k/2-hop: Fast Mining of Convoy Patterns With Effective Pruning

Orakzai, Faisal Moeen; Calders, Toon; Pedersen, Torben Bach

Published in:
Proceedings of the VLDB Endowment

DOI (link to publication from Publisher):
10.14778/3329772.3329773

Publication date:
2019

Document Version
Publisher's PDF, also known as Version of record

k/2-hop: Fast Mining of Convoy Patterns With Effective Pruning

Faisal Orakzai[†], Toon Calders[¬], Torben Bach Pedersen[‡]
[†]Department of Computer & Decision Engineering (CoDE) Université Libre de Bruxelles , Belgium
[¬]Department of Mathematics and Computer Science, University of Antwerp
[‡]Department of Computer Science Aalborg University , Denmark
[†]ofaisal@ulb.ac.be ,[¬]Toon.Calders@uantwerpen.be ,[‡]tbp@cs.aau.dk

ABSTRACT

With the increase of devices equipped with location sensors, mining spatio-temporal data for interesting behavioral patterns has gained attention in recent years. One of such well-known patterns is the convoy pattern which can be used, e.g., to find groups of people moving together in public transport or to prevent traffic jams. A convoy consists of at least m objects moving together for at least k consecutive time instants where m and k are user-defined parameters. Convoy mining is an expensive task and existing sequential algorithms do not scale to real-life dataset sizes. Existing sequential as well as parallel algorithms require a complex set of data-dependent parameters which are hard to set and tune. Therefore, in this paper, we propose a new fast exact sequential convoy pattern mining algorithm "k/2-hop" that is free of data-dependent parameters. The proposed algorithm processes the data corresponding to a few specific key timestamps at each step and quickly prunes objects with no possibility of forming a convoy. Thus, only a very small portion of the complete dataset is considered for mining convoys. Our experimental results show that k/2-hop outperforms existing sequential as well as parallel convoy pattern mining algorithms by orders of magnitude, and scales to larger datasets which existing algorithms fail on.

PVLDB Reference Format:
Faisal Orakzai, Toon Calders and Torben Bach Pedersen. k/2-hop: Fast Mining of Convoy Patterns With Effective Pruning. *PVLDB*, 12(9): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/3329772.3329773

The extended version of this paper is available as [17]. Implementation of all algorithms can be found at https://bit.ly/2SMOwDE.

1. INTRODUCTION

The massive amounts of GPS data today can be analyzed to study collective mobility behaviour. One pattern studied extensively in this context is the *Convoy* pattern [13, 14, 1, 10, 25]. A convoy is a group of at least m objects moving together (within *eps* distance of each other) for at least k time instants. Finding convoy patterns is useful in many application domains. It can be used to find groups of people traveling together for a certain time to analyze the feasibility of starting a public transport service in the areas where higher numbers of convoys are found. It can also be used to determine potential candidates for carpooling. For instance, to find potential car-pooling routes, we could use $m >= 2$ so we can pool at least 2 persons. The choice of the value of k depends on the minimum time duration of the trips we are interested in pooling. Persons/vehicles forming convoys repeatedly every morning and evening could be persons working in the same area and taking similar routes to their work location and thus could be good candidates for car-pooling. Also in traffic jams, many vehicles are generally located near to each other for long times. If we want to detect all traffic jams of duration more than 15 mins and involving 50 cars or more, we would set m to 50 and k to 15 (if the sampling frequency of the data is 1 min). The choice of *eps* determines what objects are considered to be "together" while mining convoys. For the car-pooling example, this could be a few metres within a city and a few hundred meters for inter-city convoys. Note that m, k and *eps* are user parameters whose values are determined based on the use-case.

The groups of objects forming convoys are usually found by performing, at each time instant, a density-based clustering algorithm such as DBSCAN [6] (with parameter *eps*) on the object locations followed by combining the found clusters over the time dimension into convoys. For mining convoy patterns, various algorithms [13, 14, 25, 1] have been proposed. The existing sequential algorithms, however, have been tested on small datasets that can easily fit into memory only and thus completely ignore data access optimizations.

To alleviate this problem, a scalable Distributed Convoy Mining algorithm called *DCM* has been proposed [18, 16]. Although DCM can operate on huge datasets, choosing reasonable values for the parameters of the algorithm is a challenging task and requires a deep understanding of the algorithm as well as the distribution of the input data. Even though the DCM algorithm can run on a cluster of machines in parallel, the cost of the algorithm is high as it has to process all the points in a dataset and thus, it cannot benefit from indexing techniques or run on operational data stores efficiently.

The Star Partitioning and ApRiori Enumerator (SPARE) framework proposed in [7] is the state of the art in convoy mining. The framework considers the clustering process as a pre-processing step and focuses only on the part in which the

Proceedings of the VLDB Endowment, Vol. 12, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3329772.3329773

clusters are matched together to mine convoys. For convoy mining, the clustering phase is the most expensive one as it involves scanning and clustering the whole dataset. SPARE, however, ignores the clustering part. This approach doesn't yield much benefits because the part that is optimized is dominated by the clustering phase, which is not optimized.

In this paper, instead of scaling the convoy mining algorithm by parallelizing, we focus on improving the efficiency of the sequential algorithm. In our proposed algorithm, we use simple but effective heuristics to prune the data corresponding to objects which have no chance of forming a convoy. We define the notion of *benchmark points*, which are timestamps spread evenly at a distance $k/2$ (k being the minimum length of the convoy to be mined). We prove that, in order to find convoys of length at least k, we need to fully cluster only the data corresponding to the benchmark points. For the rest of the timestamps, we only cluster a subset of the data. The reduction of the search space greatly improves the performance of convoy mining as shown in our experiments.

This paper has the following contributions:

- We propose a fast and efficient convoy mining algorithm called k/2-hop which can operate on a variety of operational data stores and can easily be parallelized.
- We propose a correction to the fully connected convoys validation algorithm *DCVal* proposed in [25].
- We present an experimental evaluation of k/2-hop and show that k/2-hop prunes more than 99% of the data in most cases and is orders of magnitude faster than the existing sequential and parallel algorithms.

The remainder of this paper is structured as follows. Section 2 describes the related work. Section 3 explains the convoy mining problem. Section 4 describes the k/2-hop algorithm, Section 5 describes the different storage options suitable for k/2-hop, and the experiments are covered in section 6. Section 7 concludes the paper.

2. RELATED WORK

A number of mobility patterns and their detection algorithms have been proposed. One of the first such patterns is the *flock* pattern [9, 24]. A flock is a group of objects moving together for a time period t in the sense that at each time instant the convoy can be covered by a disk with a given radius. Although flocks are a good way of identifying objects moving in groups, the disk constraint limits its ability to represent objects moving together either in a shape other than a disk or in a disk shape greater than the size specified by the user. The choice of an appropriate size for the disk itself is a challenging problem.

To avoid the size and shape constraints required by the flock pattern, Jeung et al. have proposed the *convoy* pattern in [13, 14]. Unlike the *flock* pattern, in a *convoy* pattern, objects which are density-connected to each other are considered to be together. This relieves the *convoy* pattern of any restriction of size and shape.

The group movement pattern mining algorithms can be categorized by the type of the movement behaviour they capture. Kalnis et al. [15] proposed the notion of a *moving cluster* which is a sequence of spatial clusters with a certain percentage θ of similarity between the clusters of consecutive timestamps. A *moving cluster* maintains its identity while objects join or leave, whereas in a *convoy*, the objects should be together throughout the convoy's lifespan. Jung et al. proposed another type of group movement pattern in [1] called *evolving convoys* in which objects can join and leave a convoy during its lifespan. Tang et al. [23] proposed a buddy based approach for finding traveling companions from streaming trajectories. In the convoy pattern proposed in [13, 14], the objects in a convoy can be connected to each other through objects that are not part of the convoy. Yoon et al. [25], discovered that the convoy mining algorithms proposed by Jeung et al. [14] have serious problems with accuracy and recall. They refer to the convoy pattern proposed by Jeung et al. as a *partially connected convoy* pattern. They present a corrected version of the *CMC* algorithm called *PCCD* (Partially Connected Convoy Discovery). As the *CuTs* family of algorithms [25] is based on *CMC*, they also have serious accuracy issues. Yoon et al. [25] proposed a contextually different version of the *convoy* pattern called *Valid Convoy* which is a group of objects that are density-connected to each other through only those objects that are also part of the convoy. With the motivation of improving the naming convention used in [25], we call valid convoys *Fully Connected Convoys*.

The primary reasons for the poor performance of existing algorithms are the disk I/O involved in touching the whole dataset and the cost complexity of the original DBSCAN which is $\mathcal{O}(n^2)$ for each run where n is the number of objects present at the corresponding time. The DBSCAN algorithm issues a nearest neighborhood query for each point. Hence 3 hours of movement data of a million objects with a sampling frequency of 2 seconds, result in 540 million nearest neighborhood queries which may take days to process on a PC. The cost complexity of DBSCAN can be reduced to $\mathcal{O}(n \log n)$ by using an indexing structure. The memory cost of DBSCAN is $\mathcal{O}(n)$. To speed up DBSCAN, a distance matrix of size $(n^2 - n)/2$ can be constructed in memory but this increases the required memory to $\mathcal{O}(n^2)$. Existing algorithms, e.g., *CMC* [14] and *VCoDA/PCCD* [25] are plagued with expensive disk I/O and clustering as clustering needs to be done for each timestamp and as a consequence they cannot scale to huge datasets. Parallel DBSCAN algorithms [20, 12, 4] are good for one large run but not suitable for many small DBSCAN runs.

