Pontifícia Universidade Católica
do Rio de Janeiro

Marcos Paulino Roriz Junior

DG2CEP: An On-line Algorithm for Real-time Detection of Spatial Clusters from Large Data Streams through Complex Event Processing

Tese de Doutorado

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática.

Advisor: Prof. Markus Endler

Rio de Janeiro
March 2017

Pontifícia Universidade Católica
do Rio de Janeiro

Marcos Paulino Roriz Junior

DG2CEP: An On-line Algorithm for Real-time Detection of Spatial Clusters from Large Data Streams through Complex Event Processing

Thesis presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Doutor em Ciências - Informática. Approved by the undersigned Examination Committee.

Prof. Markus Endler
Advisor
Departamento de Informática – PUC-Rio

Prof. Marco Antonio Casanova
Departamento de Informática – PUC-Rio

Prof. Hélio Côrtes Vieira Lopes
Departamento de Informática – PUC-Rio

Prof. Francisco José da Silva e Silva
Departamento de Informática – UFMA

Prof.ª Flávia Coimbra Delicato
Departamento de Ciência da Computação – UFRJ

Prof. Márcio da Silveira Carvalho
Vice Dean of Graduate Studies
Centro Técnico Científico – PUC-Rio

Rio de Janeiro, March 22nd, 2017

Marcos Paulino Roriz Junior

The author received his Bachelor degree in Computer Science from the Instituto de Informática (INF) of Universidade Federal de Goiás (UFG) in 2011. He also received his Master degree in Computer Science from INF – UFG in 2013. During his academic career, he participated in several research projects from private and public agencies, such as Microsoft Research, Dell, RNP, and FAPERJ. During his PhD at PUC-Rio he received the prestigious FAPERJ "Nota 10" scholarship. Currently, he is a professor at Instituto Federal de Goiás.

Bibliographic data

Roriz Junior, Marcos Paulino

DG2CEP: An On-line Algorithm for Real-time Detection of Spatial Clusters from Large Data Streams through Complex Event Processing / Marcos Paulino Roriz Junior; advisor: Markus Endler. – 2017.

v., 121 f: il. color. ; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2017.

Inclui bibliografia

1. Informática – Teses. 2. Aglomeração Espacial. 3. Aglomeração em Fluxo de Dados. 4. Aglomeração em Tempo Real. 5. Detecção On-line de Aglomerados. 6. Processamento de Fluxo de Dados. I. Endler, Markus. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

For my beloved grandmother Rosalha Maria Borges.

Acknowledments

First, I would like to thank my family: my grandmother Rosalha Borges, my mother Luciane Borges, my aunt Crisitiane Borges, and my brother Diogo Borges, for always being there for me and supporting me in life, especially in difficult moments such as this one where I had to absent from their presence to accomplish this work. Without them I would not be able to fulfill this work.

I would like to thank my adviser, Prof. Markus Endler, to whom I have an immense gratitude and admiration for his excellent guidance, availability, patience and for having given me the honor of working with him. His research vision has inspired and shaped me into becoming a better researcher. I'm sure our research partnership will continue after the end of my PhD.

I would also like to thank Prof. Francisco Silva. His critical view guided and shaped numerous parts of this thesis. I learned a lot from him. In the same way, I would also like to thank Prof. Marco Casanova and Prof. Hélio Lopes for encouraging me to work with this topic and providing numerous suggestions, especially the insight for the density heuristic.

I would like to thank my roommates during the PhD journey: Àlan Guedes, Eduardo Araújo, Derlyane Simão, Katia Vega, and Thais Abreu. I learned a lot living with them and they made the PhD journey with more joy and happiness. I will miss you all. In addition to my funny roommates, I would also like to thank all my friends (1801!) in Rio during this journey, especially: Aline Saettler, André Brandão, André Moreira, Andy Alvarez, Daniel Pires, Guilherme Lima, Hugo Gualandi, Lisseth Saavedra, Lívia Ruback, Luis Talavera, Paula Ceccon, Patrícia Carrion, Rodrigo Santos, and Roberto Azevedo. In addition, I would also like to thank my friends from Goiânia: Ivahy Santos, João Guilherme, and Marco Aurélio Lino Massarani.

I would also like to thank everyone from LAC, especially André Mac Dowell, Bruno Olivieri, Luis Talavera, Igor Vasconcelos, and Rafael Vasconcelos. I would like to deeply thank Prof. Luiz Fernando Soares (*in memoriam*) and all the Telemídia members. I am proud to be an honorary member of Telemídia.

I would like to thank all professors and staff from PUC-Rio that taught and helped me a lot during the PhD. More specifically, Regina Maria Zanon da Silva, for her patience and availability throughout the PhD. Thank you for helping me with all the numerous bureaucratic questions and procedures!

Finally, I would like to thank CNPQ, FAPERJ (even with the late payments!), and Microsoft Research for their financial aid.

Abstract

Roriz Junior, Marcos Paulino; Endler, Markus (Advisor). **DG2CEP: An On-line Algorithm for Real-time Detection of Spatial Clusters from Large Data Streams through Complex Event Processing**. Rio de Janeiro, 2017. 121p. PhD Thesis – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Spatial concentrations (or spatial clusters) of moving objects, such as vehicles and humans, is a mobility pattern that is relevant to many applications. A fast detection of this pattern and its evolution, e.g., if the cluster is shrinking or growing, is useful in numerous scenarios, such as detecting the formation of traffic jams or detecting a fast dispersion of people in a music concert. An on-line detection of this pattern is a challenging task because it requires algorithms that are capable of continuously and efficiently processing the high volume of position updates in a timely manner. Currently, the majority of approaches for spatial cluster detection operate in batch mode, where moving objects location updates are recorded during time periods of certain length and then batch-processed by an external routine, thus delaying the result of the cluster detection until the end of the time period. Further, they extensively use spatial data structures and operators, which can be troublesome to maintain or parallelize in on-line scenarios. To address these issues, in this thesis we propose DG2CEP, an algorithm that combines the well-known density-based clustering algorithm DBSCAN with the data stream processing paradigm Complex Event Processing (CEP) to achieve continuous and timely detection of spatial clusters. Our experiments with real world data streams indicate that DG2CEP is able to detect the formation and dispersion of clusters with small latency while having a higher similarity to DBSCAN than batch-based approaches.

Keywords

Spatial Clustering Stream Clustering Real-time Clustering On-line Clustering Complex Event Processing

Resumo

Roriz Junior, Marcos Paulino; Endler, Markus. **DG2CEP: Um Algoritmo On-line para Detecção em Tempo Real de Aglomerados Espaciais em Grandes Fluxos de Dados através de Processamento de Fluxo de Dados**. Rio de Janeiro, 2017. 121p. Tese de Doutorado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Clusters (ou concentrações) de objetos móveis, como veículos e seres humanos, é um padrão de mobilidade relevante para muitas aplicações. Uma detecção rápida deste padrão e de sua evolução, por exemplo, se o *cluster* está encolhendo ou crescendo, é útil em vários cenários, como detectar a formação de engarrafamentos ou detectar uma rápida dispersão de pessoas em um show de música. A detecção on-line deste padrão é uma tarefa desafiadora porque requer algoritmos que sejam capazes de processar de forma contínua e eficiente o alto volume de dados enviados pelos objetos móveis em tempo hábil. Atualmente, a maioria das abordagens para a detecção destes *clusters* operam em lote. As localizações dos objetos móveis são armazenadas durante um determinado período e depois processadas em lote por uma rotina externa, atrasando o resultado da detecção do cluster até o final do período ou do próximo lote. Além disso, essas abordagem utilizam extensivamente estruturas de dados e operadores espaciais, o que pode ser problemático em cenários de grande fluxos de dados. Com intuito de abordar estes problemas, propomos nesta tese o DG2CEP, um algoritmo que combina o conhecido algoritmo de aglomeração por densidade (DBSCAN) com o paradigma de processamento de fluxos de dados (Complex Event Processing) para a detecção contínua e rápida dos aglomerados. Nossos experimentos com dados reais indicam que o DG2CEP é capaz de detectar a formação e dispersão de clusters rapidamente, em menos de alguns segundos, para milhares de objetos móveis. Além disso, os resultados obtidos indicam que o DG2CEP possui maior similaridade com DBSCAN do que abordagens baseadas em lote.

Palavras-chave

Aglomeração Espacial Aglomeração em Fluxo de Dados Aglomeração em Tempo Real Detecção On-line de Aglomerados Processamento de Fluxo de Dados

Table of contents

List of figures

List of tables

List of codes

1
Introduction

This thesis investigates the possibility and limitations of an on-line[1] and *near* real-time (few seconds) detection of spatial clusters from large position data streams generated by moving objects (*e.g.*, humans, vehicles, drones). Spatial clusters [2] are concentrations of moving objects in some region, for example, a massive street protest, a music concert, a traffic jam, *etc.* A fast detection of this pattern and its evolution, *e.g.*, if the cluster is shrinking or growing, is useful in numerous scenarios, such as detecting the formation of traffic jams or detecting a fast dispersion of people in a music concert [3, 4, 5].

On-line detection of such clusters and its evolution is a complex task because it requires algorithms that are capable of continuously and efficiently processing the high volume of position data in a timely manner [3, 6]. Currently, the majority of approaches for cluster detection in position data streams use spatial data structures [7, 8, 9], which can be troublesome to maintain for on-line detection. Further, they operate in batch [10, 11], where position updates are stored during time periods of certain length and then batch processed by an external routine, thus delaying the result of the cluster detection until the end of the time period.

To address these issues, in this thesis, we investigate if it is possible to perform an on-line detection of spatial cluster from large position data streams and monitor its evolution in *near* real-time. Furthermore, we investigate if this process is scalable, that is, if on-line and *near* real-time results can be obtained as the data stream volume increases. As a response to these questions, we propose DG2CEP, a grid-based algorithm that combines the well-known density-based clustering algorithm DBSCAN [12] with the data stream processing paradigm Complex Event Processing (CEP) [13, 14] to achieve scalable, on-line, and *near* real-time detection of spatial clusters.

The remainder of the chapter provides an overview of the thesis motivation, followed by the main challenges of current data stream clustering algorithms. Furthermore, it presents the main research questions, the goals and the thesis contributions. Finally, we present how the thesis chapters are organized.

[1] On-line algorithms are those that can compute without having the complete input, *i.e.*, it can start processing and receive the remaining input piece-by-piece as it compute. Contrary to that, off-line algorithms are those that assume and require a complete input to process [1].

1.1 Motivation

Advances in mobile computing enabled the popularization of portable devices, such as smartphones and tablets, with internet connectivity and location sensing. Collectively, these devices can produce large position data streams while using mobile location-based applications and services [15, 16], such as location-based games (*e.g.*, Pokémon GO [17]), exploration services (*e.g.*, Foursquare [18]), dating applications (*e.g.*, Tinder [19]), public transport services (*e.g.*, data.rio[2] [20]), and intelligent transportation services (*e.g.*, Waze [21]).

An interesting question is how large and how fast the position data stream produced by the portable devices while using such application systems can become. Precisely, what is the throughput, *i.e.*, number of data items per second, of the data stream? There is no consensus for this answer in the literature [22]. Thus, in this thesis, we consider a large position data stream as one that produces thousands of data items per second. This definition is based on the throughput of real-world citywide application and services data streams [23, 24, 25, 26]. For example, thousands of buses in Rio de Janeiro contain a portable device that continuously send their position updates to the data.rio service.

Location-based applications and services can explore their user's data stream to identify mobility patterns, *i.e.*, if the moving objects (*e.g.*, vehicles, humans, drones) are located in some region or moving in a given pattern. For instance, when numerous buses are moving slowly on a given road, this may indicate that there might be an accident or a traffic jam in that place. Applications can also benefit from early detection of such situations, *i.e.*, by receiving timely notifications when several nearby moving objects present a specific mobility pattern, to enable the application to act as soon as possible [6]. For instance, it may be important to rapidly detect that many buses are moving slowly to enable and plan a timely reaction as soon as possible, e.g., change route, or open additional traffic lanes.

A mobility pattern that is particularly relevant to many applications is a spatial cluster [2, 27, 28], a concentration of moving objects in some area, *e.g.*, a massive street protest, a music concert, or a traffic jam. Similarly, a rapid (*near* real-time) and on-line (continuous) detection of spatial clusters from position data streams is desirable in numerous applications and scenarios [3, 15], such as for optimizing traffic flows and ensuring users' safety.

[2]data.rio is an open data service that continuously provides public urban data from the city of Rio de Janeiro, such as its buses location and car accident statistics.

For metropolitan traffic control, for example, it is useful and important to detect in *near* real-time traffic accidents, *e.g.*, represented by a quick formation of a cluster of vehicles, so to remove the corresponding obstruction as fast as possible. On the other hand, it may also be important to early detect the dispersion of a cluster, *e.g.*, a crowd rushing away from some specific spot in a live or disaster scenario [29]. This information can be useful for dispatching additional rescue personal to the place of the disaster emergency. Finally, in various situations it is important to detect and monitor the evolution of such clusters, that is, how a cluster changes over time. More specifically, if a cluster is growing/shrinking or if it is merging/splitting to/from other clusters. For example, in a traffic control application, it is important to known if a traffic jam is growing or shrinking over time. In a security and surveillance application, *e.g.*, in a carnival music concert, it is important to rapidly detect the location and period of a cluster dispersion or split.

1.2 Problem Setting

As can be seen from these examples, an on-line detection of spatial clusters and monitoring its evolution in *near* real-time is a recurrent mobility pattern of interest in many applications. However, implementing timely spatial cluster detection from large position data streams poses several challenges to those applications [10, 30]. First, it has to employ efficient algorithms and data structures to cope with the high arrival rate of the position data (location updates) stream and intrinsic complexity of mutually comparing the location updates of all moving objects. Second, it must be able to detect arbitrary clusters shapes, for example, a traffic jam that reaches over several neighborhoods or a human crowd that spans across the seashore of a city. Third, it has to provide timely results that reflect the current clustering scenario to enable a fast reaction by managing applications. Finally, it must be able to track the evolution of clusters, for example, providing a continuous view on how clusters are growing, shrinking, merging, or splitting.

The majority of data stream clustering algorithms found in literature [31, 32, 33, 34] do not address, or solve only partially, these problems. For example, they use spatial data structures, such as R-Trees and Quad-Trees, which provide efficient indexes and query functions for spatial data. However, for continuous-mode cluster detection in data streams, these data structures can become troublesome due to the difficulty in accessing and modifying the spatial tree in parallel [7, 8, 9], *i.e.*, to process several location updates at once. Further, due to frequent spatial tree modifications, there is an additional cost of

frequently balancing the tree in order to reduce its height. For example, to avoid inconsistencies in the spatial tree index, these algorithms process each location update sequentially, which can lead to scalability issues when considering a data stream with thousands of data items per second.

Another major issue with existing spatial data stream clustering algorithms is that they operate in an on/off-line batch framework [10, 11], where position data is first accumulated during a given time period (on-line phase) and then are processed in batch by a specific cluster detection function (off-line phase). The main problem with this approach is that this function only processes the data items within each batch separately and defers the cluster detection process until the end of the off-line phase. Thus, since it delivers results only at discrete points of time it is complicated to provide fresh results and a continuous view of the clusters' evolution.

Furthermore, if the batch interval is large enough to include more than one consecutive instances of position data from a same moving object, then it may happen that spatial clusters which have been formed and immediately dispersed by follow-up location updates, may not be detected. On the other hand, if the batch interval is so small that it holds at most one location update per moving object, it can happen that temporally and spatially close location updates end up in different batches, possibly preventing a cluster to be detected.

1.3 Research Questions

Motivated by the problem setting and the limitations of current spatial data stream clustering algorithms, this thesis explores means of achieving *on-line (continuously) and rapidly (near real-time) detection of spatial clusters from large position data streams*. This problem has the following three sub questions:

1. *How similar is the on-line and near real-time clustering result to the ground-truth result, i.e.*, the one obtained using the traditional DBSCAN's [12] off-line algorithm? Precisely, how accurate, *w.r.t.* the moving objects, are the spatial clusters discovered in on-line/real-time to the ones obtained through DBSCAN.

2. *How scalable is this approach w.r.t the data stream volume?* More specific, we want to answer if is it possible to provide or maintain the result quality (similarity with ground-truth) when increasing the throughput of the data stream. In essence, we want to discover and understand the scalability limits of an on-line and near real-time spatial clustering approach.

3. Finally, *can this approach continuously monitor, in near real-time, the spatial cluster's evolution*? Specifically, is it able to react and detect in less than few seconds if and how the spatial cluster is changing, *e.g.*, if it is growing, shrinking, merging or splitting with/from another spatial cluster.

1.4 Goals and Proposed Approach

The goal of this thesis is to investigate the main and its sub research questions presented in the previous section. To reach this goal, this thesis proposes DG2CEP (Density-Grid Clustering using Complex Event Processing), a grid-based (counting) algorithm that combines the traditional density-based clustering algorithm DBSCAN [12] with Complex Event Processing (CEP) [13, 14] to achieve a scalable, on-line, and *near* real-time detection of spatial clusters and its evolution.

One of the main ideas in DG2CEP is to change the problem semantic from distance computations (between the moving objects location) to counting. To do this, we subdivide the spatial domain into a grid, an efficient index data structure for spatial data. Then, rather than measuring the distance between each pair of moving objects, we count the number of objects mapped to each cell. Cells that contain more than a given threshold of moving objects are further analyzed. This process triggers an expansion step that recursively merges a dense cell with its adjacent neighbor. Since cells are aligned in a grid, the expansion step is straightforward. With this method, the main performance bottleneck is no longer the distance comparison between moving objects, but the number of grid cells.

This entire approach, grid and cell discovery and transformations, is described using the CEP data stream processing paradigm. CEP provides a set of real-time data stream analytics and pattern primitives [35, 36] through continuous queries, such as *filter*, *join*, *enrich*, *negation*, and *sequence*. Each continuous query is performed by a CEP processing stage known as Event Processing Agent (EPA). An EPA continuously receives incoming events, analyzes, manipulates them, and outputs derived (complex) events that are further delivered to event consumers, *e.g.*, other EPAs processing stages or endpoint users' applications. By interconnecting each processing stage (EPA), DG2CEP builds an Event Processing Network (EPN) that works together to continuously and timely detect the spatial cluster and monitor their evolution from analyzing the incoming moving objects' position data event stream.

1.5 Contributions

The main contributions of this thesis are:

- A novel on-line counting algorithm based on grid-density clustering, designed as a network of CEP continuous query and pattern primitives, that is able to continuously and timely detect (*near* real-time) spatial clusters and its evolution from large position data streams.
- A counting heuristic that mitigates the collateral effects of the answer loss (blind spot) problem [37, 38] that appears due to the usage of a grid data structure to index and cluster spatial data.
- A scalable event processing network architecture that can process data in parallel and be distributed to process higher data stream throughputs.
- An extensive discussion about the experimental results and tradeoffs of using an on-line and real-time spatial clustering approach with real-world position data streams.

1.6 Organization

This thesis is organized as follows. In the next chapter, Chapter 2, we further elaborate the thesis problem and present the two fundamental topics used in the thesis to address it: Spatial Clustering algorithms and the Complex Event Processing data stream processing paradigm. After that, Chapter 3 presents and discusses the main related-works and the employed techniques to address the aforementioned problems.

Chapter 4 presents the proposed algorithm, DG2CEP. In that chapter, we discuss the algorithm steps and how they can be translated to CEP's continuous query and real-time primitives. In Chapter 5, we present a counting heuristic that mitigate some of the collateral effects of transforming the problem from distance comparison to counting.

Chapter 6 presents the evaluation experiment used to validate the proposed algorithm. The experiment is based on the real-world position data stream generated by the bus fleet of the city of Rio de Janeiro. We discuss the experiment results and show how they respond the raised research questions. Finally, Chapter 7 presents the concluding remarks, and the limitations of our approach. It also points to future works that can address or explore these issues.

2
Fundamental Concepts

To investigate if it is possible to on-line detect spatial clusters and monitor its evolution in *near* real-time, this thesis explores two main topics: Spatial Clustering and Complex Event Processing. This thesis argues that the combination of concepts from these two topics can enable the construction of an algorithm (DG2CEP) that address this problem. Aiming at providing the basis for the next chapters' discussions, this chapter briefly reviews those topics

In Section 2.1, we present and discuss spatial clustering algorithms and how they can be applied to position data streams [39, 40]. First, we discuss the different ways that algorithms defines a cluster and how they detect them. Particularly, we focus the discussion on the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm [12] due to its ability to discover clusters with arbitrary shapes, *e.g.*, a traffic jam that ranges over several streets. In essence, DBSCAN's density-based cluster is a dense concentration of points, in our case, moving objects location updates, that are within a given distance. Following that, we discuss a data streaming framework [11] that can be used to cluster position data streams. This framework operates on batch and is divided in an on-line and an off-line phase. In the on-line phase, the data stream is summarized. Then, the off-line phase executes from time to time and batch processes the summarized data using an off-line clustering algorithm. We discuss the key problems with this approach when considering on-line and real-time scenarios.

Section 2.2 describes Complex Event Processing (CEP) [13, 14], a data stream processing paradigm. The basic unit in CEP is an event, an entity that represents something that happened [41], *e.g.*, a location update of a moving object. CEP provides real-time stream primitives, such as *filter* and *enrich*, that are able to react, process, and output complex (derived) events based on the incoming event stream. Other primitives can further consume the output events. Using these primitives, it is possible to build a network of real-time primitives to continuously process the event stream.

Finally, the last section of this chapter summarize the main points of these two concepts and discuss how they can be used to define the DG2CEP algorithm.

2.1 Spatial Clustering

Clustering is the process of grouping objects into one or more sets (cluster), such that objects in the same set are highly similar to each other than to those objects located in other sets [39, 42]. For example, group students in a university that are spatially close to each or to a given building location. Another example of clustering, in a medical scenario, is to group concentration of stains that appears in a medical exam.

Spatial clustering is the process of clustering a specific type of object: spatial data [39]. Spatial data objects are have geometric coordinates in a given referential space referential, such as GPS position data objects (latitude, longitude) that refers to the earth's surface, or a 2D medical image that refers to a spatial model of a human brain. In this thesis, we are interested in on-line clustering the geographic position data stream produced by moving objects, *e.g.*, vehicles, humans, and drones, in *near* real-time. Note that the term moving object is interchangeably used to refer to the entity current spatial location.

The primary similarity factors used to group position data into spatial clusters are the density and distance between the moving objects. Roughly speaking, a spatial cluster is a dense concentration of connected spatial data in some area [2, 27], *e.g.*, a massive street protest, a music concert event, a traffic jam, *etc.* To exemplify this concept, consider the urban scenario of spatial position data produced by moving objects (*e.g.*, vehicles) in Figure 2.1. This figure shows an urban scenario with three moving object clusters. Each cluster contains at least five moving objects, which need to be close and connected to one another. Note that moving objects that are not close to other objects and are isolated are considered noise.

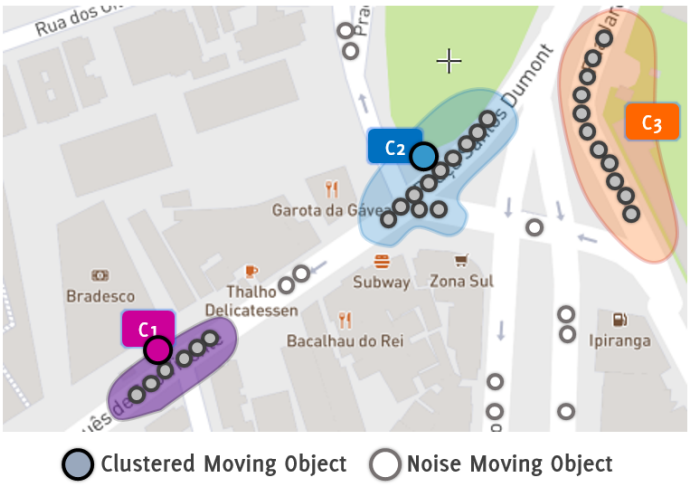


Figure 2.1: Spatial clustering of moving objects in an urban scenario.

There is an extensive set of clustering algorithms that can be used to cluster position data [3]. Many of them are variants of the k-Means [40] algorithm, which divides the entire position dataset into k clusters. However, since in many cases it is impossible to know the number k of clusters in advance, we need to use a different approach. Furthermore, k-Means-based algorithms require several passes over the data to converge and discover the clusters.

On the other hand, the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [12] cluster definition uses density thresholds to discover the clusters of position data rather than a fixed set of cluster k defined a priori. We decided to follow this definition for two reasons. First, by using a density approach, it is able to discover arbitrary cluster shapes, *e.g.*, a cluster represented by a complex polygon such as a traffic jam that spans several different streets. Second, DBSCAN only requires a single pass over the position data to converge, *i.e.*, to identify the clusters.

Figure 2.2, adapted from [43], illustrates the main difference between k-Means and DBSCAN clustering results. DBSCAN results use the density of position data as the cluster-forming criteria, while k-Means use the parameter k to split the set of position data into k clusters. Note that DBSCAN clustering results shape follows the position data shape, which is useful in numerous scenarios, for example, to discover a cluster of persons waiting in line for a music concert [44]. Due to its highest accuracy, DBSCAN is considered a benchmark of clustering algorithms. In the following subsection, we present and discuss the DBSCAN algorithm.

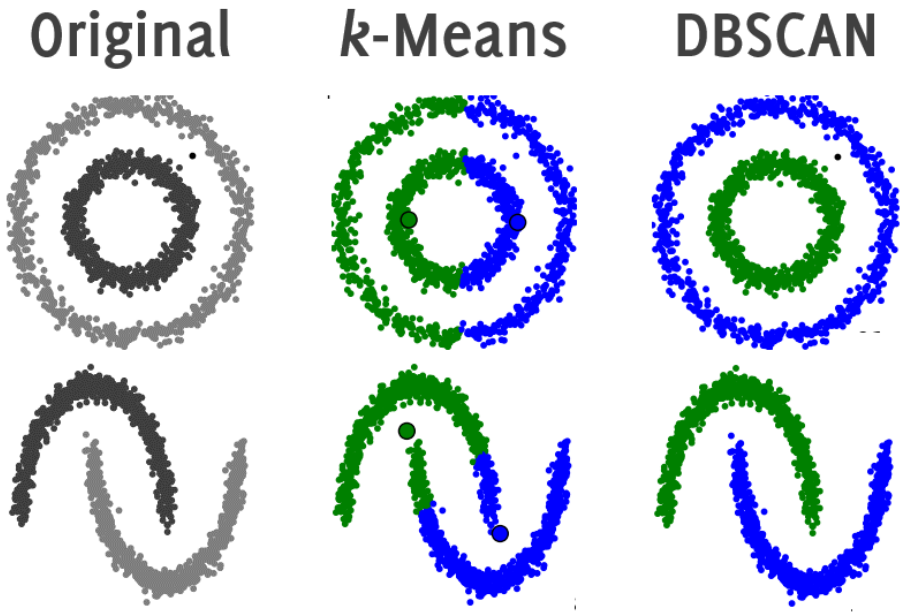


Figure 2.2: Overall difference between k-Means and DBSCAN clustering results.

2.1.1 Density-Based Spatial Clustering of Applications with Noise

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a clustering algorithm [12] based on the concept of density, *i.e.*, it does not requires the user to specify *a priori* the number of cluster to be found. Instead, the algorithm searches for concentrations of spatial data points, in our case, moving objects current location updates, to detect clusters. To do that, the algorithm uses two density parameters, an ε radius and the minimum density $minPts$ of spatial points, to specify the density-based cluster definition.

A moving object p that has more than $minPts$ other moving objects in its ε–$Neighborhood$ is known as a *core* moving object, where the ε–$Neighborhood$ of p is the set $N_\varepsilon(p) = \{q \in \mathcal{D} \mid distance(p, q) \leq \varepsilon\}$ and $\mathcal{D}$ is the set of all current moving objects location updates. Moving objects in the $N_\varepsilon(p)$ set of a core moving object p are said to be *directly density reachable* from p [12, 45]. Neighboring moving objects, in an ε of a core object p, but whose core object, but whose density is less than $minPts$ are classified as *border* objects.

Intuitively, a spatial cluster is composed of a several core moving objects that are close to one another. Hence, to connect core and border moving objects, DBSCAN extends the *direct reachability* concept. DBSCAN defines that a moving object q is *density reachable* from a moving object p if there is a chain of moving objects $o_1, o_2, \ldots, o_n$, such that $o_1 = p$ and $o_n = q$, where each o_{i+1} is *directly* density reachable from o_i, with the possible exception of $o_n(q)$ that can be a border object.

To illustrate these definitions consider the cluster scenario in Figure 2.3, with ε being the circle radius and $minPts = 3$. Moving objects C, D, and F are all directly density reachable from the core moving object E, since they are in its ε–$Neighborhood$. Further, the border moving object A is density reachable from E, since there is chain ($E \rightarrow C \dashrightarrow B \dashrightarrow A$) of core moving objects that connect them. However, this relationship is not symmetric. Although A is density reachable from E, the contrary is not valid since A is not a core moving object.

To include border moving objects in the resulting cluster elements, DBSCAN introduces the density connectivity relation. A moving object p is *density connected* to a moving object q if there is a moving object o such that both, p and q, are density-reachable from o. For example, considering the scenario in Figure 2.3, the moving object A is density connected to E because they both are density reachable from B. This relation is symmetric since it uses a moving object o to reach the input moving objects p and q.

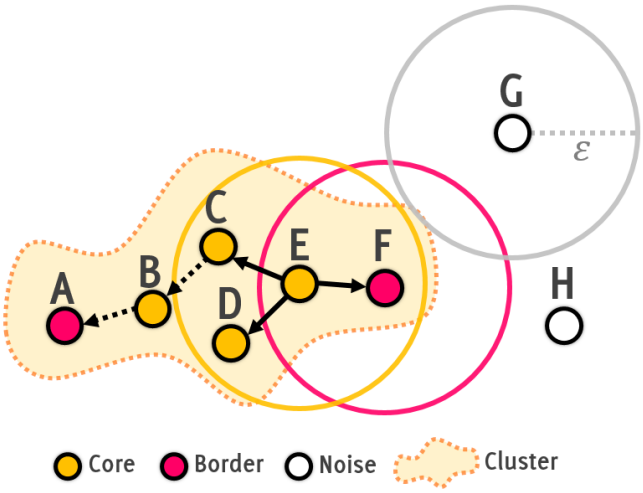


Figure 2.3: Example of core (E), border (F), and noise (G) moving objects.

Based on these concepts, DBSCAN gives the definition of density-based cluster [12]. A spatial cluster is defined as following. Let $\mathcal{D}$ be the snapshot of current moving objects' location. A spatial cluster $\mathcal{C}$ is, *w.r.t.* ε and $minPts$ is a non-empty subset of $\mathcal{D}$ that satisfy the following conditions:

1. $\forall p, q \in \mathcal{D}$: if $p \in \mathcal{C}$ and q is density-reachable from p, then $q \in \mathcal{C}$;
2. $\forall p, q \in \mathcal{C}$: p is density-connected to q.

This definition expresses the connectivity and reachability of moving objects included in a cluster. In essence, a density-based spatial cluster is a reachable chain of one or more core moving objects delimited by border objects.

The first step to detect a new cluster is to identify a core moving object p. From that, the main idea of the DBSCAN algorithm, shown in Algorithm 1, is to recursively visit each moving object q in the ε–$Neighborhood$ of p, in order to check if q is also a core moving object, *i.e.*, if it contains at least $minPts$ neighbors in its neighborhood. If it does, q neighbors' ($N_\varepsilon(q)$) are added to the cluster and further visited in the next round. By such, the cluster is recursively expanded until no further moving objects are added to the cluster, that is, all objects checked in the recursion step have less than $minPts$ neighbors or have been previously visited.

When applied to large numbers of moving objects, the main bottleneck of DBSCAN becomes the GetNeighbors function [39], which computes the $N_\varepsilon(p)$ set, *i.e.*, the ε–$Neighborhood$ of a given moving object p. Specifically, DBSCAN needs to compare the pairwise distance between the moving object p and all the remaining moving objects in order to select those that are within ε distance. This pairwise evaluation of inter-objects distances makes the computational complexity of DBSCAN be $\mathcal{O}(n^2)$.

Algorithm 1: DBSCAN

Input: A dataset of moving objects $\mathcal{D}$, the ε distance threshold, and the minimum number of points $minPts$

Output: A set of clusters K

```
1  K ← ∅
2  foreach moving object p ∈ D do
3      if p has not been visited then
4          mark p as visited
5          N_ε(p) ← GetNeighbors(p)
6          if |N_ε(p)| < minPts then
7              mark p as noise
8          else p is a core moving object
9              C ← ExpandCluster(p, N_ε(p))
10             K ← K ∪ {C}
11         end
12     end
13     return K
14 end
15 procedure ExpandCluster(p, N_ε(p)):
16     C ← {p}          ▷ Initiate a new cluster with the core moving object
17     N ← N_ε(p)                                   ▷ Neighbors to visit
18     foreach neighbor q ∈ N do
19         N ← N \ {q}
20         if q has not been visited then
21             mark q as visited
22             N_ε(q) ← GetNeighbors(q)
23             if |N_ε(q)| ≥ minPts then q is a core moving object
24                 N ← N ∪ N_ε(q)          ▷ Add q's neighbors to be visited
25             end
26         end
27         if q is not part of any cluster then
28             C ← C ∪ {q}
29         end
30     end
31     return C
32 end
```

Since DBSCAN has been designed for processing of complete - and static - datasets, one can optimize it by storing the moving objects position data in a spatial data structures (*e.g.*, R-Tree or Quad-Tree). These data structures provides efficient indexes and spatial query functions, which can optimize the computation of moving objects' ε–$Neighborhood$ significantly. More specifically, this optimization reduce the computation of the GetNeighbors function to $\mathcal{O}(\lg n)$ per moving object, and hence the overall complexity of DBSCAN to $\mathcal{O}(n \times \lg n)$.

