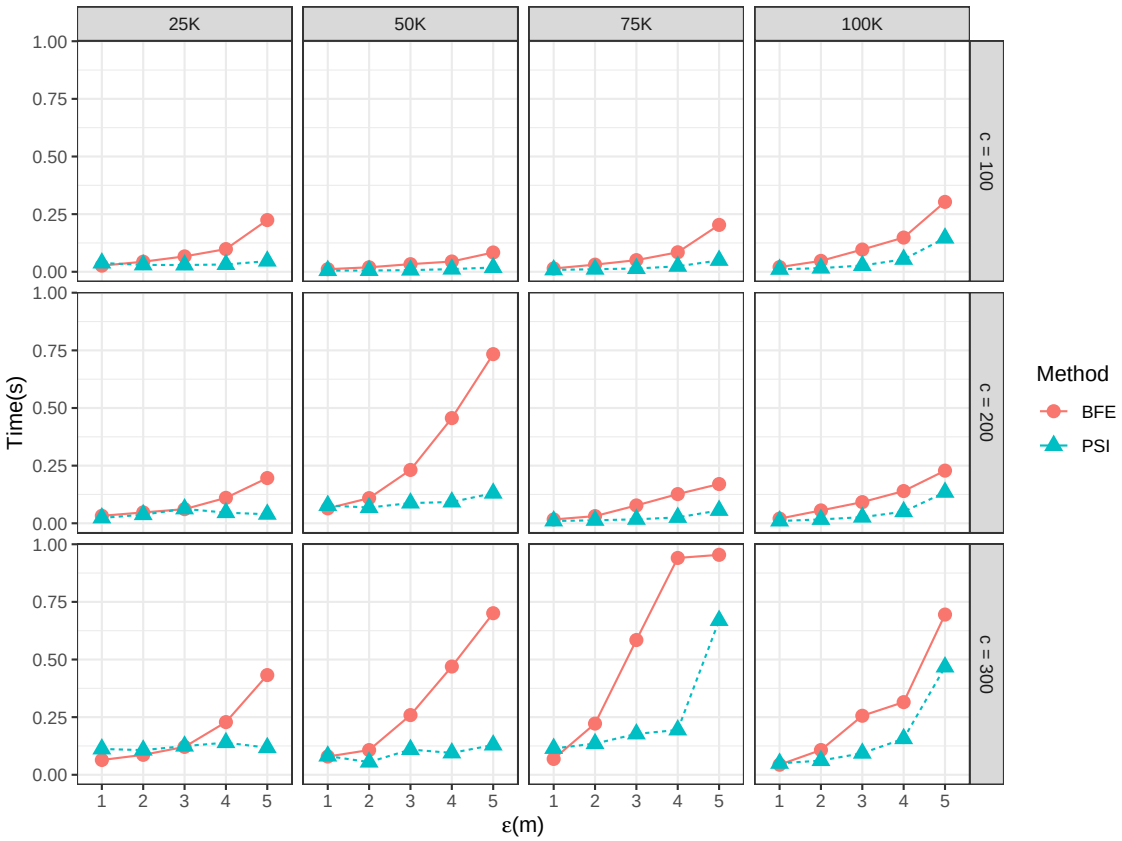


Fig. 17. Performance in an uniform dataset analysing density and capacity with diverse values for epsilon.

Overall, PSI demonstrated better performance than BFE, though there were cases (particularly with smaller ε values) where BFE outperformed PSI. In these cases, the smaller ε generates fewer pairs, and the additional ordering step required by PSI becomes an overhead. However, in the subsequent experiments focusing on temporal joins (phase 2, flock creation), we concentrate on the scalable performance of PSI.

5.6 Evaluation of Phase 2: Temporal join.

Phase 2 focuses on joining maximal disks across time instants to form flocks. In Section 4.2, we discussed four alternatives: Master, By-Level, LCA, and Cube-based. For these experiments, we used the scalable PSI approach due to its robust performance. First, we compared the Master and By-Level alternatives while varying ε from 20m to 40m using the Berlin10K dataset (see Figure 18). For the By-Level approach, we tested different step values ranging from 1 to 6. The