The sequential algorithms for convoy mining, *CuTS*, *CuTS+* and *CuTS** [14] try to reduce the cost of DBSCAN by reducing the number of objects n in each run using a filter-and-refine paradigm. In the 1st phase, each object's trajectory is simplified by reducing the number of time-location pairs using the *Douglas-Peucker* algorithm (*DP*) [5]. The simplified trajectories are then partitioned into pieces each corresponding to a time duration λ. For each time duration λ, the pieces(sub-trajectories) are clustered using the DBSCAN algorithm. This step reduces the dataset to only those object trajectories which have the potential to form convoys. In the second step, the *CMC* algorithm is applied on the reduced dataset. Although this method has proven to be faster than the CMC algorithm, the cost of trajectory simplification is $\mathcal{O}(T^2 N_t)$ where T is the number of points in a trajectory and N_t is the number of trajectories in the dataset. Using trajectory simplification also disallows us from using the same indexing structure as that of DB-

SCAN. The index for DBSCAN is based on location whereas the index required for trajectory simplification is based on object identity. The fact that the trajectory simplification process cannot use the existing spatial index constructed for DBSCAN has completely been ignored as the implicit assumption is that the data fits in memory and disk seeks are not performed. This assumption renders these algorithms inappropriate for huge data sizes as our experiments demonstrate. Additionally, finding the right combination of parameters for the *CuTs* family, for it to give acceptable performance is very hard and may involve multiple expensive iterations with no guarantee for success.

In [21], Phan et. al propose the "All in one" framework for mining multiple movement patterns. The framework seems tempting because of its ability to mine multiple patterns, but just like other sequential algorithms, it require all the data to be clustered before it could start mining movement patterns. Thus, the framework ignores the most expensive part and focusses on the cheaper part. Algorithms for mining other patterns, e.g., Swarm, are also plagued by this problem. In [11], a unifying movement pattern mining framework capable of mining multiple movement patterns calle *get_move* is proposed. This framework operates on a cluster matrices leading to the same problem of having to run the expensive clustering process.

The algorithms proposed in the above mentioned papers are sequential in nature and are not scalable enough to tackle large mobility datasets. For instance, the most efficient sequential convoy pattern mining algorithm out of the algorithms proposed in [14] took 100 seconds during an experimental run on a small dataset containing a couple of hours of movement data of only 13 cattle. The huge sizes of current mobility datasets and the limitations of existing algorithms led to the development of a parallel algorithm that can run on a set of loosely connected machines (cluster) and can produce results faster [18, 16]. Orakzai et al. [18] discuss various partitioning strategies for mining convoy pattern in a distributed setting. [16] presents a generic *Distributed Convoy Mining* algorithm *DCM* with an implementation using the Hadoop MapReduce framework [8]. DCM is based on the *CMC* algorithm [14] which is inherently inefficient because of its optimistic search for convoys touching each data point more than once. DCM's performance is highly dependent on the partition/split size and bad choices can lead to performance worse than the sequential algorithms. Additionally, maintaining/running Hadoop clusters makes the algorithm costly in a production environment.

The state of the art in convoy mining is the work in [7]. The authors propose a generic framework *GCMP* for mining co-movement patterns. The authors propose two parallel implementations of GCMP framework using Apache Spark as an underlying platform, a baseline implementation and a more optimal one called the Star Partitioning and ApRiori Enumerator (SPARE) framework. The experiments show huge performance gains over sequential algorithms. In both implementations, there are two stages of MapReduce jobs connected in a pipeline manner. The first stage deals with spatial clustering of objects in each timestamp (which they call a snapshot), which can be seen as a preprocessing step for the subsequent pattern mining phase. In particular, for the first stage, the timestamp is treated as the key in the map phase and objects within the same snapshot are clustered (DBSCAN or disk-based clustering) in the reduce phase. Finally, the reducers output clusters of objects in each snapshot, represented by a list of key-value pairs. The clusters are then provided to the second phase of the pipeline as input in which patterns are mined from the clusters. For convoy mining, the first phase is the most expensive one, however, the baseline implementation and SPARE both focus only on the second phase of the pipeline and consider the first phase as a preprocessing phase. This approach leads to larger overall convoy mining execution times.

3. CONVOY MINING PROBLEM

3.1 Density-based Clustering

Before we explain the convoy pattern, it is necessary to understand the concept of density connectedness [6]. Consider a point p in a set of points S and a distance threshold eps. The eps-neighborhood of point p can be defined as $NH(p, eps) = \{q \in S \mid d(p, q) \leq eps\}$ where $d(p, q)$ represents the euclidean distance between two points. Given a point p, a distance threshold eps and an integer m, a point q is said to be *directly density reachable* from p if $q \in NH(p, eps)$ and $|NH(p, eps)| \geq m$ and is denoted by $p \rightarrow q$. If there exists a chain of points $p_1, p_2, \ldots, p_n$ such that for all i, $p_i \rightarrow p_{i+1}$, p_n is said to be *density reachable* from p_1 and is denoted by $p_1 \rightarrow^* p_n$. Based on the previously defined terms, we can now define the notion of *density connected*.

Definition 1. ***(Density-Connected)*** *Given a set of points S, a point $p \in S$ is* density-connected *to a point $q \in S$ (denoted by $p \leftrightarrow^* q$), with respect to eps and m if there exists a point $x \in S$ such that $x \rightarrow^* p$ and $x \rightarrow^* q$.*

A set of density connected objects at a time instant is called an *(m,eps)-cluster* and is defined as follows:

Definition 2. **((m,eps)-Cluster)** *Given an integer m and distance threshold eps, a set c is called a* (m,eps)-cluster*, if for all $p, q \in c$ such that $p \neq q$, $p \leftrightarrow^* q$ and there does not exist an object $r \notin c$ such that $p \rightarrow^* r$.*

3.2 Convoys

At a physical level, movement data is stored in a 4-column table with schema $< oid, x, y, t >$ where *oid* is the object id, x and y represent the location in two dimensions, and t is the time instant at which the object was at that location. The trajectory of an object o can be extracted from this data by retrieving all the tuples with $oid = o$ and ordering them by time column t.

In Figure 1 (reproduced from [18]), for $m = 3$, clusters c_{1-1}, c_{2-1} and c_{3-1} are (3,eps)-clusters as they are maximal sets of density connected points with $size \geq m$. The (m,eps)-clusters at a time t_x can be found by querying all tuples with $t = t_x$ to retrieve all objects present at time t_x and their locations, and performing density based clustering. A convoy can be defined as follows:

Definition 3. ***(Convoy)*** Let Obj be a set of objects and s, e be timestamps with $s <= e$. $(Obj, [s, e])$ is called a (m, eps)-convoy if for all $t \in [s, e]$, there exists a (m, eps)-cluster C at timestamp t such that $Obj \subseteq C$.

From now on, we will assume that m and eps have been fixed and omit them from the notation; i.e., cluster will denote (m,eps)-cluster.

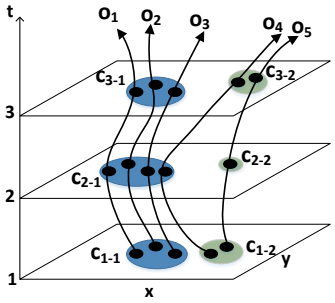


Figure 1: Convoy Pattern (in blue) with $m = 3$ and $k = 3$

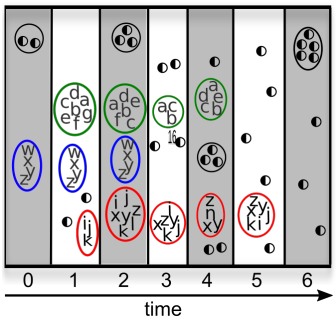


Figure 2: Convoy Types

If v is a convoy, we denote its set of objects, respectively timestamps with $O(v)$, respectively $T(v)$. In the figure 2, for $m = 3$ and a distance threshhold *eps* , $(\{a, b, c\}, [1, 4])$, $(\{x, y, z\}, [2, 5])$, $(\{i, j, k\}, [1, 3])$, and $(\{a, b, c, d, e, f\}, [1, 2])$ are convoys. Note that a single (m,eps)-cluster c at time t forms an (m,eps)-convoy $(c, [t, t])$.

Definition 4. ***(Fully Connected (FC) Convoy)*** Let OB' be a set of objects in a database DB, then we define the restriction of the database DB to objects OB' as $DB|_{OB'} := \{(oid, x, y, t) | oid \in OB'\}$. A *(m,eps)*-convoy $v = (OB', [t_s, t_e])$ is called a fully connected *(m,eps)*-convoy (FC) if v is a *(m,eps)*-convoy in $DB|_{OB'}$.

In a fully connected convoy v, the objects $O(v)$ are density connected without the help of any other object that is not part of $O(v)$; i.e., $O(v)$ is self-sufficient in satisfying the density connected property throughout the lifetime of the convoy. For example, in Figure 2, $(\{x, y, z\}, [1, 5])$ is a convoy but not a fully connected convoy because at timestamp 4 the objects of the convoy $\{x, y, z\}$ are density connected to each other through object n and do not form a cluster without object n. Also $(\{a, b, c\}, [1, 4])$ is a $(3, eps)$-convoy but not fully connected. At all times, the objects of the convoy are density connected without the involvement of other objects that are not part of the final convoy except at timestamp 4. On the other hand $(\{a, b, c\}, [1, 3])$ is an FC (3,eps)-convoy.

Definition 5. ***(Sub-Convoy and Strict Sub-Convoy)*** v is a *sub-convoy* of a convoy w if $O(v) \subseteq O(w)$ and $T(v) \subseteq T(w)$. In that case, w is called a *super-convoy* of v. If v is a sub-convoy of w and $v \neq w$, then v is called a *strict sub-convoy* of v.

