FlinkCEP zur räumlichen Clustererkennung

Niels Schneider
Fakultät Informatik
Hochschule Furtwangen
Furtwangen, Deutschland
niels.schneider@hs-furtwangen.de

Cornelius Schaub
Fakultät Informatik
Hochschule Furtwangen
Furtwangen, Deutschland
cornelius.schaub@hs-furtwangen.de

***Abstract*—Diese Arbeit behandelt das Erkennen von Personengruppen hinsichtlich Verstößen gegen die Corona Auflagen in kontinuierlichen Positionsereignisströmen mittels FlinkCEP. Das Erfassen räumlicher Cluster mobiler Objekte, z. B. Personen oder Fahrzeuge, bietet zahlreiche Anwendungsgebiete. Dabei birgt die Echtzeiterkennung solcher Cluster in großen Ereignisströmen einige Herausforderungen. Complex Event Processing (CEP) bietet eine Technologie, Muster in Ereignisströmen nahezu in Echtzeit zu erfassen. Eine solche Technologie kann verwendet werden, um Personengruppen in einem kontinuierlichen Ereignisstrom an Positionsdaten auszumachen. Vorangegangene Arbeiten zeigen, dass gitterbasierte Cluster-Algorithmen zur Anwendung mithilfe CEP vielversprechend sind. Aufgrund dessen wurde im Zuge dieser Arbeit ein dichtebasierter Gitter-Clustering-Algorithmus unter der Verwendung von Complex Event Processing mit FlinkCEP implementiert.**

***Index Terms*—Complex Event Processing, CEP, FlinkCEP, Clustererkennung**

I. Einleitung

Während in der Vergangenheit Daten zu Stapeln (Batches) gesammelt wurden, nimmt die Relevanz von Echtzeitdaten heutzutage stetig zu [1]. Diese Daten werden kontinuierlich produziert und konsumiert, allgemein ist dies auch als Streaming bekannt [2]. Traditionelle Geografische Informationssysteme (GIS) sind für die Stapelverarbeitung von Daten optimiert und stehen vor großen Herausforderungen im Angesicht der inhärenten Eigenschaften von Streaming-Daten [2]. Während die Forschung um Streaming-Daten ihr Augenmerk auf Finanztransaktionen oder industrielle Produktion legt, rücken räumlich referenzierte Daten zunehmend zu einem neuen Schwerpunkt in der Wissenschaft und Industrie zusammen [1], [3]. Die Aggregation individueller Positionsdaten kann verwendet werden, um das kollaborative Verhalten einer Gruppe zu analysieren, wie z. B. die Darstellung der Verkehrsbedingungen in einer urbanen Umgebung auf Grundlage von GPS-Sensoren, sowie das epidemiologische Verhalten einer Gruppe [3]. Complex Event Processing ist eine Technologie definierte Muster in einem Ereignisstrom zu erkennen und so eine Echtzeiterkennung von Clustern zu ermöglichen [4].

II. Grundlagen

A. Räumliches Clustering

Räumliches Clustering beschreibt den Prozess, räumliche Daten, wie sie beispielsweise durch bewegte Objekte auftreten, einzelnen Clustern zuzuweisen. Um dieses Konzept zu veranschaulichen, betrachten wir die räumlichen Daten von Personen in Abbildung 1. Die Abbildung zeigt die Cluster C_1, C_2 und C_3. Jedes Cluster enthält mindestens zwei sich nah zueinander bewegende Objekte, in diesem Fall Personen. Personen, welche sich abseits von andern Personen befinden, gehören keinem Cluster an und werden als $Rauschen$ betitelt. Betrachtet man nun den zeitlichen Aspekt und die Bewegungsrichtung der einzelnen Personen, kann man unter anderem annehmen, dass die Cluster C_1 und C_3 in Zukunft verschmelzen werden, sich aber vorab die Gruppenanzahl der Personengruppe C_1 um eine Person verringern wird.

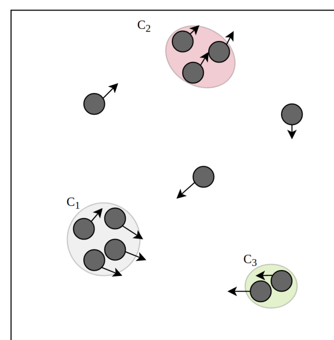

Abb. 1. Räumliche Cluster sich bewegender Objekte

B. Dichtebasiertes Räumliches Clustering

Zur Bestimmung eines Clusters können zahlreiche Clustering-Algorithmen verwendet werden. Hierbei unterscheiden sich diese in ihrer Funktionsweise sowie Parametrisierung. Betrachtet man die Punkte in Abbildung 1, so lassen sich drei Cluster identifizieren. Cluster werden anhand der Anzahl und Distanz zwischen den Punkten erkannt. So ist die Punktdichte außerhalb eines Clusters deutlich geringer als die Dichte in einem der Cluster.