However, for on-line position data stream clustering scenarios, the usage of these data structures can become troublesome. It is difficult to access and modify the spatial tree in parallel when handling thousands of position data at once [7, 8, 9]. Further, due to frequent spatial tree modifications, there is an additional cost of frequently balancing the tree height. For example, to avoid inconsistencies in the spatial tree index, these algorithms process each location update sequentially, which can lead to scalability issues when considering a data stream with thousands of position data items per second. With that said, the next subsection presents the main issues and approaches of applying or adapting DBSCAN to a data stream scenario.

2.1.2 Clustering Large Position Data Streams

Applying DBSCAN to discover spatial clusters in large position data streams is not a straightforward task, primarily because moving objects location update arrive continuously and the data stream size is unbound [10]. Further, due to its unbound nature, the data stream throughput can evolve and dynamically increase or decrease along time. Indexing and processing such data streams in near real-time thus require efficient data structures and methods [3]. As previously mentioned – and mainly due to the difficulty of accessing and modifying data items in parallel [7, 8, 9] – using spatial tree data structures (*e.g.*, R-Tree and Quad-Tree) is also costly. Aside from providing on-line and near real-time spatial clustering results, another issue is the ability to rapidly adapt and detect changes in the data streams. For example, detect that a new spatial cluster appeared or that it is, in fact, the result of the split of an existing cluster.

The main method for clustering large position data streams is the on/off-line framework [10], proposed by Aggarwal *et al.* [11]. This two phases framework operates on batch, as illustrated in Figure 2.4. The overall idea is to use batching to provide results at discrete and specific periods, rather than continue (on-line) and real-time results. The on-line phase, also known as abstraction step, summarizes the data stream in a given data structure, *e.g.*, buffer, sampling, grid, *etc.* During this step, the incoming location update stream is continuously inserted into the summary data structure. For example, keep up only the latest moving objects location update or discard moving objects that are too far from the remaining ones.

From time to time, the framework switch from the on-line phase to the off-line. In the off-line phase, it receives as input the buffered and summarized spatial data from the on-line phase. Then, it executes a classic off-line

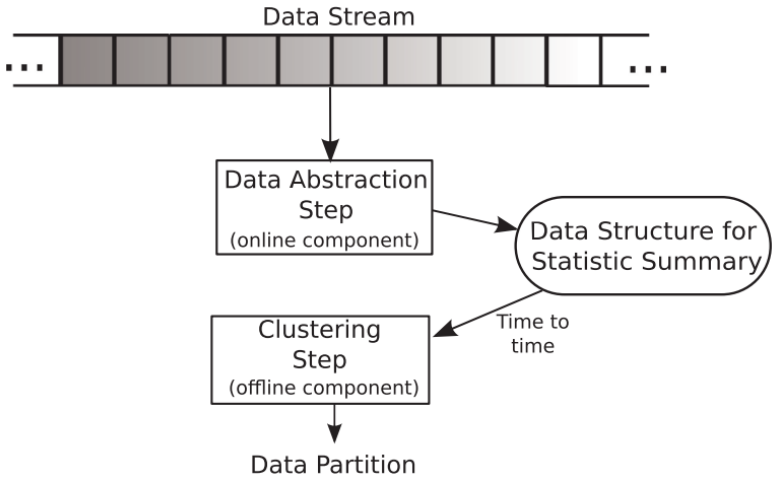


Figure 2.4: Data Stream Clustering Framework.

clustering algorithm, *e.g.*, k-Means or DBSCAN. Since the input data has been summarized, the algorithm can timely execute and detect the spatial clusters. Following this model, the framework is able to handle the data stream and provide a discrete view of the spatial clusters.

One of the main issue of the above two-step approach is that it is not able to on-line detect or monitor the spatial clusters evolution in real-time, since the clustering step happens at specific periods. Hence, the detection of spatial clusters that happened in the on-line phase is delayed until the end of the next off-line phase. By providing results only at discrete time periods it is complicated to provide fresh results or a continuous view of the cluster's evolution. For time critical scenarios, such as the detection of a traffic accident or a rapid dispersion of a crowd in a public event, such delay can be a problem since the earlier a situation is detected the fastest it is addressed.

The main problem with this framework is that it only processes the data items within each batch separately and defers the cluster detection process until the end of the off-line phase. Thus, since it delivers results only at discrete points in time it is complicated to provide fresh results or provide a continuous view of the clusters' evolution.

One way to address this issue is to decrease the batch period. However, by reducing the batch interval it can happen that temporally and spatially close moving objects are placed in different batches. Since the off-line phase only considers data items that are within the summarized batch, the algorithm may fail to detect a spatial cluster even though there are moving object that are spatially and timely close. To exemplify this, consider the illustration in Figure 2.5. Although the moving objects identified by 35 and 49 are spatially and temporally close to the ones identified by 2 and 15, the off-line clustering

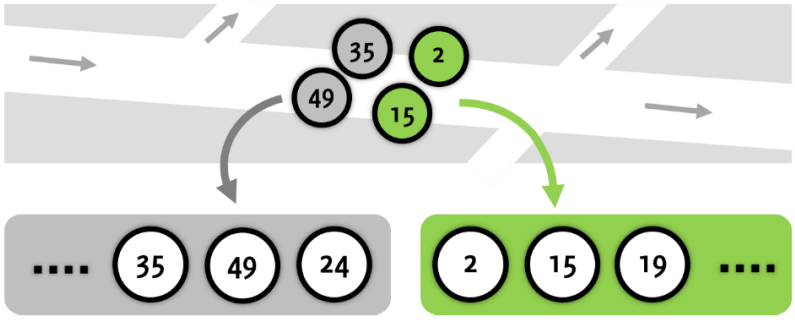


Figure 2.5: Scenario where spatially and timely close moving object are located in different batches.

algorithm will not consider them because they are placed in different batches. Similar situations can potentially lead to a failure in detecting a cluster.

On the other hand, if the batch interval is increased, the off-line phase is further delayed and the time to react to the appearance or dispersion of a spatial cluster is also increased. In addition, if the batch interval is large enough to include more than one consecutive instances of position data from the same moving object, then it may happen that spatial clusters, which have been formed and immediately dispersed by follow-up location updates, may not be detected.

To address these issues, we have decided to explore the Complex Event Processing (CEP) real-time data stream processing paradigm. This decision is based on CEP providing temporal window (*e.g.*, sliding, batch) and real-time processing primitives as first-class function to process the event data stream. Precisely, the CEP programming model was designed to provide such abstractions. Our idea is to design a DBSCAN-like data stream clustering algorithm using CEP primitives instead of relying on a traditional off-line algorithm. By doing so, we aim at taking advantage of CEP real-time nature to build an on-line and *near* real-time detection algorithm.

2.2 Complex Event Processing

Complex Event Processing (CEP) is a programming paradigm that supports reactions to a stream of event data in real time [35, 41]. In contrast to DBMSs, in which data is first stored and then queried later, CEP stores continuous queries and runs data through them, *i.e.*, rather than storing the data, CEP focuses on continuously analyzing and processing the data whiles it passes, using the stored queries. Each continuously query implement one or more CEP real-time primitives, such as *filter*, and *negation*.

Event streams are the main input source of a continuous query. The continuous query combines CEP real-time primitives to react, process, and derive other higher-level (complex) events. In CEP, events are created through producers, which are entities (*e.g.*, sensors, client applications) that encode interesting occurrences of the application domain. Events are characterized by a type, a timestamp, and a payload [41]. The event type defines the payload schema, *i.e.*, the corresponding attributes name and domain that represent the given event occurrence. For example, we can define the event type *LocationUpdate* to represent a moving object position update using the following payload schema: *uuid*, latitude, longitude, and timestamp.

An event data stream is the resulting sequence of events created and sent by producers [14, 13]. Precisely, events in a data stream follow the same type and their order are based on the timestamp tag of each event. A continuous query uses its real-time primitives, such as *filter*, *split*, *project*, *sequence*, and *negation*, to react and to process the incoming event data stream as it passes. For instance, filter the *LocationUpdate* event stream to discover moving objects that are close to a given point of interest.

Continuous queries output events are known as complex event since they represent higher-level information [41]. Both, normal (raw) events and complex events, can be used as part of the definition of another complex event. Precisely, it is possible to create hierarchies of events, in which intermediate events can be used to define other higher-level complex events. As an example, a complex *TrafficJam* event can be built by combining multiple *LocationUpdate* events in the same area and period.

Each continuous query is executed by a CEP processing stage known as Event Processing Agent (EPA) [14]. An EPA stage continuously: reacts to incoming events; analyzes and manipulates them; and outputs derived events to event consumers, other EPAs processing stages or endpoint applications. By interconnecting EPAs, it is possible to build an Event Processing Network (EPN), a topology workflow that analyzes the input event stream as it passes. Further, the EPN topology structure, a directed graph, facilitates the distribution of EPAs into different machines. Each EPA in this network is responsible for receiving events, processing the continuous query (CEP real-time primitive), and if there is a derivative step, output the complex events to the next EPA or endpoint applications.

Figure 2.6 exemplifies a CEP workflow. In the depicted scenario, event producers (buses and metro) frequently generate and transmit two types of location update events. EPAs in the EPN border continuously receive and process these events. If the analyzed event satisfies the processing primitive

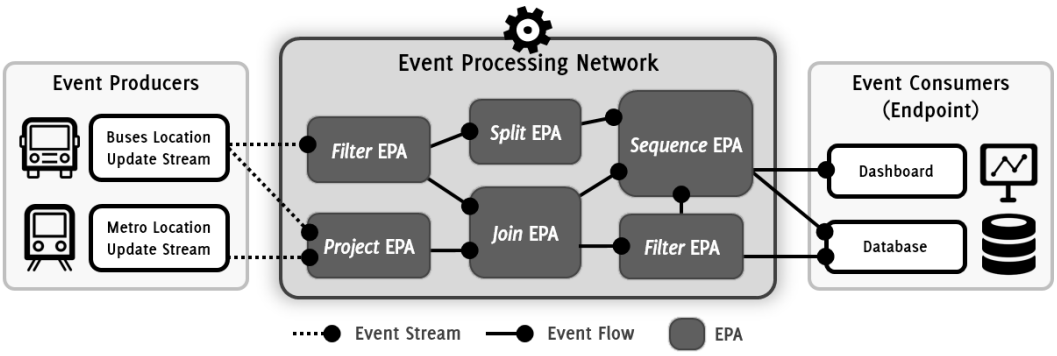


Figure 2.6: An overview of an Event Processing Network (EPN) workflow.

logic, the EPA emits a complex (higher-level) event as output. Such events are delivered to interested (connected) EPAs to be further processed. By using a network of EPAs continuous queries (real-time primitives) it is possible to further analyze, refine, combine, and process the input events to detect a given situation. In the end of this chain, there are border EPAs that send complex events (*e.g.* a detected situation) to interested endpoint applications or resources, such as a dashboard and database systems.

2.2.1 CEP Engine and Continuous Queries Languages

A CEP engine instantiates the presented CEP concepts. It provides operations for defining event types (payload schema) and real-time primitives to express the continuous queries (EPAs) and to interconnect them (EPN). There are several CEP engines implementation, for example, Esper [46], Sase [47], Microsoft StreamInsight [48], Apache Flink [49], Red Hat Drools [50], and TelegraphCQ [51]. The main difference between such engines are the language constructs used to define the event types and the supported real-time primitives. In general, CEP's language constructs can be divided into two models: stream-oriented and rule-oriented [14, 36, 52].

The stream-oriented language model follows the Continuous Query Language (CQL) [53] formal design. CQL is an SQL extension to support stream operators. Several CEP engines uses a CQL-like language due to its formalism and its similarity with the SQL language. To illustrate the expressiveness of CEP's stream-oriented language model, consider the continuous query written in Esper's Event Processing Language (EPL) [46] in Code 2.1. The EPL continuously count the number of filtered *LocationUpdate* events in a latitude and longitude interval within a sliding window of 10 seconds.

```
1 INSERT INTO FilterLocationUpdate
2 SELECT COUNT(*)
3 FROM LocationUpdate.TIME(10 s)
4 WHERE (lat > −21 AND lat < −23 AND
5        lng > −42 AND lng < −43)
```

Code 2.1: Sample continuous query written in Esper's stream-oriented language.

To implement this EPA, the CEP engine does the following. When the continuous query receives the incoming event it first retrieves the events in the specified sliding window, *i.e.*, all *LocationUpdate* events received in the past 10 seconds *w.r.t.* the analyzed event timestamp. This subset event stream is transformed into a temporary relation. Using this relation the continuous query filters the events whose payload values are within the latitude and longitude intervals. Then, the continuous query counts the number of filtered tuples and wrap this value into a new *FilterLocationUpdate* complex event.

Note that an EPA's continuous queries, written in a stream-oriented language, can implement one or more CEP real-time primitives. In the illustrated case, the EPL query implements the *filter*, *aggregate count*, and *project* primitives. In addition, as said, note that the CQL syntax is very similar to an SQL query. One of the differences is that the output stream of a continuous query can be streamed into another event stream to be consumed by other continuous queries. In this case, the continuous query continuously produce *FilterLocationUpdate* events. Another difference from standard SQL is the ability to use time windows, *i.e.*, to analyze and correlate incoming events with a temporal subset of the event streams. We will cover time window in a later subsection of this chapter.

In contrast to SQL-like languages, CEP engines that employs rule-oriented languages are based on inference and event-condition-action (ECA) clauses [36]. Inference and ECA rules separates event handling, condition checking and action into different clauses. First, the EPA's continuous query is triggered when the given event happens. Then, a constraint or condition is verified. Finally, if the condition is satisfied the rules executes the action clause.

To exemplify a rule-oriented language consider the rule illustrated in Code 2.2. It is a continuous query similar to the previous CQL example, but now written in Red Hat Drools [50] rule language. The ECA rule is triggered when a *LocationUpdate* event arrives. Then the condition clause is verified, *i.e.*, it checks if the event is within the specified latitude and longitude interval. Further, it accumulates all *LocationUpdate* events within the spatial interval

for a period of 10 seconds. Finally, it counts the number of events received and wraps this value into a new *FilterLocationUpdate* complex event, which is delivered to other CEP rules through the insert statement.

```
1  RULE "Count Filtered Position During a Given Period"
2      WHEN
3          LocationUpdate (lat > -21 AND lat < -23 AND
4                          lng > -42 AND lng < -43)
5          ACCUMULATE (
6                  $lus = LocationUpdate (lat > −21 AND lat < −23 AND
7                                         lng > −42 AND lng < −43)
8                  OVER WINDOW:TIME(10 s),
9                  $cnt = COUNT($lus)
10         )
11     THEN
12         INSERT(FilterLocationUpdate($cnt))
13 END
```

Code 2.2: Sample continuous query written in Drools' rule-base language.

The primary difference between stream-oriented and rule-oriented languages is the syntax used to build continuous queries. Precisely, stream-oriented languages extends existing SQL queries with other primitives and stream operators (*e.g.*, time windows), whereas rule-oriented languages uses the event-condition-action (ECA) clauses to process the incoming events. Both language models, can implement the same CEP real-time primitives. For instance, consider the sample continuous query described by Code 2.1 and Code 2.2. In both cases, the query languages implements the *filter* primitive.

Nevertheless, in practice, CEP engines support a different set of processing primitives [52]. In general, engines that employs stream-oriented language focus on supporting *transformation* primitives (*e.g.*, *filter*, *split*, *combine*), while those that are based on rule-oriented languages concentrate on supporting CEP's *pattern detection* primitives (*e.g.*, *sequence*, *negation*). Figure 2.7 illustrates a classification of CEP primitives, adapted from [14, 52], that are usually supported in each language model. However, it is important to note that some CEP engines, *e.g.*, Esper [46] and Microsoft StreamInsight [48], support both types of language models. Hence, they usually contain both type of primitives. In the following subsection, we define these class of CEP primitives and briefly discuss the function of each one of them.

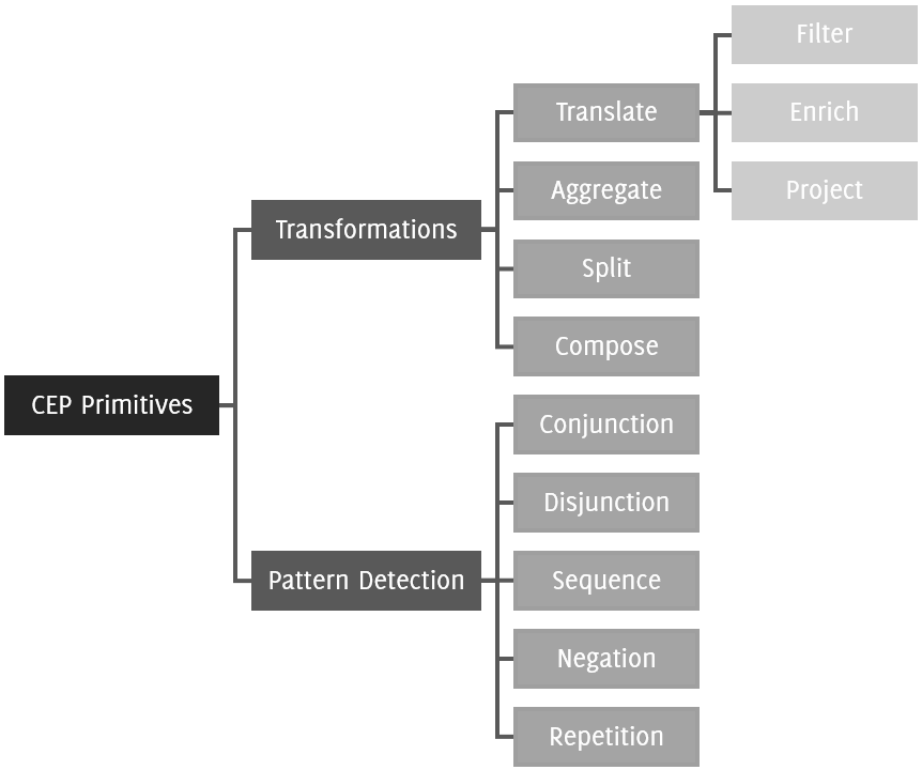


Figure 2.7: Classification of CEP real-time primitives.

2.2.2 CEP Primitives

Transformation primitives are those that either filter, modify, or correlate the incoming events. For example, translate primitives can convert the input event into another derived event using its attributes or an external data source. As in relational algebra, the *project* primitive creates a derived event using a subset of its attributes, while the *enrich* primitive uses an external data source (*e.g.*, in-memory data table, or a database) to create a derived event that contains new or modified information from the original input event. For instance, a *LocationUpdate* event can be enriched with existing Point of Interest of a database to generate a complex *LocationUpdateNearPoI* event to express that a moving object is in the vicinity of a point of interest.

Aggregate primitives combines multiple input events in a single output event, *e.g.*, multiple *LocationUpdate* of a given region in a complex *TrafficJam* event. In addition, this primitive can employs conventional SQL aggregate functions, such as *avg*, *collapse*, *count*, *min*, *max*, and *sum*. *Compose* joins two or more input event streams, looks for match using a given criteria, and creates derived events using the match result. As an example, correlate the *TrafficJam* event stream with *CarAccident* events, using the location distance as matching criteria, to discover if there is a relationship between them. Finally,

the *split* primitive divides an input event into multiple events, *e.g.*, divide the complex *CarAccident* event into two or more *LocationUpdate* events to retrieve the driver's information (uuid).

On the other hand, pattern detection primitives are based on an event pattern template [14]. Event patterns use logic operators – such as *conjunction*, *disjunction*, *repetition*, *negation*, and *sequence* – to define a template of events that continuous query should look for [52, 54]. For example, the *conjunction* primitive defines an event pattern that uses the following template: E_1 `AND` E_2 `AND` ... `AND` E_n. This primitive continuously analyzes the event streams and is satisfied when every event in the template definition, E_1, E_2, ..., E_n, is detected. Note that the template definition is independent of the event order, *i.e.*, it does not depend on the timing or ordering of detected participant events. To illustrate this primitive, consider the following event pattern: *TrafficJam*($|lu| > 50$) `AND` *CarAccident* ($|lu| \geq 3$). This continuous query is triggered when there is at least one *TrafficJam* event with more than 1000 location updates and a *CarAccident* event that involves at least three vehicles (location updates). As result, pattern primitives outputs the detected events in a complex event, *e.g.*, both events, *TrafficJam* and *CarAccident*.

Similar to the conjunction primitive, the disjunction primitive defines an event pattern using the `OR` logic operator, *e.g.*, E_1 `OR` E_2 `OR` ... `OR` E_n. This primitive is satisfied when the continuous query detects one of the events specified in the template. CEP also uses other relation operators, such as negation. The *negation* primitive defines an event pattern that detects the absence of a given event. For instance, the event pattern $\neg E$ [$10min$] means that the continuous query should be triggered if no event E is detected within 10 minutes. To exemplify the negation primitive, consider the following continuous query: $\neg$ *TrafficJam*($|lu| \geq 5$) [$30min$]. This continuous query will be triggered if it does not receive any *TrafficJam* event, whose payload includes a total of 5 moving objects, in a period of 30 minutes.

CEP also provides primitives for detecting events that happen in a given order. The *sequence* primitive defines an event pattern template that can capture the specified orders, *e.g.*, $E_1 \rightarrow E_2 \rightarrow E_1 \rightarrow E_3$ is triggered when the continuous query receives the following events in order E_1, E_2, another E_1, and E_3. For instance, in the sample scenario, the sequence event pattern *CarAccident*($|lu| \geq 2$) $\rightarrow$ *TrafficJam* [1 hour] $\rightarrow$ $\neg$ *TrafficJam* [60 min], detects the situation that a *CarAccident* event with at least two location updates is followed by a *TrafficJam* event within one hour. After that, the pattern looks for the absence of a *TrafficJam* event in 60 minutes. This situation can indicate that the traffic jam generated by a car accident has disappeared.

The repetition primitive uses an event pattern that detects when a given event type repeats at least n times within a given window (*e.g.*, time, length). For instance, consider the repetition primitive with the following event pattern E $[n]$ [3 minutes]. This primitive continuously consume and process events until it founds at least n events of type E within 3 minutes. The event pattern $LocationUpdate(lat > -23, lat < -21, lng > -43, lng < -42)$ [100] [5 minutes] detect the occurrence of 100 location in a time period of 5 minutes within the latitude $[-23, -21]$ and longitude $[-43, -42]$ intervals.

2.2.3 CEP Context and Time Windows

Several presented primitives requires the concept of a window (*e.g.*, a time window) to process the incoming event stream. For instance, the *aggregate* primitive combines correlated events that occurs within a given time window in a single complex event. CEP provides means for grouping related events in a context (window) to enable them to be processed in a related way. A CEP context subdivides the event stream into one or more partitions [14, 13] using logical and/or temporal predicates. Further, each context partition represents a subset of the partitioned event stream.

As an example, consider the *SameObjectLU* context partition declaration and illustrated in Figure 2.8. This context subdivides the *LocationUpdate* event stream according to the *uuid* attribute value. The resulting context partition is a subset stream of *LocationUpdate*, such that all events located in

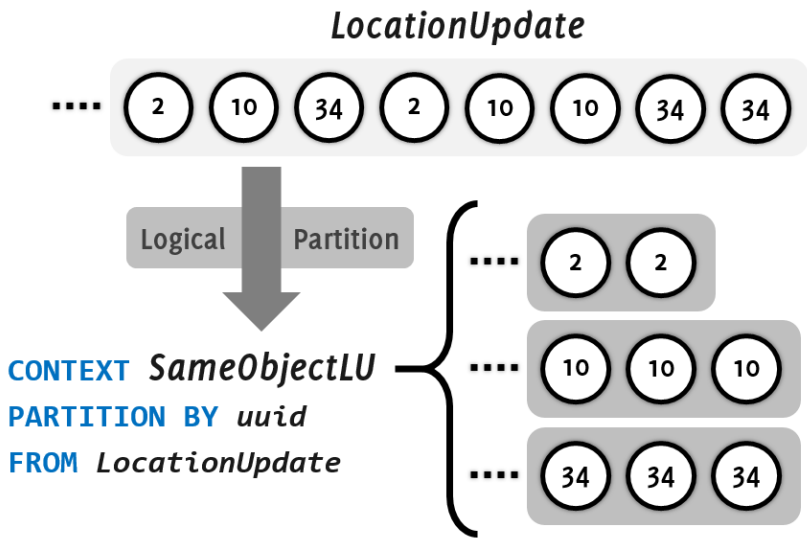


Figure 2.8: An overview of a context partition, *SameObjectLU*, that subdivides the *LocationUpdate* event stream using each event's *uuid* value.

a given partition contains the same *uuid*, that is, the events emitted from the same moving object. Since context partition are event streams (a subset), all CEP real-time primitives works on them. For example, the following sequence primitive, $SameObjectLU \rightarrow SameObjectLU$ [60 s] can be applied to detect a situation in which the delay between location updates of a specific moving object is larger than 60 seconds.

Time windows are also context partition. Precisely, a time window is a temporal context [14, 13] that subdivides an event stream into time intervals using the *timestamp* attribute. A time window partitions the event stream using the timestamp tag of each event, such that it includes only events that are within the given interval, *e.g.*, $(now - \Delta, now)$ [3, 22], where now is the current timestamp. For example, the $LocationUpdate$ [SLIDE 30 second] time window declaration automatically creates a temporal context partition that retains the latest ($\Delta = 30$ seconds) $LocationUpdate$ events. When an EPA process an event with timestamp t in such window, the real-time primitives will only consider events received in the previous $(t - 30, t)$ interval. This concept is useful to ensure that only the latest events from the data stream are considered.

The two primary kinds of time windows in CEP are Landmark and [22, 53], as illustrated in Figure 2.9. Landmark time windows provides the ability to process the event stream in batch. It buffers all event produced during a Δ time interval and then applies the continuous queries to the whole set of events. Precisely, there can be a delay between the event's arrival time and its processing time. For example, in Figure 2.9 (a), while events E_1 and E_2 have arrived at time $t = 1$, they will only be processed when the batch period is over $(t = 2)$.

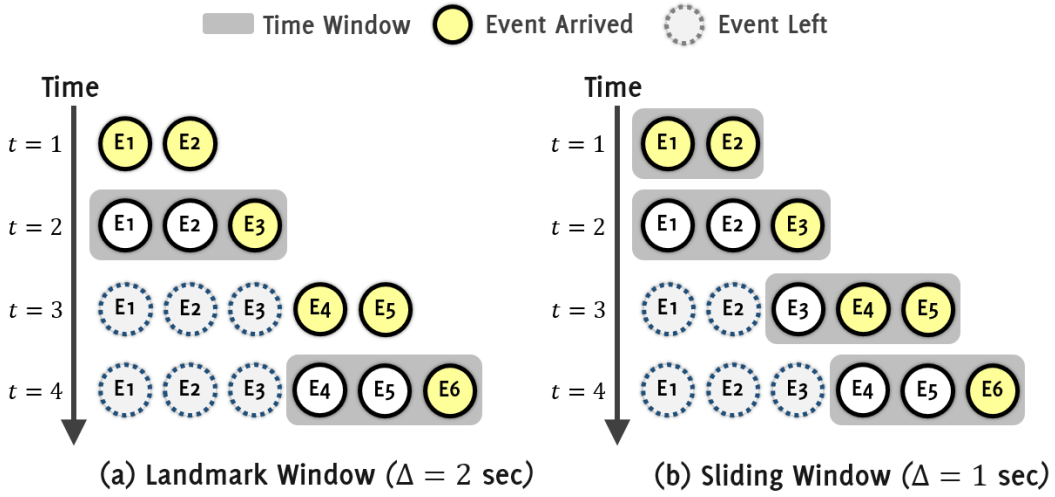


Figure 2.9: Example of Landmark and Sliding Time Windows.

By buffering the events, it is possible to use aggregate primitives' functions, such as *min* and *max*, to summarize the entire batch content in a complex event. However, if events that are meant to be processed together, *e.g.* by a *sequence* primitive, are placed in adjacent batches the primitive will fail to detect the correlation between the events'. One way to mitigate this issue is to increase the Δ batch period, but this causes an additional delay overhead to process an event.

Sliding windows handle this problem by moving the time window along with received events. Thus, instead of having predefined batch periods, the time window boundaries slide to the current event timestamp. More specifically, a sliding window is a moving landmark window that contain events in the past Δ time units. To illustrate this scenario, consider the event stream with a sliding time window of 1 second in Figure 2.9 (b). For instance, when $t = 3$, the time window includes the events from the past second ($t = 2$) to the current time. This sliding movement, adjusting the time window bounds referential to the current event being analyzed, mitigates the issue of having correlated adjacent events in different landmark (batch) periods. By sliding the time window it is possible to include the previous adjacent events that would be placed in different batches. By using sliding time window with real-time primitives it is possible to glimpse in the pasts events and provide continuous and near real-time event processing.

Finally, there are fading windows. A fading window is a sliding window where a decay factor λ is applied to the events according to their age to differentiate their importance for the event processing, *i.e.* more recent events have higher importance than older events. It is very useful and provider a richer information that sliding window. However, the continuous computation of each event weights using the decay factor is too costly for on-line processing of streams [3].

2.3 Summary

This chapter presented the two fundamental topics that underpins this thesis, Spatial Clustering algorithms and the Complex Event Processing (CEP) data stream processing paradigm.

First, we presented the density-based DBSCAN off-line clustering algorithm [12]. We discussed how DBSCAN's bottleneck is the computation of the distance between moving objects to discover the neighbors that are within an ε radius. To speed up this computation, some approaches employ spatial data structures, which are able to efficiently retrieve a moving object neigh-

bors through a series of range queries. However, this computation can become troublesome in data stream scenarios, due to the difficulty of accessing and modifying the spatial tree in parallel alongside the constant cost or balancing the tree [7, 8, 9].

To mitigate these problems, the main framework for clustering position data streams operates in batch [10, 11]. One of the main problem with this approach is that it only processes spatial data when the batch finishes. Thus, it delays the clustering results until the next batch. In addition, the batch-based nature provides a discrete view of the spatial clusters instead of a fresh or continuous view of its evolution. Further, if the batch interval is reduced, it can happen that temporally and spatially close moving objects being placed in different batches, which can potentially lead to a failure in detecting a spatial cluster. Contrary to that, if we increase the batch period it will take longer to react to a discovery or disappearance of a given spatial cluster.

We also presented the Complex Event Processing (CEP) real-time data stream processing paradigm [35, 41]. CEP provides logical operators, primitives, and time windows to react and process event streams in real time. These concepts are used to specify continuous queries, that analyze data stream as it passes rather than afterward. Each continuous query is executed by a processing stage known as Event Processing Agent (EPA). Each EPA continuously receives incoming events, analyzes, manipulates them, and outputs derived events that are delivered to event consumers, other EPAs processing stages or endpoint applications. EPAs can be interconnected to build a network of continuous queries, which in turn can be used to detect a higher-level situation.

In the next chapter, Chapter 3, we continue to present the main benefits and limitations of how current approaches handle large position data streams in on-line and/or in near real-time. After that, in Chapter 4, we present DG2CEP, our approach for processing such scenario.

3
Related Work

In this chapter, we present several recent approaches for clustering large position data streams in on-line and/or in near real-time. Overall, they can be classified according to the applied technique: sampling, micro-clustering, grid, and CEP-based. Respectively, Sections 3.1 to 3.4, present and briefly discuss the advantages and limitations of these approaches. Finally, Section 3.5 summarizes their main limitations.

3.1
Sampling

DENSE [55] is an algorithm that uses sampling to cluster large position data streams. By reducing the data stream size through sampling, it aims to speed up the clustering process for on-line and real-time scenarios. It uses a sampling function to define which moving objects from the collected data will be sampled. To build this function, every Δ time units, it collects all the moving objects' position data. Then, using the collected data it computes the spatial clusters using an off-line DBSCAN-like algorithm. After that, it selects the k most representative moving objects from the resulting clusters. Such representatives are picked using two factors: highest number of neighbors and distance to already picked representative. To do so, DENSE uses kernel estimation [56] to effectively pick the k^{th} most representative moving objects . After defining the sampling set, DENSE will only process the position data from moving objects located in this set. Precisely, during the next data stream Δ time interval, DENSE will update the detected cluster using only the position data of the k^{th} representatives. At the end of this period, DENSE receives all moving objects, recomputes the new set of clusters, and elects a new set of k representatives from them.

Although the sampling approach is interesting, it may delay the detection of emerging clusters since DENSE only processes the position data of the k^{th} moving object that are *already* in a given cluster. Clusters that rapidly appear and disappear during the sampling process may not be detected since its moving objects are not within the k^{th} most representative moving objects. More specifically, before electing a new set of k^{th} representatives, DENSE only

update the existing clusters. Hence, similar to batch-based approaches, it does not address the situations where spatially and temporally close position data fall into different batches. Finally, this approach cannot provide a continuous view of the clusters' evolution, *e.g.*, if an existing cluster has merged with another one, since cluster will only be recomputed at discrete Δ periods.

3.2 Micro-Clustering

Micro-Clustering is a summarization technique for clustering based on cluster features [31], a characteristic vector. The overall idea is to summarize the cluster characteristics, such as its centroid, the number of moving objects, its radius, rather than the data points itself. The algorithms using this concept associate each new data point with some neighboring micro-cluster. If no such micro-clusters exists, *i.e.*, if a new incoming data is not within range of an existing micro-cluster centroid (*w.r.t.* ε) a new micro-cluster is created with the single data point.