Definition 6. ***(Maximal Convoy)*** An *(m,eps)*-convoy v is called *maximal* if there does not exist another convoy w such that v is a strict sub-convoy of w.

Definition 7. ***(Maximal FC Convoy)*** A fully connected *(m,eps)*-convoy v_{fc} is called *maximal* if there does not exist another fully connected convoy w_{fc} such that v_{fc} is a strict sub-convoy of w_{fc}.

For instance, in Figure 2, $(\{a, b\}, [1, 2])$ is an FC (2,eps)-convoy but it is not maximal because it is a sub-convoy of another FC (2-eps) convoy $(\{a, b, c\}, [1, 3])$. $(\{a, b, c\}, [1, 3])$ is a maximal FC convoy because there doesn't exist any other FC convoy of which it is a sub-convoy. In the Figure 2, $v_1 = (\{w, x\}, [0, 2])$ is a sub-convoy of $v_2 = (\{w, x, z\}, [0, 2])$ which is a sub-convoy of $v_3 = (\{w, x, y, z\}, [0, 2])$. Note that v_1 and v_3 are FC convoys whereas v_2 is not.

Definition 8. ***(FC Convoy Mining Problem)*** Given a dataset D, clustering parameters m, eps, and minimum length of convoy k, find all maximal FC (m, eps)-convoys v such that $|T(v)| >= k$.

The following lemmas describe the relationship between a convoy and an FC convoy.

LEMMA 1. *Every FC convoy is a convoy and hence, every maximal FC convoy is a sub-convoy of a maximal convoy.*

Let V be the set of all (m,eps)-convoys in a dataset and V_{FC} be a set of all FC (m,eps)-convoys in that dataset, then each convoy in V_{FC} is a sub-convoy of at least one convoy in V. Thus, using the validation algorithm proposed in [25], V can be reduced to V_{FC}.

LEMMA 2. *If v is a convoy, then for any $O' \subseteq O(v)$ and $T' \subseteq T(v)$, (O', T') is a convoy as well.*

However, the above lemma doesn't hold for FC convoys. For instance, in Figure 2, $(\{a, b, c\}, [1, 3])$ is an FC (2,eps)-convoy but its sub-convoy $(\{a, c\}, [1, 3])$ is a (2,eps)-convoy but not an FC (2,eps)-convoy because at timestamp 2, object a is density connected to object c through object b which is not a part of the convoy.

Table 1: Summary of Notation

Symbol	Meaning
v	A convoy
DB	Dataset
T_s	Start time of the dataset
T_e	End time of the dataset
$DB[T]$	dataset restricted to time interval T
$DB_{\|O}$	dataset restricted to objects in set O
B^n	Set of n^{th} order benchmark points
H_i	i^{th} hop-window
$V_{sp}(i)$	Spanning candidate convoy set from H_i
V_M	Merged convoy set
V_{sp}^{mr}	Maximal right closed convoy set
V_{FC}	Set of fully connected convoys
L	Lifespan of a convoy
CC_i	Candidate Clusters for hop-window H_i
CC_i^l	Candidate Clusters for level l hop-window H_i
CC_i^{l*}	Intermediate CC_i^l
C_i	Set of clusters at b_i
$t_s(v)$	Start time of convoy v
$t_e(v)$	End time of convoy v
$O(v)$	Objects in convoy v

4. k/2-HOP ALGORITHM

Convoy mining is an expensive task. Existing algorithms are plagued by multiple expensive runs of DBSCAN, which involve clustering all the points corresponding to each timestamp. From our experiments on real world datasets, we have observed that the Convoy pattern is not a frequent pattern. Most of the detected clusters do not become part

of any convoy, thus wasting valuable compute and I/O cycles. Therefore, in this section, we propose the k/2-hop algorithm, that is capable of pruning most of the data points and clustering only the relevant convoy member points with 100% accuracy. k/2-hop has six steps as shown in Algorithm 1. Each of the steps is explained in the following sections.

4.1 Benchmark Points and Clusters

We call every $k/2^{th}$ timestamp in the dataset a *benchmark point* and the time period between two consecutive benchmark points, a *hop-window*. The i^{th} benchmark point is denoted by b_i and the i^{th} hop-window is denoted by H_i. B represents the set of all benchmark points and H the set of all *hop-windows*. Figure 3 shows the main ideas of k/2-hop. This figure represents the different timestamps $[0, 16]$ and the convoys are represented by rounded rectangles. The algorithm is based on the following observations (see appendix for proofs):

LEMMA 3. *Let k be the minimum length of a convoy to be mined. Then for each convoy v with lifespan $L = [t_s, t_e]$ such that $|L| \geq k$, there must exist at least two consecutive benchmark points b_i and b_{i+1} such that $b_i \in L$ and $b_{i+1} \in L$.*

PROOF. See full version [17] of the paper. □

Algorithm 1 k/2-hop

Input: Dataset DB, Convoy size parameter m, Convoy length parameter k, Distance threshold *eps*
Output: Set of fully connected convoys V_{FC}
1: $C = \{C_i | i = 1, 2, \ldots, n\}$ such that $C_i \leftarrow DBSCAN(b_i)$ and $b_i = i * \lfloor k/2 \rfloor$
▷ The set of benchmark cluster sets
2: $CC = \{CC_i = C_i \cap^{set} C_{i+1} | i = 1, 2, \ldots, n\}$
▷ The set of Candidate cluster sets
3: $V_{sp} = \{V_{sp}^i : HWMT(H_i, CC_i) : H_i \in H\}$
▷ 1st order spanning candidate convoys
4: $V_M = DCM_{merge}(\{V_{sp}^1, V_{sp}^2, ..., V_{sp}^n\})$
▷ Maximal spanning candidate convoys
5: $V_E \leftarrow$ extend(V_M, DB)
▷ Semi-connected convoys
6: $V_{FC} \leftarrow$ DCVal(V_E)
▷ Fully Connected convoys
7: **return** V_{FC}

Figure 3 shows convoys with $k = 8$. It can easily be seen that for a convoy with length greater than or equal to k, crossing at least two consecutive benchmark points is inevitable.

The set of (m,eps)-clusters at a benchmark point b_i is called the *benchmark clusters* set and is denoted by C_i. Note that as the objects of a convoy remain together for the whole length of the convoy, this is also true for the benchmark points it crosses, which implies that the objects of the convoy crossing b_i must be a subset of one of the benchmark clusters in C_i.

LEMMA 4. *Let C be the set of clusters at a benchmark point b and $O(x)$ be the set of objects of a convoy or cluster x. Then for every convoy v crossing b, there exists a $c \in C$ such that $O(v) \subseteq c$.*

PROOF. See full version [17] of the paper. □

4.2 Candidate Clusters

As a convoy must cross at least two consecutive benchmark points, the objects of a convoy must be part of a benchmark cluster at both of these benchmark points. In other words, if we take a set-wise intersection of the two benchmark cluster sets at the two benchmark points, the objects of the convoy must be part of at least one of the sets in the result.

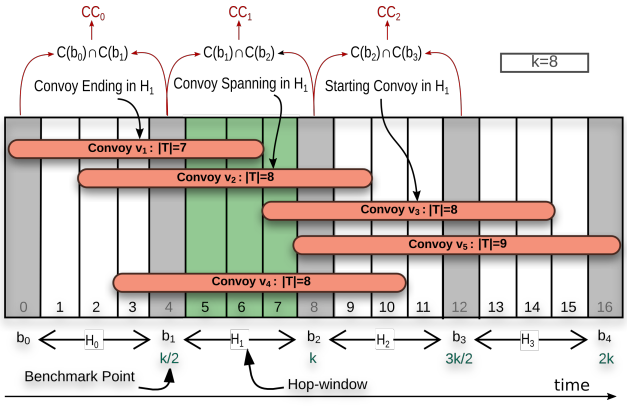


Figure 3: Red rectangles show convoys with k=8, b_i is a benchmark point, H_i is a hop-window. v_1 ends in H_1, v_2 crosses H_1 and v_3 starts in H_1

LEMMA 5. *Let v be a convoy crossing two consecutive benchmark points b_i and b_{i+1} and let C_i and C_{i+1} be the respective sets of clusters. Then there must exist clusters $c_i \in C_i$ and $c_{i+1} \in C_{i+1}$ such that $O(v) \subseteq c_i \cap c_{i+1}$.*

PROOF. See full version [17] of the paper. □

This lemma plays a crucial role in pruning the data of the objects that cannot be part of a convoy in a hop-window. For instance, let C_1 and C_2 be the sets of clusters at consecutive benchmark point b_1 and b_2:

$$C_1 = \{\{a, b, c, d\}, \{e, f, g, h\}, \{i, j, k\}\}$$
$$C_2 = \{\{a, b, c\}, \{d, e\}, \{f, g, h\}, \{i, j\}\}$$

Then the element-wise set intersection of C_1 and C_2 results in $\{\{a, b, c\}, \{d\}, \{e\}, \{f, g, h\}, \{i, k\}\}$. Assuming we are looking for convoys with $m = 3$, we discard the sets with size less than 3 to get the *Candidate Clusters* set $\{\{a, b, c\}, \{f, g, h\}\}$ for hop-window H_1 denoted by CC_1. To find the convoys that span the hop-window, we can filter out the data corresponding to all objects other than the members of CC_1.