Der *Density-Based Spatial Clustering of Applications with Noise* (DBSCAN) [5] Algorithmus ist ein klassisches Beispiel zur Bestimmung einer unbekannter Anzahl an Clustern aufgrund ihrer Punktdichte. Die grundlegende Idee des Algorithmus ist, dass jeder Punkt, der in seiner mit einem Radius (ϵ) bestimmten Nachbarschaft eine Mindestanzahl weiterer Punkte $minPts$ enthält, zu einem Cluster gehört. Die Form

einer Nachbarschaft wird hierbei durch die Wahl einer Abstandsfunktion für die zwei Punkte p, q bestimmt und mit $dist(p, q)$ angegeben. Jede Abstandsfunktion, wie z. B. die Euklidische, kann dabei verwendet werden [5]. Formal lässt sich die ϵ-Nachbarschaft eines Punktes p aus einem Satz an Positionsdaten D wie folgt beschreiben:

$$N_\epsilon(p) = \{q \ \in D | dist(p, q) \leq \epsilon\}$$

Um nun ein Cluster zu bestimmen wird jeder Punkt, welcher in seinem Radius ϵ den Schwellwert $minPts$ überschreitet als Kernpunkt betitelt. Da weitere Punkte zu einem Cluster gehören könnten, welche den Anforderungen einer Schwellwertüberschreitung in ihrer Nachbarschaft nicht genügen, wird jeder Punkt im Radius ϵ eines Kernpunktes, welcher selbst kein Kernpunkt ist, als Randpunkt markiert (siehe Abbildung 2). Weitere Punkte im Radius des Randpunktes werden nicht dem Cluster hinzugefügt und gelten als Rauschen.

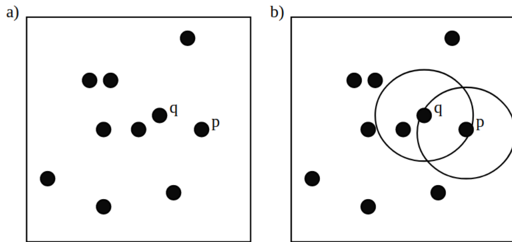

Abb. 2. Bestimmung von Kernpunkten q und Randpunkten p mit $minPts = 3$

Punkte gehören zum selben Cluster, wenn diese dichteerreichbar sind. Als dichteerreichbar gilt ein Punkt p in Bezug auf ϵ, sollte eine Kette an Punkten $p_1, ..., p_n$ existieren, durch welche die Punkte p_1 bis p_n direkt miteinander verbunden sind (siehe Abbildung 3). Existiert eine solche Erreichbarkeit, gelten alle beteiligten Punkte als dichteverbunden [5].

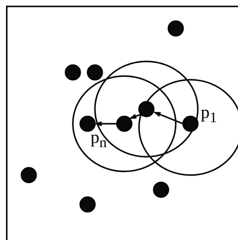

Abb. 3. Dichte-Erreichbarkeit von p_1 zu p_n

Ist ein Cluster zu einem anderen dichteerreichbar, verschmelzen diese miteinander zu einem größeren Cluster. Ist ein Cluster nicht von einem anderen erreichbar, existiert dieses getrennt. Dadurch ist es möglich, eine vorab unbekannte Anzahl an Clustern zu ermitteln.

Zur Ermittlung der ϵ-Nachbarschaft $N_\epsilon(p)$ muss der Abstand eines Punktes p zu allen übrigen Punkten bestimmt werden. Dies kann zwar durch die Verwendung räumlicher Datenstrukturen wie R-Bäume optimiert werden, jedoch ist dieser Ansatz für kontinuierlich eintreffende Daten eines Datenstroms schwierig anpassbar [4]. Um den Ansatz des DBSCAN-Algorithmus in Ereignisströmen verwenden zu können, wird dieser in einen Online- und Offline-Part aufgeteilt [6]. Hierbei werden Daten während der Online-Phase aus dem Datenstrom D zu einem Stapel gesammelt und anschließend in einer Offline-Phase zu Clustern verarbeitet. Diese Arbeitsweise ermöglicht jedoch keine kontinuierliche Auslieferung aktueller Cluster-Informationen.