Many works following this approach [33, 57] limit the number of position data represented by a micro-cluster, that is, a micro-cluster is split when the number of summarized position data exceed a given value. The summarization of an incoming position data into a micro-cluster may be costly, since it requires to compute the distance between the new position data and all existing micro-clusters. Specially in scenarios with many micro-clusters this may incur in much processing. In fact, the size of a micro-cluster is a recurring trade-off in these approaches. If the micro-cluster size is large, it requires less comparison, which in turn makes the algorithm faster. However, in such cases, the clustering results may become inaccurate, since it may group a large number of moving objects that are not close to each other in the same micro-cluster. On the other hand, if the micro-cluster size is small, the algorithm must do more comparison, but the result will be more accurate since there is less discrepancy within the moving objects located in the same micro-cluster.

By using the micro-cluster's centroid as a "virtual position data" those algorithms can further cluster these data points using a standard algorithm, such as DBSCAN, instead of the original position data. Usually, micro-cluster approaches also operate in an on/off-line mode, where the on-line phase summarizes the incoming data points into the micro-cluster feature vector, and an off-line phase that clusters the data points of the micro-clusters itself using DBSCAN.

Based on this concept, Forestiero *et al.* [58] propose FlockStream, a bio-inspired [28] and micro-cluster method for data streams that follows Aggarwal

et al. [10, 11] two phase batch-based framework. Their approach is based on the multi agent paradigm and is able to perform clustering on any kind of data (not just spatial data), including high-dimension. In FlockStream, each moving object denotes an agent. It maps the moving objects high-dimensional data (*e.g.*, latitude, longitude, altitude, velocity) to 2D agents (x, y) in a grid-like structure. Each agent is responsible for discovering its ε–$Neighborhood$ by communicating with neighboring agents in its radius. By doing that FlockStream is able to speed up the distance computation, since it is performed independently by each agent. Then, a central agent communicate and retrieve, from time to time, the ε–$Neighborhood$ of each agent (moving object) to compute the spatial clusters.

The primary contribution of FlockStream is the significant reduction and parallelization of distance computation of incoming and existing position data to neighboring agents in a visibility radius (a distance threshold). However, to reduce the distance comparison FlockStream assumes that each agent knows each other and is able to communicate with one another, which is not feasible for many application scenarios. For instance, in a city wide monitoring application, a city bus may not known or have means to communicate with a vehicle that is passing by. In fact, to enable this approach, one needs to employ a scalable peer discovery mechanism in each moving objects. In addition, the central agent concept is similar to the batch-based approach discussed before and inherits all its limitations.

Kranen *et al.* [33] tackle a specific problem related to clustering large data streams, where a data stream throughput can vary over time and no assumption can be made about minimal times between consecutive items in the stream (*e.g.* at peak times). To handle this problem the authors present ClusTree an index structure for storing a compact view of the current clustering result and an anytime clustering algorithm that adapts its update process to the arrival rate of the items in the data stream, delivering a fast but coarse/unprecise result for very fast streams, and more refined results for slower stream flows. Their approach proposes a hierarchical organization of micro-clusters in a balanced multidimensional index structure, (similar to an R-tree) where higher-level entries in the tree represent aggregated information about the micro-clusters of their sub-trees. When updating the structure for each newly arrived item, due to the current stream flow there might not always be sufficient time to reach the leaf node (*i.e.* closest micro-cluster). So the work proposes interrupting the insertion process and temporarily storing the item in a local aggregate (some temporary micro-cluster) that is also buffered in the tree entries and is piggybacked in future descends of the tree, when there is sufficient time.

ClusTree inserts a position data by finding its closest micro-cluster (tree node). Precisely, it navigates the tree by comparing the incoming position data location to the closest intermediary micro-cluster centroid. However, it is difficult to ensure that the position data is in fact closer to this micro-cluster since its centroid is essentially an estimate of the underlying micro-cluster centroid locations. Thus, this approach can lead to a new position data being inserted in a wrong micro-cluster. In fact, ClusTree provides several insertion traversal heuristic for dealing with this issue.

ClusTree provides the on-line phase (summarizing) of micro-clusters clustering. However, as mentioned before, to discover the clusters boundary it still need to cluster the indexed micro-cluster in an off-line phase. While the number of distance computations of incoming data point in ClusTree is reduced, due to the tree-like data structure, the algorithm needs to maintain the tree balance, which can be troublesome for high speed data streams. Further, since ClusTree is based on micro-clusters, and those record only summary information about the data points, such as the cluster centroid, the information of the data points (*i.e.* the moving objects that form the cluster) is not preserved. As a consequence, it is no able to identify the moving objects that are placed in such cluster. This can be an issue in numerous scenarios, for instance, it is interesting to known the buses lines located in a traffic jam or to known the vehicles involved in a given accident to activate their insurance.

Jensen *et al.* [57] propose an interesting and novel approach for on-line clustering of moving objects. Similar to ClusTree, it combines a micro-cluster approach with spatial index. But, by assuming that cluster shapes are circular it is able to predict when a cluster will split. It does that by using a maximum radius and the moving object's velocity. One of the main contribution of their work is the ability to predict cluster splits or merges by using the moving object's velocity direction and speed vectors and assuming a linear movement from them. However, it also imposes a restriction over the cluster shape (circular), which may not be feasible for many applications. More specifically, they are only able to detect that have a circular shape. In addition, like other micro-cluster approaches, it requires an off-line phase to compute and detect the cluster.

3.3 Grid-based

Grid-based clustering algorithms have been proposed as a means of scaling the clustering process [3, 10, 59]. In this approach, moving objects are mapped to rectangular grid cells (*a.k.a.* grid partitions), and the cost of this mapping is only proportional to the number of cells [3] since grids provide a static reference

for the moving objects. Thus, each grid partition only "holds" the objects whose position data falls into the cell's geographic area. If we assume that position data density is a good metric for clustering, then the density of each cell would be the number of position data mapped into to the cell divided by the size of the cells. Note that this approach gives a slightly different density semantics than DBSCAN's notion of density. Here, clusters are detected based on the density of cells rather than the density of the neighborhood of a core node, as in DBSCAN.

D-STREAM [59] and DENGRIS-STREAM [60] are two well-known representatives of grid-based algorithms for clustering that are generic and not focused on any particular application domain. As with other approaches, both algorithms follow Aggarwal *et al.* [10, 11] on/off-line two-phase batch framework. At regular time intervals, *e.g.* every 30 seconds, the on-line phase collects streamed data, maps them to the grid cells and makes some summarization process, while in the off-line phase the clustering function is executed over the set of data accumulated during the on-line phase. This on/off-line operation mode has the disadvantage that clusters which occur during the off-line processing phase are only detected at the following time period. Furthermore, such approaches do not address the issue of spatially and temporally close moving objects being placed in different batches and/or to different (adjacent) grid cells. In the former case, if the algorithm aims to consider the location data placed in previous batches it needs to implement and maintain a time window for each location update. This process is costly and further complicates the algorithm logic, since it is mixed with the clustering part. In the latter case, it can happen that spatially close objects with the same batch (*w.r.t.* ε) are mapped to adjacent grid cells. Thus, although they are spatially and temporally close, they would not contribute to the density of the grid cell which may lead to failing to detect a spatial cluster.

Moreover, in both algorithms the clustering function does a global search for all dense and modified cells of the grid, which involves high processing cost over the streamed data, specially for large grids. This function also updates the summarization of all cells individually, which further increases the processing cost. Finally, both approaches are only able to provide a discrete view of the spatial cluster evolution. Thus, not being able to provide a smooth and continuous view of its evolution, nor detect the rapid formation and dispersion of spatial clusters within the same batch.

Several approaches, such as MR-DBSCAN [8] and DBCURE-MR [61], have tackled the problem of modeling large clustering datasets to the MapReduce programming model by also subdividing the space as grid. By using a

sequence of Map and Reduce phases they are able to correctly identify the spatial clusters. While MapReduce is a powerful paradigm for highly parallel processing, both work does not handle streaming data, only large datasets, and also operate in batch due to the stateless nature of the map and reduce primitives. Reduce primitives needs to wait for the MapReduce coordinator to send its input entries, which in turn needs to wait for all the corresponding map functions to finish it. Further, because of the stateless nature of the programming model and the clustering recursive nature, there is a high communication and data transfer cost between each MapReduce phase and intermediary data need to be saved and accessed in the distributed file system.

3.4 Complex Event Processing

Considering CEP-based solutions for clustering position data stream, the majority of works encountered [62, 63, 64, 65] provide generic range-query based primitives for detecting when a region is dense. For example, Mouza and Rigaux [62] reduce the problem of mobility pattern detection to regular expression. They assume that the space is divided into a set of continuous regions, each one represented by a token. For example, Figure 3.1 illustrates a spatial domain divided into six regions: *a, b, c, d,* and *e.*

Their model map a given moving object location (latitude, longitude) to the region it falls into. Then, the mobility model of a given moving object is represented by a string containing a sequence of region tokens. Enter events add a token to the object mobility string, while leave events add a time quantifier to that region. For example, the sequence `"[a{4}.c{2}.f{1}]"`, means that the moving object stayed four minutes in region *a*, two in region *c* and in the *f* region. LIFT [63] and Mobiiscape [65] extend these primitives with other

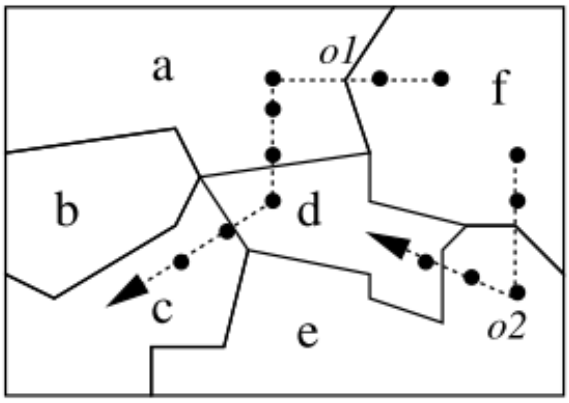


Figure 3.1: Regions represented by tokens. Patterns can be extracted using "regex".

operations. For example, they add an additional primitive called *visit*, that combines the *enter*, *stay*, and *leave* primitives. Nevertheless, since these works are all based on these primitives, they are limited to patterns that associate the movement of a mobile object with a specific region.

Although these primitives can be used to detect dense regions, they require developers to specify *a priori* information of the possible locations of clusters, which is not feasible in many systems. In addition, since the user need to specify the possible clustering regions, they are not able to detect clusters with arbitrary shapes, such as a traffic jams in a road/avenue that crosses several regions, since the cluster boundary is already predefined in the range pattern query. Contrary to that, this thesis aims to detect both the formation and dispersion of arbitrary clusters (in a given domain) without the need of specifying the clustering region *a priori*.

3.5 Summary

This chapter presented several current approaches for clustering large position data stream in real-time or in an on-line manner. Overall, the primary problem with such approaches is that they follow Aggarwal *et al.* two phases data stream framework [10, 11], as illustrated in Table 3.1. One of the major issue with this framework is that the detection process is delayed until the next batch period. Precisely, it will only provide clustering results at discrete times. Thus, it is not able to provide a continuous view of the cluster evolution neither able to detect cluster that rapidly emerge and then dissipate within the batch period.

Table 3.1: Comparison of related works

Algorithm	Technique	Inspiration	Phases	Window
DENSE [55]	Sampling	OPTICS	2 phases	Batch
FlockStream [58]	Agents	k-Means	2 phases	Batch
ClusTree [33]	Micro	k-Means	2 phases	Batch
Jensen *et al.* [57]	Micro	DBSCAN	2 phases	Batch
D-Stream [59]	Grid	DBSCAN	2 phases	Fading
DENGRIS-Stream [60]	Grid	DBSCAN	2 phases	Sliding
MR-DBSCAN [8]	Grid + MR	DBSCAN	2 phases	Batch
DBCURE-MR [61]	Grid + MR	DBSCAN	2 phases	Batch

To address these issues, in the next chapter we present DG2CEP. Our idea is to design a DBSCAN-like data stream clustering algorithm using CEP primitives and contextual time windows instead of relying on a traditional

off-line algorithm. By doing so, we aim to take advantage of CEP real-time primitives and sliding time window concepts to build an on-line and near real-time detection algorithm.

4
Density-Grid Clustering using Complex Event Processing

In this chapter, we present Density-Grid Clustering using Complex Event Processing (DG2CEP) a density-grid data stream clustering algorithm expressed as a network of CEP primitives. DG2CEP combines the grid index [3, 39] with CEP's data stream processing primitives [14, 13] to enable the continuous detection of spatial clusters and monitor their evolution from large streams of position data in near real-time.

Contrary to DBSCAN, which computes a pairwise distance between moving object positions, DG2CEP employs a counting based semantic. To do so, it divides the spatial domain into a context partition grid. Then, each moving object position data is mapped into one grid cell (*a.k.a.* context partition). When one context partition cell contains more than $minPts$ unique location updates – within a sliding window of a Δ period – a core cell event is derived. This event is enriched with the adjacent cell border events to form a Cell Cluster event (core plus border cells' content). On the other hand, a Cell Disperse event is generated to indicate that the cell cluster has become sparse, that is, when it no longer contains $minPts$ location updates.

Grid cluster events are composed of one or more adjacent Cell Cluster events. Precisely, they contain a set of adjacent core cells and their corresponding border cells. Incoming cell cluster and cell disperse events are correlated existing grid clusters to either create, destroy, update, merge, or split them. As a result, this process produces as output the resulting grid cluster and its corresponding semantic, *e.g.*, add, merge, split, disperse.

Overall, DG2CEP, illustrated in Figure 4.1, can be expressed as a network of CEP primitives and is divided in three parts:

- Fetching and mapping the moving object's position data and building a stream of location updates (*Stream Receiver* EPN);
- Mapping and identification of grid cells which have become dense or sparse over time (*Cell* EPN);
- Correlate dense and sparse cells with existing grid clusters to create new, update, augment, or split them (*Grid* EPN).

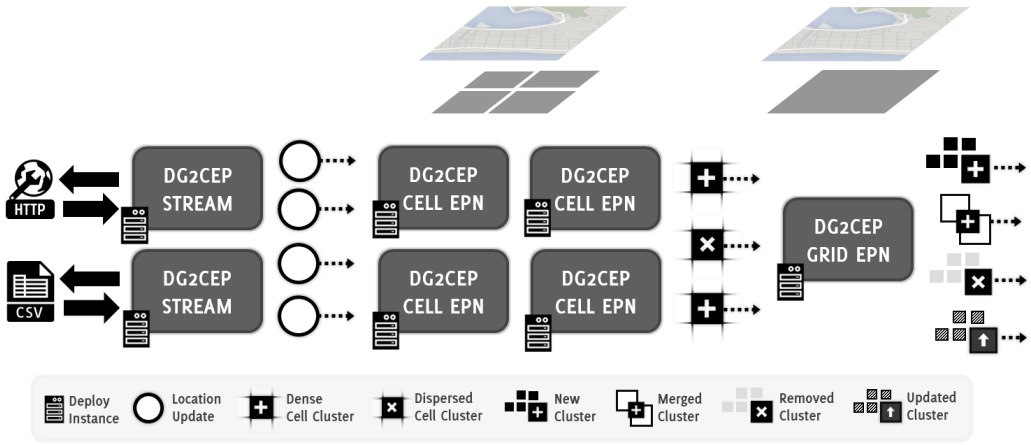


Figure 4.1: Overview of DG2CEP distributed event processing architecture.

Each stage of the algorithm, which expresses a subset of the EPN, can be deployed in parallel on different machines in a cloud or cluster, so as to distribute the workload. Distributed instances within DG2CEP EPN are interconnected through a message-oriented middleware, such as publish/subscribe, that provides an uncoupled communication channel between each stages. For example, a *Cell* EPN subscribe to a topic containing *Location Updates* events and publishes as output to the *DenseCellCluster* and *DispersedCellCluster* topics.

To distribute the workload, DG2CEP subdivides the event stream topics into different spatial ranges using spatial filters. Then, each deployed instance subscribes to position data of a spatial range to only handle events from that region, as suggested by Figure 4.1, where each of the CELL EPNs only consume events from one of the previous squares (top). In addition, the CEP network design also facilitates parallel processing, since multiple events can be processed in parallel by different EPA continuous queries. For instance, while DG2CEP is filtering dense cells, it can detect in parallel cells that have become sparse. Overall, with the exception of the final stage (GRID), the parallelization factor of DG2CEP is the number of grid cells.

In the remainder of this chapter, we present each DG2CEP algorithm stage. Section 4.1 addresses the *Stream* EPN, which is responsible for receiving and publishing the position data to distributed instances. Section 4.2 presents the *Cell* EPN, which is responsible for detecting cell cluster and disperse events. Section 4.3 presents the *Grid* EPN, which combines the cell cluster and disperse event to either create, merge, update, or destroy grid cluster events. Section 4.4 discusses DG2CEP's computational complexity and trade-offs in the choice of its parameters, while Section 4.5 addresses some of its limitations. Finally, the last section of this chapter summarizes DG2CEP workflow.

4.1 Stream Receiver EPN

DG2CEP's first algorithm stage, *Stream Receiver*, is responsible for continuously receiving, mapping and publishing the incoming moving objects positions data as a *LocationUpdate* event stream. Moving objects periodically inform their location by sending the following position data: $\langle id, lat, lng, t \rangle$, where id is the moving object's identifier, lat and lng are their current position (latitude and longitude values, respectively), and t is the timestamp the data was sampled.

Incoming moving objects position data stream tuples are mapped to a grid cell $\langle i, j \rangle$ to avoid the pairwise distance comparison between them. The overall idea is to reduce the problem semantic to counting – instead of pairwise comparison – the location updates mapped to each cell. By doing so, the algorithm can rely only on counting the number of location updates in each grid cell to discover dense and sparse cells, that will constitute the spatial clusters.

Thus, the entire monitored spatial domain, a rectangular region defined by $[lat_{min}, lat_{max}]$ and $[lng_{min}, lng_{max}]$, is divided into a grid G of grid cell size $\frac{\varepsilon}{\sqrt{2}} \times \frac{\varepsilon}{\sqrt{2}}$. The choice for this respectively grid cell size is to guarantee that the maximum distance between any two location updates within the grid cell is ε, similar to DBSCAN ε–*Neighborhood* , as shown in Figure 4.2. Thus, G is segmented in the following intervals:

- $i \rightarrow [lng_{min}, lng_{min} + i \times \frac{\varepsilon}{\sqrt{2}}, lng_{min} + (i+1) \times \frac{\varepsilon}{\sqrt{2}}, \ldots, lng_{max}]$ and
- $j \rightarrow [lat_{min}, lat_{min} + j \times \frac{\varepsilon}{\sqrt{2}}, lat_{min} + (j+1) \times \frac{\varepsilon}{\sqrt{2}}, \ldots, lat_{max}]$

for longitude and latitude respectively. To calculate each interval of the domain DG2CEP uses a spatial offset function[1], which receives as input a latitude and longitude coordinate alongside an $\frac{\varepsilon}{\sqrt{2}}$ distance, in meters, and returns the offset latitude and longitude coordinates.

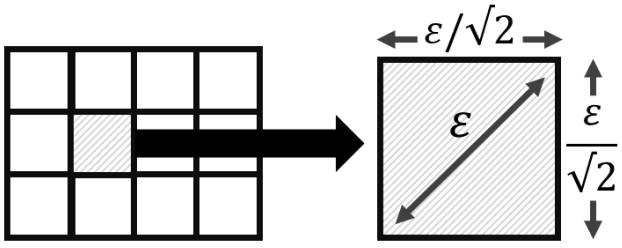


Figure 4.2: Example of DG2CEP $\frac{\varepsilon}{\sqrt{2}} \times \frac{\varepsilon}{\sqrt{2}}$ grid cell division.

[1]See Ed. Willlians Aviation Formulary `http://www.williams.best.vwh.net/avform.htm` for a sample implementation of an offset spatial function.

Algorithm 2 describes the required steps to translate the incoming moving object positions data to *LocationUpdate* events. The incoming data stream is first mapped to a grid cell G_{ij} by mapping its latitude and longitude attribute to the grid intervals. DG2CEP stores the domain latitude and longitude intervals in a static data structure, such as a segment tree or a binary tree, which allows efficient retrieval of the corresponding cell grid index. It is worth mentioning that DG2CEP needs to build this interval data structure only once, since the scope of the monitored domain is immutable, *i.e.*, its intervals' boundary do not change over time.

As an alternative, it is possible to directly compute the position data cell index using an approximation formula: $\lfloor \frac{lat - lat_{min}}{\varepsilon\sqrt{2}} \rfloor$ and $\lfloor \frac{lng - lng_{min}}{\varepsilon\sqrt{2}} \rfloor$, where lat and lng are the position data location, and lat_{min} and lng_{min} are the domain region's lower boundaries. This formula represents the roughly, rounded, number of $\frac{\varepsilon}{\sqrt{2}}$ units required to index the moving object's incoming position data. Finally, DG2CEP emit a *LocationUpdate* event that enriches the incoming position data $\langle id, lat, lng, t \rangle$ with its corresponding grid cell i and j index.

The Stream Receiver stage of DG2CEP (see Figure 4.1) can be expressed in CEP primitives as following. First, the concept of a grid cell is translated to a CEP context partition by dividing the *LocationUpdate* event stream in a series of partitions according which causes the splitting of the entire *LocationUpdate* event stream into $i \times j$ context partitions. For example, Code 4.1, written in Esper Event Processing Language (EPL) [46], creates such context partition (*CellContext*) following this definition. By doing so, continuous queries that use the *CellContext* partition will only consider events that are in the same grid cell, *i.e.*, those that have the same i and j index.

Algorithm 2: DG2CEP (*Stream Receiver*)

Input: An input stream of position data $\mathcal{D}$, the ε distance threshold, the minimum number of moving objects $minPts$, and the latitude $[lat_{min}, lat_{max}]$ and longitude $[lng_{min}, lng_{max}]$ intervals
Output: An out. stream of *LocationUpdate* $\langle id, lat, lng, t, i, j \rangle$ events

1 $lat_{intervals} \leftarrow$ **segment** the latitude interval by $\frac{\varepsilon}{\sqrt{2}}$
2 $lng_{intervals} \leftarrow$ **segment** the longitude interval by $\frac{\varepsilon}{\sqrt{2}}$
3 **while** *data stream* $\mathcal{D}$ *is active* **do**
4 $\quad rawlu \leftarrow$ **read** position data $\langle id, lat, lng, t \rangle$ from $\mathcal{D}$
5 $\quad i \leftarrow$ FindIndex(lat, $lat_{intervals}$) $\quad \triangleright \approx \lfloor \frac{lat - lat_{min}}{\varepsilon\sqrt{2}} \rfloor$
6 $\quad j \leftarrow$ FindIndex(lng, $lng_{intervals}$) $\quad \triangleright \approx \lfloor \frac{lng - lng_{min}}{\varepsilon\sqrt{2}} \rfloor$
7 $\quad$ **emit** *LocationUpdate*($\langle rawlu, i, j \rangle$)
8 **end**

```
1 CREATE CONTEXT CellContext
2 PARTITION BY i AND j
3 FROM LocationUpdate
```

Code 4.1: DG2CEP grid as a context partition (in EPL).

Also the translation of incoming moving object positions data to *LocationUpdate* events can be done with CEP's primitives, as shown in Code 4.2. The difference between an incoming position data tuple and a *LocationUpdate* event is that the latter includes the mapped i and j cell indexes in addition to all the position data tuples attributes ($\langle id, lat, lng, t, i, j\rangle$). By using the *project* and *enrich* primitive it is possible to continuously translate the incoming position data tuple streams into *LocationUpdate* events, as illustrated in Figure 4.3. Here, first, the EPA project the incoming position data values to the *Location Update* event. Then, it enriches this output by computing two additional values, i and j, using the incoming position data latitude and longitude position, lat and lng respectively. As output, it generate the complex *LocationUpdate* event that contains the original position data values and its mapped cell index (i, j).

An EPA with this continuous query can be deployed and executed in parallel, *i.e.*, without any collateral effects due to the stateless nature of the used CEP primitives. Both, *project* and *enrich*, which computation is solely based on the analyzed data. In the following section, we present and discuss how the received *LocationUpdate* events can be used to detect dense and sparse grid cells.

```
1 INSERT INTO LocationUpdate
2 SELECT id, lat, lng, t, ⌊(lng − lng_min)/(ε√2)⌋ AS i, ⌊(lat − lat_min)/(ε√2)⌋ AS j
3 FROM PositionData
```

Code 4.2: Translation EPA (in EPL).

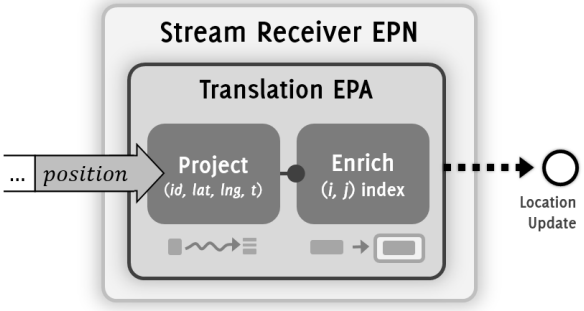


Figure 4.3: Stream Receiver EPN and Translation EPA.

4.2 Cell EPN

DG2CEP second algorithm part, *Cell*, is responsible for receiving the incoming *LocationUpdate* event stream to discover dense and sparse grid cells. Thus, first it uses the communication middleware to subscribe to all *LocationUpdate* events that are within a given latitude and longitude range. By subdividing the spatial domain into different regions, it is possible to distribute the workload between the Cell EPN instances, as shown in Figure 4.4. This scenario configuration contains four deployed instances, which reflect the domain division. Note that the spatial domain can be subdivided in smaller regions to further distribute the workload of the system.

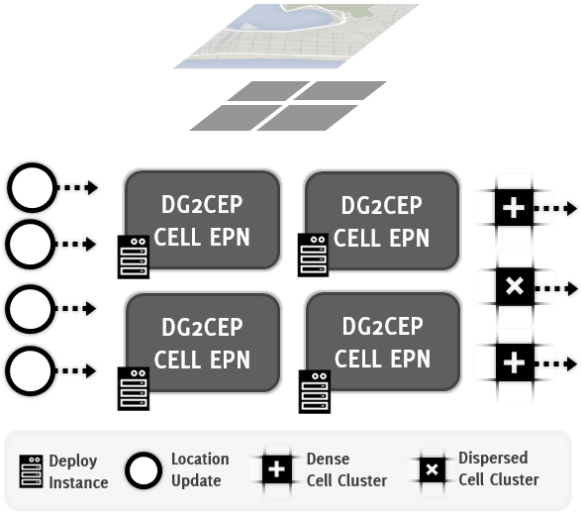


Figure 4.4: Overview of DG2CEP *Cell* EPN.

4.2.1 Dense Cell Discovery

Algorithm 3 procedurally describes how DG2CEP detects the formation and dispersion of dense and sparse cells. As mentioned previously, such algorithm is based on counting the density of each grid cell (context partition) rather than computing the distance between moving object location updates. Therefore, each grid cell gets assigned a density value, which is the number of unique *LocationUpdate* events mapped to the cell within a given Δ time window. *LocationUpdate* events assigned to a cell are all within ε distance (ε-*Neighborhood*) apart from each other since the cell length is $\frac{\varepsilon}{\sqrt{2}}$, which means that the maximum distance between two moving objects in the same grid cell is $\frac{\varepsilon}{\sqrt{2}} \times \sqrt{2} = \varepsilon$.

Algorithm 3: DG2CEP (*Dense and Disperse Cells*)

Input: A stream $\mathcal{D}$ of *LocationUpdate* events, the ε and $minPts$ thresholds, Δ period, and the latitude and longitude intervals
Output: A stream of *DenseCellCluster* $\langle Cell, Neighbors \rangle$ events, and a stream of *DispersedCellCluster* $\langle Cell \rangle$

```
1  G ← create a grid dividing the lng (i) and lat (j) intervals by ε√2
2  L ← create a map to store the latest cell of each moving object
3  while data stream D is active do
4      lu ← read location update ⟨id, lat, lng, t, i, j⟩ from D
5      update lu in G_ij
6      C ← all unique location updates in G_ij in the past Δ period
7      ▷ Verify the cell density
8      if |C| ≥ minPts then G_ij is a dense cell
9          ▷ Retrieve the core's neighboring (adjacent) cells
10         N ← ∅
11         for a ← i − 1 to i + 1 do
12             for b ← j − 1 to j + 1 do
13                 if a ≠ i and b ≠ j then
14                     N ← N ∪ {G_ab}
15                 end
16             end
17         end
18         emit DenseCellCluster (⟨C, N⟩) event
19     end
20     ▷ Now we check if the update generated a sparse cell
21     if L(id) ≠ ⟨i, j⟩ and L(id) ≠ ∅ then mov. obj. changed cell
22         ▷ Verify if previous cell will become sparse when removing lu
23         remove lu from G_L(id)
24         P ← location updates in G_L(id) in the past Δ period
25         if |P| = minPts − 1 then previous cell has dispersed
26             emit DispersedCellCluster (⟨P⟩) event
27         end
28     end
29     ▷ Update the last cell of the moving object
30     L(lu.id) ← ⟨lu_i, lu_j⟩
31 end
```

To detect a dense grid cell the algorithm retrieves all unique *LocationUpdate* events in the data stream, within a Δ period, that are in the incoming event grid cell index (*e.g.*, G_{ij}). This value can be easily computed through CEP, since DG2CEP can take advantage of the *CellContext* partition. Thus, whenever CEP receives a *LocationUpdate* event it can use its (i, j) index to select the appropriate context partition sub-stream, which contains only *LocationUpdate* events from that specific cell index. Using the incoming *LocationUpdate* event

timestamp t, CEP can provide a sliding window abstraction to only consider events whose timestamp are within the $[t - \Delta, t]$ period in the sub-stream. By taking advantage of CEP's sliding window concept, DG2CEP mitigate the issues related to batching, such as spatially close location updates being place in different batches, since it's time window slide (moves) based on the analyzed event. The resulting sliding window sub-stream can potentially have more than one *LocationUpdate* event from the same moving object. CEP is able to further filter the sub-stream, using the moving object *id* value, to retain only the latest *LocationUpdate* event per moving object.

Then, it can use the *aggregate collapse* and *count* primitives, to group and count all unique *LocationUpdates* that are in such context partition sub-stream in the past Δ time interval, as shown in Code 4.4. *Collapse* puts the grouped events into a single set, while the *count* primitive is used to compute the cell density, *i.e.*, the number of unique *LocationUpdate* events in that specific grid cell. Finally, as a result, this continuous query creates a complex *CellContent* event that summarizes the grid cell information, including its (i, j) index, its density value, and a set that contains all unique *LocationUpdate* events witnessed during the last Δ time interval.


```
1 CONTEXT CellContext
2 INSERT INTO CellContent
3 SELECT i, j, COLLAPSE(*) AS locationUpdates, COUNT(*) AS density
4   FROM LocationUpdate [SLIDING Δ period, LAST UNIQUE id]
```

Code 4.3: Cell Density EPA (in EPL).

Likewise core moving objects in DBSCAN, grid cells whose density value is greater than or equal to $minPts$ are considered dense and are classified as core, since the maximum distance between moving objects inside such cell is ε. Therefore, to discover core cells, DG2CEP filter the *CellContent* event stream to identify events whose density value is equal or higher than $minPts$. This simple task can be continuously and timely done in CEP through the usage of the *filter* and *project* primitives, as shown in Code 4. If the analyzed *CellContent* event density surpass the $minPts$ threshold, the EPA produces a derived complex CellCore event to indicate that the grid cell is dense.

```
1 INSERT INTO CellCore
2 SELECT i, j, locationUpdates, density
3 FROM CellContent
4 WHERE |density| ≥ minPts
```

Code 4.4: Cell Density EPA (in EPL).

Analogously to DBSCAN, in DG2CEP, the simplest cluster is formed by the combination of a core grid cell ($\mathcal{C}$) and its neighboring border cells ($\mathcal{N}$), which are then further visited in a later part of the algorithm. Thus, DG2CEP needs to *enrich* the *CellCore* events with its adjacent border cells to create a complex event named *DenseCellCluster*, to be further analyzed and expanded in DG2CEP. This task can be designed as an EPA continuous query that joins the incoming *CoreCell* event with existing *CellContent* events received in the past Δ period. It produces an intermediary result that pairs the core cell $\mathcal{C}$ with the existing G_{ij} cells. Then, it *filters* the pairs that contain neighboring cells, *i.e.*, those that are in the border of the incoming core cell index. A neighboring border cell is one that has both cell index (i, j) different, but each index differs by one value, at most. Finally, DG2CEP uses the *aggregate collapse* primitive to collect all the neighboring cells in a set $\mathcal{N}$. As a result, the continuous query produces the complex *DenseCellCluster* event, which contains the *CellCore* event ($\mathcal{C}$) and the resulting collection set containing its neighboring cells ($\mathcal{N}$). A description of this EPA in EPL is shown in Code 4.5.

```
1 INSERT INTO CellCluster
2 SELECT C, COLLAPSE(OtherCell.*) AS N
3 FROM CellCore AS C
4       CellContent [SLIDING Δ period, UNIQUE (i, j)] AS OtherCell
5 WHERE (C.i ≠ OtherCell.i OR C.j ≠ OtherCell.j)
6   AND (|C.i − OtherCell.i| ≤ 1 AND |C.j − OtherCell.j| ≤ 1)
```

Code 4.5: Cell Cluster EPA (in EPL).