We formally define the candidate clusters for hop-window i as follows:

$$CC_i := \{c_i \cap c_{i+1} \mid c_i \in C_i, c_{i+1} \in C_{i+1},\ |c_i \cap c_{i+1}| \geq m|\}$$

As shown above, instead of touching the whole dataset, we first find clusters only at the benchmark points and find CC by the set-wise intersection of the cluster sets from the adjacent benchmark points, i.e., $CC = \{CC_1, CC_2, ...CC_n\}$. As in real word cases, e.g., vehicular traffic, convoy is an uncommon pattern, CC contains only a few objects. This allows us to prune all data corresponding to the objects not part of any cluster in CC. Mining convoys in the reduced data is much faster.

4.3 Hop-Window Mining Tree (HWMT)

Up to this point, we have a CC for each hop-window. The next task is to mine convoys in the hop-window using CC. Before moving forward it is important to know what kind of convoys we can find in a hop-window.

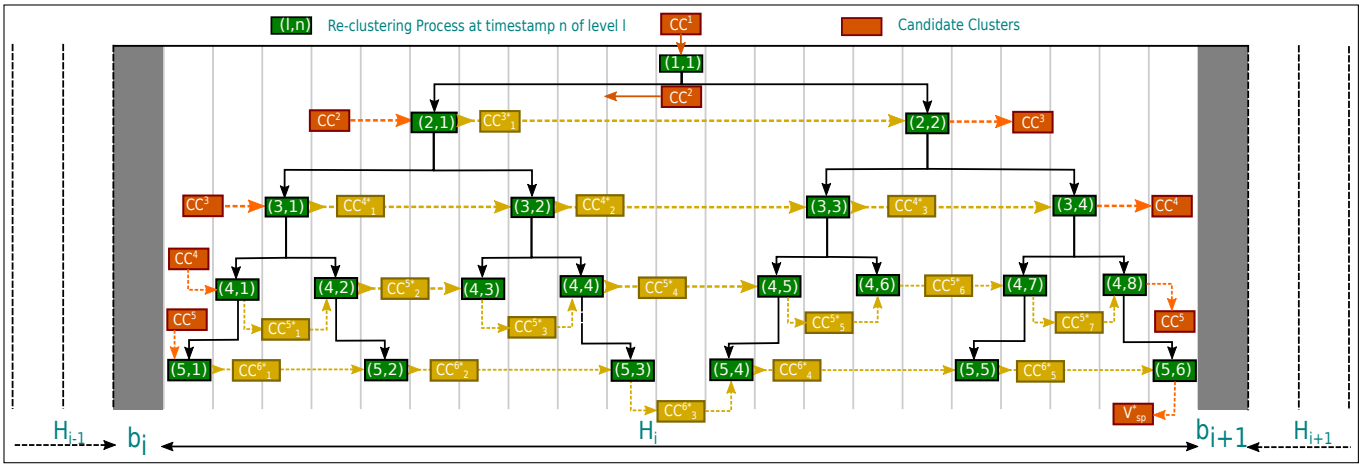


Figure 4: Hop-Window Mining Tree for Spanning Convoys. The cluster sets from b_i and b_{i+1} are intersected to form the candidate cluster set CC^1 (input to HWMT level 1) which is clustered at the timestamp b^2 to form the candidate cluster set CC^2 (input to HWMT level 2).

LEMMA 6. *If a convoy v with lifespan L overlaps a hop-window H_i, i.e., $L \cap H_i \neq \emptyset$, only one of the following can be true:*

1. *$L \cap \{b_i, b_{i+1}\} = \{b_i, b_{i+1}\}$, i.e., v spans H_i entirely.*
2. *$H_i \cap \{t_s(v)\} \neq \emptyset$, i.e., v starts in H_i.*
3. *$H_i \cap \{t_e(v)\} \neq \emptyset$, i.e., v ends in H_i.*

PROOF. See full version [17] if the paper. □

There can be three types of convoys whose lifespan overlaps with a hop-window, i.e., the convoys which span/cross the hop-window (*spanning* candidate convoys), the convoys which end in the hop-window (*ending* candidate convoys) and the convoys which start in the hop-window (*starting* candidate convoys). The number of hop-windows a candidate spanning convoy spans is called the *order* of the candidate spanning convoy. Therefore, the candidate spanning convoys mined in a hop-window are 1st order candidate spanning convoys. Our proposed HWMT algorithm which is a part of the global k/2-hop algorithm, operates on a hop-window H_i and mines the 1st order candidate spanning convoys efficiently by reducing the size of CC_i at each step. Applying the HWMT algorithm on all hop-windows gives us all the 1st order spanning convoys. Working on individual hop-windows independently of other hop-windows makes the HWMT algorithm a good candidate for distributed execution.

The HWMT algorithm tries to reduce the effect of coincidental togetherness. The algorithm exploits the fact that the chance of objects being coincidentally together in adjacent timestamps is higher than the chance of them being coincidentally together in distant timestamps. Therefore it picks up the farthest timestamps in the hop-window for processing in each step.

The HWMT algorithm operates on the Hop Window Mining Tree, which is a binary search tree of timestamps with the middle timestamp of the hop-window as the root node. The nodes in the next levels represent timestamps in the middle of the timestamps of the previous level, and so on. Figure 4 shows a sample HWMT. The candidate cluster set of the hop-window is reclustered at the root node. The result is then reclustered at the left timestamp of the 2nd level, the result of which is then reclustered at the right timestamp. This process is repeated for all the levels of the tree until no clusters are found at any timestamp. After each reclustering step, the result contains either the same or fewer objects than the input. The reclustering of clusters at a timestamp validates the togetherness of the objects of the clusters at that timestamp.

Algorithm 2 *HWMT*

Input: Dataset $DB[H_i]_{|CC_i}$, Hop-window H_i, Candidate cluster set CC_i
Output: 1st order spanning convoys set V_{sp}
▷ Constructs a binary tree of the timestamps within H_i as shown in Figure 4
1: $V_{sp} \leftarrow CC$
2: **for** each level $l \in hwmt$(in ascending order) **do**
3: **for** each timestamp $t \in l$ **do**
4: $V'_{sp} \leftarrow \emptyset$
5: **for** each convoy $v \in V_{sp}$ **do**
6: $V'_{sp} \leftarrow V'_{sp} \cup DBSCAN(DB[t]_{|O(v)})$
7: **if** $V'_{sp} = \emptyset$ **then**
8: return V'_{sp}
9: $V_{sp} \leftarrow V'_{sp}$
10: **for** $v \in V_{sp}$ **do**
11: $t_s(v) \leftarrow t_s(H_i) - 1; t_e(v) \leftarrow t_e(H_i) + 1$ ▷ convoy starts and ends in the bordering benchmark points which are not included in the hop-window
12: return V_{sp}

Algorithm 2 shows the pseudo code of the HWMT algorithm. The algorithm operates on a hop-window and takes as input the dataset restricted to the hop-window and the candidate cluster set for that hop-window, CC_i and returns the set of convoys spanning the hop-window H_i. The algorithm first creates a hop-window mining tree for the hop-window. The algorithm processes each level of the tree starting from the highest one, i.e., the root of the tree (line 2). The candidate clusters in V_{sp} are reclustered at each timestamp in T_l to find new candidate clusters (line 6). If no clusters are found after the re-clustering process (line 7), no spanning convoys exist in the hop-window so we do not need to go any deeper in the tree and the algorithm terminates. Once the HWMT has been processed up to the leaves of the tree, V_{sp} contains the surviving clusters which contain objects which form a cluster in each timestamp of the hop-

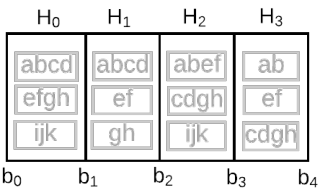


Figure 5: Spanning Convoys

Table 2: Steps of HWMT for example in Fig 6 to be seen in conjunction with Fig 4. (l,n) represents the timestamp number n of the level l of HWMT. t represents the actual timestamp from the dataset corresponding to (l, n). CC^1 is provided as input to level 1 of HWMT where it is clustered at the first timestamp of level 1 i.e. (1,1) which corresponds to timestamp 4 and is the root of HWMT. The result CC^2 is passed on to the level 2 where it is clustered at (2,1), the result of which is clustered at (2,2) producing CC^3. CC^3 is passed on to level 3 and the process is repeated until either the CC^i_j becomes empty or no timestamp in HWMT is left unprocessed. In both the cases, the last CC^i_j is the result of the HWMT algorithm.

(l,n)	t	Cluster Set	Value
	0	C_0	$cluster(DB[0]) = \{\{abcdefghij\}, \{xyx\}, \{mno\}\}$
	8	C_8	$cluster(DB[8]) = \{\{abcd\}, \{xyz\}\}$
		CC^1	$C_0 \cap^* C_8 = \{\{abcd\}, \{xyz\}\}$
(1,1)	4	CC^2	$reCluster(DB[4]_{\|CC^1}) = \{\{abcd\}\}$
(2,1)	2	CC^{3*}_1	$reCluster(DB[2]_{\|CC^2}) = \{\{abcd\}\}$
(2,2)	6	CC^3	$reCluster(DB[6]_{\|CC^{3*}_1}) = \{\{abcd\}\}$
(3,1)	1	CC^{4*}_1	$reCluster(DB[1]_{\|CC^3}) = \{\{abcd\}\}$
(3,2)	3	CC^{4*}_2	$reCluster(DB[3]_{\|CC^{4*}_1}) = \{\{abcd\}\}$
(3,3)	5	CC^{4*}_3	$reCluster(DB[1]_{\|CC^{4*}_2}) = \{\{abcd\}\}$
(3,4)	6	CC^4	$reCluster(DB[6]_{\|CC^{4*}_3}) = \{\{abcd\}\}$

window. The clusters are thus eligible to be considered as spanning candidate convoys. The start and end times of the clusters is updated to be the left and the right benchmark points bordering the hop-window which makes them represent spanning convoys of the current hop-window (line 11). Figure 6 shows an example of a hop-window for which the steps of the HWMT algorithm are described in Table 2.