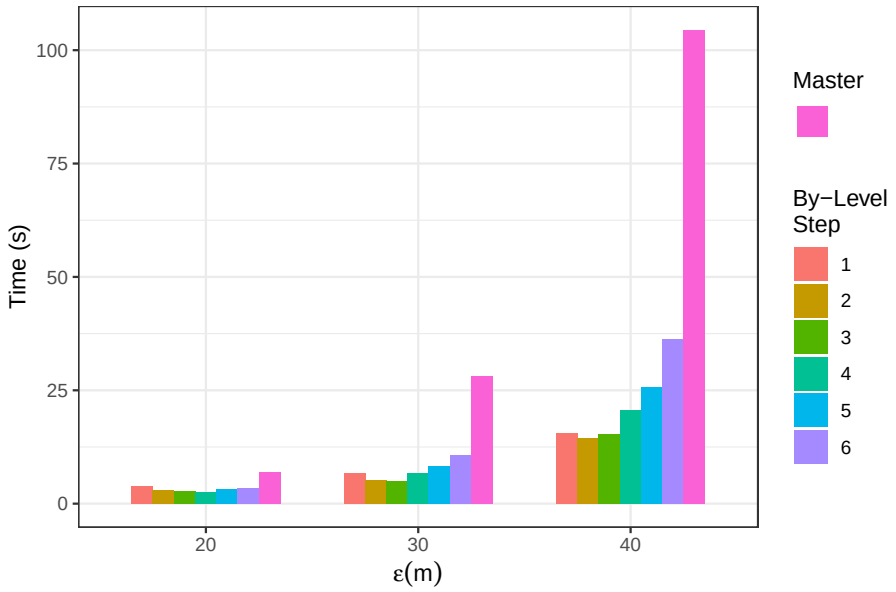


Fig. 18. Root and step alternative for temporal join using the Berlin dataset.

Master approach proved to be the slowest, due to the overhead of sending all CPFs to the root node. The performance of the By-Level approach depends on the step size. A smaller step value (e.g., step 1) introduces overhead because CPFs may need to be evaluated at more intermediate nodes before completion. On the other hand, a larger step value reduces parallelism by sending more CPFs to intermediate nodes. Based on these experiments, we determined that Step=3 offers the best balance.

We also evaluated the optimal value for the *interval* parameter in the Cube-based approach. Using the LA25K dataset with $\varepsilon = 30m$, we tested various interval values, ranging from 2 to 12 time instants. This dataset contains 30 time instants in total. The results, shown in Figure 19, illustrate the trade-offs involved. Lower interval values result in higher parallelism, as more cubes can be processed independently. However, this also increases the number of cube crossings for CPFs that need to be checked, which adds to the execution time. Conversely, larger interval values reduce parallelism but also decrease the number of CPF crossings. Based on these findings, we selected $interval = 6$ as the optimal value for the Cube-based approach.

Finally, we compared the optimized versions of the By-Level and Cube-based approaches with the Master and LCA methods. Figure 20 shows the results, including the sequential PSI algorithm as a reference. This experiment was conducted using the LA25K dataset with ε values ranging from 5m to 30m. Clearly, all parallel approaches offer significant improvements over the sequential PSI.

To further analyze the relative performance of the scalable approaches, Figure 21 focuses on the parallel algorithms for the same experiment. Interestingly, for very small ε values, the Master approach performs best —primarily because the limited number of flocks makes sending the CPFs to a single node fast and efficient. However, as ε increases, the Cube-based approach becomes the most effective, leveraging greater parallelism. By-Level also improves over the Master approach as ε grows, as explained in Figure 18. Similarly, for larger ε values, the LCA approach outperforms By-Level because it more quickly identifies the node that can complete the CPF operations.