The degree of parallelism of the *Cell* processing stage is associated with the number of grid cells, that is, this stage can process in parallel one event for each grid cell index since its computation is based on each grid cell. Specifically, they can process an event of each grid cell at the same time. This upper limit is used to avoid inconsistency. For example, consider that two *LocationUpdate* events from different moving objects, but mapped to the same grid cell $\langle i, j \rangle$ index, are being processed in parallel. In this scenario, due to the stateful nature of the *aggregate count* primitive, it may miss the count of the other *LocationUpdate* that is being processed in parallel. However, it is important to note that this degree of parallelism refer to each EPA and not the entire EPN, that is, several events with the same cell index can coexist in the EPN, but at different EPAs stages, in the pipeline. Precisely, once an event with grid cell index i and j leaves a given EPA, other event with the same index can be processed in that stage.

4.2.2 Sparse Cell Discovery

It is important to detect when a dense cell becomes sparse, that is, when the density of the corresponding cell drops below the $minPts$ parameter. Such situation happens whenever moving objects changes grid cell or if they stop sending their position.

To detect when a moving object changes its grid cell, DG2CEP stores their latest grid cell in a map $\mathcal{L}$, as shown in line 30 of Algorithm 3. For instance, if the latest location update of a moving object with id equal to 7 was placed in grid cell $(4, 9)$, then $\mathcal{L}(7) = (4, 9)$. DG2CEP checks if a moving object has changed it cell by comparing if its incoming *LocationUpdate* event cell index differs from its than its previous *LocationUpdate*, more specifically, if $\mathcal{L}(id) \neq \langle i, j \rangle$, where i and j are the current location update cell indexes. This situation can be timely verified by an EPA continuous query using the CEP *sequence* and *filter* primitives, as shown in Code 4.6. To do that, first the EPA employs the select primitive to extract consecutive *LocationUpdate* events. Then, it *filters* the event if their grid cell index differs. As output, the EPA produces a complex *CellRecheck* event containing the moving object previous grid cell index (i, j). This event is further analyzed with the intent to verify if the cell is dense *w.r.t.* $minPts$.

```
1 INSERT INTO CellRecheck
2 SELECT prev.i AS i, previous.j AS j
3 FROM PATTERN [prev = LocationUpdate → current = LocationUpdate]
4 WHERE (prev.id = current.id)
5   AND (prev.i ≠ prev.i OR prev.j ≠ current.j)
```

Code 4.6: Cell Changed EPA (in EPL).

Whenever DG2CEP detects that a moving object changed from its previous grid cell P to a new one it rechecks the density of the previous cell. If the cell density is equal to $minPts$, it indicates that the P grid cell cluster will disperse after removing the *LocationUpdate* event. This scenario can be detected in CEP through the usage of the *join*, *filter*, and *project* primitives, as shown in Code 4.7. It *joins* the incoming *CellRecheck* event with the *CellContent* event stream producing an intermediary result that pairs the cell index to be rechecked with the existing *CellContent* events. Then, the EPA uses the *filter* primitive to correlate and extract the corresponding cell density using the *CellRecheck* indexes. Finally, if P density value is equal to $minPts$ the continuous query generates a complex *DispersedCellCluster* event to indicate that the cell will become sparse.

```
1 INSERT INTO DispersedCellCluster
2 SELECT CellDensity
3 FROM CellRecheck AS P
4       CellContent [SLIDING Δ period, LAST UNIQUE (i, j)]
5 WHERE (P.i = CellContent.i AND P.j = CellContent.j)
6  AND (|P.density| = minPts)
```

Code 4.7: Cell Disperse EPA (in EPL).

Moving objects can also stop sending *LocationUpdate* events, which can lead to a cell cluster becoming invalid even though they have not changed cells. DG2CEP is able to timely detect such situation through the usage of CEP sequence, negation, and absence pattern primitives combined with a Δ sliding window period, as shown in Code 4.8. This EPA verifies if a previous *DenseCellCluster* event in a grid cell (i, j) is not followed by another such event, *i.e.*, in the same grid cell index, during a Δ period. The absence of this event throughout this period indicates that the number of location updates mapped to the cell (context partition) dropped to less than $minPts$, and that it is no longer a cell cluster. By setting up a Δ time window value equal to the moving objects' location update frequency, DG2CEP makes sure that all location updates have been considered. Finally, the continuous query produces a complex *DisperseCellCluster* event containing the analyzed cell to indicate that it is no longer dense.

```
1 INSERT INTO DispersedCellCluster
2 SELECT PrevCellCluster.core
3 FROM PATTERN [
4     DenseCellCluster AS PrevCellCluster →
5     (Δ period AND NOT DenseCellCluster AS FollowCellCluster)
6 ]
7 WHERE PrevCellCluster.x ≠ FollowCellCluster.x AND
8       PrevCellCluster.y ≠ FollowCellCluster.y
```

Code 4.8: Cell Disperse by Time EPA (in EPL).

Similar to the dense cell discovery, the degree of parallelism for the sparse detection phase is associated with the number of context partitions (grid cells). It can process an event for each grid cell in parallel without having collateral effects. Further, the sparse detection algorithm can be processed in parallel with the dense cell discovery one, that is, both algorithms can be processed

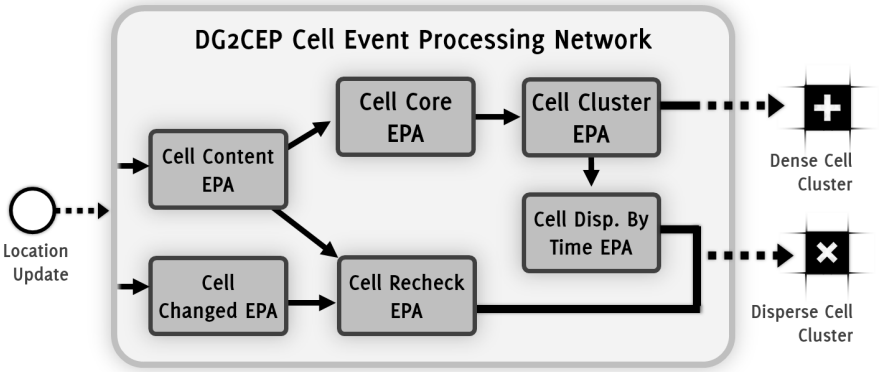


Figure 4.5: Cell Event Processing Network.

at the same time. Figure 4.5, express this relationship, where both, density and disperse EPAs, are executed in parallel in the Cell Event Processing Network. In addition, it illustrates the Cell EPN and how the described EPAs are interconnected to continuously and timely detect the formation of dense and disperse cells.

4.3 Grid EPN

With the EPAs presented so far, DG2CEP is only detecting the formation and dispersion of individual cell clusters. However, in order to detect spatial clusters of arbitrary shapes, DG2CEP also needs to implement the successive merge and unmerge of cells, similar to DBSCAN expansion step. Figure 4.9, illustrates an overview of the Grid EPN structure with cell cluster and disperse events as input, while having grid formation, merge, update, split, and destroy events as output.

The *Grid* EPN is responsible for receiving and handling cell cluster and disperse events to create, destroy, and evolve spatial grid clusters. In terms of CEP workflow this boils down to merging and expanding grid clusters when receiving *DenseCellCluster* events, while removing cells from, and occasionally splitting, existing grid clusters when receiving *DisperseCellCluster* events.

Analogous to DBSCAN, where clusters are collections of density-connected core and border moving objects, in DG2CEP, grid clusters are the resulting combination of one or more adjacent *DenseCellCluster* events. In turn, each *DenseCellCluster* event contains a core grid cell and its corresponding neighbors. Thus, a grid cluster contains core cells (from the cell cluster events) and border cells (the core cells neighbors). Finally, each cell contain the individual moving objects location update. The remaining subsections will discuss how to build and manage such clusters.

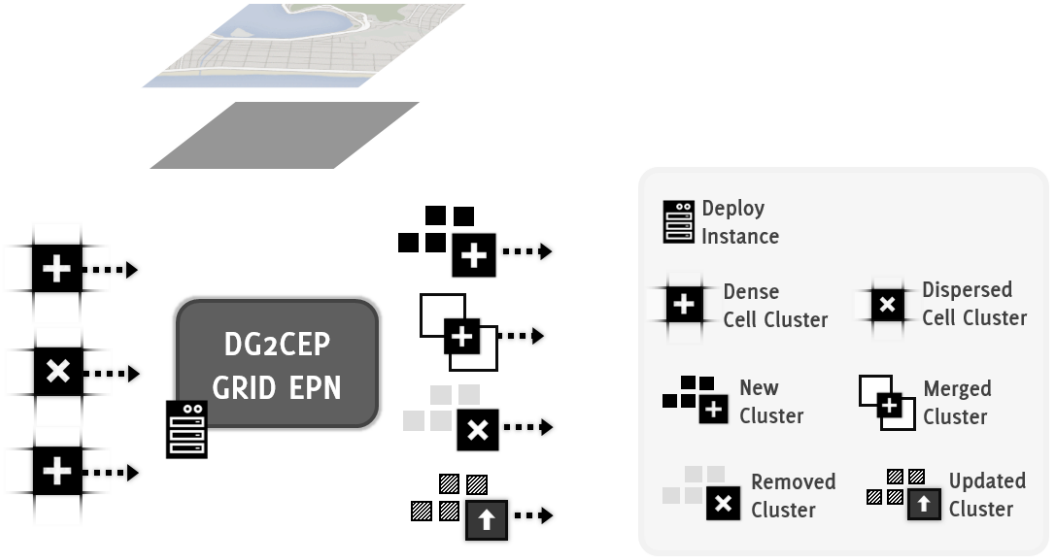


Figure 4.6: Overview of *Grid* EPN.

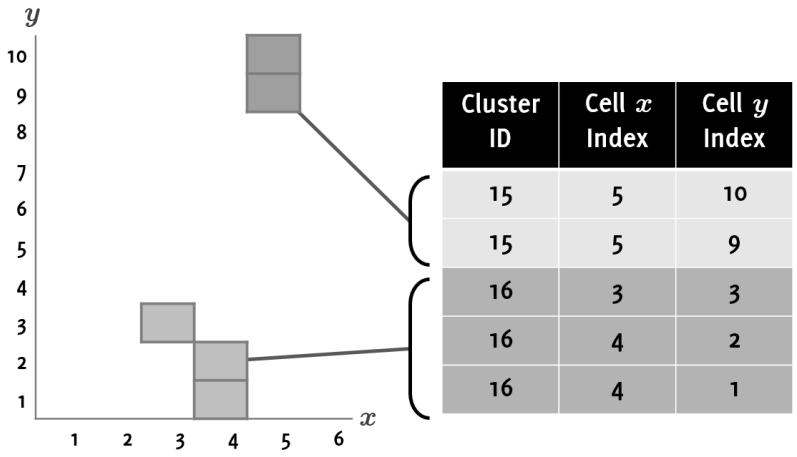
4.3.1 Grid Cluster Representation

To represent a grid cluster in CEP, DG2CEP uses a streaming relation. In CEP this concept is known as stream windows and implemented as an in-memory table. The difference between a streaming relation (stream window) and a conventional database relation is the distinctive capability of referencing and manipulating the relation content within continuous queries and usage of context windows, *e.g.*, time and partition. Thus, by storing the current grid cluster list in a stream relation DG2CEP is able to search, add, update, and remove clusters within its other continuous queries.

The cluster streaming window schema is defined as follow: $\langle cid, x, y \rangle$, where *cid* is the cluster identifier and x and y are the core cell cluster index[2]. To exemplify this schema, consider the cluster streaming window illustrated in Figure 4.7. Here, a grid cluster with $cid = 15$ contains two core cells, with indexes $(5, 9)$ and $(5, 10)$, while the cluster with $cid = 16$ contains three core cells: $(3, 3)$, $(4, 2)$ and $(4, 1)$. The name window only index the grid clusters core cells, since its border ones can be easily retrieved through the core cells adjacency.

The following subsection will discuss how such structure can be managed in parallel. Further we also discuss how to correlate incoming *DenseCellCluster* and *DispersedCellCluster* events with the *Clusters* streaming window.

[2]We use x and y to represent a cluster core cell in the streaming window to avoid confusion with the incoming grid cell i and j indexes



Cluster ID	Cell x Index	Cell y Index
15	5	10
15	5	9
16	3	3
16	4	2
16	4	1

Figure 4.7: An example of the *Clusters* streaming window.

4.3.2 Grid Add, Update, and Merge

Algorithm 4 describes how DG2CEP expands *DenseCellCluster* events to create and evolve grid clusters. First, DG2CEP unwrap the complex event and adds its core and border cells to a grid G. Then, before further processing the event, it has to verify if it is creating, augmenting, merging, or just updating an existing grid cluster. We decide to untangle these different cases by checking for a cluster update since it is the most frequent situation, *i.e.*, when a cell cluster belongs to a grid cluster and updates its content, *e.g.*, the number or location of moving objects.

DG2CEP updates an existing cluster if it contains a core cell which index is equal to the incoming event cell This can be computed by evaluating if $\sigma_{(x=i,y=j)}(Clusters) \neq \varnothing$, that is, using the relational algebra select operator over the *Clusters* streaming window. In the case that result is non empty, it means that G_{ij} is contained and is a core cell in the given cluster. Thus, an *OutputGridCluster* event is emitted with the cluster ID alongside an `UPDATE` tag to indicate that it has updated. An output EPA, which is discussed later, consume this event to produce the resulting cluster.

However, when no existing clusters contains the incoming cell event index, $\sigma_{(x=i,y=j)}(Clusters) = \varnothing$, it means that either the core cell can form a cluster or can be merged to an existing one. Nevertheless, in both cases, the conclusion of such operation is a new cluster, either one with this single cell or the result of the merged one. Thus, DG2CEP insert a new grid cluster in the streaming window containing the incoming cell indexes and a new cluster id. Subsequent EPAs will process the input event in the *Clusters* streaming window to filter out each case, either add or merge.

Algorithm 4: DG2CEP (*Grid Add/Merge/Update*)

Input: A stream $\mathcal{D}$ of *DenseCellCluster* $\langle \mathcal{C}, \mathcal{N} \rangle$, the ε threshold, and the latitude and longitude intervals

Output: Continuously output pairs $\langle \mathcal{G}, \mathcal{S} \rangle$ of grid clusters $\mathcal{G}$ and their corresponding semantic $\mathcal{S}$

$G \leftarrow$ **create** a grid dividing the lng (i) and lat (j) intervals by $\varepsilon\sqrt{2}$
Clusters $\leftarrow$ **create** a relation $\langle cid, x, y \rangle$ to store the grid clusters cells
while *data stream* $\mathcal{D}$ *is active* **do**
    DenseCellCluster $\leftarrow$ **read** dense cell cluster $\langle \mathcal{C}, \mathcal{N} \rangle$ event from $\mathcal{D}$
    add the dense core cell $\mathcal{C}$ and its neighbors $\mathcal{N}$ cells to G
    $i, j \leftarrow$ the dense core cell index $\langle \mathcal{C}_i, \mathcal{C}_j \rangle$
    ▷ *Now we check if the core cell belong to a grid cluster in Clusters*
    if $\sigma_{x=i,y=j}(Clusters) \neq \varnothing$ **then** G_{ij} is in a grid cluster
        ClusterID $\leftarrow \pi_{cid}(\sigma_{x=i,y=j}(Clusters))$
        $\mathcal{S} \leftarrow$ `UPDATE`
    else G_{ij} is not in any existing grid cluster
        ClusterID $\leftarrow$ generate a new cluster ID
        Clusters $\leftarrow Clusters \cup \{\langle ClusterID, i, j \rangle\}$
        $\mathcal{N}' \leftarrow \varnothing$ ▷ *Check if it is neighbor of an existing grid cluster*
        foreach *adjacent cell* $\langle i', j' \rangle$ *from* G_{ij} **do**
            $\mathcal{N}' \leftarrow \mathcal{N}' \cup \pi_{cid}(\sigma_{x=i',y=j'}(Clusters))$
        end
        if $\mathcal{N}' = \varnothing$ **then** there are no grid clusters adjacent to G_{ij}
            $\mathcal{S} \leftarrow$ `ADD`
        else
            $\mathcal{S} \leftarrow$ `MERGE`
            foreach *neighboring cluster id ncid in* $\mathcal{N}'$ **do**
                GridClusterCoreCells $\leftarrow \sigma_{cid=ncid}(Clusters)$
                foreach *gric cell gc in GridClusterCoreCells* **do**
                    ▷ *Update the tuple cid to the new ClusterID*
                    Clusters $\leftarrow Clusters - \{\langle ncid, gc_x, gc_y \rangle\}$
                    Clusters $\leftarrow Clusters \cup \{\langle ClusterID, gc_x, gc_y \rangle\}$
                end
            end
        end
    end
    ▷ *Retrieve the grid cluster core and border cells using ClusterID*
    $\mathcal{G} \leftarrow \varnothing$ ▷ *Grid cluster cells*
    GridClusterCells $\leftarrow \sigma_{cid=ClusterID}(Clusters)$
    foreach *core grid cell gc in GridClusterCoreCells* **do**
        $\mathcal{G} \leftarrow \mathcal{G} \cup \{G_{gc_i,gc_j}\}$
        foreach *adjacent cell* $\langle i', j' \rangle$ *from gc* **do**
            $\mathcal{G} \leftarrow \mathcal{G} \cup \{G_{i',j'}\}$
        end
    end
end
emit *GridCluster* ($\langle \mathcal{G}, \mathcal{S} \rangle$) event

To avoid having inconsistency issues when processing parallel events, we decided to apply a lock for this rule for each cell index. This means that the degree of parallelism associated with this EPA is also the number of cells. While an event with cell index G_{ij} is being processed, subsequent events with index i and j are queued, while events with different cell indexes can be process in parallel.

Now to continuously and timely express this decision, to either update or add/augment an existing cluster, DG2CEP uses EPA described in Code 4.9. This EPA uses the SQL `MERGE` primitive which atomically update or insert a tuple in a relation depending on a given criteria, in our case, if the incoming core cell exists in the *Clusters* streaming window, that is, if there is a *Clusters* tuple with index x and y equal to the incoming core cell i and j index.

```
1  ON DenseCellClluster AS denseCell
2  MERGE INTO Clusters
3  WHERE denseCell.x = Clusters.x AND denseCell.y = Clusters.y
4  WHEN MATCHED
5      THEN INSERT INTO OutputGridCluster
6                SELECT Clusters.cid, "UPDATE"
7  WHEN NOT MATCHED
8      THEN INSERT INTO Clusters, HandleNewGridCluster
9                SELECT nextClusterID++ AS cid,
10                      denseCell.i AS x, denseCell.j AS y
```

Code 4.9: Grid Cell Check EPA (in EPL).

This EPA can be expressed using the *join*, *filter*, and *project*. Precisely, it *joins* the incoming *DenseCellCluster* events with the *Clusters* streaming window using the core cell indexes. If there is a match, that is, if the core cell indexes is already in a give grid cluster the EPA outputs a complex *OutputGridCluster* update event by using the corresponding grid cluster id. As said, this event is later consumed by other EPA to build the grid cluster output. However, if there is no match, then the EPA insert a new entry to the *Clusters* streaming window containing the incoming core cell indexes and a newly generated cluster id. It also generates a complex *HandleNewGridCluster* event with the same content to handle and process the new grid cluster, *i.e.* to either add or merge it with existing clusters.

To completely either add or merge a cluster and avoid inconsistency it is necessary to lock the *Clusters* streaming window. During this period the remaining EPAs will only be able to read entries from the streaming window. It

is important to note that this lock only happens when modifying the structure of the clusters, *e.g.*, merging a cluster, which usually does not happen frequently.

Now, DG2CEP needs to process the newly created grid cluster, *i.e.*, the one inserted in the *Clusters* streaming window using the incoming *DenseCellCluster* event indexes. Thus, first, DG2CEP identify if there are adjacent grid clusters to the newly created one. This task can be done by querying the Clusters streaming window using each border cell index (i', j'), *e.g.*, using the relational algebra selection and project primitives $\pi_{cid}(\sigma_{(x=i',j')}(Clusters))$.

This can be easily expressed in CEP through the usage of a *join*, *filter*, and *project* primitive, as shown in Code 4.10. The continuous query joins the incoming *HandleNewGridCluster* events with the *Clusters* named window using the cell indexes. The intermediary result of this join is a set of pairs containing the newly dense cell cluster with the existing grid clusters core cells. Then, the EPA use the filter primitive to extract that pairs whose cell indexes are different and differ at most by one. Such pairs, *i.e.*, grid clusters are considered merge candidate. For each merge candidate, *i.e.*, neighboring grid cluster, the EPA uses a project primitive to extract its current cluster ID and the newly created cluster ID. Then, it emits a complex *MergeCluster* event with these two IDs for each merge candidate. However, if there is no match, that is, there are no border grid cluster, then CEP creates and emits a complex *OutputGridCluster* event with the newly created cluster ID indicating that the grid cluster has been added.

```
1 INSERT INTO MergeCluster
2 SELECT cellCluster.cid AS newClusterID, gc.cid AS oldClusterID
3 FROM HandleNewGridCluster AS cellCluster, Clusters AS gc
4 WHERE (|cellCluster.x − gc.x| ≤ 1 AND |cellCluster.y − gc.y| ≤ 1)
5   AND (|cellCluster.x − gc.x| ≠ 1 AND |cellCluster.y − gc.y| ≠ 1)
```

Code 4.10: Grid Check Merge EPA (in EPL).

If the EPA returns an empty set $\mathcal{N}'$ of neighbors, that is, if there are no grid clusters in the streaming window which are neighbors of the incoming dense cell, then there is no need to merge the recently created cluster. However, if the $\mathcal{N}'$ set is non-empty, DG2CEP will merge the neighboring grid clusters in $\mathcal{N}'$ to the newly created cluster. The result of this merge is a grid cluster composed by the newly created cluster and the union of all its neighboring grid clusters. This is due to the newly created grid cluster serving as a link to connect all its neighboring grid clusters.

To efficiently merge the cluster, DG2CEP needs to update up all core cells of adjacent grid clusters to the cluster id of the newly created one (from the DenseCellCluster event). Hence, the algorithm removes from the *Clusters* streaming window the old grid cluster core cells and add it using the new cluster ID. This task can be continuous and timely done in CEP through the usage of the *join*, *update*, and *project* primitive, as described in Code 4.11. The continuous query reacts to the incoming *MergeCluster* event by updating the Clusters streaming window to its new cluster ID.

```
1 ON MergeCluster AS mergedCluster
2 INSERT INTO OutputGridCluster
3 UPDATE Clusters
4   SET cid = mergedCluster.newClusterID
5 WHERE cid = mergedCluster.oldClusterID
```

Code 4.11: Grid Merge EPA (in EPL).

To exemplify this process, consider the scenario illustrated by Figure 4.8. In this case, the *Clusters* streaming window contains tree grid clusters. Further, consider that the incoming *DenseCellCluster* event is the hashed cell, with cell index equal to $(4, 4)$. According to the described EPAs, DG2CEP first verifies if the incoming *DenseCellCluster* event is already contained within a given grid cluster. In this case, the incoming dense cells is not located in any grid cluster. Thus, DG2CEP creates a new grid cluster using the incoming event cell and a new cluster ID, *e.g.* 17. Then, it looks for adjacent grid clusters in its neighboring cells. In this case, there are three adjacent grid clusters. Thus, DG2CEP produces three *MergeCluster* events pairing the new cluster ID (17) with adjacent grid cluster IDs (12, 14, and 16). The resulting of this merge is a single grid cluster with ID 17 and composed by the union of all adjacent grid clusters (12, 14, and 16), since the incoming core cell interconnect them. To do this change, DG2CEP only needs to swap the grid clusters IDs, as shown in the EPA described in Code 4.11.

Finally, in addition to the event semantics (*e.g.*, add, update, merge), the grid cluster output should also include the core and border cell contents. This can be done through querying the *Clusters* streaming window using the old cluster ID, in case of an update, or its new one, in the case of an add or merge occurrence. After that, DG2CEP build the grid cluster by retrieving each core cell and border contents. The resulting grid cell set is wrapped alongside the event semantic in a complex *GridClusterOutput* event. This event is intended to be consumed by endpoint applications or to be further processed by other EPAs continuous queries.

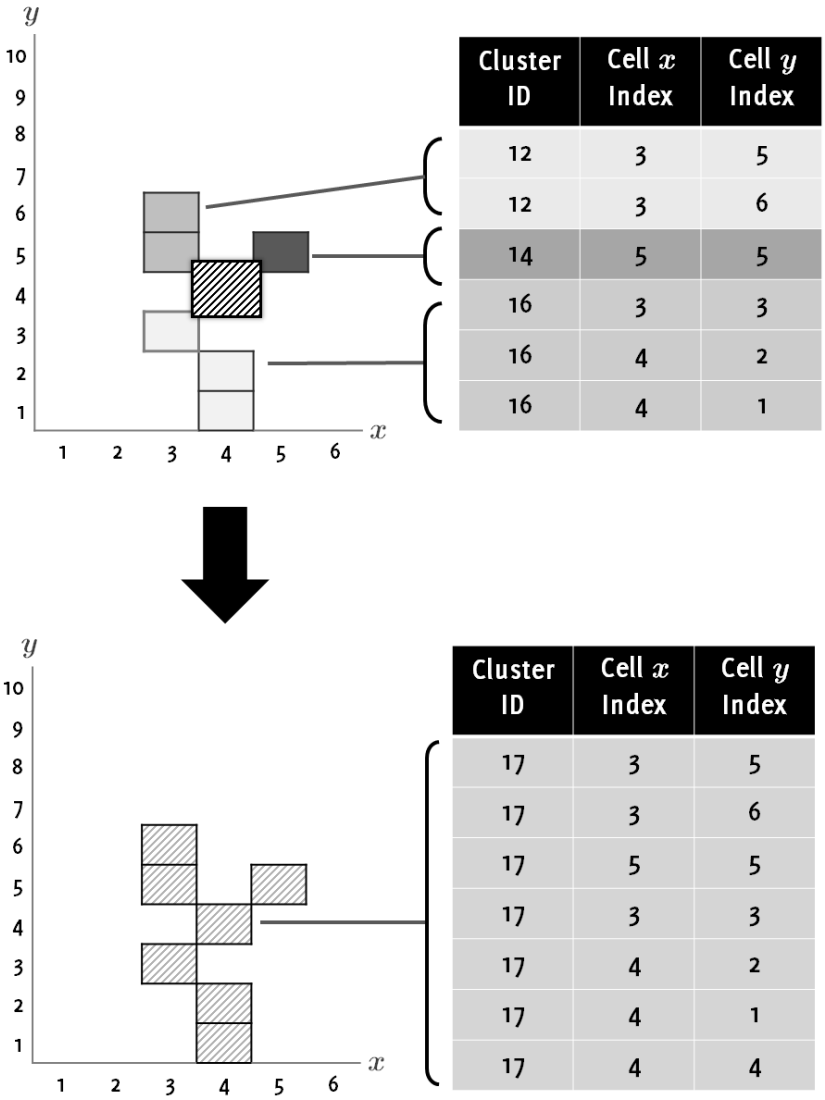


Figure 4.8: Sample scenario of merging grid clusters in DG2CEP.

The construction of the *GridClusterOutput* can be expressed in CEP through the usage of the *join*, *filter*, *aggregate*, and *project* primitives, as described in Code 4.12. An EPA *joins* the incoming *GridCluster* event with the *Clusters* relation with a sliding window of the *CellContent* event stream. This sliding window only includes unique *CellContent* events in the past Δ period. Then, the *filter* primitive matches the incoming GridCluster cluster ID with the *Clusters* core cells cluster ID. In turn, all *CellContent* events whose cell indexes are within a maximum distance of one, *w.r.t.* the core cells index, are selected. Then, the *aggregate collapse* function group the selected cells in a single set. Such set is projected alongside the clustering semantic into the final complex *GridClusterOutput* event. This event can be consumed by endpoint applications or be further processed by other EPAs.

```
1 INSERT INTO GridCluster
2 SELECT COLLAPSE(cell.*) AS Cells, gridCuster.semantic AS Semantic
3 FROM OutputGridCluster AS gridCuster,
4       Clusters AS cw,
5       CellContent [SLIDING Δ period, LAST UNIQUE (i, j)] AS cell,
6 WHERE cw.cid = gridCluster.cid
7       AND |cw.x − cell.i| ≤ 1 AND |cw.y − cell.y| ≤ 1
```

Code 4.12: Grid Output EPA (in EPL).

4.3.3 Grid Disperse

When a cell cluster disperses, *i.e.*, when receiving a *DisperseCellCluster* event, it is necessary to timely reflect this change in the *Clusters* streaming window, as described in Algorithm 5. To do so, DG2CEP needs to identify the cluster that contains the dispersed cell. Since this change will alter the *Clusters* window, it is necessary to lock it during this timely operation to avoid inconsistencies.

After locking the streaming window an EPA extracts the grid cluster id of the cell represented in the *DisperseCellCluster* event. This is done by comparing the incoming disperse cell index to the grid clusters' core cells indexes in the *Clusters* streaming window. This operation can be continuous and timely processed in CEP through the usage of the *join*, *filter*, and *project* primitives, as shown in Code 4.13.

```
1 INSERT INTO DisperseCluster
2 SELECT cluster.cid AS ClusterID
3 FROM DisperseCellCluster AS disperse,
4       Cluster AS cluster
5 WHERE |disperse.x − cluster.x| = 0 AND |disperse.y − cluster.y| = 0
```

Code 4.13: Grid Discover EPA (in EPL).

Incoming *DisperseCellCluster* events are joined with the *Clusters* window by correlating the disperse cell index with existing grid cluster core cells indexes. Then, the output of such continuous query is the id of the grid cluster that contains the dispersed cell. The EPA computation is equivalent to the following relational algebra equation: $ClusterID = \pi_{cid}(\sigma_{x=i,y=j}(Clusters))$. Using the dispersed grid cluster id, DG2CEP selects all its core cells. This is done to verify if the cluster in question will be completely removed or split when removing the dispersed cell. The EPA described in Code 4.14 does this task.

Algorithm 5: DG2CEP (*Disperse*)

Input: A stream $\mathcal{D}$ of *DispersedCellCluster* $\langle \mathcal{C} \rangle$, the ε and $minPts$ thresholds, Δ period, and the latitude and longitude intervals
Output: Continuously output zero or more pairs $\langle \mathcal{G}, \mathcal{S} \rangle$ of residual grid clusters $\mathcal{G}$ and their corresponding semantic $\mathcal{S}$

```
1  G ← create a grid dividing the lng (i) and lat (j) intervals by ε√2
2  Clusters ← create a relation ⟨cid, x, y⟩ to store the grid clusters cells
3  while data stream 𝒟 is active do
4      DispersedCellCluster ← read dispersed cell ⟨𝒞⟩ event from 𝒟
5      i, j ← the dispersed cell index ⟨𝒞_i, 𝒞_j⟩
6      ▷ Now we check if the dispersed cell belong to a grid cluster
7      if σ_{x=i,y=j}(Clusters) ≠ ∅ then G_ij is in a grid cluster
8          ClusterID ← π_cid(σ_{x=i,y=j}(Clusters))
9          ▷ Remove the dispersed cell from the given grid cluster
10         Clusters ← Clusters − {⟨ClusterID, i, j⟩}
11         ▷ Retrieve the residual core cells
12         CoreCells ← σ_{cid=ClusterID}(Clusters)
13         ▷ Compute the residual grid clusters
14         ℛ ← ∅
15         foreach residual core cell c from CoreCells do
16             foreach residual grid cluster r in ℛ do
17                 if r contain a grid cell adjacent to c then
18                     ▷ Add c to the residual grid cluster
19                     r ← r ∪ {c}
20                 end
21             end
22             if no residual grid cluster contain c then
23                 ▷ Create a new residual grid cluster with c
24                 ℛ ← ℛ ∪ {r}
25             end
26         end
27         if ℛ = ∅ then there are no residual grid clusters
28             emit OutputGridCluster ⟨C, DESTROYED⟩
29         else there are residual grid clusters
30             delete all residual core cells in the Clusters relation
31             foreach residual grid cluster r from ℛ do
32                 newClusterID ← generate a new cluster ID
33                 foreach core cell c in r do
34                     Clusters ← Clusters ∪ ⟨newClusterID, c_i, c_j⟩
35                 end
36                 emit OutputGridCluster ⟨r, SPLIT⟩
37             end
38         end
39     end
40 end
```

```
1 SELECT *
2 FROM DisperseCluster AS disperse,
3       Cluster AS cluster
4 WHERE disperse.disperseClusterID = cluster.cid
```

Code 4.14: Grid Discover EPA (in EPL).

After extracting the cluster core cell, DG2CEP only needs to identify and handle possible residual (*e.g.*, split) clusters. This is done by first removing the dispersed cell and then scanning through the remainder core cells to group them into disjoint sets, as shown in the algorithm. A list of clusters $\mathcal{R}$ is created to hold the residual clusters. Each member of this list is a set r (residual cluster) whose elements are the core cell. To verify if a core cell belongs to a residual cluster, DG2CEP check if a residual cluster contains an adjacent cell to the one being analyzed. If it does the remainder core cell is added to the residual cluster, otherwise a new residual cluster is created with this cluster. After this step, the cluster containing the dispersed cluster is deleted. Then, each of the residual clusters are reinserted in the streaming window. This is done by generating a new cluster id with the combination of the core cells belonging to this residual cluster. If there are no residual grid cluster, *i.e.*, $\mathcal{R} = \varnothing$, a destroyed semantic is emitted to indicate that grid cluster has faded. Otherwise, the remainder grid clusters are reinserted into *Clusters* and semantically considered a split, *i.e.*, a sub-set of the original grid cluster.