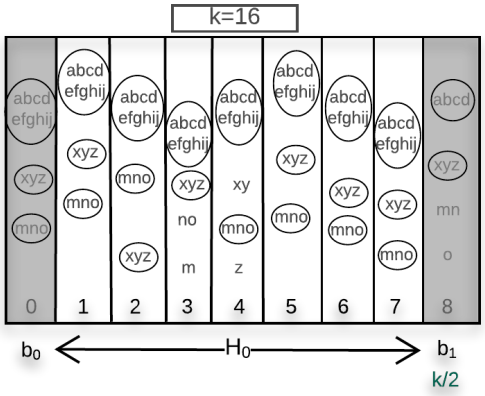


Figure 6: Hop-Window Mining Tree Example. X-axis represents timestamps. Grey timestamp blocks b_0 and b_1 are benchmark points. Each letter represents an object and their relative position shows their distance amongst each other. Encircled objects represents clusters of minimum size 3.

4.4 Finding Maximal Spanning Convoys

In this step, the adjacent 1st order spanning convoys are merged together recursively to find maximal spanning candidate convoys (which cannot be merged with adjacent spanning candidate convoys any further). A maximal spanning convoy can be defined as follows:

Table 3: Finding Maximal Spanning Convoys in Fig 5. The 1st merge column contains the result of merging the spanning convoys from hop-windows H_1 and H_2. The 2nd merge column contains the result of merging the convoys from 1st merge and the hop-window H_1 and so on. The convoys with grey background got extended during the merge process and hence are candidates of further merge with the next hop-window.

1st merge		2nd merge		3rd merge	
O	T	O	T	O	T
{a,b,c,d}	[b0,b2]	{a,b,c,d}	[b0,b2]	{a,b}	[b0,b4]
{e,f,g,h}	[b0,b1]	{a,b}	[b0,b3]	{c,d}	[b0,b4]
{e,f}	[b0,b2]	{c,d}	[b0,b3]	{e,f}	[b0,b4]
{g,h}	[b0,b2]	{e,f}	[b0,b3]	{g,h}	[b0,b4]
{i,j,k}	[b0,b1]	{g,h}	[b0,b3]	{c,d,g,h}	[b2,b4]
		{a,b,e,f}	[b2,b3]	{a,b,e,f}	[b2,b3]
		{c,d,g,h}	[b2,b3]	{c,d,g,h}	[b2,b3]
		{i,j,k}	[b2,b3]	{i,j,k}	[b2,b3]

Definition 9. ***(Maximal Spanning Convoy)*** Let $O(v)$ be the set of objects of a spanning convoy v and $T = [b_m, b_n]$ be its lifetime. v is a maximal spanning convoy if there does not exist a spanning convoy w such that v is a strict sub-convoy of w.

For merging 1st order spanning convoys to find maximal spanning convoys, we use the DCM_{merge} algorithm presented in [16]. Figure 5 shows the 1st order spanning convoys in 4 adjacent hop-windows. We use the DCM_{merge} algorithm to merge spanning convoys from the adjacent hop-windows from left to right. Table 3 shows the results of each phase of the merge process for the convoys in Figure 5. First we merge the 1st order spanning convoys of H_0 and H_1. The results are shown in the column for the 1st merge. The convoys marked with dark background end at the second benchmark point b_2, and can thus be merged further with the convoys in H_2. The convoys without any background colour are maximal as they cannot be merged further. The convoys with dark background are then merged with the spanning convoys of H_2 and the results are shown in the 2nd merge column. This process is repeated until the spanning convoys from the last hop-window are not processed, which in our example is H_3. The convoys with white background colour form the set of maximal spanning convoys as they cannot be merged further and are not strict sub-convoys of any spanning convoy.

4.5 Extending Maximal Spanning Convoys

Up to this point, we have discovered all the possible spanning convoys, however, according to Lemma 6, we still need to mine the hop-windows for the starting and ending candidate convoys which have the following properties:

LEMMA 7. *Given a convoy v with lifespan L. If $t_e(v) \in H_i$ then $b_{i-1}, b_i \in L$; i.e., if a convoy v ends in H_i then it spans H_{i-1}.*

PROOF. See APPENDIX[17]. □

LEMMA 8. *Given a convoy v with lifespan L. If $t_s(v) \in H_i$ then $b_{i+1}, b_{i+2} \in L$; i.e., If a convoy v starts in H_i then it spans H_{i+1}.*

PROOF. Symmetric case of lemma 7 □

From the above Lemmas 7 and 8, we know that to mine the ending (resp. starting) convoys in a hop-window, we need the (maximal) spanning candidate convoys from the previous (resp. next) window. In this phase, the maximal spanning convoys from the previous phase are extended within

the hop-windows to find their actual starts and ends. First, the convoys are extended to the right and then the resulting convoys are extended to the left. Algorithm 3 shows the pseudo code of the extension algorithm to the right. The algorithm takes as input the maximal spanning convoys set V_M from the previous phase and returns the *right-closed* maximal spanning convoys set V_{sp}^{mr}. A *right-closed* convoy is a convoy which cannot be extended to the right anymore. Each convoy is extended individually and may result in one or more convoys after extension. The convoy to be extended is put in the set V_{prev} which represents the convoys to be extended further to the right (line 2). In each iteration, a convoy $v \in V$ is re-clustered in the timestamp next to it's end time which is the current timestamp and the resulting convoy(s) are put in G_t (line 6). If G_t is empty, the convoy v can no more be extended. Hence, it is put in the result set V_{sp}^{mr} (line 8). If G_t is not empty, the start and end time of

Algorithm 3 *extendRight*

Input: Dataset DB, Maximal spanning convoys V_M
Output: Right extended spanning convoys V_{sp}^{mr}
1: **for** each convoy $v_{sp} \in V_M$ **do**
2: $V_{prev} \leftarrow \{v_{sp}\}$
3: **for** each time $t \in [t_e(v_{sp})+1, t_e(DB)]$ **do**
4: $V_{next} \leftarrow \emptyset$
5: **for** each $v \in V_{prev}$ **do**
6: $G_t \leftarrow$ reCluster$(v, DB[t])$
7: **if** $G_t = \emptyset$ **then**
8: $V_{sp}^{mr} \leftarrow$update$(V_{sp}^{mr}, \{v\})$
9: **else**
10: **for** each $c \in G_t$ **do**
11: $V_{next} \leftarrow V_{next} \cup \{c, [t_s(v), t]\}$
12: **if** $O(v) \notin G_t$ **then**
13: $V_{sp}^{mr} \leftarrow$update$(V_{sp}^{mr}, \{v\})$
14: $V_{prev} \leftarrow V_{next}$
15: **if** $V_{prev} = \emptyset$ **then**
16: **break**
17: update(V_{sp}^{mr}, V_{prev})
18: return V_{sp}^{mr}

each new convoy is set and the set is added to V_{next} (line 11) so that the convoys in the set could further be extended to the right in the next iteration. If the set G_t does not contain the original convoy v which was reclustered (line 12), two things could have occurred. Either the convoy v got split into two or more convoys or it got reduced in size. In both the cases, the convoy v is added to the result (line 13) as it did not get extended in its current shape. The *update* function makes sure that v is only added to the result if it is not a sub-convoy of existing convoy in the result set and all existing convoys that are sub-convoys of v are removed from the result set. The convoys in set V_{next} are put in the set V to be extended in the next timestamp (line 14). If the set V_{prev} is empty (line 15), we do not have any convoy to extend to the next timestamp, thus, the process is stopped. In the extension process, if the algorithm reaches the end time of the dataset, the convoys in V can be extended no further, thus, they are added to the result using the *update*() method (line 17).

While adding convoys to the result in the *extendRight* algorithm, we never check if the convoys satisfy the minimum length constraint of k. This is because, even if a convoy does not satisfy the k constraint now, it could potentially extend to the left and satisfy k. The *extendLeft* algorithm takes as input the right extended spanning convoys and extends them to the left. The algorithm works just like the *extendRight* algorithm with a few differences. First, it starts extending each convoy to the left, starting from the timestamp just before the starting time of a convoy and goes all the way to the start time of the dataset. Second, since, after the extension to the left, no more extension is possible, we expect the convoys to satisfy the k constraint. So, all the convoys which do not satisfy the k constraint, are discarded. The resultant convoys are passed on to the validation phase in which the maximal fully-connected convoys are mined.

4.6 Mining Fully Connected Convoys

After extending the spanning convoys to the left and right, we still need an additional validation step to ensure that the final convoys we output are fully connected. The reason is that whenever we reduce the set of objects in a convoy we are extending in the algorithm, we did not check if the reduced set of objects was fully connected in the timestamps that were already in the convoy. For instance, suppose we have a convoy $(abcde, [1,5])$ and at timestamp 6 the only cluster intersecting $abcde$ is $abcd$. Then we output $(abcde, [1,5])$ as it is maximal, and continue with $(abcd, [1,6])$. At this point, however, it is not guaranteed that $abcd$ is fully connected in the timestamps 1 to 5; maybe object e is needed to connect the object d to abc in, let's say timestamp 3. In that case $(abc, [1,6])$ may be the real fully-connected convoy while our algorithm outputs $(abcd, [1,6])$. Therefore we need an additional step to reduce the candidates output by our algorithm sofar to FC-convoys.