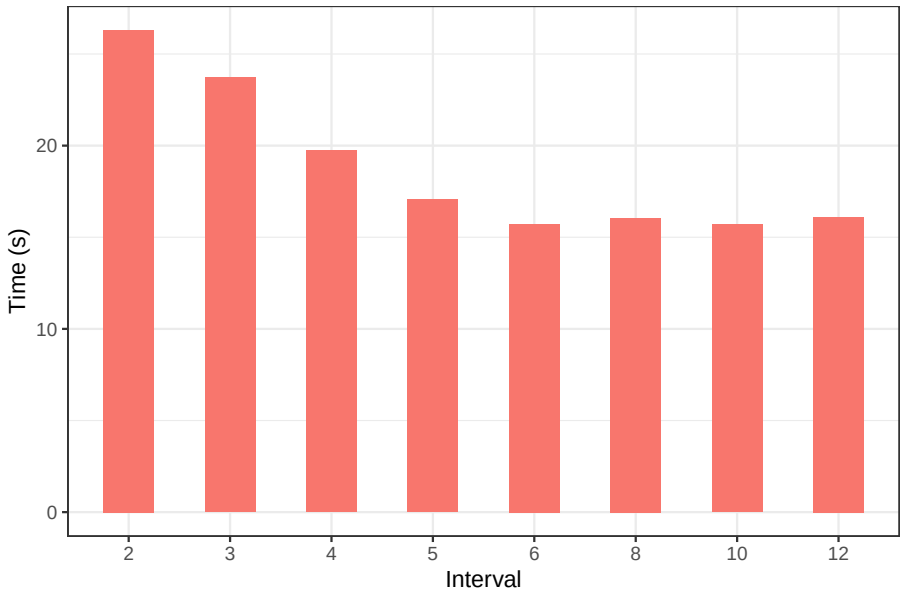


Fig. 19. Interval optimization for the Cube-based alternative for temporal join using the LA25K dataset.

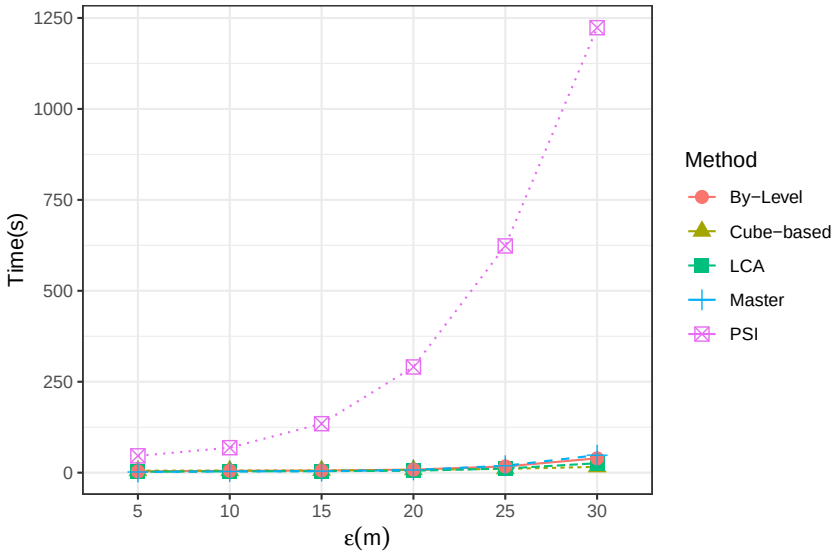


Fig. 20. Performance comparing parallel and sequential alternatives in the LA25K dataset.

We repeated the same experiment with the LA50K dataset, varying ε from 4m to 20m. The results, shown in Figure 22, once again demonstrate that the Cube-based approach offers the best performance as ε increases.

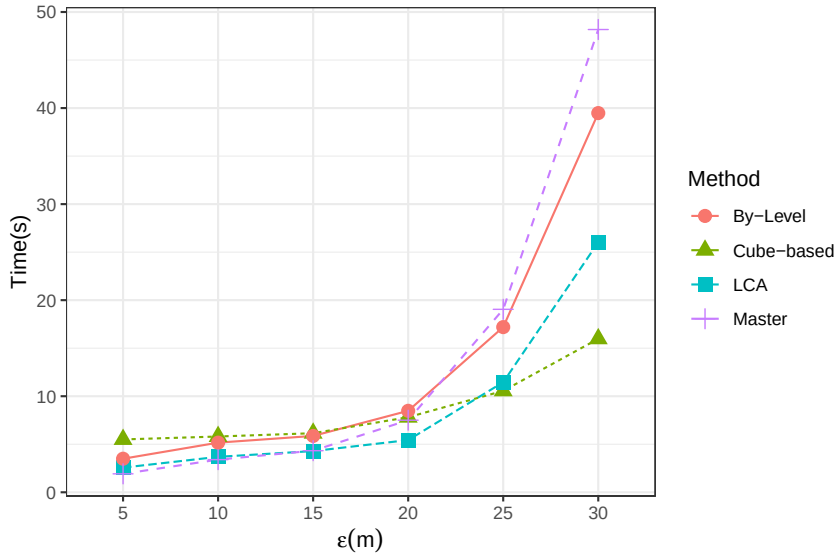


Fig. 21. Performance of the 4 parallel alternatives in the LA25K dataset.