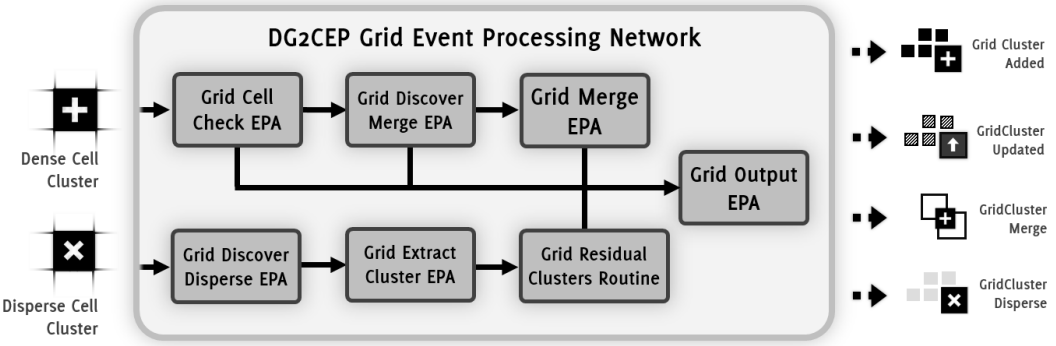


Figure 4.9: Grid Event Processing Network.

This last algorithm and EPA cannot be easily paralyzed since it modifies the grid cluster structure, *i.e.*, the *Clusters* streaming window. During the EPA processing, it is required to lock the streaming window for removing the grid cluster that contains the dispersed cell. This operation can be quickly done since it is based on comparing the cell indexes, from the dispersed cell and current grid clusters. Similarly, residual grid clusters can be quickly computed since it,

also, only uses the cell indexes. It is important to note that the lock duration only impacts EPAs responsible for adding or merging new grid clusters to existing ones, which usually does not happens frequently. One way to mitigate the locking issue is to deploy the Grid EPN, illustrated in Figure 4.9, in different machines. By restricting the deploy instance to a subset of the spatial domain the disperse cell lock duration will only affect the grid cluster add/merge EPAs of that instance.

4.4 Discussion

A primary benefit of DG2CEP when compared to DBSCAN is that it substitutes the distance comparison problem to counting. The DBSCAN clustering approach requires, for every location update, a pairwise comparison between all the moving objects, which has quadratic complexity. By using an $\frac{\varepsilon}{\sqrt{2}} \times \frac{\varepsilon}{\sqrt{2}}$ square shaped grid cells, DG2CEP reduces the problem to counting the number of moving objects that fall into each partition. Similar to DBSCAN, in DG2CEP still needs to expand the core grid cells, *i.e.*, those that are dense *w.r.t.* the $minPts$ parameter, but this process is much simpler and more efficient as they are disposed in a grid. Hence, the main performance factor is not anymore the number of moving objects location updates but instead the number of grid cells g, or context partitions, which solely depends on ε.

The tuning of the ε parameter and the frequency of location updates sent by moving objects for a concrete application (in terms of the objects size and speed) is very complex, and requires deep knowledge and expertise in the application domain [3, 10]. Since DG2CEP is not focused on a specific application scenario, we assume that such information is well known in advance. Regarding the sliding window size (Δ), it has a direct relation with the expected frequency/periodicity of the moving objects location update. Ideally, Δ should be large enough to ensure that the latest location updates of every mobile node are being considered. Unfortunately, for real-world applications it is usually impossible to know the largest update period, since each moving object has a specific location update frequency. In practical terms, it is usually sufficient to choose the size of Δ such that a large enough percentage of moving objects (e.g., 75%) generate location updates in periods smaller than Δ.

Regarding algorithmic complexity, it is hard to calculate DG2CEP complete computational cost given the reactive and its event-based nature. Considering that most CEP real-time primitives and context operations can be implemented in near real-time [14, 66], the DG2CEP computational cost is determined by its longest event path. In worst case, a moving

object location update will pass through all EPAs that detect a grid cluster formation/dispersion. Thus, in this case, the computational cost of DG2CEP per location update, would be:

$$\begin{aligned} DG2CEP &= \mathcal{O}(StreamReceiver) + \mathcal{O}(Cell) + \mathcal{O}(Grid) \\ &= \mathcal{O}(\lg g) + \mathcal{O}(1) + \mathcal{O}(g) \\ &= \mathcal{O}(g) \end{aligned}$$

where g is the number of grid cells ("context partitions").

The *Stream Receiver* cost is associated with g, the number of grid cells (or context partitions). Using a data structure that holds intervals, such as Segment Tree or Binary Tree, DG2CEP can identify the grid cell indexes of a location update in $\mathcal{O}(\lg g)$. The computation of an approximation cell index can be done in constant time, $\mathcal{O}(1)$, since it can be directly computed through the location update latitude and longitude values, roughly the number of $\frac{\varepsilon}{\sqrt{2}}$ units required to index this position.

After mapping the moving objects location update to a grid cell, DG2CEP checks if that specific cell forms a cluster, that is, if it the number of unique location updates events mapped to the given cell within a Δ period is bigger or equal to $minPts$. When this happen, the algorithm retrieve its adjacent neighbors and builds a complex *DenseCellCluster* event to be further expanded. Further, it also checks if the previous grid cell of the moving object location update will dispersed, that is, the cell density will drop below the $minPts$ threshold. It verifies that whenever the moving object change its cell. In both cases, for detecting dense and sparse grid cells, the computation can be done in constant time since it involves basic operators.

Finally, the last algorithm part, *Grid*, consumes the dense and disperse cell events. The *DenseCellCluster* event triggers the EPA to check if the incoming dense cell will form a new grid cluster or is part of an existing one. This process requires an iteration over the existing grid clusters cells. In the worst case, each cluster could be a single cell in the grid, which would require an iteration over all g grid cells. Hence, its computational cost is $\mathcal{O}(g)$. For *DisperseCellCluster* events the process is similar. Here, the worst case scenario is that a cell becomes sparse in a grid cluster composed of every cell. In this case, the algorithm would need to iterate over all the g grid cluster cells to discover residual clusters. Thus, its computational cost is also $\mathcal{O}(g)$. Considering therefore all the aforementioned steps, the total computational cost for the entire DG2CEP algorithm is $\mathcal{O}(g)$.

However, it is important to note that the DG2CEP worst-case scenario is unlikely to occur for all events in the stream. It would require all location updates in the Δ time window to have their corresponding moving objects located in dense cells. In most cases, most location update events will pass only through the *Stream Receiver* and stop at the *Cell* part, which have a constant cost, $\mathcal{O}(1) + \mathcal{O}(1)$ if using the approximate index, or an $\mathcal{O}(\lg g) + \mathcal{O}(1)$ when using a data structure to compute the location update cell index. It is also important to note that DG2CEP's space complexity is $\mathcal{O}(n)$ since all location updates are held only for a Δ period under the assumption that the maximum update frequency of the n moving objects is Δ.

4.5 Limitations

Complementing the previous discussion, in the following we discuss some limitations of DG2CEP, which in fact, are common to all grid-based clustering approaches.

While the DG2CEP counting approach of grid cells (context partitions) gives a performance advantage over DBSCAN, it also entails what we call the *blind spot* or *answer loss* problem: the difficulty to detect a dense grid cluster when spatially and temporally close location updates are mapped to adjacent grid cells [37, 38, 67]. The blind spot problem happens in any grid-based approach because the spatial domain is segmented in ε square shaped grid cells, and moving objects that are ε distance apart from each other may be mapped to different cells, not contributing to the required $minPts$ density in a specific grid cell. For example, suppose that $minPts = 4$ and the grid cells have the following location updates illustrated in Figure 4.10. In this case, no grid cell would be considered dense since their density is below the $minPts$ threshold even though there is a clear high density of location updates in the picture close to the borders of all 4 cells.

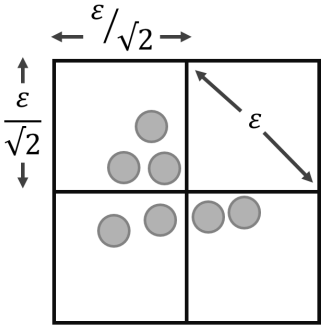


Figure 4.10: Blind Spot Scenario in DG2CEP, for $minPts = 4$.

For a $i \times j$ grid, in worst case there would be $(i-1) \times (j-1)$ *blind spots*, when all location updates would be distributed to the corners, and all grid cells would contain less than $minPts$. However, this worst case is very unlikely to happen since it would require a very specific, and regular, location update distribution of location updates, that would have to have, so to say, *knowledge* of the underlying grid.

It is possible to compare DG2CEP's and DBSCAN's clustering results. Considering that there are no *blind spots*, DG2CEP clustering results are a superset of DBSCAN one. Suppose that a grid cluster in DG2CEP has c core and b border grid cells. Hence, all location updates in the c core grid cells would also be included in DBSCAN result since they are all within the ε distance. DG2CEP default expansion includes all the border grid cells. Therefore, in the worst-case scenario, the neighbor grid cells should not be included, since their content is beyond the ε distance. To illustrate this scenario, consider the following scenario illustrated in Figure 4.11, with $minPts = 4$ and a grid cluster with two core and eight border cells.

The number of location updates detected by DG2CEP in this cluster would be $|c| + |b| = 12$, where $|c|$ and $|b|$ are the number of location updates placed in the cluster's core and border cells respectively. This result is a superset of DBSCAN's outcome, which is 9 location updates. This clustering error is limited by following equation: $|b| \times (minPts - 1)$. This means that, since DG2CEP includes all the content of neighbors partitions, in the worst-case scenario, all moving objects in the neighbor partitions are not within ε distance of the DBSCAN. However, these partitions are limited by $minPts$, otherwise they would have been included as core partitions.

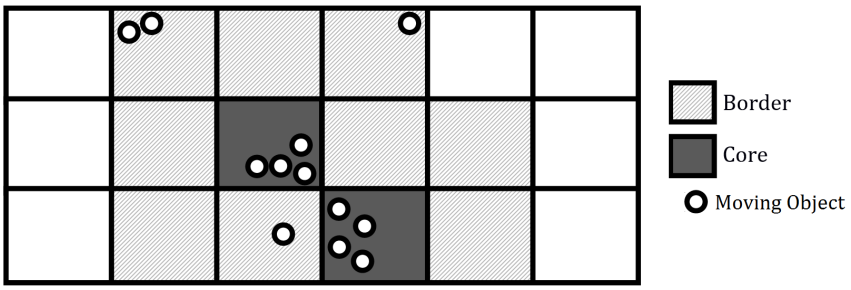


Figure 4.11: DG2CEP result as a superset of DBSCAN one.

4.6 Summary

This chapter presented DG2CEP, a grid/density-based clustering algorithm designed for large position data stream. The algorithm is expressed as

a network of CEP real-time processing primitives and employs a counting approach instead of using distance comparisons between moving objects' location updates to reduce the computational cost. In general, the overall idea is to first map the location updates to grid cells (or context partitions, *i.e.*, CEP sub-streams) instead of comparing each individual location update with the remainder ones. Then, similar to core point in DBSCAN, the algorithm filter the grid cells whose density is equal or higher than the $minPts$ threshold. These core cells form a grid cluster. Further, like DBSCAN, it expands these core grid cells to augment or split the grid cluster. During this chapter, we explained step-by-step how these tasks can be expressed as a CEP real-time primitives.

Although the counting semantics enables grid-based approaches to scale and provides faster results over other approaches, they may fail to identify some spatial clusters, a problem known as *answer loss* (or *blind spot*) [37, 38, 67], that was briefly presented in Section 4.5. Answer loss is a problem that happens due to the discrete division of the space domain into grid cells, which can lead to spatial and temporally close moving objects being mapped to different cells and, thus, not contributing for a cell density that exceeds the $minPts$ even though the objects are closer than ε to each other, as illustrated in Figure 4.10. Next chapter, will present a counting heuristic that is able to mitigate this problem and increase DG2CEP ability to detect the grid clusters that contains this situation.

5
Answer Loss Heuristic

In this chapter, we present a heuristic to address the *answer loss* or *blind spot* subproblem [37, 38, 67]. Answer loss is a subproblem that happens in DG2CEP, and other grid-based approaches, due to the discrete division of the space domain into grid cells, which can lead to spatial and temporally close moving objects locations updates being mapped to different cells. Although the moving objects are close *w.r.t.* ε, such cluster will not be detected since no grid cell is dense *w.r.t.* $minPts$, as illustrated in Figure 4.10.

The remainder of the chapter is structured as follows. Section 5.1 presents the heuristic approach, a density heuristic that is sensible to location updates in adjacent grid cells. Section 5.2 shows the usage of the heuristic in two example scenarios and discusses its effectiveness, tradeoff, and limitations. Finally, Section 5.3 discusses some related work to address this subproblem, while Section 5.4 makes the concluding remarks and summarizes the main concepts of the proposed heuristic.

5.1
Transient Heuristic

To address the answer loss subproblem, while retaining DG2CEP counting semantic, we propose a density heuristic that logically divides each grid cell into $\mathcal{S}$ inner slots (strips), in both directions, horizontal and vertical. Then, the density function counts the number of mapped location updates in those inner slots in a way that slots closer to G_{ij} have higher weight than those that are more distant. The exact process is detailed below.

Each location update mapped to a grid cell G_{ij} is also mapped to a horizontal and vertical slot index s, such that s varies from 0 to $\mathcal{S} - 1$, the first and last slot respectively. This operation can be effectively done in constant time during DG2CEP *Stream Receiver* phase by comparing the location update position with the width and length size of each slot.

After that, the heuristic need to detect the transient grid cells, *i.e.*, those whose density is less than the $minPts$ parameter, but higher than a lower-bound $lowerPts$ value. This can be continuously and timely verified in CEP through the usage of the filter and project primitives, as described in Code 5.1.

The continuous query reacts to the *CellContent* event produced by an incoming *LocationUpdate* event. If the density value is within the transient threshold interval a new complex *CellTransient* event is generated with the corresponding grid cell values.

```
1 INSERT INTO CellTransient
2 SELECT i, j, lu, density
3 FROM CellContent
4 WHERE |density| ≥ lowerPts AND |density| < minPts
```

Code 5.1: Cell Transient EPA (in EPL).

To apply the heuristic, DG2CEP needs to analyze the neighborhood of the transient grid cell. Therefore, it needs to enrich the *CellTransient* event (*e.g.*, G_{ij}) with its adjacent grid cells (*e.g.*, $N_\varepsilon(G_{ij})$). Similar to what DG2CEP does to enrich *CellCore* events into *CellCluster*, this task can be timely done by joining the incoming *CellTransient* event with the existing *CellContent* events received in the past Δ time window. Then, DG2CEP can correlate the paired cell indexes to filter and group the neighboring grid cells. Finally, DG2CEP enriches the original transient grid cell event with its neighboring grid producing a complex *CellTransientCheck* event. A description of an EPA performing this task is shown in Code 5.2.

```
1 INSERT INTO CellTransientCheck
2 SELECT G_ij, COLLAPSE(Cell.*) AS N_ε(G_ij)
3 FROM CellTransient AS G_ij,
4       CellContent [SLIDING Δ period, UNIQUE (i,j)] AS Cell
5 WHERE (G_i ≠ Cell_i OR G_j ≠ Cell_j)
6   AND (|G_i − Cell_i| ≤ 1 AND |G_j − Cell_j| ≤ 1)
```

Code 5.2: Cell Transient Enrich EPA (in EPL).

The next step is to update the clustering density function by considering the inner density of its neighboring grid cells, as illustrated by Figure 5.1. Not only the number of location updates in a grid cell G_{ij} is considered, but also the distribution of moving objects in the inner slots of each neighboring cell in $N_\varepsilon(G_{ij})$. To do this, we propose a discrete decay weight function that counts the number of location updates inside each inner slots of each neighboring grid cell in such way that slots closer to G_{ij} receive a higher weight. The heuristic density is the sum of the location updates' placed in each neighboring close slot with their corresponding weight. However, the closer slots indexes vary according to

the position of the neighboring cell, as shown by darker tones in Figure 5.1. Thus, to avoid handling the different neighboring position when computing the density function, DG2CEP "normalize" their inner slots distribution. To do so, DG2CEP reorder the neighbor cells $n \in N_\varepsilon(G_{ij})$ inner slots n_s in such a way that the first slot, $s = 0$, is closer to the evaluated cell, while the last slot, $s = \mathcal{S} - 1$, is the farthest one, to enable them to be handled as if they were aligned in the same position.

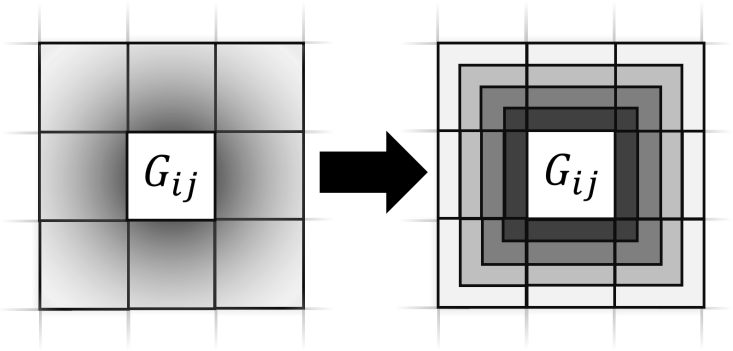


Figure 5.1: Density neighborhood of a given cell. Note that the neighbor's n closer inner slots n_s is relative to the position of G_{ij}.

After normalization, the density function can be described as:

$$density(G_{ij}, N_\varepsilon(G_{ij})) = |G_{ij}| + \sum_{n \in N_\varepsilon(G_{ij})} \left(\sum_{s=0}^{\mathcal{S}-1} n_s \times w_s \right),$$

where $|G_{ij}|$ is the number of location updates contained in G_{ij}, $n \in N_\varepsilon(G_{ij})$ is a adjacent cell neighbor, $\mathcal{S}$ is the total number of inner slots, n_s is the number of location updates in the s^{th} slot index of a neighboring grid cell n, and w_s is the s^{th} decay weight.

CEP engines implementations commonly provide the ability to encapsulate external and user defined functions as continuous query primitives[1,2]. DG2CEP makes use of this to compute and provide the density heuristic as an external filter function $density(G_{ij}, N_\varepsilon(G_{ij}))$. Then, it can refer to the heuristic function in the continuous query, as illustrated in the EPA written in Code 5.3. If the analyzed grid cell density, using the heuristic, is larger or equal to the *minPts* parameter, DG2CEP produces a *DenseCellCluster* event. To enable compatibility to EPAs that consume the dense cell cluster events, the EPA emits the same *DenseCellCluster* event type regardless if the grid cell is considered dense *w.r.t minPts* or the heuristic.

[1]Esper Functions: `http://www.espertech.com/esper/release-5.3.0/esper-reference/html/functionreference.html#epl-function-user-defined`.

[2]Apache Flink UDF: `https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/api_concepts.html` .

```
1 INSERT INTO DenseCellCluster
2 SELECT G_ij, N_ε(G_ij)
3 FROM CellTransientCheck
4 WHERE density(G_ij, N_ε(G_ij)) ≥ minPts
```

Code 5.3: Cell Transient Heuristic EPA (in EPL).

We propose two discrete weight decay functions, a linear and an exponential one, illustrated by Figure 5.2 (for $\mathcal{S} = 4$ inner slots). The linear decay weights can be computed as $w_s = \frac{-s}{\mathcal{S}} + 1$, where s is the given grid cell inner slot index. For example, considering $\mathcal{S} = 4$, the slots weights are $w_0 = 1$, $w_1 = 0.75$, $w_2 = 0.50$, and $w_3 = 0.25$. This means, that the weight of location updates placed in the closest inner slot w_0 is 1, while location updates placed in the last inner slot is 0.25. Thus, when computing the grid cell density, the number (count) of location updates in the first slot contribute directly, since they are multiplied by 1, while the sum of those placed in the last slot only contribute by 0.25 to the grid cell density.

Likewise the linear weight, the exponential decay weights can be computed as $w_s = k^s$, where k is a number between 0 and 1 such that $k^{\mathcal{S}} \approx 0$. Based on this definition, k varies accordingly to the number of inner slots $\mathcal{S}$. For example, considering that cells have $\mathcal{S} = 4$ inner slots, k value is approximately 0.3162, *i.e.*, $0.3162^4 \approx 0$, while for grid cells that have $\mathcal{S} = 10$ inner slots, k is approximately 0.6309, since $0.6309^10 \approx 0$. To discover k, one can assume $k^{\mathcal{S}} = 0.01$, then $\ln k^{\mathcal{S}} = \ln 0.01$, which yields $k = e^{(\frac{\ln 0.01}{\mathcal{S}})}$. For example, as said, for $\mathcal{S} = 4$, k is approximately 0.3162 and the slot weights are $w_0 = 1$, $w_1 = 0.3162$, $w_2 \approx 0.3162^2 \approx 0.099$ and $w_3 \approx 0.3162^3 \approx 0.0312$.

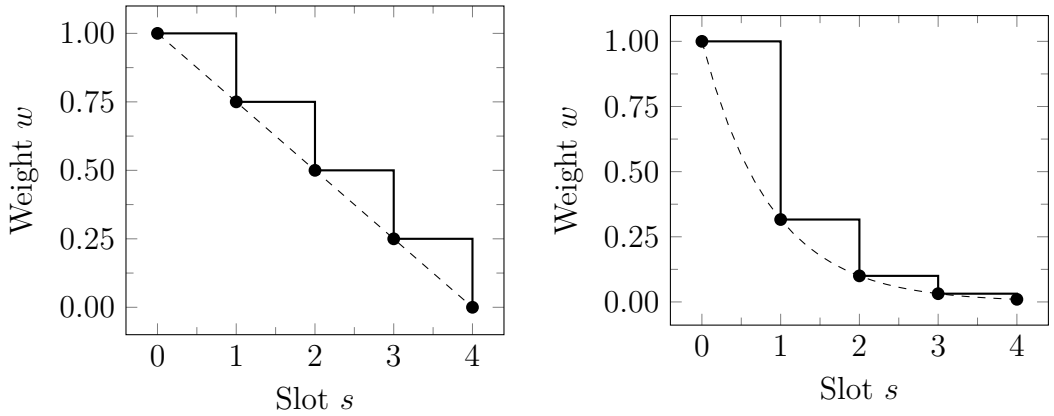


Figure 5.2: Linear and exponential weights for the heuristic inner slots ($\mathcal{S} = 4$).

5.2 Usage and Limitations

By applying the discrete weight function to the neighboring cells inner slots, the proposed heuristic can detect several answer loss clustering scenarios. For example, consider the clustering scenario of Figure 5.3 (a) and parameters $\mathcal{S} = 4$ and $minPts = 4$. Since the analyzed grid cell density is 2, thus less than $minPts$, DG2CEP would not detect the cluster. Using the proposed heuristic, with a linear weight decay, the computed density will be $2+(1\times1)+(4\times0.75) = 5 \geq minPts$, thus, the cluster would be detected. An exponential decay weight will also detect this cluster, since the computed density would be $2 + (1 \times 0.3162^0) + (4 \times 0.3162^1) = 4.26 \geq minPts$.

On the other hand, as a collateral effect of considering moving objects of neighboring cells when calculating the cell density, the proposed heuristic would detect a non-existing cluster (a false positive) in some situations, as illustrated in the cell configuration of Figure 5.3 (b), for $\mathcal{S} = 4$ and $minPts = 4$. In this scenario, DG2CEP would correctly not detect the cluster, since the cell density is 1. However, the linear weight decay would wrongly detect the cluster, since the cell density in this case would be $1+(1\times1)+(1\times0.75)+(2\times0.5)+(1\times0.25) = 4 \geq minPts$. Nevertheless, in this scenario, the exponential weight decay would correctly not detect such cluster, since the computed density would be $1+(1\times0.3162^0)+(1\times0.3162^1)+(2\times0.3162^2)+(1\times0.3162^3) = 2.54 \leq minPts$.

To mitigate the heuristic collateral effect of detecting non-existing clusters, we propose to only apply the method when evaluating transient cells, that is, cells whose density are lower than $minPts$, but higher than a lower-bound $lowerPts$ threshold, where $lowerPts \leq minPts$. By using a lower-bound threshold, we can restrict the heuristic application to specific scenarios, *e.g.*, to

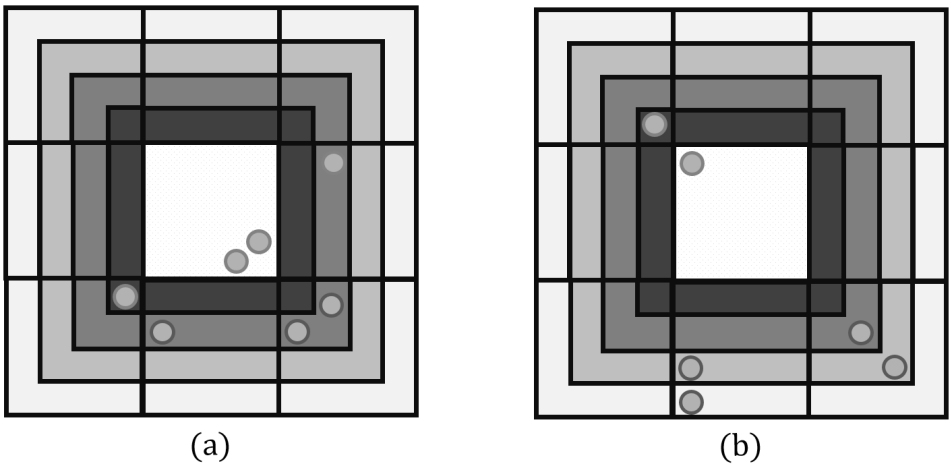


Figure 5.3: Cell configuration scenarios. In (a) the scenario forms a cluster, while in (b) it does not.

almost dense grid cells. For example, consider a $lowerPts = 2$ threshold in the clustering scenario of Figure 5.3 (b). In this configuration, the linear weight heuristic will correctly not detect the cluster, since the cell density would be 2.

In addition to mitigating false positive answers, transient cells also reduce the overall heuristic computational cost, since the heuristic will only apply to grid cells whose density are within the transient interval $lowerPts \leq minPts$. For example, a $lowerPts = 0$ means that the heuristic is applied to every moving objects location update not placed in a dense grid cell. However, using a high $lowerPts$ threshold value may again raise the answer loss problem, since the heuristic would only apply to values close to minPts. For example, in Figure 5.3 (a), for $lowerPts = 3$, the cluster would not be detected, since the heuristic density would not be applied.

5.3 Related Work

This section compares the proposed heuristic with other solutions that address the Answer Loss subproblem. In general, these approaches are aimed at off-line scenarios and extensively use spatial index and operators, contrarily to DG2CEP's and the proposed heuristic's counting semantic.

Ni *et al.* [38] presented two techniques, an exact and an approximate method that solve the answer loss subproblem in spatial dense queries. Both methods rely on a spatial index TPR-Tree, an R*-tree variant, to index the moving object trajectories. Furthermore, they extended the cell density concept to dense points, as if each point in the cell had its own ε–$Neighborhood$ radius. The first method uses a filtering and refinement strategy. Cells that contain less than $minPts$ location updates, but when combined with their neighborhood cells surpass this threshold, are filtered to be further analyzed. The refinement step applies a detailed plane-sweep algorithm to count the number of location updates in the grid cell's neighborhood. To do so, it executes a sequence of spatial-temporal range queries in the TPR-tree. Then, by combining the queries answers, they are able to discover the moving objects that are within the cell radius. Although interesting, the approach requires a sequence of range-queries and spatial index operations, during the plane-sweep step, which can be troublesome to guarantee in data streams scenarios that requires timely responses. Hence, this approach is better suited to off-line scenarios where response time is not the primary concern, but correctness.

Their second technique is an approximation method that provides a function to represent the density distribution of moving objects location updates. Contrary to their first method, this function represents the entire grid density,

rather than considering each individual cells. More precisely, the function returns all dense regions (clusters) of the grid, *w.r.t. minPts*, in a given timestamp. The function is based on Chebyshev polynomials, a recursive function that uses several geometric primitives such as cos and arccos. While using a single function to discover any dense cluster is appealing, the computational cost of such function is higher than counting the number of location updates. In addition, the function can cause an overhead or delay the response since it calculates all dense regions at once, even those that do not suffer from the answer loss subproblem.

Jeung *et al.* [67, 68] proposed an interesting supervised solution, which combines off-line processing and hidden Markov chains, for solving the answer loss problem in trajectory clustering. Their overall goal is to extract clusters from trajectories datasets. To do so, first, they preprocess a trajectory dataset using DBSCAN to discover the clusters location and their moving objects. Using the clustering result and hidden Markov chains, they create a trajectory model by discovering the set of cells and their probability to be associated with the cluster location, where the probability of each cells expresses the percentage of moving objects of such cluster in that cell. Based on this model, they can correlate the clusters location to the respective moving object cells. There are two main differences between their approach and ours. First, our proposed heuristic does not need to do any *a priori* processing. Second, their approach is focused on clusters solely from the *a priori* trajectory dataset, while our approach can dynamically discover and detect clusters from data streams.

5.4 Summary

This chapter presented a counting density heuristic that is sensitive to the number of location updates in adjacent cells. The overall idea is to further subdivide internally each grid cell into "logical" horizontal and vertical slots. Then, when computing a cell density, we consider the distribution of location updates inside the slots of adjacent grid cells. We consider two discrete functions (linear and exponential) to weight the adjacent cells inner slot distributions and to combine it with the cell's own density value.

However, since the heuristic considers moving objects in adjacent cells when calculating the cell density, it can wrongly detect a cluster that does not exists as a collateral effect. Thus, we proposed that the heuristic should be used only when the evaluated cell has a transient density, *i.e.*, the number of its objects is less than the required parameter, but larger than a lower threshold $lowerPts$.

The following chapter presents a complete evaluation of DG2CEP with and without the proposed heuristic. Overall, the evaluation investigates the elapsed time required by DG2CEP to react and detect a cluster formation/dispersion and how close this result is to the DBSCAN ground-truth result. In addition to these experiments, we discuss the tradeoff between the transient cell threshold and the similarity of the clusters found in DG2CEP to the ones in DBSCAN. We also discuss how the heuristic impacts the DG2CEP performance.

6
Evaluation

DG2CEP was evaluated using a real world data stream of position data generated by the bus fleet of the city of Rio de Janeiro[1]. With the intent to answering the thesis research questions, the evaluation had the following goals: First, measure the elapsed time required by DG2CEP to detect the formation, dispersion, and evolution of spatial clusters when compared to the baseline DBSCAN off-line algorithm. We also measure the elapsed time required by the well-known grid and batch-based D-STREAM [59] algorithm to indirectly compare it to DG2CEP. Further, we intend to measure how does the elapsed time varies according to the number of moving objects and the partition size ε.

The evaluation also measured how similar DG2CEP's clustering result is to DBSCAN results throughout the entire data stream using a second-by-second analysis. By doing that, we aim to verify if DG2CEP's clustering result is able to "keep up" to DBSCAN by measuring how their similarity evolves throughout the data stream. For this we used the *Rand Index* [69], which measures the similarity between clusters, considering the number of true positive, true negative, false positive, and false negative moving objects placed in a given cluster.

We also evaluated the proposed heuristic under different transient thresholds. Here, the goal was to compare the heuristic-enhanced DG2CEP similarity index with the original DG2CEP and DBSCAN. In addition, we evaluate if the number of inner slots impacts the number of clusters found.

The remainder of this chapter is organized as follows. Section 6.1 briefly present the main technologies and implementation aspects of DG2CEP. Section 6.2 describes the input data stream used throughout the experiments. Then, Section 6.3 presents and analyzes an experiment that evaluates the proposed heuristic effectiveness. After that, Section 6.4 presents the experiment used to compute the elapsed time required by DG2CEP to discover the spatial clusters. Following, Section 6.5 discusses the experiment used to discover the similarity index between DG2CEP, DBSCAN, and D-STREAM clustering results.

[1]The resulting dataset is available to be downloaded and reproduced at `http://www.lac.inf.puc-rio.br/dg2cep/`.

6.1 Implementation

DG2CEP was implemented using the Java programming language and several open source libraries. The reason for using Java is due to the numerous open source libraries, framework, and middleware platforms available in this language. For example, we opted to use the Esper CEP Engine [46], one of the leading open source CEP engines, which is available as a Java library. Esper provides a continuous query CQL-like declarative language that supports CEP's transformation and pattern-based primitives. We implemented DG2CEP event processing network as a network of Esper's continuous query.

DG2CEP also uses the SDDL communication middleware [70] to interconnect the different parts of the event processing network. SDDL provides publish/subscribe communication with real-time guarantees for local, mobile, and cloud services, and is based on the OMG DDS standard. SDDL is also written in Java and uses OpenSplice, an open source implementation of the DDS standard.

A central element of the DG2CEP implementation is a configuration manager that runs on each distributed instance as a daemon. This manager component implements the command pattern and can receive local and remote configuration commands such as reading and applying parameters, deploying and destroying EPAs, or creating publish/subscribe topics. This strategy enables DG2CEP to be extended by implementing new commands.

Each DG2CEP distributed instance also implements a wrapper for life-cycle management and interaction with the Esper CEP Engine, to dynamically create and destroy EPAs. All these functions are also implemented as commands. Exposing such CEP functions as commands enable EPAs to be deployed and interconnected locally or remotely, dynamically creating the necessary listener and subscriber routines to receive or route input/output events. This strategy enables DG2CEP instances to be as flexible as needed. For example, one can deploy the entire EPN in a single machine, or subdivide into different machines.