An important observation on which our validation is based, is that whenever (O,T) is an FC-convoy, then (O,T) is a convoy in the dataset we obtain by reducing the objects to O and the timespan to T, *and vice versa*. Indeed, suppose $(abc, [1,6])$ is an FC-convoy. Then, if we reduce our dataset to only objects abc and timestamps in $[1,6]$, any convoy mining algorithm will find $(abc, [1,6])$ as a convoy, *including our own HWMT-based algorithm*! Alternatively we could have used an algorithm called "*DCVal*" from [25] for this task, yet we decided to use a variant of the HWMT algorithm, called HWMT*, which we will specify later. The validation algorithm now proceeds as follows: for each candidate convoy (O,T) output by our algorithm sofar, we run a separate instantiation of the validation algorithm. The validation algorithm for (O,T) restricts the dataset and runs HWTM*. This results either in one convoy (O,T) which is then guaranteed to be an FC-convoy, or it outputs smaller convoys (O_1,T_1), ..., (O_k,T_k) for which we run the validation algorithm again. For instance, if $HWMT*$ run for convoy $(abcde, [1,5])$ returns $(abcd, [1,5])$ and $(de, [2,5])$, then these two convoys will be validated again. This continues until we either reach one or more FC convoys, or they become too small.

The complete procedure is depicted in Algorithm 4. The for loop in lines 2-8 iterates over all candidates v_{in} output by our algorithm sofar. v_{in} is removed from V_E and $HWMT*$ is run on it (lines 3 and 4). If this results in the same convoy, then it is added to the output set V_{FC}, otherwise all convoys in the result of $HWMT*$ are added to the set of convoys to be validated and the procedure continues (line 8).

HWMT* is based on HWMT, but has some subtle differences. We illustrate the differences with an example convoy $(O,T) = (abcde, [1,6])$ to be validated:

1. HWMT* will not consider objects except those in O.

2. HWMT* does not start clustering at the benchmark points b_i, but instead at the extremes of T. In our running example this means that we start by clustering the objects *abcde* in points 1 and 6. Then, using the same rationale as HWMT, the next point that is clustered is $\frac{1+6}{2}$ (3 or 4), etc.

3. If HWMT meets a point where for a certain timestamp no cluster are found, it stops. This is because HWMT works on a hop-window and finds spanning convoys only. HWMT* does not stop at such point, because it also needs to output convoys that do not completely span T. HWMT* only stops when no more convoys of length k or more can be found.

Algorithm 4 *validate*

Input: Extended convoys V_E, Convoy size parameter m, Minimum convoy length k
Output: Maximal fully-connected Convoys V_{FC}

```
1: V_FC ← ∅                                  ▷ initialize result set to empty
2: for each convoy v_in ∈ V_E do
3:     V_E ← V_E \ V_out
4:     V_out ← HWMT*(v_in)
5:     if V_out = {v_in} then
6:         V_FC ← V_FC ∪ V_out
7:     else
8:         V_E ← UPDATE(V_E, V_out)
9: return V_FC
```

Proof of Correctness & The Cost Complexity See full version [17] of the paper for the proof of correctness. k/2-hop algorithm's strength lies in its data pruning capability. Its cost increases with the number of convoys in the dataset as lesser data can be pruned. In the worst case, each object in the dataset would be part of at least one cluster at each timestamp, the maximum size of the cluster would be m, and each object would be part of at least one convoy at each timestamp of its existence in the dataset. In such a case, the whole dataset participates in convoys, and it is impossible for k/2-hop to prune any data. The cost complexity of k/2-hop in such a case becomes equal to the baseline PCCD/VCoDA algorithms. However, in the average case the data that doesn't participate in any convoy is pruned by k/2-hop. From our experiments, we have seen that up to 99% of the data is pruned by k/2-hop.

5. PERSISTENT STORAGE STRUCTURE

k/2-hop shows improved performance because of its meagre requirement for disk I/O. Therefore, use of an efficient storage structure, tailored for its access behaviour can greatly enhance k/2-hop's performance. k/2-hop requires to read all the data corresponding to the benchmark points but within a hop-window, it needs to read only the points corresponding to the objects which are part of the 1^{st} order candidate clusters. Thus, its disk access requirements can be categorized as follows: (1) For data corresponding to the benchmark points, the capability of doing fast scans on the benchmark points is needed and (2) for non-benchmark points data, fast random access indexed by object ids and timestamp is required. In addition to that, to support large amounts of data, it is desirable that the storage structure supports (3) fast data inserts, (4) distributed storage and (5) parallel querying with linear horizontal scaling. Moreover, (6) the storage format/data arrangement should be independent of the choice of the convoy mining parameters (m, k and eps). Otherwise for each convoy query with a different parameter, the data would have to be rearranged on disk. Also, (7) the structure should be efficient in terms of memory and disk space consumption.

For the purpose of experiments, we tested 3 storage structures, namely, flat files, relational and Log-Structured Merge-Tree (LSMT) [19]. Flat files are good for scans but are not suitable for random access, thus k/2-hop does not benefit from it. However, relational and LSMT based storage structures are more suitable for k/2-hop.

5.1 Relational Data Storage

A relational table could be used for data storage having each tuple of the form $(timestamp, oid, x, y)$ with a multi-column clustering index on *timestamp* and *oid*. To get benchmark points data, a SELECT query on the *timestamp* would fetch all the points corresponding to the benchmark point. For the HWMT algorithm, a SELECT query on the *timestamp* as well as the *oid* is required.

5.2 Log-Structured Merge-Tree

Log-Structured Merge-Tree [19] is a storage structure designed for high performance transaction applications and makes a hybrid use of memory and disk. It stores the freshly inserted data in the form of logs to minimize disk seeks and stores the data in the form of key-value pairs. The data is sorted by keys which allows faster range query performance on the keys because similar keys are co-located. Note that the data can be queried through the keys only. For achieving better range scan and random access results for k/2-hop, we create a composite key (t,oid) consisting of the timestamp and the *oid* of the object with the location coordinates (x,y) of the object stored as the value of the key-value pair. For fetching the data of benchmark points, we issue a query by timestamp which results in a range scan from (t,0) to (t,max(oid)). As all the data corresponding to a timestamp t is co-located, the data is fetched with a single seek, hence better I/O performance. For fetching the data for HWMT, a point query is issued for each (timestamp,oid) pair.

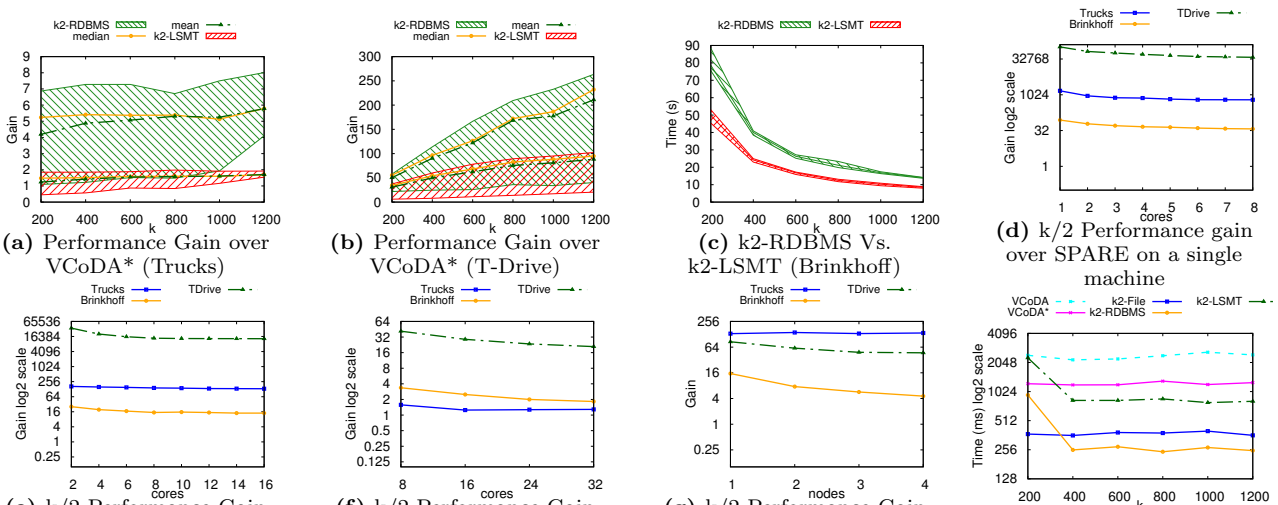
For huge datasets, LSMT is more suitable than a relational storage structure. It is horizontally scalable, allows fast inserts and does not need special indexing structures to be created or disk reorganization.

6. EXPERIMENTS

We compare k/2-hop algorithm with sequential as well as distributed algorithms. As k/2-hop is a sequential algorithm, it is fair to compare it with sequential algorithms running on a single machine, however, this puts the distributed algorithms at a disadvantage as they are designed to be run and take benefit of multiple cores and nodes. Keeping this in view, in addition to comparing the sequential and distributed algorithms with the k/2-hop algorithm on a single machine, we test the performance of the distributed algorithms (SPARE and DCM) on multi-core and multi-node setups and see how do they compare to the single machine performance of k/2-hop.

6.1 Setups

For the experiments, three different hardware setups were used, each for single instance, multi-core and multi-node experiments. Following are the details of each setup. Unless specified otherwise, all figures correspond to experimental setup A.