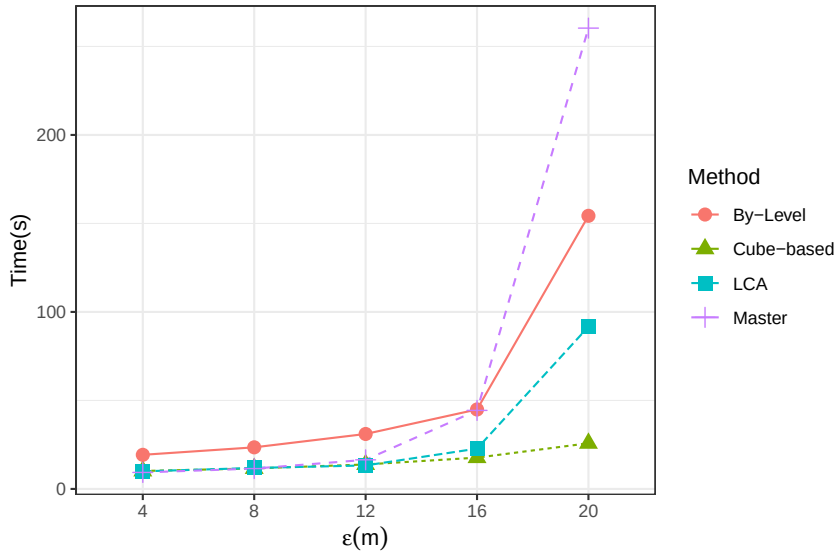


Fig. 22. Performance of the 4 parallel alternatives in the LA50K dataset.

6 CONCLUSIONS

We present a novel, scalable approach to discovering moving flock patterns in large trajectory databases. By leveraging distributed frameworks, the proposed method overcomes the limitations of sequential algorithms that struggle with large-scale spatio-temporal datasets. Through partitioning and replication, as well as improvements in pruning and temporal joins, this approach efficiently handles dense data, offering significant performance improvements over

traditional methods. The evaluation results demonstrate the scalability and effectiveness of the approach, making it a valuable contribution for analyzing complex movement patterns.

REFERENCES

[1] T. Amor, S. Reis, D. Campos, H. Herrmann, and J. Andrade. 2016. Persistence in Eye Movement during Visual Search. *Scientific Reports* 6 (2016), 20815.

[2] H. Arimura, T. Takagi, X. Geng, and T. Uno. 2014. Finding All Maximal Duration Flock Patterns in High-Dimensional Trajectories. (2014).

[3] M. Benkert, J. Gudmundsson, F. Hübner, and T. Wolle. 2008. Reporting Flock Patterns. *Computational Geometry* 41, 3 (2008), 111–125.

[4] C. Bron and J. Kerbosch. 1973. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM* 16, 9 (1973), 575–577.

[5] A. Calderon. 2011. *Mining Moving Flock Patterns in Large Spatio-Temporal Datasets Using a Frequent Pattern Mining Approach.* Master's thesis. University of Twente.

[6] G. Di Lorenzo, M. Sbodio, F. Calabrese, M. Berlingerio, F. Pinelli, and R. Nair. 2016. AllAboard: Visual Exploration of Cellphone Mobility Data to Optimise Public Transport. *IEEE TVCG* 22, 2 (2016), 1036–1050.

[7] A. Eldawy. 2014. SpatialHadoop: Towards Flexible and Scalable Spatial Processing Using Mapreduce. In *SIGMOD PhD Symposium*. 46–50.

[8] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining* (Portland, Oregon) *(KDD'96)*. AAAI Press, 226–231.

[9] M. Fort, J. Antoni, and N. Valladares. 2014. A Parallel GPU-Based Approach for Reporting Flock Patterns. *IJGIS* 28, 9 (2014), 1877–1903.

[10] X. Geng, T. Takagi, H. Arimura, and T. Uno. 2014. Enumeration of Complete Set of Flock Patterns in Trajectories. In *IWGS*. 53–61.

[11] J. Gudmundsson and M. van Kreveld. 2006. Computing Longest Duration Flocks in Trajectory Data. In *ACM SIGSPATIAL*. 35–42.