In this evaluation we also decided to implement the D-STREAM batch-based algorithm [59] with the intent to indirectly compare it to DG2CEP through their clustering result *w.r.t.* DBSCAN ground-truth. D-STREAM is a grid-based batch on/off-line clustering algorithm, which also maintains a grid that summarizes the moving object densities for each grid cell (context partition). Similar to DG2CEP, during the on-line phase, the algorithm identifies and maps each location update to a grid cell index.

During the on-line period, modified grid cells are added to a *grid list* data-structure. In the off-line phase, after the waiting period exceeds a threshold,

D-STREAM updates the grid density of modified grid cells (grid list) and cells that are in a grid cluster. After that, the algorithm analyzes both, modified and grid clusters cells, using a iterative routine. The idea is to identify grid cells that have become dense (new clusters) or sparse (dispersed) and iteratively merge or remove them from clusters. Finally, at every round the algorithm output the cell content of the current grid clusters. It is important to note that during D-STREAM's off-line phase, the entire on-line phase is blocked, since it needs to iterate through the grid state. Thus, the time required by the off-line routine delays the beginning of the next on-line phase.

6.2 Data Stream

The experiments used a real world data stream produced by the bus fleet of the city of Rio de Janeiro, Brazil. We crawled the data stream from the `data.rio` open platform and produced a dataset containing the trajectory data for the city's 11 324 buses for an one hour (from 17:30 to 18:30) for the week of July 12^{th} to 19^{th} of 2016. We choose the rush hour period because it contained the largest throughput, and probably the largest number of spatial clusters.

While crawling the data stream, we learned that, in average, each bus update its location every 60 seconds. Hence, considering 11,234 buses, each second contains in average 187 location updates. With the intent to increase the volume and provide a real-time aspect to the data stream, we augmented the data stream using linear interpolation between buses location updates in such a way that a location update is emitted every second for every bus. More precisely, between two consecutive buses location update points we generated (interpolated) additional location updates on the direct line between these two points. Using this method, we produced four resulting data streams with 2500, 5000, 7500, and 10000 location updates events per second.

We established the data stream ground truth clustering results by computing DBSCAN[2] at every second of the one hour period, as illustrated by Figure 6.1. For example, at time t, DBSCAN found cluster C1, C2, and C3. Each one with its own moving object's location updates. As a result, we have a snapshot of the spatial clusters and its content (moving object's location updates) that appears at every second of the data stream. This is an expensive computing task. In fact, it took more than 24 hours to compute the second-by-second ground truth results. Using this information, we are able to evaluate how close DG2CEP is to the optimal off-line result.

[2]For this, we adapted the Apache Math implementation of DBSCAN, for more information see: `http://commons.apache.org/proper/commons-math/`.

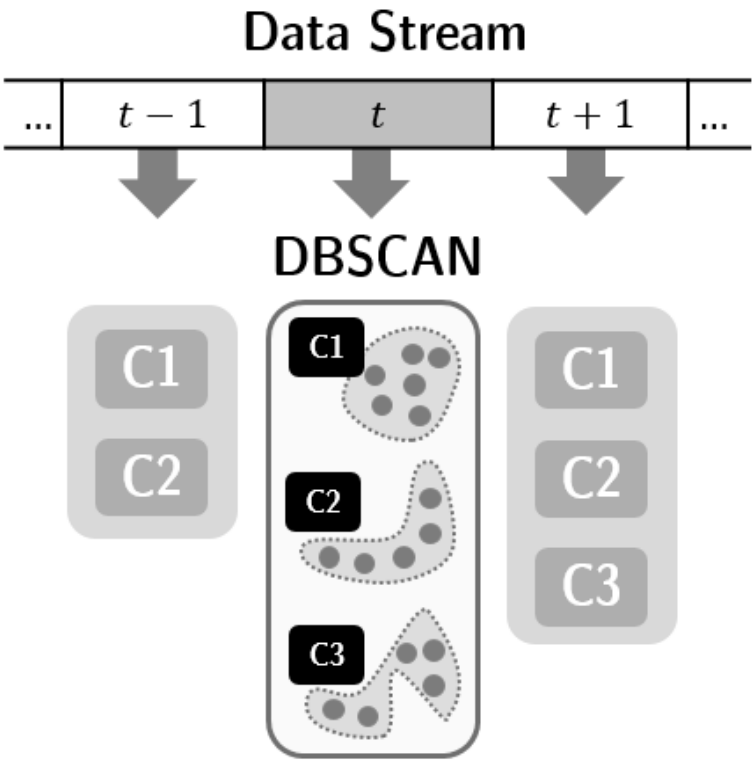


Figure 6.1: Second-by-second DBSCAN ground truth result (second t).

6.3 Answer Loss

The first experiment evaluates the proposed heuristic impact in the DG2CEP performance with respect to similarity and number of wrongly detected and undetected clusters. We start by evaluating the proposed heuristic since the later experiments will utilize the heuristic. Using the heuristic-enhanced DG2CEP, the evaluation had two goals:

- Measure the percentage and similarity of the on-line cluster results found, when compared with the original DG2CEP and the baseline DBSCAN off-line algorithm. Furthermore, investigate how these results vary with different $lowerPts$ values that define transient cells.
- Investigate if the number of correct and incorrect clusters found and their similarity with the baseline DBSCAN algorithm vary when using different number of inner slots s.

The second-by-second DBSCAN ground-truth result enabled us to compare and measure the result of the proposed heuristic clustering with the original grid-based DG2CEP and DBSCAN output. Whenever he enhanced-heuristic DG2CEP discovers a cluster we take a snapshot of its content (moving objects' location updates) to analyze at a later time. Using this information, we

compare the discovered clusters with their counterparts in the ground-truth log. A cluster C is discovered in DG2CEP at timestamp t if in the DBSCAN ground-truth log in the same timestamp t exists a cluster D such that the overlap between them is higher or equal to 50% ($C \cap D \geq 0.5$), that is, the discovered cluster in DG2CEP contain at least 50% of the content of the ground-truth cluster. If the heuristic wrongly detects a cluster, *i.e.*, no similar cluster exists in the ground-truth log, that is ($C \cap D < 0.5$), then this detected cluster is marked as false positive (FP). All clusters not detected by the heuristic but present in the ground-truth log are marked as false negatives (FN). By comparing these metrics, the percentage of incorrectly detected clusters (FP) and missed clusters (FN), to the total number of clusters in the ground-truth log, we can measure DG2CEP effectiveness of handling the answer loss problem.

6.3.1 Experiment Parameters

Since the primary interest of this experiment is to measure the heuristic impact in DG2CEP we fixed a set of values. First, we used a data stream throughput of 5 000 location updates per second, a grid size of $\varepsilon = 100$ meters, and $minPts = 20$. We also set DG2CEP's sliding window to be $\Delta = 60$ seconds, that is, we consider the location updates received within the last 60 seconds.

To measure the impact of the proposed heuristic, we considered $lowerPts$ thresholds ranging from 90% to 30% of the $minPts$ density threshold. For example, since $minPts = 20$, we evaluate the following $lowerPts$ thresholds: 18, 16, 14, 12, 10, 8, and 6. We chose to vary the $lowerPts$ threshold until it is 30%, because lower threshold values tend to produce more false positive clusters due to the heuristic collateral effects.

Finally, for investigating the transient threshold experiment we subdivide the grid cells into ten slots ($\mathcal{S} = 10$). Hence, since $\varepsilon = 100$ m, each inner slot $s = \frac{100}{\mathcal{S}}$ width is 10 meters. We choose these values considering that the GPS accuracy is approximately between 10 to 20 meters. However, in this experiment we also evaluate if the total number of slots $\mathcal{S}$ impact the number and similarity of the detected clusters. For this test, we have considered the following number of slots: 10, 50, and 100. As a result, we have the experiment configuration shown in Table 6.1.

Table 6.1: Parameters for DG2CEP's Heuristic Experiment

$lowerPts$	Total # Slots $\mathcal{S}$	Fixed
$18, 16, 14, 12, 10, 8, 6$	10, 50, 100	ε, $minPts$, Δ, Throughput

6.3.2 Experiment Setup

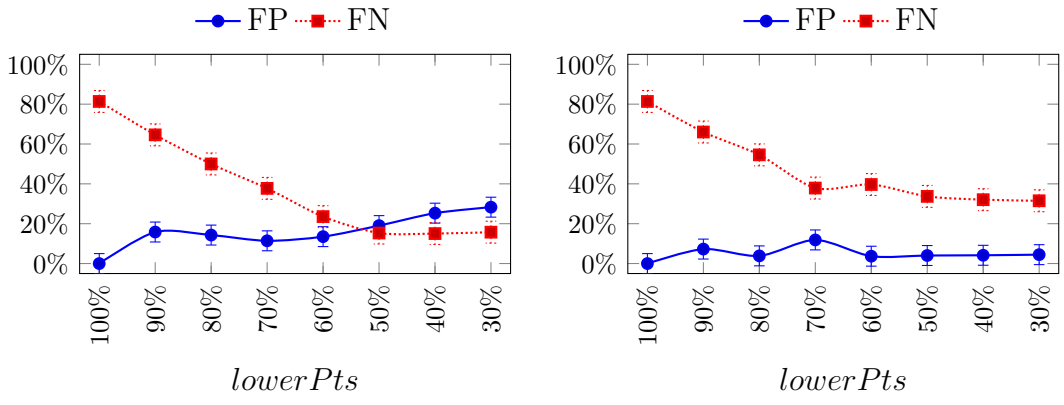
We executed these experiments in the Microsoft Azure Cloud platform using two virtual machines running Ubuntu GNU/Linux 14.04.3 64-bit and the OpenJDK 1.7.91 64-bit Java runtime. One of the virtual machines replayed the data stream, while the second one contained an instance of DG2CEP with its entire EPN. The virtual machines were interconnected through a Gigabit link/bus and had the following hardware configuration:

- Intel® Xeon CPU E5-2673 v3 @ 2.40 GHz
- 28 GiB Memory RAM

6.3.3 Result and Analysis

In this subsection, we present and discuss the evaluation results. Each experiment was run 10 times and the error bars in the graphs represent a 95% confidence interval. As said, each test run replayed the rush-hour data stream (17:00–18:00) for the heuristic-enhanced DG2CEP.

Figure 6.2 illustrates the average percentage of missed clusters (False Negative – FN) and incorrectly detected clusters (False Positive – FP) of the proposed heuristic in DG2CEP at a given second, when compared with the ground-truth clustering results in the specified second for a one-hour test period and parameters $\varepsilon = 100$, $minPts = 20$, and $\mathcal{S} = 10$. The two line graphs illustrated by Figure 6.2 (a) and (b) represent the values obtained when evaluating the heuristic with linear and exponential weights respectively.



(a) Linear Weight (b) Exponential Weight

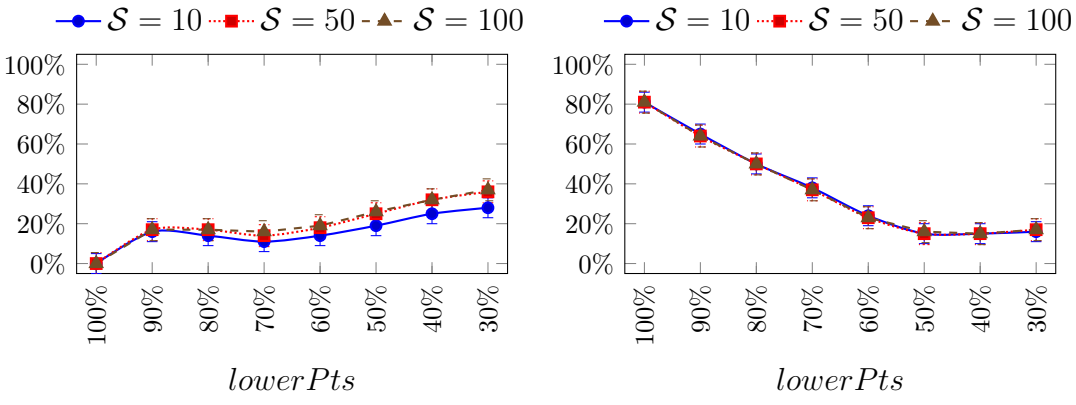
Figure 6.2: Percentage of Incorrectly Detected (FP) and Undetected Clusters (FN) in heuristic-enhanced DG2CEP (for $\varepsilon = 100$, $minPts = 20$, and $\mathcal{S} = 10$).

In both cases (linear and exponential), due to the answer loss problem, the graph shows that $lowerPts$ equal or close to $minPts$ (100% and 90%) will fail to detect some clusters (false negatives). However, since the threshold is closer to $minPts$, the original density, there will be few collateral effects, hence, the low percentage of incorrect detected clusters (false positives).

According to the linear weight graph, Figure 6.2 (a), the $lowerPts$ thresholds that yielded the best tradeoff results were 60% and 50% of $minPts$. These thresholds reduced the number of missed clusters from 80% to 23.57% and 15.32%, respectively, with a collateral effect of incorrect clusters of 13.51% and 19.05%, respectively. More specifically, using such parameters (*e.g.* a lowerPts of 50%), a single heuristic-enhanced DG2CEP instance was able to provide in real-time at a given second a clustering result that is 84.68% similar to the off-line DBSCAN result at that second.

To exemplify such results consider the graphic comparison illustrated by Figure 6.4, which shows the similarity between the detected clusters of DBSCAN and DG2CEP, using a linear heuristic and $lowerPts = 50\%$, in a given second. Each marker (in red) represent a cluster centroid. As can be seen, the clusters found in real-time by the on-line DG2CEP algorithm are very similar to their ground-truth off-line DBSCAN counterparts.

The heuristic exponential weight graph, Figure 6.2 (b), presented better results as $lowerPts$ decreases. This illustrates that, when using the exponential weight, the heuristic is more tolerant to collateral effects. For example, the number of incorrect clusters (false negative) results is 4.47% for a $lowerPts$ equal to 30% of $minPts$. However, for this parameter, the heuristic reduced the number of cluster not detected due to the answer loss problem, from 80% to 31.51%, instead to 13.51% when using linear weights.



(a) False Positive (b) False Negative

Figure 6.3: Relationship between heuristic results and the total number of cell slots subdivisions $\mathcal{S}$ for linear weights.

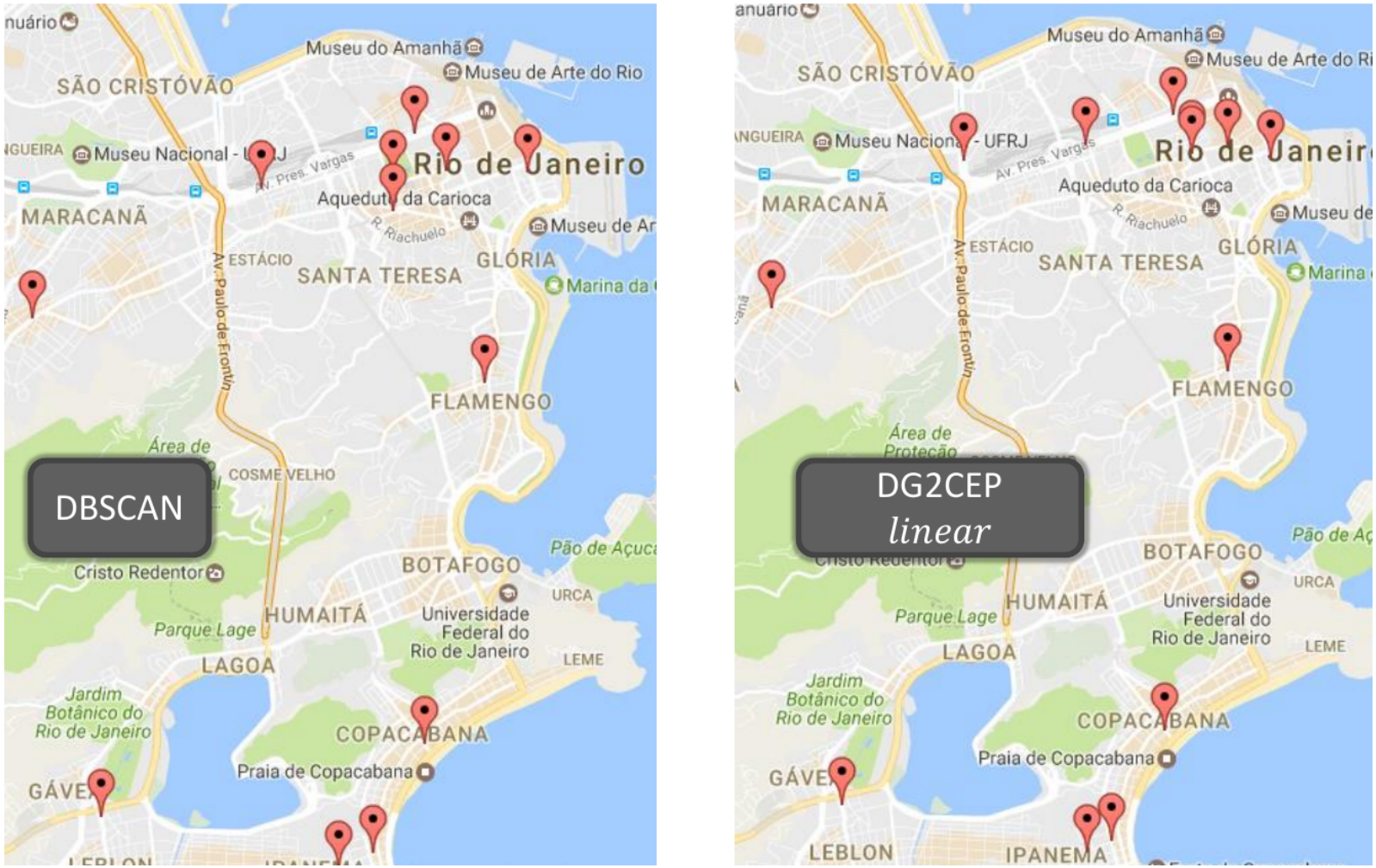
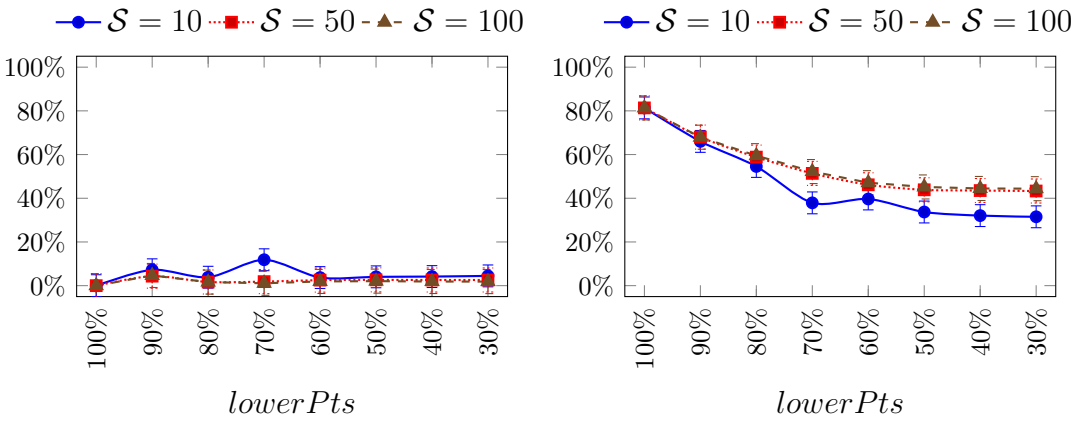


Figure 6.4: Graphical comparison between the off-line DBSCAN clustering result and DG2CEP on-line clustering result.



(a) False Positive (b) False Negative

Figure 6.5: Relationship between heuristic results and the total number of cell slots subdivisions $\mathcal{S}$ for exponential weights.

With respect to the relationship between the total number of cell subdivisions $\mathcal{S}$ and the heuristic results, Figure 6.3 and Figure 6.5 shows how the number of undetected clusters (false negative) and incorrectly detected clusters (false positive) vary for different number of cell slots when using the heuristic linear and exponential weight respectively for parameters $\varepsilon = 100$ and $minPts = 20$.

Overall, the results presented a similar behavior. Intuitively, we expected that the heuristic would present a better result as the number of slots increased. However, this was not the case. It seems that the high number of cell slots combined with the GPS error failed to grasp the correct distribution of objects within the cell. For example, when using a linear weight, subdividing the cells in a total of 10 slots, and a transient threshold of 50%, throughout the entire experiment run DG2CEP only failed to detect in a given second t 15% of the spatial clusters that appears in the same second t in DBSCAN. When subdividing the cell into 50 and 100 slots for the same transient threshold of 50%, the number of missed clusters in the given second increased to 16% for both cases.

In conclusion, the experiments indicate a heuristic-enhanced DG2CEP yields betters results that without it. Further, it shows that when using a linear weight, a smaller number of subdivision slots $\mathcal{S}$, and a transient threshold of 50%, DG2CEP produced in real-time the most similar clustering result to its off-line DBSCAN counterpart when compared to the other parameters. Hence, for the next experiments we use a heuristic-enhanced DG2CEP with such parameters ($lowerPts = 0.5 \times minPts$, $\mathcal{S} = 10$, and linear weights).

6.4 Elapsed Time

For this experiment, we first distinguish where new clusters appears in DBSCAN's ground truth result. A cluster C is said to appear at time t if there is no cluster B in the previous timestamp $t-1$ whose intersection $B \cap C$ is higher than 0, *w.r.t* the moving objects in each set. We log the timestamp and the content of such cluster. We use this information to compare to DG2CEP second-by-second clustering results to discover the elapsed time it took to find it. Precisely, the elapsed time required by DG2CEP to detect a cluster C that appeared in timestamp t in DBSCAN is the minimum timestamp $q \geq t$ that contains a cluster D who contains at least 50 % of C elements ($C \cap D \geq 0.5$).

We apply a similar logic to establish which and where the clusters from DBSCAN results dispersed. For instance, a cluster C from timestamp $t-1$ is said to have dispersed at timestamp t if there is no cluster B in t such that the intersection between $C \cap B$ is higher than 0, *w.r.t* their content (moving objects). This means that DBSCAN results show that timestamp t no longer contains a cluster that was in the previous second. Similar to the formation approach, the elapsed time required by DG2CEP to detect the dispersion of C at timestamp t is the minimum timestamp $q \geq t$ that no longer contains a cluster D whose intersection with C is higher than 0 ($C \cap D > 0$).

Finally, we do the same for cluster evolution. A cluster C evolves in DBSCAN at timestamp t when its content changes over 50 % within the previous second, that is, if exists a cluster B in timestamp $t-1$ where they contain elements in common, $B \cap C > 0$, but their intersection $B \cap C \leq 0.5$ is less than or equal to 50 %. Hence, the elapsed time required by DG2CEP to detect this evolution is a timestamp $q \geq t$ whose contain a cluster D whose intersection with C is higher or equal to 50 % ($C \cap D \geq 0.5$).

To measure these values we need to compute DG2CEP and D-STREAM second-by-second clustering result. Hence, whenever these two algorithms detect a cluster we save its content into an output file. However, there are gaps between the clustering entries since the algorithms output in discrete periods, that is, DG2CEP only yield results when a cluster is discovered, dispersed, or updated, while D-STREAM only output in specific periods (batch). To address this issue, and compare second-by-second, we need to fill these gaps intervals. We do this in the following way: suppose DG2CEP detects a cluster C at timestamp t and in timestamp $t' = t + 20$ it detects an update of C. This mean that from time t to t' DG2CEP clustering result is C, hence, we fill the missing intervals with C entries. The same logic is used to fill D-STREAM cluster entries.

6.4.1 Experiment Parameters

The primary goal of the experiment is to discover the elapsed time required by DG2CEP and D-STREAM to detect the distinguished clusters and how this time varies under different data stream volumes. To do so, we executed the experiment using three data stream throughputs: of 2500, 5000, and 7500 moving object's location updates per second, as described in Section 6.2.

A primary parameter of our algorithm is the size of ε-squared grid cells (context partitions). To verify the impact of ε in the measured elapsed time, we further tested the experiments using three grid cell sizes: 50, 100, and 150 meters. In addition, in all test runs we set the sliding window Δ to be 60 seconds, to reflect the maximum interval used by the bus fleet to send their location update. Further, we fixed the value of $minPts$ to be 20. Also, based on the previous experiment, we use the proposed heuristic with linear weights, a transient $lowerPts = 0.5 \times 20 = 10$ density, and $\mathcal{S} = 10$ subdivision slots. As a result, we have the experiment configuration shown in Table 6.2. We executed each experiment scenario 10 times, totalizing 90 executions.

Table 6.2: Parameters for DG2CEP's Elapsed Detection Experiment

ε	Data Stream Throughput	$minPts$	Δ
50 m	2500 lu/s, 5000 lu/s, 7500 lu/s	20	60s
100 m	2500 lu/s, 5000 lu/s, 7500 lu/s	20	60s
150 m	2500 lu/s, 5000 lu/s, 7500 lu/s	20	60s

To compare D-STREAM results to DG2CEP, through DBSCAN ground-truth, we also executed the experiment configurations using D-STREAM with the following batch periods: 30, 45, and 60 seconds with a $\Delta = 60$ second fading time window. We choose these values because to understand the relationship between a lower, medium, and higher batch period.

6.4.2 Experiment Setup

We executed all experiments in the DigitalOcean Cloud, where we used virtual machines running the Ubuntu GNU/Linux 14.04.5 64-bit operating system. All virtual machines were interconnected through a Gigabit link/bus and had the following hardware configuration:

- 4 × Intel Xeon CPU E5-2660 @ 2.20GHz
- 8 GiB Memory RAM

For this experiment, we used four different setup configurations. The first experiment setup configuration, which we called DG2CEP Single Instance, contains two virtual machines. One of the virtual machines replayed the data stream, while the second one contained an instance of DG2CEP with its entire EPN. Similarly, we created a D-STREAM Single Instance setup. On this case, instead of DG2CEP, the second virtual machine contains an instance of D-STREAM.

We also interested in measuring how the number of deploy instances impacts the experiment. For this, aside for a virtual machine to replays the data stream, we also executed the experiment with four and eight distributed DG2CEP instances. In the first case, here called DG2CEP 2–2, we subdivided the spatial domain into two parts and used a total of four virtual machines (two for the CELL EPN and the remaining two for the GRID EPN). In the second case, called DG2CEP 4–4, we subdivided the spatial domain into four parts. Similarly, we use four virtual machines for the CELL EPN and the other four to the GRID EPN instances.

6.4.3 Results and Analysis

This subsection presents the experiment results. Each experiment run was tested 10 times and error bars represent a confidence interval of 95%.

6.4.3.1 Formation

Figure 6.6 shows the elapsed time, in seconds, that DG2CEP and D-STREAM required to detect a cluster formation when compared to DBSCAN second-by-second ground-truth information. The graph indicates that the size of ε has some impact on the detection time. As expected, a smaller ε yields a smaller detection time when compared to the one with a large ε. A smaller ε divides the spatial domain into a larger number of context partitions, which in turn increases the cost of identifying the context partition index for each location update. In addition, the CEP engine will also have to manage a larger number of context partitions. However, a larger ε can also increase the detection time for cluster formation when compared to a lower ε value. The primary reason is the increase of workload in the processing network. Since more moving objects are mapped to the same grid cell, which in turn generate more events that pass through the processing network, this additional load is reflected in the detection time.

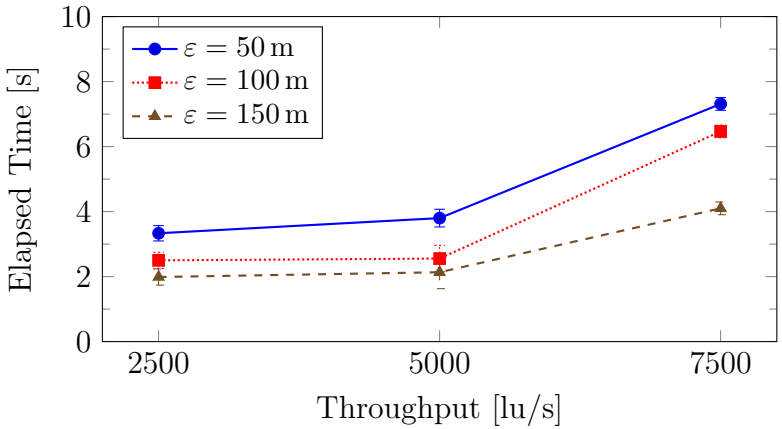
The experimental results show that a single instance of DG2CEP can detect cluster's formation in few seconds, as illustrated in Figure 6.6 (a). For example, in the scenario of 5 000 moving objects sending their position every second, DG2CEP detected the cluster formation in 3.80 s, 2.55 s, and 2.13 s for $\varepsilon = 50$, $\varepsilon = 100$, and $\varepsilon = 150$ respectively. The experiments also indicate that the algorithm scales with the number of moving objects, showing a linear increase in the cluster formation and dispersion detection times when increasing the data stream throughput.

With respect to scalability, the experiment results, illustrated in Figure 6.6 (c) and (d), indicates that the elapsed time required by DG2CEP to detect the cluster's formation reduced significantly when increasing the number of distributed instances. For instance, considering $\varepsilon = 100$ m in a data stream scenario of 7 500 location updates per second, the detected time reduced from 6.46 s (single machine) to 4.40 s for DG2CEP 2–2 configuration and to 1.83 s for its 4–4 configuration.

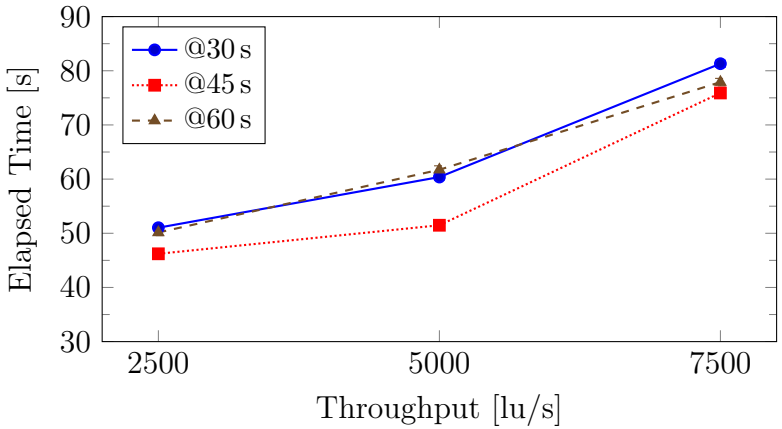
Although more instances speed up DG2CEP process, in some cases it can lead to an increase on the elapsed time due to the overhead involved in communicating and transferring data between instances. For example, for $\varepsilon = 50$ m and a data stream of 2 500 location updates per second, a DGCEP 2–2 setup configuration was able to detect the cluster formation faster (1.66 s) than a 4–4 configuration (2.11 s). However, this difference disappears as the data stream volume, and consequently the workload, increases.

With respect to batch-based approaches, D-STREAM required more time to detect the cluster formation than any DG2CEP configuration under all batch periods. For example, for $\varepsilon = 100$ m and a data stream throughput of 5 000 location updates per second, it required approximately 60.38 s, 51.48 s, and 61.79 seconds to detect the cluster formation for batch periods of 30, 60, and 90 seconds respectively.

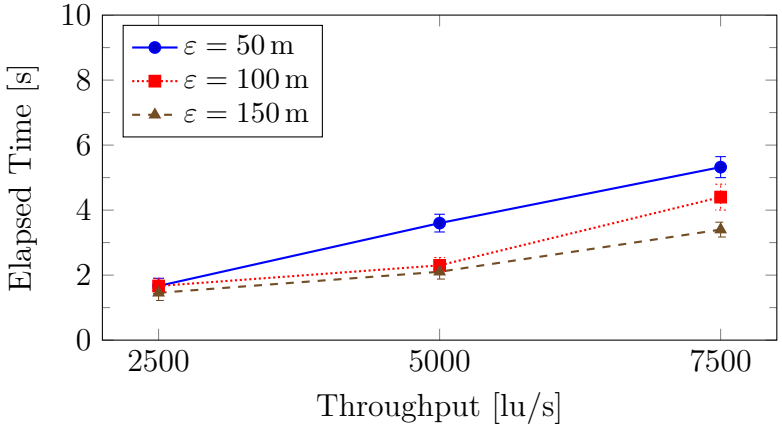
As expected, a smaller and higher batch period required more time to detect the cluster formation that a median one. With a smaller batch period the costly off-line processing is done more frequently than for the other periods, and thus more frequently D-STREAM will have to stop and compute the clusters while halting the on-line phase. Although a larger batch period also considers a higher number of moving objects in its buffer, the large waiting period between batches means that the cluster result is usually outdated. Thus, the medium batch period of 45 seconds made a better balance between batch size and off-line processing, yielding a better but slower detection time to DBSCAN.



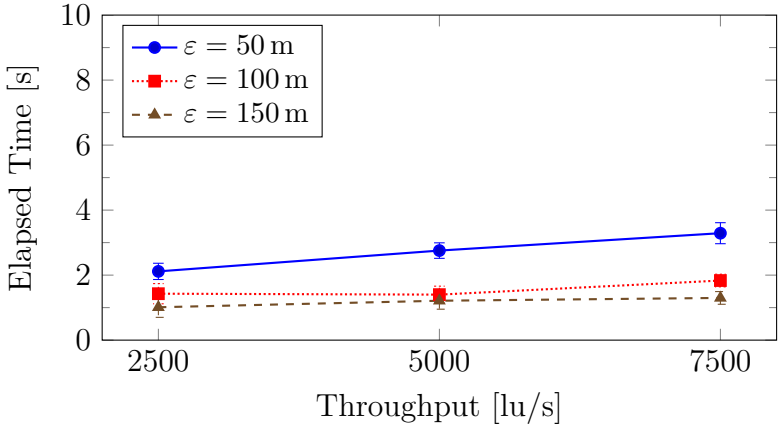
(a) DG2CEP Single Instance



(b) D-STREAM Single Instance ($\varepsilon = 100$ m)



(c) DG2CEP 2–2



(d) DG2CEP 4–4

Figure 6.6: Elapsed time to detect a cluster formation *w.r.t.* DBSCAN.