(a) Performance Gain over VCoDA* (Trucks)

(b) Performance Gain over VCoDA* (T-Drive)

(c) k2-RDBMS Vs. k2-LSMT (Brinkhoff)

(d) k/2 Performance gain over SPARE on a single machine

(e) k/2 Performance Gain over SPARE on YARN (Setup B)

(f) k/2 Performance Gain over SPARE on NUMA Machine (Setup C)

(g) k/2 Performance Gain over DCM on YARN (Setup B)

(h) Trucks: Effect of varying k on performance

Figure 7: k/2-hop Performance Experiments

6.1.1 Setup A

For experiments requiring a comparison with sequential algorithms, a machine with a quad core Intel(R) Core(TM) i7-4700MQ processor with 2.4GHz frequency was used. The machine had 16MB RAM and a Samsung MZ7TD256 SCSI Solid State Drive. It ran a 64-bit Linux Mint 17.3 Rosa distribution with Linux 3.19.0-32-generic (x86_64) kernel. All the algorithms were implemented in Java and run on Java(TM) SE Runtime Environment (build 1.8.0_101-b13) using the Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode) implementation. All the experiments were run with the maximum heap size set to 6GB.

6.1.2 Setup B

For measuring the performance of the SPARE framework running on Spark over YARN and of DCM running on YARN using the MapReduce framework, a cluster of 5 machines was used with one machine configured as the Resource Manager and 4 machines configured as Node Managers. Each machine had Intel(R) Xeon(R) E5520 processor running at 2.27GHz2. The processor contained 2 sockets, 4 cores per socket and 2 threads per core. The machines had 24GB of RAM. For the experiments, the same JVM as of setup A was used.

6.1.3 Setup C

For measuring the multi-core performance of the distributed algorithms (DCM and SPARE) in, we used a machine with 4 AMD Opteron(tm) 6376 processors running at 2.3 GHz. Each processor had 16 cores equally divided between 2 NUMA nodes. Thus the machine had 8 NUMA nodes, each node containing 8 cores making the cluster's total core count to be 32. We used spark in Standalone mode to reduce the overhead of YARN container allocation. The machine had 256 GB of physical memory. For the experiments, the same JVM as of setup A and B was used.

6.2 Data Sets

Following datasets were used for the experiments:

6.2.1 Trucks Dataset

The truck dataset consists of 276 trajectories of 50 trucks delivering concrete to several construction places around Athens metropolitan area in Greece. The locations in latitude and longitude were sampled approximately every 30 seconds for 33 days. To make the experiments compatible with the experiments performed on the trucks dataset in the previous papers [13, 14], a single day's of a truck's movement was considered as a trajectory of a truck. The next day's trajectory of the same truck was considered as a different truck's trajectory to increase the number of objects in the dataset and hence, to find more convoys.

6.2.2 T-Drive Taxi Dataset

[12] This dataset [26, 27] contains the GPS trajectories of 10,357 taxis during the period of Feb. 2 to Feb. 8, 2008 within Beijing. The total number of points in this dataset is about 15 million (29 million after interpolation) and the total distance of the trajectories reaches to 9 million kilometers. The average sampling interval is about 177 seconds with a distance of about 623 meters.

6.2.3 Brinkhoff Generator's Dataset

For testing k/2-hop's performance on synthetic datasets, we used the well-known Brinkhoff Generator [2, 3] which can generate network based traffic data based on a real-world dataset and user specified parameters using simulation. It is open-source and publicly available. Table 4 shows different properties of the generated Brinkhoff dataset.

6.3 Results

6.3.1 Performance Gain

Figures 7a and 7b show the performance gain of the k/2-hop algorithms over $VCoDA^*$ on the Trucks and T-Drive dataset, respectively. The top and bottom lines of an area

[1] `http://research.microsoft.com/apps/pubs/?id=152883`

[2] `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/User_guide_T-drive.pdf`

represent the maximum and the minimum gain of the algorithm for different values of *eps* and m where as the lines inside the marked areas represent mean and the median of different gain values. It can be seen that k2-RDBMS is up to 8 and 260 times faster than $VCoDA^*$ on Trucks and T-Drive dataset, respectively. As the Trucks dataset is quite small, the results of T-Drive dataset is more relevant. Moreover, the $VCoDA^*$ algorithm could not finish when run on the Brinkhoff dataset (Figure 7c), whereas the k/2-hop algorithms were able to finish without any problems. While k2-RDBMS performs the best on the Trucks and T-Drive dataset, k2-LSMT performs the best on the largest dataset, i.e., the Brinkhoff dataset.

Table 4: Brinkhoff Dataset Properties

Property	Value	Property	Value
MaxTime	25000	data space height	26915
ObjBegin	5000	number of nodes	6105
ObjTime	100	number of edges	7035
ExtObjBegin	100	maximum time	25000
ExtObjTime	2	moving objects	2505000
data space width	23572	points	122014762

6.3.2 Comparison with SPARE Framework

Figures 7d,7e and 7f show the performance gain of k/2 algorithm over the distributed SPARE framework[7] on a single machine, scale-out and scale-up scenarios respectively. As k/2 algorithm is a sequential algorithm and we used a non-threaded implementation for the experiments, k/2 was effectively running on a single core whereas SPARE was running in parallel on multiple cores. It can be seen that k/2 algorithm is up to 43000 times faster than SPARE running on a single core and up to 32000 times faster than SPARE running on 8 cores.

6.3.3 Comparison with DCM Algorithm

Figure 7g shows the performance gain of k/2 algorithm over the distributed convoy mining algorithm DCM[16] on a cluster with different number of nodes. With the increase in the number of nodes, the DCM runtime decreases leading to a decrease in the gain of k/2, however, k/2, even being a sequential algorithm, outperforms DCM running in parallel on multiple nodes and is up to 140 times faster.

6.3.4 Data Pruning Performance

Table 5 shows the minimum and the maximum number of points processed by k/2-hop for different convoy mining parameters. It can be seen that k/2-hop is able to prune more than 99% of the data in most cases.

6.3.5 Effect of k

Figures 7h, 8a and 8b show the effect of variation in k on performance of the convoy mining algorithms. It can be seen that the performance of VCoDA and VCoDA* does not vary much with the change in k. This is because both the algorithms touch all the data points for clustering purposes irrespective of the parameters. The execution time of the k2-* algorithms decreases with the increase in k. For the Trucks dataset, change in k does not reduce the execution time of the k2-* algorithms significantly for values of $k >= 600$ because there is not much time spent on the HWMT phase as no convoys exist for this range of k . VCoDA crashed while processing the Brinkhoff dataset, throwing an out of memory exception. Increase in k causes decrease in the expected

Table 5: k/2-hop: Data Pruning Performance

	Trucks	T-Drive	Brinkhoff
Total Number of Points	366202	29384000	122014762
Min Points Processed	571	49038	205331
Max Points Processed	57031	500691	1221697
Min Pruning	84.43%	98.3%	99%
Max Pruning	99.84%	99.83%	99.83%

number of convoys, thus, k2-* algorithms is able to prune more data and get better performance. The k2-LSMT and k2-RDBMS algorithms benefit from this effect more than the rest of the algorithms. This is because the k2-LSMT and k2-RDBMS algorithms are also able to reduce disk I/O costs in addition to the clustering costs. k2-RDBMS performs the best on the Trucks and T-Drive datasets, however, k2-LSMT performs best on the Brinkhoff algorithm (the largest in size). k2-File algorithm performs better than k2-LSMT on the Trucks dataset because it can easily load the whole dataset in memory and mine convoys.

6.3.6 Effect of m

Figures 8c, 8d and 8e show the effect of variation in m on performance of the convoy mining algorithms. It can be seen that the performance of k2-* algorithms increases with the increase in m. In Figure 8d, the execution time decreases when m is changed from 3 to 6 but for further increase, the decrease is not very significant. The reason for this is that for both values of m, i.e.,; 6 and 9, not many benchmark and candidate clusters were discovered, which allowed k2-* algorithms to save the cost of HWMT phases. We found the same behaviour on the Brinkhoff dataset however the behaviour is not very evident in Figure 8e because of the log scale. VCoDA and file based k2 algorithm crashed while processing the Brinkhoff dataset. k2-File performed the best on Trucks dataset because of its smaller size, k2-RDBMS performed the best on T-Drive dataset whereas k2-LSMT performed the best on the Brinkhoff dataset.