[12] Mordechai Haklay and Patrick Weber. 2008. Openstreetmap: User-generated street maps. *IEEE Pervasive computing* 7, 4 (2008), 12–18.

[13] K. Holland, B. Wetherbee, C. Lowe, and C. Meyer. 1999. Movements of Tiger Sharks (Galeocerdo Cuvier) in Coastal Hawaiian Waters. *Marine Biology* 134, 4 (1999), 665–673.

[14] P. Huang and B. Yuan. 2015. Mining Massive-Scale Spatiotemporal Trajectories in Parallel: A Survey. In *TAKDDM*. Vol. 9441. Springer.

[15] J. Hughes, A. Annex, C. Eichelberger, A. Fox, A. Hulbert, and M. Ronquest. 2015. GeoMesa: a distributed architecture for spatio-temporal fusion. In *Defense + Security Symposium*.

[16] H. Jeung, M. Yiu, and C. Jensen. 2011. Trajectory Pattern Mining. In *Computing with Spatial Trajectories*. Springer, 143–177.

[17] H. Jeung, M. Yiu, X. Zhou, C. Jensen, and H. Shen. 2008. Discovery of Convoys in Trajectory Databases. *VLDB* 1, 1 (2008), 1068–1080.

[18] P. Kalnis, N. Mamoulis, and S. Bakiras. 2005. On Discovering Moving Clusters in Spatio-Temporal Data. In *ASTD*. Springer, 364–381.

[19] Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. 2012. Recent Development and Applications of SUMO - Simulation of Urban MObility. *International Journal On Advances in Systems and Measurements* 5, 3&4 (December 2012), 128–138.

[20] F. La Sorte, D. Fink, W. Hochachka, and S. Kelling. 2016. Convergence of Broad-Scale Migration Strategies in Terrestrial Birds. *Royal Society: Biological Sciences* 283, 1823 (2016), 2588.

[21] Y. Leung. 2010. *Knowledge Discovery in Spatial Data.* Springer.

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

[23] H. Miller and J. Han. 2001. *Geographic Data Mining and Knowledge Discovery.* Taylor & Francis, Inc.

[24] P. Tanaka, M.. Vieira, and D. Kaster. 2016. An Improved Base Algorithm for Online Discovery of Flock Patterns in Trajectories. *JIDM* 7, 1 (2016).

[25] Etsuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. 2010. A Simple and Faster Branch-and-Bound Algorithm for Finding a Maximum Clique. In *WALCOM: Algorithms and Computation*, Md. Saidur Rahman and Satoshi Fujita (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 191–203.

[26] U. Turdukulov, A. Calderon, O. Huisman, and V. Retsios. 2014. Visual Mining of Moving Flock Patterns in Large Spatio-Temporal Data Sets Using a Frequent Pattern Approach. *IJGIS* 28, 10 (2014), 2013–2029.

[27] M. Vieira, P. Bakalov, and V. Tsotras. 2009. On-Line Discovery of Flock Patterns in Spatio-Temporal Data. In *ACM SIGSPATIAL*. 286–295.

[28] M. Vieira and V. Tsotras. 2013. *Spatio-Temporal Databases: Complex Motion Pattern Queries.* Springer.

[29] E. Welzl. 1991. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science*. Springer, 359–370.

[30] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. In *ICMD*. 1071–1085.

[31] J. Yu, J. Wu, and M. Sarwat. 2016. A Demonstration of GeoSpark: A Cluster Computing Framework for Processing Big Spatial Data. In *ICDE*. 1410–1413.

[32] Y. Zheng and X. Zhou. 2011. *Computing with Spatial Trajectories.* Springer.

A CENTER COMPUTATION.

Algorithm 1 Find the centers of given radius which circumference laid on the two input points.

Require: Radius $\frac{\epsilon}{2}$ and points p_1 and p_2.
Ensure: Centers c_1 and c_2.

function FindCenters(p_1, p_2, $\frac{\epsilon}{2}$)
  $r^2 \leftarrow (\frac{\epsilon}{2})^2$
  $X \leftarrow p_1.x - p_2.x$
  $Y \leftarrow p_1.y - p_2.y$
  $d^2 \leftarrow X^2 + Y^2$
  $R \leftarrow \sqrt{|4 \times \frac{r^2}{d^2} - 1|}$
  $c_1.x \leftarrow X + \frac{Y \times R}{2} + p_2.x$
  $c_1.y \leftarrow Y - \frac{X \times R}{2} + p_2.y$
  $c_2.x \leftarrow X - \frac{Y \times R}{2} + p_2.x$
  $c_2.y \leftarrow Y + \frac{X \times R}{2} + p_2.y$
  return c_1 and c_2
end function

B DISK PRUNING.