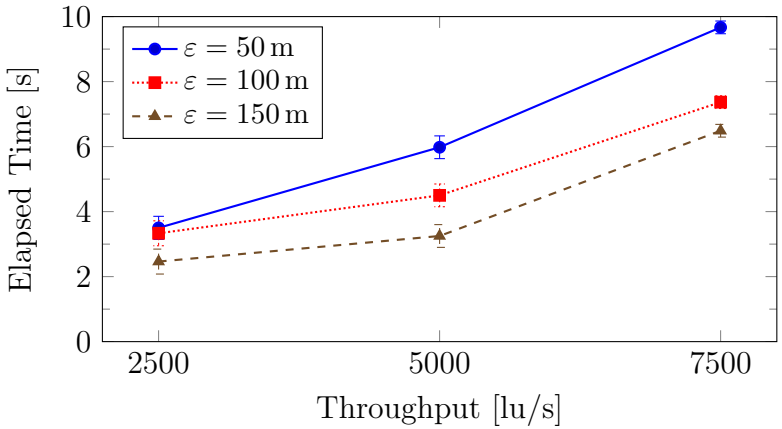
6.4.3.2 Dispersion

The elapsed time to detect a cluster dispersion by DG2CEP and by D-STREAM for all tested scenarios is illustrated in Figure 6.7. All values are higher than the ones required to detect a cluster formation. The reason is the way that dispersion of a cluster is detected in DG2CEP: a dispersion event is triggered when a moving object changes its cell or if DG2CEP does not receive a *DenseCellCluster* event within a Δ period. Hence, we expected that the time to detect a cluster dispersion to be higher than the time required to detect its formation due to being dependent on additional information (grid cell change or absence of location updates).

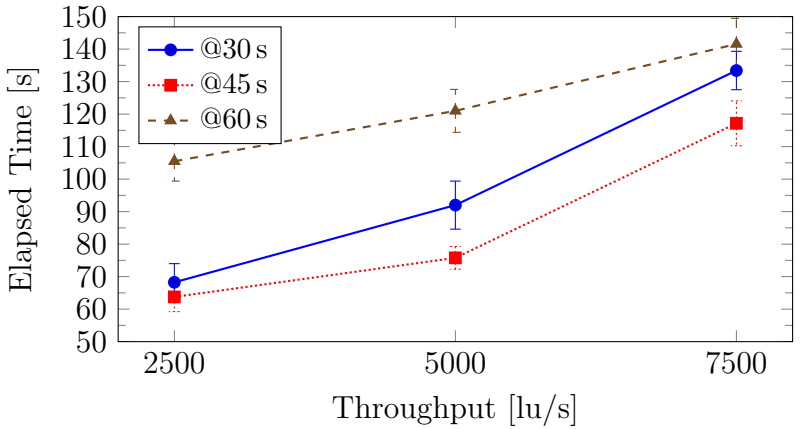
The results also indicate a direct correlation between the grid cell ε size and the elapsed time required to detect a dispersed cluster. A larger ε takes more time to detect a cluster dispersion since moving objects' location updates are mapped to fewer grid cells. In particular, those moving objects help to maintain the grid cell denser for a longer period. For example, the larger a grid cell is the more it takes a moving object location update to change it cell which in turn delay the event that triggers the dispersion.

Although the elapsed times to detect a cluster dispersion is higher than to detect its formation, a single DG2CEP instance is still able to detect it in few seconds as illustrated in Figure 6.7 (a). For example, for a throughput of 2 500 location updates per second, DG2CEP was able to detect a cluster dispersion in 3.50 s, 3.33 s, and 2.46 s for $\varepsilon = 50$, $\varepsilon = 100$, and $\varepsilon = 150$ respectively. Under a higher data stream throughput, a single DG2CEP instance was still able to detect a cluster dispersion in a few seconds. When doubling the data stream throughput, from 2 500 to 5 000 location updates per second, the elapsed time required by DG2CEP to detect the cluster dispersion increased linearly. More specifically, it took in average 5.98 s, 4.50 s, and 3.25 s for $\varepsilon = 50$, $\varepsilon = 100$, and $\varepsilon = 150$ respectively. More specifically, it took in average 5.98 s, 4.50 s, and 3.25 s for $\varepsilon = 50$, $\varepsilon = 100$, and $\varepsilon = 150$ respectively.

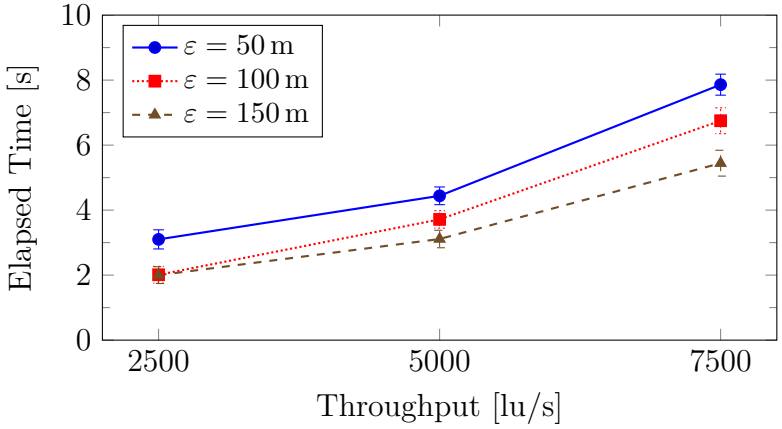
Similar to the experiment with cluster formation, the results indicate that the elapsed time to detect cluster dispersion also reduced when increasing the number of DG2CEP instances, as shown in Figure 6.7 (c) and (d). For example, consider DG2CEP's 4–4 distributed configuration. Considering $\varepsilon = 50$ m in a data stream scenario of 5 000 location updates per second, the detected time reduced from 5.98 s (single machine) to 2.55 s. Such reduction also appear under a higher data stream throughput (of 7 500 location updates per second). In this case, the detection time reduced from 9.66 s (single machine) to 5.76 s.



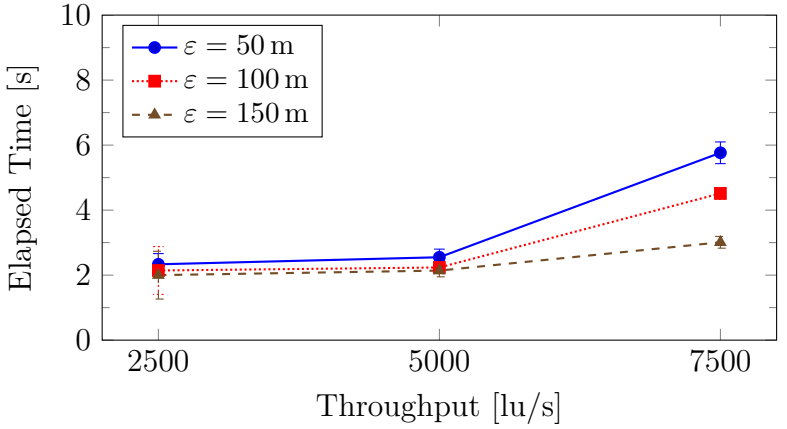
(a) DG2CEP Single Instance



(b) D-STREAM Single Instance ($\varepsilon = 100$ m)



(c) DG2CEP 2–2



(d) DG2CEP 4–4

Figure 6.7: Elapsed time to detect a cluster dispersion *w.r.t.* DBSCAN.

D-STREAM's batch-based approach presented significantly worst results for detecting cluster dispersion when compared to detecting its formation. The primary reason for this is that D-STREAM does not handle moving object that changes grid cells. Thus, it may retain the moving object's location update in multiple cells. Hence, since D-STREAM triggers a cluster dispersion only based on a cell density, the dispersion process will take more time since more moving objects are contributing to the cell density.

As expected, D-STREAM required expressively more time to detect the cluster dispersion than any DG2CEP configuration for all batch periods. For example, for $\varepsilon = 100\,\text{m}$ and a data stream throughput of 7 500 location updates per second, it required approximately 133.41 s, 117.17 s, and 141.55 seconds to detect the cluster formation for periods of 30, 60, and 90 seconds respectively. Likewise the experiment with cluster formation, the smaller and higher batch period required more time to detect the cluster dispersion that a median one.

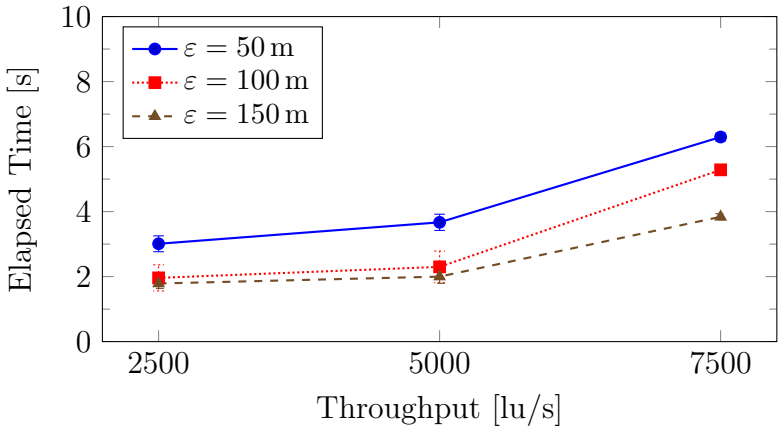
6.4.3.3 Evolution

Figure 6.8 shows the elapsed time, in seconds, that DG2CEP and D-STREAM required to detect a cluster evolution when compared to DBSCAN second-by-second ground-truth log. The graph results indicate that DG2CEP was able react and detect the cluster evolution under a few seconds. Furthermore, under all scenarios, the elapsed time required to detect a cluster evolution were lower than the required time to detect its formation and dispersion.

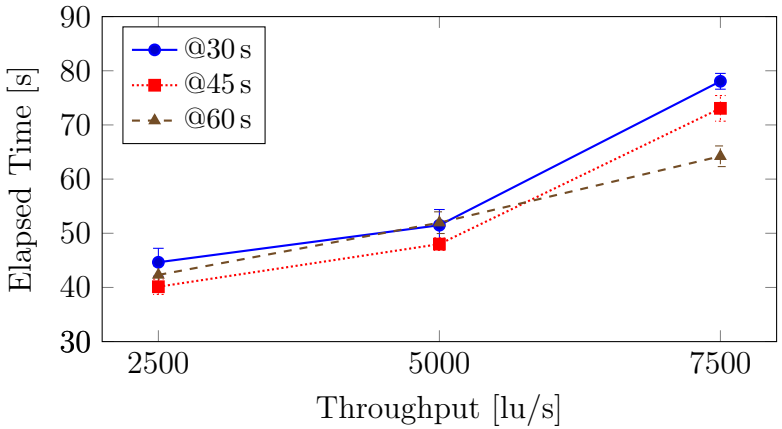
One of the reasons for a faster detection time for evolution is that DG2CEP only needs to update the cluster content instead of either adding or merging to an existing one, which add as overhead to the elapsed time. For example, considering $\varepsilon = 100\,\text{m}$ and a data stream throughput of 5 000 location updates per second, a 2–2 DG2CEP configuration required approximately 2.1 seconds to detect a cluster evolution instead of 3 seconds for its formation.

Likewise with cluster formation and dispersion, a single instance of DG2CEP was able to detect cluster evolutions within few seconds, as illustrated in Figure 6.8 (a). For example, considering $\varepsilon = 100\,\text{m}$, a single DG2CEP instance detected in average that a cluster has changed 50% of its element within 1.9 s, 2.3 s, and 5.28 s seconds for respectively the data stream throughput of 2 500, 5 000, and 7 500 location updates per second.

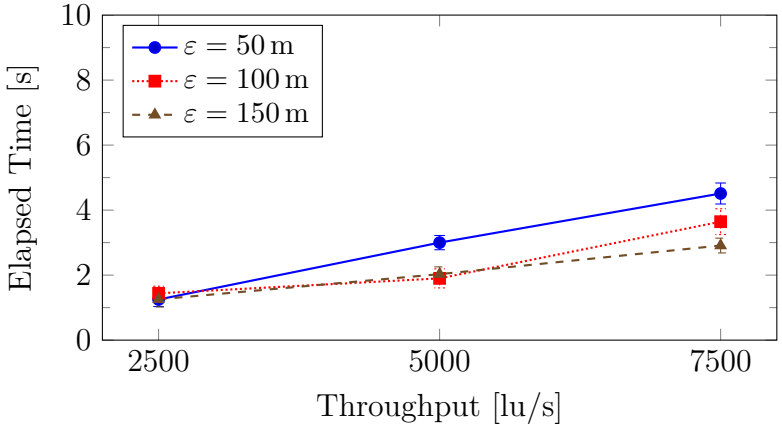
When adding new DG2CEP instances these numbers dropped *w.r.t* a single instance, as illustrated in Figure 6.8 (c) and (d). For instance, in all data stream throughputs, the 4–4 DG2CEP configuration was able to reflect the off-line DBSCAN second-by-second cluster evolution result within 1.5 seconds.



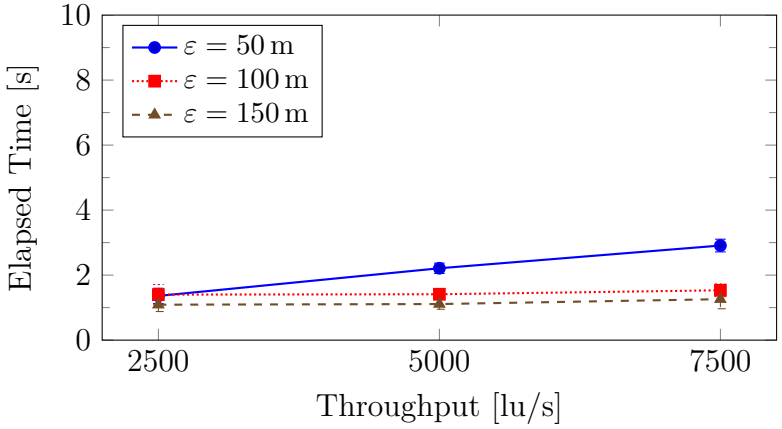
(a) DG2CEP Single Instance



(b) D-STREAM Single Instance ($\varepsilon = 100$ m)



(c) DG2CEP 2–2



(d) DG2CEP 4–4

Figure 6.8: Elapsed time to detect a cluster evolution *w.r.t.* DBSCAN.

Similar to the previous experiments, D-STREAM required more time to detect the cluster evolution than any DG2CEP configuration under all batch periods. Nevertheless, the results indicates that such results are significantly faster than the required time to compute the formation and dispersion with D-STREAM. For example, consider the scenario with $\varepsilon = 100\,\text{m}$, a batch period of 30 seconds, and a data stream throughput of 5 000 location updates per second. In this case, D-STREAM required in average 51.48 seconds to detect a cluster evolution, contrary to a requiring 60.38 and 92.01 seconds to detect its formation and dispersion respectively. In a nutshell, the findings in this experiment support the idea that a cluster evolution is detected faster than its formation or dispersion.

6.5 Similarity

The second portion of experiments had two goals. First, measure how similar (accurate) DG2CEP real-time clustering results are to DBSCAN's second-by-second ground-truth results. The second goal is to indirectly compare DG2CEP with D-STREAM, through their continuous similarity (second-by-second) to DBSCAN.

We measured the similarity for a detected cluster entry and for the entire cluster results. By detected cluster entry we mean how similar a given cluster found in DG2CEP is to its counterpart in DBSCAN. In order to measure this value, we proceeded as follows: whenever DG2CEP detects a cluster we take a snapshot of its content (moving objects' location updates) and compare it to DBSCAN second-by-second ground-truth log. To measure similarity we used the *Rand Index* [69] metric, which express the percentage of similarity between two clusters. *Rand Index* is a number between 0 and 1, where 1 means that the clusters are identical and 0 means that they are totally different, (*i.e.* that have no common moving object. *Rand Index* is expressed as $\frac{TP+TN}{TP+FP+FN+TN}$, where TP, TN, FP, FN, are the number of true positive, true negative, false positive, and false negative cases respectively, *w.r.t.* the moving objects location updates' outputted by DG2CEP and DBSCAN.

DG2CEP and DBSCAN may identify several clusters in the same snapshot. Thus, to identify the cluster D in DBSCAN that is the counterpart to C , the one discovered by DG2CEP, we need to compare C with all cluster found by DBSCAN in the snapshot. Precisely, we use C timestamp to retrieve all clusters found S in that given second. Then, we choose the cluster D with the highest *Rand Index* since it is the one in DBSCAN's output closer to the cluster found with DG2CEP, that is, the counterpart cluster D is computed as a cluster

that has the higher *Rand Index* value $D = \max((RandIndex(d, C) \mid \forall d \in S))$. We call this metric *Detected Rand Index*, as it represents the *Rand Index* of a detected cluster.

We also measured how similar are the DG2CEP complete second-by-second clustering result to DBSCAN second-by-second ground-truth information. This measurement metric, which we called *Complete Rand Index*, represents the similarity between two sets of clustering results, *i.e.*, it is a comparison between the entire set of detected cluster of DG2CEP or D-STREAM with those found by DBSCAN in a given second, not just the similarity of the detected ones. Thus, at every second, in addition to the *Detected Rand Index* we consider the number of undetected clusters in the total number of clusters.

For example, suppose in a given timestamp that DG2CEP detected 3 clusters (c_1,c_2, and c_3), while DBSCAN yields 4 clusters. Then the *Complete Rand Index* is calculated as $\frac{dri(c_1)+dri(c_2)+dri(c_3)+0}{4}$, where *dri* is the *Detected Rand Index* of the clusters detected by DG2CEP. Undetected clusters have $dri = 0$, since the algorithm did not detect them, thus, contributing to the decrease of this similarity index.

6.5.1 Experiment Parameters

Analogous to the elapsed time experiment, the goal here is to discover how similar DG2CEP and D-STREAM results are to DBSCAN one and how they vary under different data stream volumes. To measure the data stream volume influence, the experiment was executed using two different throughputs: 2500 and 5000 moving object's location updates per second, as described in Section 6.2.

The experiment also measured the influence of different ε-squared grid cells (context partitions) sizes. Since this is a time consuming experiment, we limited the parameter variation to 50 and 100 meters. In addition, in all test runs we set the sliding window Δ to be 60 seconds, to reflect the maximum interval used by the bus fleet to send their location update. Further, we fixed the value of $minPts$ to be 20. As a result, we have the experiment configuration shown in Table 6.3. We executed each experiment scenario 10 times, totalizing 40 executions.

Table 6.3: Parameters for DG2CEP's Similarity Detection Experiment

ε	Data Stream Throughput	$minPts$	Δ
50 m	2500 lu/s and 5000 lu/s	20	60s
100 m	2500 lu/s and 5000 lu/s	20	60s

6.5.2 Experiment Setup

We executed all experiments in the DigitalOceal Cloud with the same four setup configurations that were described in Subsection 6.4.1. Recalling, the first experiment setup configuration, which we called DG2CEP Single Instance, encapsulates the entire EPN into a single machine.

A second and third setup configuration tests the influence of a distributed DG2CEP deploy in the experiment performance. Particularly, the second configuration, called DG2CEP 2–2, subdivides the spatial domain into two parts and used a total of four virtual machines (two for the CELL EPN and the remaining two for the GRID EPN). The third configuration setup, called DG2CEP 4–4, subdivides the spatial domain into four parts (four to the CELL EPN and the other four to the GRID EPN).

Finally, we setup a D-STREAM instance to compute the batch-based results. For this configuration, we tested the experiment with three different batch periods: 30, 45, and 60 seconds with a $\Delta = 60$ second fading time window. We choose these values because to understand the relationship between a lower, medium, and higher batch period.

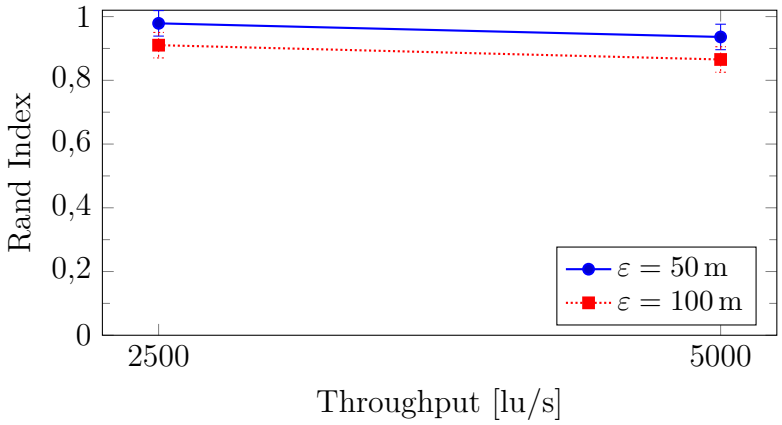
6.5.3 Results and Analysis

In this subsection, the results for the similarity experiment are presented. As done previously, we used a confidence interval of 95%. The results are further discussed in this subsection.

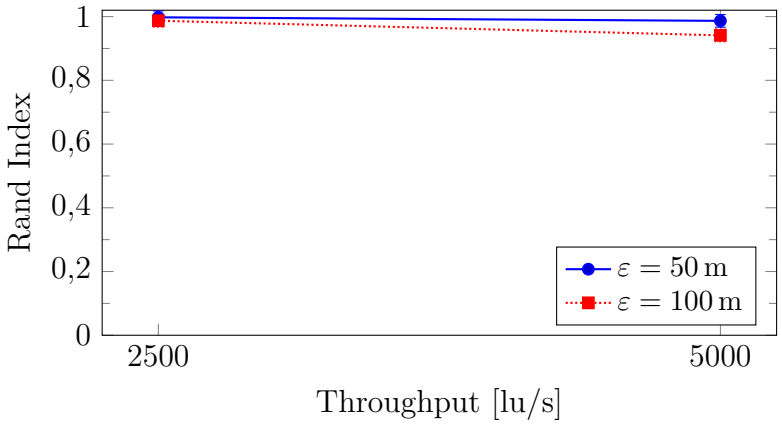
6.5.3.1 Detected Cluster's Similarity

Figure 6.9 shows the *Detected Rand Index* for each experiment scenario, that is, the similarity of the detected clusters with their counterpart in DBSCAN. The graph indicates that DG2CEP's real-time clustering result is highly similar to DBSCAN. For example, in the scenario with a throughput of 5 000 location updates per second, the detected clusters in real-time by a single DG2CEP instance (a) achieves a similarity of 93.61% and 86.54% with their second-by-second offline counterpart in DBSCAN's output for $\varepsilon = 50$ and $\varepsilon = 100$ respectively. When using a distributed deploy, for instance a 4–4 DG2CEP configuration (b), the similarity increase to 98.66% and 94.11% for $\varepsilon = 50$ and $\varepsilon = 100$ respectively.

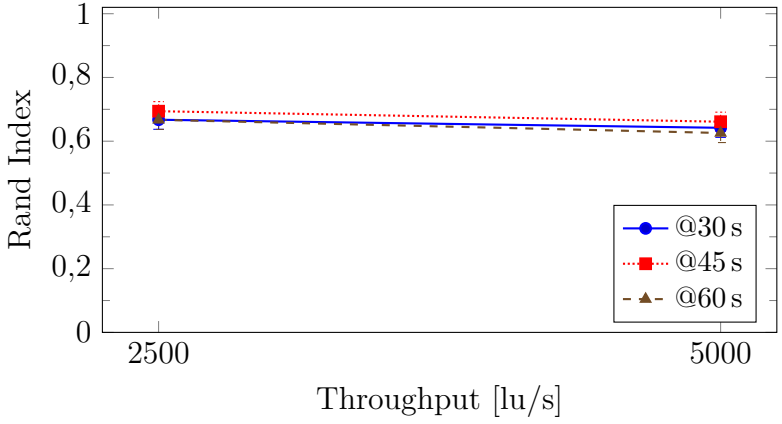
These results suggest that there is a relationship between the encountered similarity and the size of the grid cell. For instance, the scenarios that used



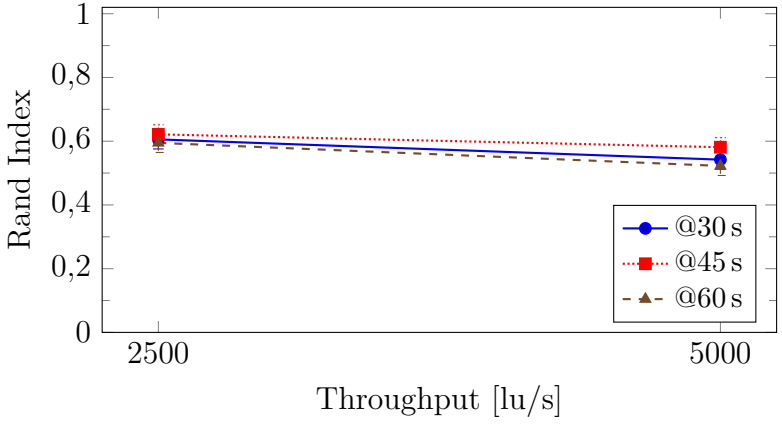
(a) DG2CEP Single Instance



(b) DG2CEP 4–4



(c) D-STREAM ($\varepsilon = 50\,m$)



(d) D-STREAM ($\varepsilon = 100\,m$)

Figure 6.9: Similarity of detect clusters with their counterpart in DBSCAN.

$\varepsilon = 50$ were the ones that presented highest similarity. The primary reason is that smaller values of ε yield smaller grid cells, which in turn have smaller areas close to the cell's borders. Thus moving objects placed in such clusters are more likely to be mutually within an ε distance.

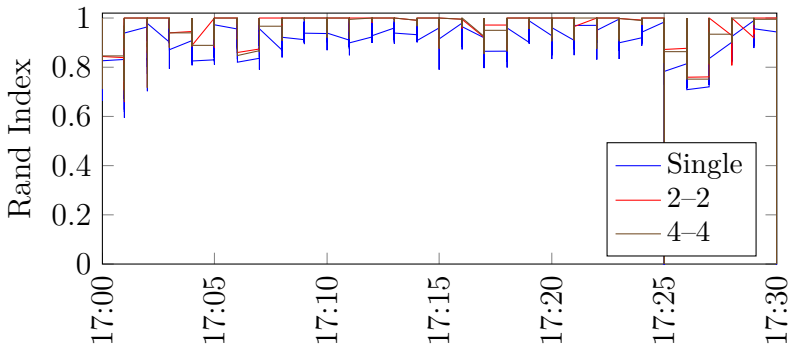
If we look at the previously elapsed time results, a smaller ε subdivides the domain into a larger number of grid cells, which in turn increases the cost of identifying the grid cell index for each moving object's location update. In addition, the CEP engine will also need to manage a larger number of grid cells. Thus, a small ε requires more time for the cluster detection than a larger one. Therefore, there is a trade-off when using DG2CEP: the smaller ε is, more similar will be the results of DG2CEP and DBSCAN at the cost of increasing the required computational effort and processing time. Hence, the user has to consider his/her application's requirements against the availability of processing resources and the expected rate of location updates to be processed.

As expected, D-STREAM results were lower than DG2CEP. For the scenario with $\varepsilon = 50\,\text{m}$ and a throughput of 5 000 location updates per second, the clusters found in D-STREAM had a similarity of 64.20%, 66.11%, and 62.57% for batch periods of 30, 45, and 60 seconds respectively. Confirming the previous experiments, the medium batch period of 45 seconds

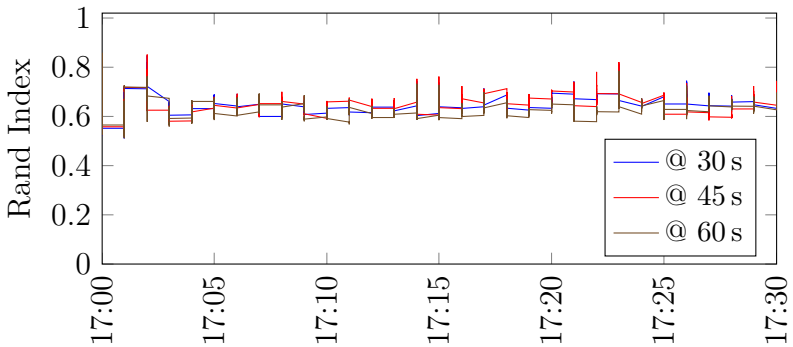
This graph also indicates that the similarity of detected clusters in DG2CEP and D-STREAM scales with the data stream throughput, showing a linear decrease in the similarity when increasing the data stream volume. For example, considering $\varepsilon = 50\,\text{m}$, the similarity of detected clusters in real-time by a single DG2CEP instance were 97.88% and 93.61% for a data stream throughputs of 2 500 and 5 000 location updates per second. D-STREAM also presented a linear behavior. For the same scenario, the similarity of the cluster detected by D-STREAM to DBSCAN were 69.41% and 66.11% for throughputs 2 500 and 5 000 using a batch period of 45 seconds.

6.5.3.2 Similarity Evolution

Figure 6.10 illustrates how the similarity of detected clusters by DG2CEP in real-time and D-STREAM in batch evolve and compare to DBSCAN throughout 30 minutes of the test period for a data stream throughput of 5 000 location updates per second and $\varepsilon = 50$ meters. Confirming our previous findings, clusters detected by DG2CEP in real-time presented a high similarity with their off-line DBSCAN counterpart. More specifically, for a data stream throughput of 5 000 location updates per second and $\varepsilon = 50\,\text{m}$, the clusters detected by a single DG2CEP instance (a) presented in average a similarity of

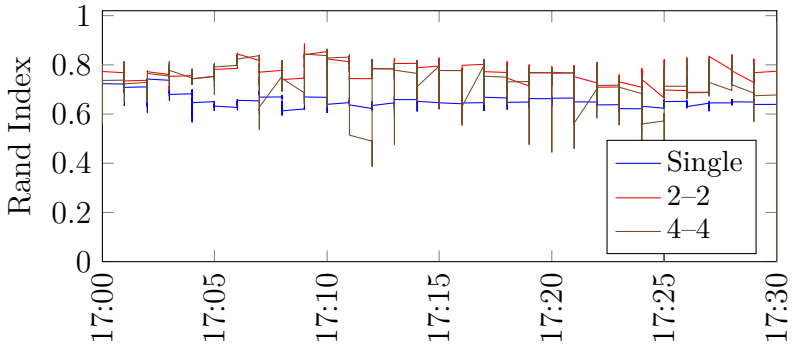


(a) DG2CEP ($\varepsilon = 50$ m)

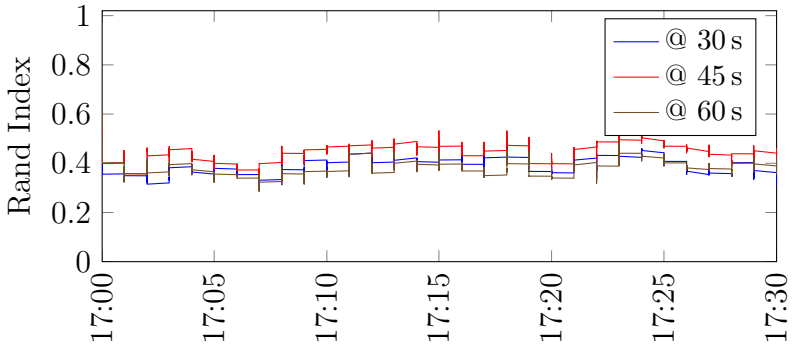


(b) D-STREAM ($\varepsilon = 50$ m)

Figure 6.10: Evolution of the Detected Rand Index ("similarity") of DG2CEP and D-STREAM with DBSCAN.



(a) DG2CEP ($\varepsilon = 50$ m)



(b) D-STREAM ($\varepsilon = 50$ m)

Figure 6.11: Evolution of the Complete Rand Index ("similarity") of DG2CEP and D-STREAM with DBSCAN.

Table 6.4: *Detected* and *Complete Rand Index* of DG2CEP and D-STREAM with DBSCAN for $\varepsilon = 50\,\text{m}$ and a throughput of 5 000 lu/s.

Algorithm	*Detected Rand Index*	*Complete Rand Index*	CI
DG2CEP Single	0.9361	0.7158	±0.04
DG2CEP 2–2	0.9588	0.7997	±0.02
DG2CEP 4–4	0.9866	0.7879	±0.02
D-STREAM @ 30s	0.6420	0.4131	±0.03
D-STREAM @ 60s	0.6611	0.4606	±0.02
D-STREAM @ 90s	0.6257	0.4039	±0.03

93.61% with DBSCAN results. Further, based on the line graph, we observe that DG2CEP was able to detect the clusters' evolution throughout the entire experiment. Table 6.4 details the average *Detected* and *Complete Rand Index* for DG2CEP and D-STREAM when compared to DBSCAN for this period.