6.3.7 Effect of *eps*

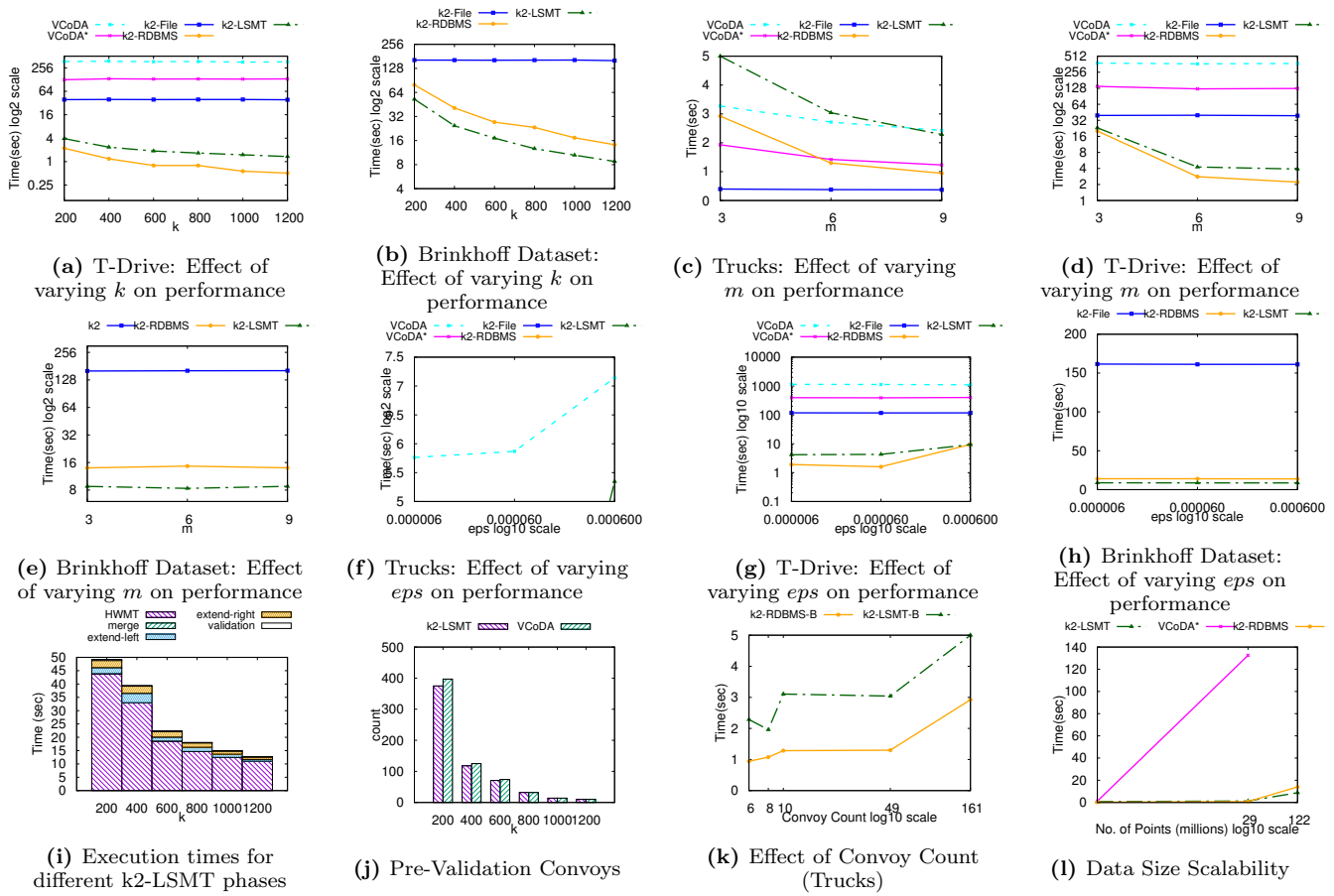
Figures 8f, 8g and 8h show the effect of the choice of values of *eps* on convoy mining algorithms. Increase in the value of *eps* causes detection of more and larger clusters which do not form convoys, hence, the object conversion ratio decreases. k2-LSMT performs better than other algorithms for all tested values of *eps*.

6.3.8 Execution Time of k2-LSMT Phases

Figure 8i shows the execution time of difference phases of the k/2 algorithms. It can be seen that most of the time is taken by the HWMT phase. In the HWMT, k/2-hop processes most of the timestamps and also involves point queries which are expensive. The extension (left and right) phases are the second most expensive parts of the k/2 algorithm. The rest of the phases take negligible time to execute.

6.3.9 Effect of No. of Pre-Validation Convoys

Figure 8j shows the number of pre-validation convoys detected by different convoy mining algorithms for different values of k. VCoDA algorithm first finds the partially-connected convoys and at the end runs a validation algorithm on them to get fully-connected convoys. In k/2-hop algorithm the size of the input convoy set to the validation algorithm is smaller than the partially-connected convoy set



(a) T-Drive: Effect of varying k on performance

(b) Brinkhoff Dataset: Effect of varying k on performance

(c) Trucks: Effect of varying m on performance

(d) T-Drive: Effect of varying m on performance

(e) Brinkhoff Dataset: Effect of varying m on performance

(f) Trucks: Effect of varying *eps* on performance

(g) T-Drive: Effect of varying *eps* on performance

(h) Brinkhoff Dataset: Effect of varying *eps* on performance

(i) Execution times for different k2-LSMT phases

(j) Pre-Validation Convoys

(k) Effect of Convoy Count (Trucks)

(l) Data Size Scalability

Figure 8: k/2-hop Performance Experiments

discovered by VCoDA because of the clustering process restricted to only the subset of the data thus saving time on the validation process. It turns out that the difference in the number of pre-validations convoys between VCoDA and k/2 algorithms is not very significant, thus the performance gain achieved by k/2-hop in terms of lesser validation time is not very significant. This effect can also be seen in Figure 8i which shows that the time spent on the validation process by VCoDA as well as k/2 algorithm is insignificant.

6.3.10 *Effect of Convoy Count*

Figure 8k shows the execution time of k/2-hop algorithms for the number of convoys found in the data set. It can be seen that generally the execution time increases with more convoys but this is not always true. It is possible to have no convoys in the dataset and a higher execution time because of lower object conversion ratio. The datasets in which many objects tend to be closer but not for long enough to form convoys, have lower object conversion ratio and hence higher execution time per convoy.

6.3.11 *Data Size Scalability*

Figure 8l shows the data size scalability of the convoy mining algorithms. The execution time of VCoDA* increases sharply with the increase in the data size but for the larger dataset, i.e.; Brinkhoff dataset, it crashes. The k/2 algorithms show sub-linear increase in the execution time and show huge performance gain (2 orders of magnitude) over VCoDA* algorithm which increases with the size of the dataset. This shows that k/2 algorithms can process very large data sets on a single machine.

7. CONCLUSION AND FUTURE WORK

Convoy is a rare pattern and mining it is computationally expensive. Existing algorithms do not scale up to the huge amounts of movement data. In this paper we propose the k/2-hop algorithm which is highly scalable and outperforms any existing sequential or parallel algorithms by orders of magnitude because of its smart sampling technique which mostly prunes more than 99% of the data. The performance of k/2-hop increases with the increase in k and m, and the decreases in the values of *eps*. k2-RDBMS performs the best in small to medium datasets, whereas k2-LSMT outperforms k2-RDBMS in large datasets. The k/2-hop technique can be applied to numerous movement pattern mining algorithms such as moving clusters [15] and flock patterns [9, 24, 22] to make them fast and efficient, and also to enable them to use operational data stores with negligible overhead.

In future, we would like to use k/2-hop to mine different movement patterns like moving clusters and flocks. We would also like to parallelize k/2-hop using distributed data processing platforms e.g Apache Spark and Apache Flink.

8. ACKNOWLEDGMENTS

This research was partially funded by "The Erasmus Mundus Joint Doctorate in Information Technologies for Business Intelligence - Doctoral College (IT4BI-DC)".

9. REFERENCES

[1] H. H. Aung and K.-L. Tan. Discovery of evolving convoys. In *Scientific and Statistical Database Management*, pages 196–213. Springer, 2010.

[2] T. Brinkhoff. Generating network-based moving objects. In *Scientific and Statistical Database Management, 2000. Proceedings. 12th International Conference on*, pages 253–255. IEEE, 2000.

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

[4] B.-R. Dai, I. Lin, et al. Efficient map/reduce-based dbscan algorithm with optimized data partition. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 59–66. IEEE, 2012.

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

[6] 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*, volume 96, pages 226–231, 1996.

[7] 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.

[8] S. Ghemawat and J. Dean. Mapreduce: simplified data processing on large clusters. In *Proc. OSDI*, 2004.

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

[10] J. Gudmundsson, M. van Kreveld, and B. Speckmann. Efficient detection of motion patterns in spatio-temporal data sets. In *Proceedings of the 12th annual ACM international workshop on Geographic information systems*, pages 250–257. ACM, 2004.

[11] P. N. Hai, P. Poncelet, and M. Teisseire. G e t_m ove: an efficient and unifying spatio-temporal pattern mining algorithm for moving objects. In *International Symposium on Intelligent Data Analysis*, pages 276–288. Springer, 2012.

[12] 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.

[13] H. Jeung, H. T. Shen, and X. Zhou. Convoy queries in spatio-temporal databases. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 1457–1459. IEEE, 2008.

[14] 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.

[15] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. In *Advances in spatial and temporal databases*, pages 364–381. Springer, 2005.

[16] F. Orakzai, T. Calders, and T. B. Pedersen. Distributed convoy pattern mining. In *Mobile Data Management (MDM), 2016 17th IEEE International Conference on*, volume 1, pages 122–131. IEEE, 2016.

[17] F. Orakzai, T. Calders, and T. B. Pedersen. k/2-hop: Fast mining of convoy patterns with effective pruning [full version]. *https://bit.ly/2VCMCT6*, 2019.

[18] F. Orakzai, T. Devogele, and T. Calders. Towards distributed convoy pattern mining. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '15, pages 50:1–50:4, New York, NY, USA, 2015. ACM.

[19] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[20] M. Patwary, D. Palsetia, A. Agrawal, W.-k. Liao, F. Manne, and A. Choudhary. A new scalable parallel dbscan algorithm using the disjoint-set data structure. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[21] N. Phan, P. Poncelet, and M. Teisseire. All in one: Mining multiple movement patterns. *International Journal of Information Technology & Decision Making*, 15(05):1115–1156, 2016.

[22] A. O. C. Romero. *Mining moving flock patterns in large spatio-temporal datasets using a frequent pattern mining approach*. PhD thesis, Master Thesis, University of Twente (March 2011), 2011.

[23] 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 *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 186–197. IEEE, 2012.

[24] M. R. Vieira, P. Bakalov, and V. J. Tsotras. On-line discovery of flock patterns in spatio-temporal data. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 286–295. ACM, 2009.

[25] H. Yoon and C. Shahabi. Accurate discovery of valid convoys from moving object trajectories. In *Data Mining Workshops, 2009. ICDMW'09. IEEE International Conference on*, pages 636–643. IEEE, 2009.

[26] J. Yuan, Y. Zheng, X. Xie, and G. Sun. Driving with knowledge from the physical world. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 316–324. ACM, 2011.

[27] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun, and Y. Huang. T-drive: driving directions based on taxi trajectories. In *Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems*, pages 99–108. ACM, 2010.