Algorithm 2 Prune disks which are duplicate or subset of others.

Require: Set of disks D.
Ensure: Set of disks D' without duplicate or subsets.

function PruneDisks(D)
  $E \leftarrow \emptyset$
  for all disk d_i in D **do**
    $N \leftarrow d_i \cap D$
    for all disk n_j in N **do**
      if d_i contains all the elements of n_j **then**
        $E \leftarrow E \cup n_j$
      end if
    end for
  end for
  $D' \leftarrow D \setminus E$
  return D'
end function

C CLIQUE AND MBC APPROACH.

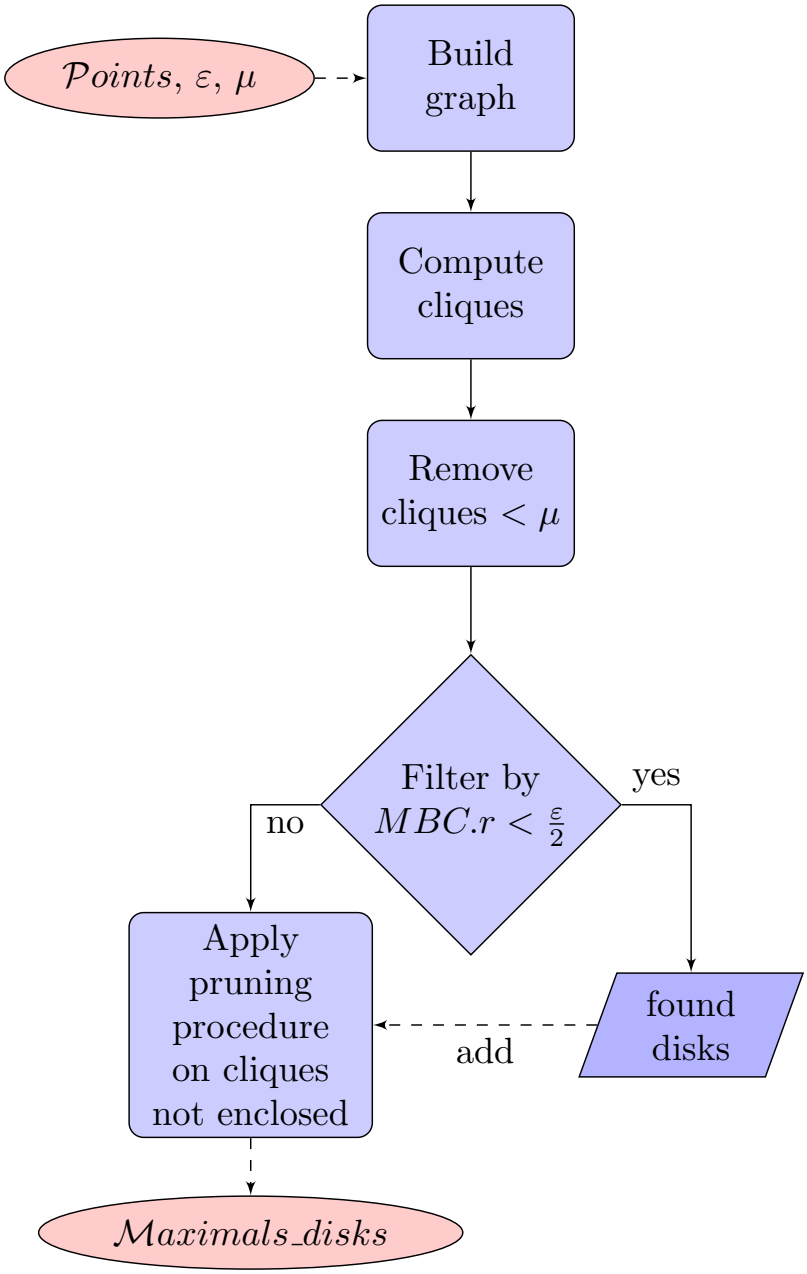


Fig. 23. Schematic description of the Clique and MBC approach.