Throughout the experiment, detected clusters in D-STREAM presented a lower similarity to their DBSCAN counterpart than the ones detected by DG2CEP. Precisely, in average, a 64.20%, 66.11%, and 62.57% similarity for batch periods of 30, 60, and 90 seconds respectively. As expected, a smaller and higher batch period presented lower similarity than one with a medium one. With a smaller batch period the costly off-line processing is done more frequently than for the other periods, and thus more frequently D-STREAM will not produce any update of clusters. Although a larger batch period also considers a higher number of mobile nodes in its buffer, the large waiting period between batches means that the cluster result is usually outdated. Thus, the medium batch period of 45 seconds made a better balance between batch size and off-line processing, yielding a 66.11% similarity with DBSCAN result. It is interesting to note that the resulting graph lines of all figures for D-STREAM contain periodic sharp edges, representing the on-line phase of the algorithm. During this phase, the algorithm similarity constantly degrades until the next off-line phase. Finally, the graph indicates that D-STREAM was also able to maintain such index throughout the rush-hour period.

The *Complete Rand Index* metric expresses how similar DG2CEP and D-STREAM clustering results, as a whole, are and to DBSCAN for every second of the test period. As illustrated in Figure 6.11, the results indicate that in average in a given second a single DG2CEP instance clustering result is approximately equal to 71.58% of DBSCAN's off-line result. When considering a distributed DG2CEP instance, its real-time similarity factor to the same given second in DBSCAN optimal results increases to approximately 80%. In addition, the graph shows that DG2CEP was able to maintain such index throughout the test period.

DG2CEP presented a higher similarity to DBSCAN results than all D-STREAM batch periods. In fact, the *Complete Rand Index* of a DG2CEP single instance is higher than all of D-STREAM individual *Detected Rand Index*, that is, then the similarity of the *detected* cluster by D-STREAM to DBSCAN. With respect to D-STREAM, as expected, the medium batch period (45 seconds) presented a higher similarity with DBSCAN when compared to the other batch periods. For the tested scenario, it presented a similarity of 46.06% to DBSCAN, followed closely by the smallest period (30 seconds), while the largest batch period (60 seconds) presented a similarity of only 40.39%.

An interesting result from the *Complete Rand Index* tests are the suddenly appearance of sharply regions in the graph, such as the one from 17:10 to 17:15 in DG2CEP graph. We investigated these regions and discovered that the decrease in the similarity is due to undetected clusters by DG2CEP. Since the *Complete Rand Index* metric compare the set of discovered clusters of DG2CEP with DBSCAN, an undetected cluster will decrease the similarity since its *Detected Rand Index* is 0 and the *Complete Rand Index* is the average of detected rand index in that second.

Nevertheless, with the exception of the answer loss scenarios, the experimental results shows that DG2CEP provides better continuous clustering result than D-STREAM for all tested batch periods (30, 45, and 60 seconds). The results also indicate that DG2CEP was able to monitor cluster evolution, that is, its clustering result was able to keep up with DBSCAN second-by-second result. The same cannot be said for D-STREAM due to its batch-based processing. In fact, smaller and largest batch periods (30 s and 60 s) had worse result than the medium ones (45 s) in both *Rand Index* variations (*Detected* and *Complete*).

7
Conclusion

To address the research questions stated in Section 1.3, precisely *if is it possible to provide an on-line clustering result in near real-time from large position data streams*, this thesis presented DG2CEP. DG2CEP is an on-line clustering algorithm that uses Complex Event Processing (CEP) [13, 14] stream-processing concepts to leverage and attain *near* real-time DBSCAN-like density clustering, in form of a network (EPN) of CEP declarative rules, from large position data streams. It is able to continuously monitor the formation, dispersion, and evolution of clusters of arbitrary size and shape. In a nutshell, DG2CEP combines density- and grid-based data stream clustering approaches and represents them as a network of CEP real-time primitives.

The algorithm performs a DBSCAN-like [12] cluster expansion procedure but using the on-line and real-time declarative CEP primitives. The main idea behind DG2CEP is to mitigate the clustering process by first mapping the location updates to a grid, with context partitions of size $\frac{\varepsilon}{\sqrt{2}} \times \frac{\varepsilon}{\sqrt{2}}$, and then successively clustering the context partitions (grid cells) rather than the moving objects' location updates. Throughout this thesis we presented the overall EPN structure of DG2CEP, the specific CEP rules and primitives used in each processing phase, and discussed some problems related to the adaptation of the density-based cluster detection of DBSCAN to a grid of context partitions.

A primary contribution of DG2CEP is that it reduces the data stream clustering problem from pairwise distance measurements to counting the number of location updates that are mapped into each grid cell. The algorithm then has to cluster grid cells with high density of moving objects, but this process is less costly due to the trivial evaluation of the distance and adjacency between two context partitions. This enables DG2CEP to provide a continuous and rapid clustering result, since the computation now depends on the number of context partitions, instead of the number of moving objects. Further, the clustering cost is mitigated (smoothed) in relation to batch-based processing approaches since data is clustered as it flows through the EPN, rather than accumulating the data and doing the processing at once.

7.1 Research Questions

Experimental results (see Chapter 6) indicate that DG2CEP addresses the thesis main and sub research questions. The results shows that DG2CEP *is able to provide an on-line and near real-time result from large position data stream w.r.t.* a second-by-second off-line and optimal DBSCAN ground-truth clustering result. For example, as illustrated in Figure 6.6, in a data stream scenario of 2 500 moving objects sending their position every second, a single DG2CEP instance setup detected the cluster formation in 3.33 s, 2.50 s, and 1.98 s for $\varepsilon = 50$, $\varepsilon = 100$, and $\varepsilon = 150$ respectively. When employing a distributed DG2CEP version, the elapsed time to detect the cluster formation significantly reduced to 2.11 s, 1.48 s, 1.01 s for the same parameters. This means that within few seconds, DG2CEP reacted and detected a cluster that appears in the off-line DBSCAN second-by-second ground-truth results.

Similar results were obtained for dispersion detection. For instance, as shown in Figure 6.7, for a data stream throughout of 2 500 location updates per second, the elapsed time required by a single DG2CEP instance to detect a cluster dispersion was 3.50 s, 3.33 s, and 2.46 s for $\varepsilon = 50$, $\varepsilon = 100$, and $\varepsilon = 150$ respectively. A distributed DG2CEP setup significantly reduced this elapsed time to 2.33 *s*, 2.14 s, and 1.99 s for the same parameters.

DG2CEP experiments results also indicate that it can rapidly react and detect cluster's evolution, thus, addressing the sub research question *that asks if such approach can continuously monitor the cluster evolution in near real-time.* For example, considering $\varepsilon = 150$ meters, a single DG2CEP instance was able to detected a cluster evolution within 1.78 s, 2.00 s, and 3.84 s seconds for respectively the data stream throughput of 2 500, 5 000, and 7 500 location updates per second.

One of the sub research questions asked *how scalable is this approach w.r.t the data stream volume.* The experimental results indicate that DG2CEP was able to provide and maintain the results quality within seconds to the off-line DBSCAN ground-truth when increasing the position data stream throughput. For example, considering $\varepsilon = 100$ meters, a single DG2CEP instance detected in average a cluster evolution within 1.9 s, 2.3 s, and 5.28 s seconds for respectively the data stream throughput of 2 500, 5 000, and 7 500 location updates per second. When adding new DG2CEP instances these numbers dropped significantly, as illustrated in Figure 6.8 (c) and (d). For instance, a 4–4 DG2CEP setup was able to react and detect such cluster evolution within 1.40 s, 1.41 s, and 1.53 s for the same respectively data stream throughputs.

Finally, to answer *how similar is the on-line and near real-time clustering result to the ground-truth result* sub research question this thesis used two similarity metrics: *Detected* and *Complete Rand Index*. *Detected Rand Index* is a measure that express how similar a detected cluster found by DG2CEP is to its counterpart in DBSCAN in the same second. Its values is a number between 0 and 1, where 1 means that the clusters are identical and 0 means that they are totally different, (*i.e.* that have no common moving object. *Rand Index* is expressed as $\frac{TP+TN}{TP+FP+FN+TN}$, where TP, TN, FP, FN, are the number of true positive, true negative, false positive, and false negative cases respectively, *w.r.t.* the moving objects location updates' outputted by the cluster in DG2CEP and in DBSCAN.

On the other hand, *Complete Rand Index*, represents the similarity between two sets of clustering results, *i.e.*, it is a comparison between the entire set of detected cluster of DG2CEP with those found by DBSCAN in a given second, not just the similarity of the detected ones. Thus, at every second, in addition to the *Detected Rand Index* we consider the number of undetected clusters in the total number of clusters. Its values also range from 0 to 1, where 1 means that both sets contains the same number of clusters with the same moving objects location updates, while 0 means that they are totally different (no common entry).

The experiments indicate that that DG2CEP's real-time clustering result is highly similar to DBSCAN, as illustrated in Figure 6.9. For example, in the scenario with a throughput of 5 000 location updates per second, the detected clusters in real-time by a single DG2CEP instance achieves a similarity of 93.61% and 86.54% with their second-by-second offline counterpart in DBSCAN's output for $\varepsilon = 50$ and $\varepsilon = 100$ respectively. When using a distributed deploy, for instance a 4–4 DG2CEP configuration (b), the similarity increase to 98.66% and 94.11% for $\varepsilon = 50$ and $\varepsilon = 100$ respectively.

With respect to the *Complete Rand Index* metric, the results illustrated by Figure 6.11 indicate that in average in a given second a single DG2CEP instance clustering result is approximately equal to 71.58% of DBSCAN's off-line result. When considering a distributed DG2CEP instance, its real-time similarity factor to the same given second in DBSCAN optimal results increases to approximately 80%. This means, that DG2CEP can provide in real-time a clustering result that is approximately equal to 80% in the same second. Such index increases as time passes, as show in the elapsed detection experiments. In addition, the graph shows that DG2CEP was able to maintain such index throughout the entire test period.

In conclusion, if we look at the previously elapsed time results, a smaller ε subdivides the domain into a larger number of grid cells, which in turn increases the cost of identifying the grid cell index for each moving object's location update. In addition, the CEP engine will also need to manage a larger number of grid cells. Thus, a small ε requires more time for the cluster detection than a larger one. Therefore, there is a trade-off when using DG2CEP: the smaller ε is, more similar will be the results of DG2CEP and DBSCAN at the cost of increasing the required computational effort and processing time. Hence, the user has to consider his/her application's requirements against the availability of processing resources and the expected rate of location updates to be processed.

7.2 Limitations

The DG2CEP counting semantic, aligned with CEP primitives, enables it to provide scalable and faster results over using DBSCAN distance comparisons approach. However, it may fail to identify some spatial clusters, a problem known as *answer loss* (or *blind spot*) [37, 38, 67], described and discussed in Chapter 5. Although, the heuristic significantly reduce the number of undetected clusters, it may still miss the detection of some clusters. For instance, considering a data stream scenario of 5 000 location updates per second and $\varepsilon = 50$ meters, in a given second a distributed DG2CEP setup clustering result is approximately equal to 80% of DBSCAN off-line result at the same second, as illustrated in Figure 6.11. Part of the remainder clusters are detected by DG2CEP in the next seconds.

However, there is a part that is not detected by DG2CEP. For instance, the heuristic results, using linear weights, indicate that DG2CEP provided a clustering result that is 84.68% similar to the off-line DBSCAN for a throughput of 5 000 location updates per second and $\varepsilon = 100$ meters, as illustrated in Figure 6.2. Hence, one of the limitations of DG2CEP on-line approach is the inability to provide an identical result to DBSCAN. By reducing the clustering problem from distance comparison, in an ε radius, to counting the number of a moving objects, in a squared ε grid cell, we are losing this precision.

Another limitation of DG2CEP is that it is not able to dynamically change its parameters, that is, its parameter does not change after deployment. This can be an issue in some scenarios. For example, considering a downtown spatial region, the clustering $minPts$ threshold may vary accordingly to the given hour of the day. During rush-hours, the $minPts$ parameters should be higher than in midnight. This is a not an easy task, since a change in the parameters

requires a change throughout the entire EPN. For example, when changing the grid cell ε size, DG2CEP need to rebuild and recompute the grid structure. This should be done without losing, discarding, or duplicating incoming and existing events.

7.3 Contributions

To summarize, the main contributions of this thesis are:

- A novel on-line counting algorithm based on grid-density clustering, designed as a network of CEP continuous query and pattern primitives, that is able to continuously and timely detect (*near* real-time) spatial clusters and its evolution from large position data streams.
- A counting heuristic that mitigates the collateral effects of the answer loss (blind spot) problem [37, 38] that appears due to the usage of a grid data structure to index and cluster spatial data.
- A scalable event processing network architecture that can process data in parallel and be distributed to process higher data stream throughputs.
- An extensive discussion about the experimental results and tradeoffs of using an on-line and real-time spatial clustering approach with real-world position data streams.

In addition to this thesis, such contributions were published in the following journal and conference papers.

- [71]: RORIZ JUNIOR, M.; ENDLER, M.; SILVA E SILVA, F.. **An on-line algorithm for cluster detection of mobile nodes through complex event processing**. Information Systems, 64:303 – 320, 2017.
- [72]: RORIZ JUNIOR, M.; ENDLER, M.; CASANOVA, M. A.; LOPES, H.; SILVA E SILVA, F.. **A Heuristic Approach for On-line Discovery of Unidentified Spatial Clusters from Grid-Based Streaming Algorithms**, p. 128–142. Springer International Publishing, Cham, 2016.
- [73]: RORIZ JUNIOR, M.; ENDLER, M.. **DG2CEP: A density-grid stream clustering algorithm based on complex event processing for cluster detection**. In: VI SIMPÓSIO BRASILEIRO DE COMPUTAÇÃO UBÍQUA E PERVASIVA, Brasília, Brazil, 2014. SBC.
- [74]: BAPTISTA, G. L. B.; RORIZ, M.; VASCONCELOS, R.; OLIVIERI, B.; VASCONCELOS, I.; ENDLER, M.. **On-line Detection of Collective Mobility Patterns through Distributed Complex Event Processing**. Pontifícia Universidade Católica do Rio de Janeiro, Technical Report MCC-06/13, ISSN 0103-9741, 2013.

7.4 Future Work

As future work, we plan to address the following issues. By default, each DG2CEP processing stage can be deployed in a different machine, forming a pipeline workflow. However, if an stage is overloaded it can impact the entire system. For example, in rush hours, the stream input stage can receive high volume of data, thus, possibly becoming a bottleneck. It is important to note that such overloads can vary unexpectedly, *e.g.*, an accident, a mass protest, *etc.* Motivated by this, we intend to investigate and propose an autonomous and elastic architecture to scale DG2CEP, with respect to the number of events received, by dynamically expanding and contracting the processing topology. Although the autonomous architecture target DG2CEP, the lessons and discussions presented can be applied to other autonomous data streaming systems and framework.

In addition, in its current version, DG2CEP requires the setting of several parameters, such as the grid size ε, the minimum number of moving objects $minPts$, and the sliding window Δ. It can be complicated for the user to specify these parameters, specially because they can change over time and/or in given regions. For example, the minimum number of moving objects to form a cluster may be different based on the specified time (*e.g.*, workhours, midnight) or in different regions (*e.g.*, downtown, home neighborhood's. Hence, we are interested in investigating if it is possible to have a parameter free version of DG2CEP. Such version would automatically adapt and change the parameters based on some information, its surrounding, historical, etc. The major issue here is how to dynamically realign or resize the grid cells without losing or duplicating events.

In addition, we are interested in investigating a hybrid weight function, which combines the benefits of linear and resilience of exponential weights. We are confident that with some small changes in the proposed heuristic we may obtain better results, and consequently, enhance even further the similarity between DG2CEP and DBSCAN.

Finally, we also intend to verify if DG2CEP can be generalized for other type of event streams, such as finance and temperature stream. More specifically, it is possible to reuse or adapt DG2CEP to other types of event streams? To examine this question, in future works we aim to investigate functions that can map the different type of event streams to DG2CEP grid cells, similarly to how we map the moving object positions.

Bibliography

[1] KARP, R. M.. **On-Line Algorithms Versus Off-Line Algorithms: How Much is It Worth to Know the Future?** In: PROCEEDINGS OF THE IFIP 12TH WORLD COMPUTER CONGRESS ON ALGORITHMS, SOFTWARE, ARCHITECTURE - INFORMATION PROCESSING '92, VOLUME 1, p. 416–429, Amsterdam, The Netherlands, 1992. North-Holland Publishing Co.

[2] DODGE, S.; WEIBEL, R.; LAUTENSCHÜTZ, A.-K.. **Towards a Taxonomy of Movement Patterns**. Information Visualization, 7(3):240–252, 2008.

[3] AMINI, A.; WAH, T.; SABOOHI, H.. **On Density-Based Data Streams Clustering Algorithms: A Survey**. Journal of Computer Science and Technology, 29(1):116–141, 2014.

[4] KARGUPTA, H.; SARKAR, K.; GILLIGAN, M.. **MineFleet: An Overview of a Widely Adopted Distributed Vehicle Performance Data Mining System**. In: PROCEEDINGS OF THE 16TH ACM SIGKDD INTERNATIONAL CONFERENCE ON KNOWLEDGE DISCOVERY AND DATA MINING, KDD '10, p. 37–46, New York, New York, USA, 2010. ACM.

[5] ANANTHANARAYANAN, G.; HARIDASAN, M.; MOHOMED, I.; TERRY, D.; THEKKATH, C. A.. **StarTrack: A Framework for Enabling Track-based Applications**. In: PROCEEDINGS OF THE 7TH INTERNATIONAL CONFERENCE ON MOBILE SYSTEMS, APPLICATIONS, AND SERVICES, MobiSys '09, p. 207–220, New York, NY, USA, 2009. ACM.

[6] MITSCH, S.; MÜLLER, A.; RETSCHITZEGGER, W.; SALFINGER, A.; SCHWINGER, W.. **A Survey on Clustering Techniques for Situation Awareness**, p. 815–826. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[7] GAROFALAKIS, M.; GEHRKE, J.; RASTOGI, R.. **Querying and Mining Data Streams: You Only Get One Look a Tutorial**. In: PROCEEDINGS OF THE 2002 ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD '02, p. 635, New York, NY, USA, 2002. ACM.

[8] HE, Y.; TAN, H.; LUO, W.; MAO, H.; MA, D.; FENG, S.; FAN, J.. **MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce**. In: PARALLEL AND DISTRIBUTED SYSTEMS (ICPADS), 2011 IEEE 17TH INTERNATIONAL CONFERENCE ON, p. 473–480, 2011.

[9] ZHENG, K.; ZHENG, Y.; JING YUAN, N.; SHANG, S.; ZHOU, X.. **Online Discovery of Gathering Patterns over Trajectories**. IEEE Transaction on Knowledge Discovery and Data Engineering, 26(8):1974–1988, 2014.

[10] SILVA, J. A.; FARIA, E. R.; BARROS, R. C.; HRUSCHKA, E. R.; DE CARVALHO, A. C. P. L. F.; GAMA, J.. **Data Stream Clustering: A Survey**. ACM Comput. Surv., 46(1):13:1–13:31, 2013.

[11] AGGARWAL, C.; HAN, J.; WANG, J.; YU, P.. **A framework for clustering evolving data streams**. In: PROCEEDINGS OF THE 29TH INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES - VOLUME 29, p. 81–92. VLDB Endowment, 2003.

[12] ESTER, M.; KRIEGEL, H.; SANDER, J.; XU, X.. **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, p. 226–231, 1996.

[13] LUCKHAM, D. C.. **The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[14] ETZION, O.; NIBLETT, P.. **Event Processing in Action**. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[15] ZHENG, Y.; CAPRA, L.; WOLFSON, O.; YANG, H.. **Urban Computing**. ACM Transactions on Intelligent Systems and Technology, 5(3):1–55, sep 2014.

[16] MCAFEE, A.; BRYNJOLFSSON, E.. **Big data: the management revolution.** Harvard business review, 90(10):61–68, 2012.

[17] NIANTIC INCORPORATION. **Pokémon Go**. `https://www.pokemongo.com`, 2016. [Online: accessed on 12/10/2016].

[18] CROWLEY, D.; SELVADURAI, N.. **Foursquare**. `https://www.foursquare.com`, 2016. [Online: accessed on 12/10/2016].

[19] INTERACTIVECORP. **Tinder**. https://www.gotinder.com, 2016. [Online: accessed on 12/10/2016].

[20] IPLANRIO – EMPRESA MUNICIPAL DE INFORMÁTICA DA CIDADE DO RIO DE JANEIRO. **data.rio**. https://data.rio, 2016. [Online: accessed on 12/10/2016].

[21] WAZE MOBILE. **Waze**. https://www.waze.com, 2016. [Online: accessed on 12/10/2016].

[22] MATYSIAK, M.. **Data Stream Mining: Basic Methods and Techniques**. Technical report, Rheinisch-Westfälische Technische Hochschule Aachen, 2012.

[23] YUAN, J.; ZHENG, Y.; ZHANG, C.; XIE, W.; XIE, X.; HUANG, Y.. **T-Drive: Driving Directions Based on Taxi Trajectories**. In: ACM SIGSPATIAL GIS 2010. Association for Computing Machinery, Inc., 2010.

[24] AMARAL, B. G. D.; NASSER, R.; CASANOVA, M. A.; LOPES, H.. **Busesinrio: Buses as mobile traffic sensors: Managing the bus gps data in the city of rio de janeiro**. In: 2016 17TH IEEE INTERNATIONAL CONFERENCE ON MOBILE DATA MANAGEMENT (MDM), volumen 1, p. 369–372, June 2016.

[25] GULISANO, V.; JIMENEZ-PERIS, R.; PATINO-MARTINEZ, M.; VALDURIEZ, P.. **StreamCloud: A large scale data streaming system**. In: PROCEEDINGS - INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, p. 126–137, 2010.

[26] VIEIRA, M. R.; FRIAS-MARTINEZ, V.; OLIVER, N.; FRIAS-MARTINEZ, E.. **Characterizing Dense Urban Areas from Mobile Phone-Call Data: Discovery and Social Dynamics**. In: IEEE SECOND INTERNATIONAL CONFERENCE ON SOCIAL COMPUTING, p. 241–248. IEEE, aug 2010.

[27] LAUBE, P.; VAN KREVELD, M.; IMFELD, S.. **Finding remo — detecting relative motion patterns in geospatial lifelines**. In: DEVELOPMENTS IN SPATIAL DATA HANDLING: 11TH INTERNATIONAL SYMPOSIUM ON SPATIAL DATA HANDLING, p. 201–215. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[28] VICSEK, T.; ZAFEIRIS, A.. **Collective motion**. Physics Reports, 517(3–4):71–140, 2012.

[29] MOUSSAÏD, M.; HELBING, D.; THERAULAZ, G.. **How simple rules determine pedestrian behavior and crowd disasters**. Proceedings of the National Academy of Sciences, 108(17):6884–6888, 2011.

[30] KHALILIAN, M.; MUSTAPHA, N.. **Data Stream Clustering: Challenges and Issues**. In: PROCEEDINGS OF INTERNATIONAL MULTI CONFERENCE OF ENGINEERS AND COMPUTER SCIENTISTS, p. 566–569, 2010.

[31] CAO, F.; ESTER, M.; QIAN, W.; ZHOU, A.. **Density-Based Clustering over an Evolving Data Stream with Noise.** In: PROCEEDINGS OF THE 2006 SIAM CONFERENCE ON DATA MINING, p. 326–337, 2006.

[32] TU, L.; CHEN, Y.. **Stream Data Clustering Based on Grid Density and Attraction**. ACM Transactions on Knowledge Discovery from Data, 3(3):12:1–12:27, jul 2009.

[33] KRANEN, P.; ASSENT, I.; BALDAUF, C.; SEIDL, T.. **The ClusTree: indexing micro-clusters for anytime stream mining**. Knowledge and Information Systems, 29(2):249–272, nov 2011.

[34] YU, Y.; WANG, Q.; WANG, X.. **Continuous clustering trajectory stream of moving objects**. China Communications, 10(9):120–129, sep 2013.

[35] FLOURIS, I.; GIATRAKOS, N.; DELIGIANNAKIS, A.; GAROFALAKIS, M.; KAMP, M.; MOCK, M.. **Issues in complex event processing: Status and prospects in the Big Data era**. Journal of Systems and Software, p. 1–20, 2016.

[36] KUDYBA, S.. **Big Data, Mining, and Analytics**. Auerbach Publications, Boca Raton, Florida, 1st edition, 2014.

[37] JENSEN, C.; LIN, D.; BENG CHIN OOI; RUI ZHANG. **Effective Density Queries on Continuously Moving Objects**. In: 22ND INTERNATIONAL CONFERENCE ON DATA ENGINEERING (ICDE'06), p. 71–71. IEEE, 2006.

[38] NI, J.; RAVISHANKAR, C. V.. **Pointwise-Dense Region Queries in Spatio-temporal Databases**. In: DATA ENGINEERING, 2007. ICDE 2007. IEEE 23RD INTERNATIONAL CONFERENCE ON, p. 1066–1075, 2007.

[39] HAN, J.; KAMBER, M.; PEI, J.. **Data Mining: Concepts and Techniques**. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[40] JAIN, A. K.. **Data clustering: 50 years beyond K-means**. Pattern Recognition Letters, 31(8):651–666, jun 2010.

[41] LUCKHAM, D.; SCHULTE, R.. **Event Processing Glossary - Version 2.0**. http://www.complexevents.com/2011/08/23/event-processing-glossary-version-2/, 2011. [Online: accessed on 12/08/2016].

[42] BRAMER, M.. **Principles of data mining**. Springer, second edi edition, 2013.

[43] PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V.; VANDERPLAS, J.; PASSOS, A.; COURNAPEAU, D.; BRUCHER, M.; PERROT, M.; DUCHESNAY, E.. **Scikit-learn: Machine Learning in Python**. Journal of Machine Learning Research, 12:2825–2830, 2011.

[44] LIN, M.; HSU, W.-J.. **Mining GPS data for mobility patterns: A survey**. Pervasive and Mobile Computing, 12:1–16, jun 2014.

[45] REHMAN, S. U.; ASGHAR, S.; FONG, S.; SARASVADY, S.. **DBSCAN: Past, present and future**. The Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2014), (FEBRUARY 2014):232–238, 2014.

[46] ESPERTECH. **Esper - Complex Event Processing**, 2014.

[47] YANLEI DIAO NEIL IMMERMAN; GYLLSTROM, D.. **SASE+: An Agile Language for Kleene Closure over Event Streams**. Technical Report UM-CS-07-03, Department of Computer Science, University of Massachusetts Amherst, 2007.

[48] MICROSOFT. **Microsoft StreamInsight**, 2015.

[49] CARBONE, P.; EWEN, S.; HARIDI, S.; KATSIFODIMOS, A.; MARKL, V.; TZOUMAS, K.. **Apache Flink: Unified Stream and Batch Processing in a Single Engine**. Data Engineering, p. 28–38, 2015.

[50] PROCTOR, M.. **Drools: A Rule Engine for Complex Event Processing**. In: PROCEEDINGS OF THE 4TH INTERNATIONAL CONFERENCE ON APPLICATIONS OF GRAPH TRANSFORMATIONS WITH INDUSTRIAL RELEVANCE, AGTIVE'11, p. 2, Berlin, Heidelberg, 2012. Springer-Verlag.

[51] CHANDRASEKARAN, S.; COOPER, O.; DESHPANDE, A.; FRANKLIN, M. J.; HELLERSTEIN, J. M.; HONG, W.; KRISHNAMURTHY, S.; MADDEN, S. R.; REISS, F.; SHAH, M. A.. **TelegraphCQ: Continuous Dataflow Processing**. In: PROCEEDINGS OF THE 2003 ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, SIGMOD '03, p. 668, New York, NY, USA, 2003. ACM.

[52] CUGOLA, G.; MARGARA, A.. **Processing Flows of Information: From Data Stream to Complex Event Processing**. ACM Computing Surveys, 44(3):1–62, jun 2012.

[53] ARASU, A.; BABU, S.; WIDOM, J.. **The CQL continuous query language: semantic foundations and query execution**. The VLDB Journal, 15(2):121–142, jul 2005.

[54] SAGIROGLU, S.; SINANC, D.. **Big data: A review**. International Conference on Collaboration Technologies and Systems (CTS), p. 42–47, 2013.

[55] BOUTSIS, I.; KALOGERAKI, V.; GUNOPULOS, D.. **Efficient Event Detection by Exploiting Crowds**. In: PROCEEDINGS OF THE 7TH ACM INTERNATIONAL CONFERENCE ON DISTRIBUTED EVENT-BASED SYSTEMS, DEBS '13, p. 123–134, New York, NY, USA, 2013. ACM.

[56] WAND, M. P.; JONES, M. C.. **Kernel Smoothing**. Monographs on Statistics and Applied Probability. Chapman and Hall, London, 1994.

[57] JENSEN, C. S.; LIN, D.; OOI, B. C.. **Continuous Clustering of Moving Objects**. Knowledge and Data Engineering, IEEE Transactions on, 19(9):1161–1174, 2007.

[58] FORESTIERO, A.; PIZZUTI, C.; SPEZZANO, G.. **A single pass algorithm for clustering evolving data streams based on swarm intelligence**. Data Mining and Knowledge Discovery, 26(1):1–26, 2013.

[59] CHEN, Y.; TU, L.. **Density-based Clustering for Real-time Stream Data**. In: PROCEEDINGS OF THE 13TH ACM SIGKDD INTERNATIONAL CONFERENCE ON KNOWLEDGE DISCOVERY AND DATA MINING, KDD '07, p. 133–142, New York, NY, USA, 2007. ACM.

[60] AMINI, A.; YING, W.. **DENGRIS-Stream: A density-grid based clustering algorithm for evolving data streams over sliding window**. In: PROC. INTERNATIONAL CONFERENCE ON DATA MINING AND COMPUTER ENGINEERING, p. 206–210, 2012.

[61] KIM, Y.; SHIM, K.; KIM, M.-S.; SUP LEE, J.. **DBCURE-MR: An efficient density-based clustering algorithm for large data using MapReduce**. Information Systems, 42(0):15–35, jun 2014.

[62] MOUZA, C.; RIGAUX, P.. **Mobility Patterns**. GeoInformatica, 9(4):297–319, 2005.

[63] FLORESCU, S.-C.; MOCK, M.; KÖRNER, C.; MAY, M.. **Efficient mobility pattern stream matching on mobile devices**. In: 2ND WORKSHOP ON UBIQUITOUS DATA MINING, UDM 2012 : IN CONJUNCTION WITH THE 20TH EUROPEAN CONFERENCE ON ARTIFICIAL INTELLIGENCE (ECAI 2012), MONTPELLIER, número Ecai, p. 23–27. ECCAI, 2012.

[64] BAROUNI, F.; MOULIN, B.. **An extended complex event processing engine to qualitatively determine spatiotemporal patterns**. In: PROCEEDINGS OF GLOBAL GEOSPATIAL CONFERENCE 2012, p. 201, Quebec City, 2012.

[65] KIM, B.; LEE, S.; LEE, Y.; HWANG, I.; RHEE, Y.; SONG, J.. **Mobiiscape: Middleware support for scalable mobility pattern monitoring of moving objects in a large-scale city**. Journal of Systems and Software, 84(11):1852–1870, 2011.

[66] SUHOTHAYAN, S.; GAJASINGHE, K.; LOKU NARANGODA, I.; CHATURANGA, S.; PERERA, S.; NANAYAKKARA, V.. **Siddhi: A Second Look at Complex Event Processing Architectures**. In: PROCEEDINGS OF THE 2011 ACM WORKSHOP ON GATEWAY COMPUTING ENVIRONMENTS - GCE '11, p. 43, New York, New York, USA, 2011. ACM Press.

[67] JEUNG, H.; SHEN, H. T.; ZHOU, X.. **Mining Trajectory Patterns Using Hidden Markov Models**. In: Song, I. Y.; Eder, J.; Nguyen, T. M., editors, DATA WAREHOUSING AND KNOWLEDGE DISCOVERY, chapter Mining Tra, p. 470–480. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[68] JEUNG, H.; YIU, M. L.; ZHOU, X.; JENSEN, C. S.; SHEN, H. T.. **Discovery of convoys in trajectory databases**. Proc. VLDB Endow., 1(1):1068–1080, Aug. 2008.

[69] MANNING, C. D.; RAGHAVAN, P.; SCHÜTZE, H.. **Introduction to Information Retrieval**. Cambridge University Press, New York, NY, USA, 2008.

[70] DAVID, L.; VASCONCELOS, R.; ALVES, L.; ANDRÉ, R.; ENDLER, M.. **A DDS-based middleware for scalable tracking, communication**

and collaboration of mobile nodes. Journal of Internet Services and Applications (JISA), 4(1):1–15, 2013.

[71] RORIZ JUNIOR, M.; ENDLER, M.; SILVA E SILVA, F.. **An on-line algorithm for cluster detection of mobile nodes through complex event processing**. Information Systems, 64:303 – 320, 2017.

[72] RORIZ JUNIOR, M.; ENDLER, M.; CASANOVA, M. A.; LOPES, H.; SILVA E SILVA, F.. **A Heuristic Approach for On-line Discovery of Unidentified Spatial Clusters from Grid-Based Streaming Algorithms**, p. 128–142. Springer International Publishing, Cham, 2016.

[73] RORIZ JUNIOR, M.; ENDLER, M.. **DG2CEP: A density-grid stream clustering algorithm based on complex event processing for cluster detection**. In: VI SIMPÓSIO BRASILEIRO DE COMPUTAÇÃO UBÍQUA E PERVASIVA, Brasília, Brazil, 2014. SBC.

[74] BAPTISTA, G. L. B.; RORIZ, M.; VASCONCELOS, R.; OLIVIERI, B.; VASCONCELOS, I.; ENDLER, M.. **On-line Detection of Collective Mobility Patterns through Distributed Complex Event Processing**. Pontifícia Universidade Católica do Rio de Janeiro, Technical Report MCC-06/13, ISSN 0103-9741, 2013.