C. Dichtebasiertes Gitter-Clustering

Bei der Verwendung hochskalierbarer Cluster-Algorithmen erfreuen sich dichtebasierte Gitter-Clustering-Algorithmen an zunehmender Beliebtheit [7]. Dieser Ansatz teilt den durch $[x_{min}, x_{max}]$ und $[y_{min}, y_{max}]$ definierten Raum in das namensgebende Gitter G auf. Wie in Abbildung 4 ersichtlich, wird die diagonale Größe einer Gitterzelle durch den Parameter ϵ bestimmt. Dabei gewährleistet ϵ den maximalen Abstand zweier Positionspunkte in einer Gitterzelle.

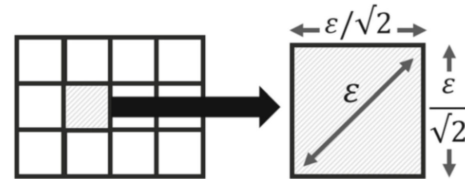

Abb. 4. Gitterzellengröße $\frac{\epsilon}{\sqrt{2}} \times \frac{\epsilon}{\sqrt{2}}$

Jedes sich im Gitter befindende Objekt wird auf einer Gitterzelle G_{ij} abgebildet. Dies wird effizient durch die Verwendung der Formeln

$$i = \frac{x - x_{min}}{\epsilon\sqrt{2}}$$
$$j = \frac{y - y_{min}}{\epsilon\sqrt{2}}$$

ermöglicht. Wird ein Objekt auf einer Gitterzelle abgebildet, so steigt ein Zähler für diese. Ist der Zähler einer Gitterzelle größer als ein vorab definierter Schwellwert $minPts$, gilt die Zelle als dicht (Dense-Cell). Ist eine Zelle dicht, zählt diese bereits als Cluster. Befinden sich dichte Zellen direkt aneinander angrenzend, bilden sie ein zusammenhängendes Cluster. Fällt die Anzahl an Objekten einer dichten Zelle unter $minPts$, zählt diese nicht weiter als dicht (Sparse-Cell).

Dichtebasierte Gitter-Clustering-Algorithmen eignen sich für hochparallelisierte Anwendungen. Dies resultiert unter anderem aus der guten Teilbarkeit eines Gitters in eine Vielzahl kleinerer Bereiche. Die Partitionierung des Gitters in viele kleinere Gitter ermöglicht in Betracht auf Streaming-Daten bereits eine Gruppierung einzelner Ereignisse in getrennte Datenströme [4]. Ebenso muss nicht wie beim DBSCAN eine aufwendige Berechnung der Distanz stattfinden, sondern ausschließlich ein Zähler je Gitterzelle G_{ij} verwaltet werden.

D. Complex Event Processing

Complex Event Processing Systeme erhalten einen kontinuierlichen Datenstrom an Ereignissen und generieren mittels

eines vorab definierten Musters (Pattern) in Echtzeit einen Mehrwert aus diesen (siehe Abbildung 5).

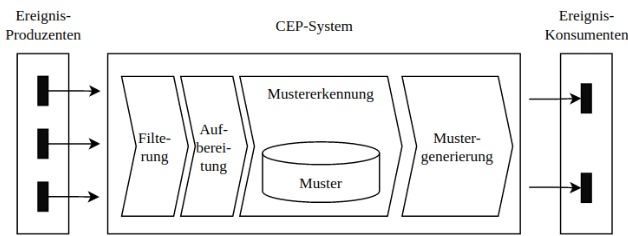

Abb. 5. Prinzip des Complex Event Processing

Ein typisches Ereignis (Event) ist strukturell als Tupel an Werten aufgebaut. Die grundlegenden Attribute eines einfachen Ereignisses sind dabei sein Typ (z. B. PositionsEvent) sowie ein Zeitstempel der Entstehung des Events [8]. Des Weiteren kann ein Ereignis eigene Attribute, wie im Falle eines Positionsereignisses die x und y-Positionen, enthalten. CEP-Systeme definieren nun eine ausdrucksstarke Sprache zur kontinuierlichen Erkennung komplexerer Ereignisse (CE) ce. Die primären Operatoren einer solchen Sprache laut Alevizos et al. [9] sind:

- *Sequenz*: Eine Reihe an zeitlich aufeinander folgenden Ereignissen ($ce ::= ce_1; ce_2$).
- *Disjunktion*: Das Auftreten mindestens eines von zwei zeitlich unabhängigen Ereignissen ($ce ::= ce_1 \lor ce_2$).
- *Iteration*: Ein Ereignis, welches N-mal hintereinander auftritt, wobei $N \geq 0$ ($ce ::= ce^*$).
- *Konjunktion*: Zwei Events, welche unabhängig ihres zeitlichen Zusammenhanges auftreten ($ce ::= ce_1 \land ce_2$).
- *Negation*: Die Abwesenheit eines Events ($ce ::= \neg ce_1$).
- *Selektion*: Die Auswahl der Events, dessen Attribute einem Satz an Prädikaten und oder Relationen entsprechen ($ce ::= \sigma_\theta(ce)$).
- *Projektion*: Die Rückgabe eines Ereignisses, dessen Attributwerte eine möglicherweise transformierte Teilmenge der Attributwerte seiner Unterereignisse sind ($ce ::= \pi_m(ce)$).
- *Zeitfenster*: Das Auftreten eines Events in einem vorgegebenen festen Zeitfenster $[T_1, T_2]$ ($ce ::= [ce]\frac{T_2}{T_1}$).

Komplexe Events bilden sich aus mehreren Events, welche einem Muster aus den gelisteten Operatoren genügen. Ein Muster kann dabei in unterschiedlichen Modellen definiert werden.

Da temporale Operatoren der CEP-Sprache regulären Ausdrücken ähneln, erweisen sich unter anderem nichtdeterministische Automaten als sehr beliebt zur Modellierung dieser [8]. Dabei wird in der Regel ein Automat verwendet, um die Semantik des Musters zu beschreiben. Ein solcher Automat erhält einen Strom einfacher Events und ändert seine internen Zustände. Zustandsübergänge erfolgen je nach Prädikat für eine ausgehende Transition. Bei einem Zustandsübergang kann das dafür verantwortliche Event gespeichert werden, sollte es zu einem späteren Zeitpunkt gebraucht werden. Ist der Endzustand des Automaten erreicht, besteht eine vollständige Übereinstimmung mit dem vorab definierten Muster. Ist der Automat in einem nicht finalen Zustand, besteht eine teilweise Übereinstimmung des Patterns. Auch wenn die Modellierung eins CEP-Musters in einer anderen Sprache erfolgt, wird diese häufig später zu einem Automaten kompiliert [8].

Als wichtige Anwendungsgebiete für CEP-Systeme nennt Hedtstück [10] hierbei:

- *Smart Home*: Zur Erkennung gefährlicher Situationen, wie ein unbewachter eingeschalteter Herd ohne anwesenden Personen im Haus.
- *Überwachung von Fahrzeugen*: GPS-Daten eines Fahrzeuges zur frühzeitigen Erkennung von Staubildungen.
- *Aktienhandel*: Erkennung von lokalen Minima und Maxima zur Ermittlung geeigneter Kauf- und Verkaufsmomente.
- *Geschäftsprozessmanagement*: Erkennung von Schwellwertsunterschreitungen für Lagerbestände.
- *Digitalisierung in der Fertigungsindustrie*: Fertigungsüberwachung.
- *Patienteüberwachung*: Frühzeitige Erkennung gesundheitlicher Probleme anhand Vital-Events.
- *Betrugserkennung im Onlinebanking*: Erkennung verdächtiger Kontoaktivitäten.

E. FlinkCEP Bibliothek

Ein Vertreter der automatenbasieten CEP-Systeme ist FlinkCEP [11]. FlinkCEP ist eine Open-Source Bibliothek der Apache Software Foundation [12] und Teil des Apache Flink Frameworks [13] zur Erkennung komplexer Ereignisse. FlinkCEP verwendet keine gesonderte Sprache zur Definition von Mustern wie bspw. *ESPER* [2]. Ein Pattern wird als Java oder Scala Code mithilfe der von Flink gegebenen API definiert (siehe Abbildung 6). Eine komplexe Mustersequenz entsteht durch die logische Verknüpfung mehrerer Muster. Muster haben einen Namen und eine oder mehrere Bedingungen, die für dieses Muster erfüllt werden müssen. Bedingungen können Wahrheitsaussagen über ein oder mehrere Events, die Angabe des erforderlichen Eventtypen, die Anzahl notwendiger Auftritte, oder eine Zeitspanne sein, die eine zeitliche Einschränkung definiert, damit das Muster gültig ist [11]. Die Definition einer komplexer Mustersequenzen beginnt immer mit dem Aufruf von `Pattern.begin` [11]. Zugleich ist dies der Beginn des ersten einfachen Musters. Weitere einfache Muster können durch den Aufruf von `next`, `followedBy`, `followedByAny`, `notNext` und `notFollowedBy` miteinander verknüpft werden [11].

FlinkCEP unterstützt dabei die folgenden Stärken der Kontiguität zwischen Ereignissen verschiedener Muster [11]:

- *Strenge Kontiguität*: Alle übereinstimmenden Ereignisse müssen streng hintereinander erscheinen, ohne dass zwischen ihnen weitere übereinstimmende Ereignisse auftreten.
- *Entspannte Kontiguität*: Ignoriert alle nicht übereinstimmende Ereignisse, die zwischen den übereinstimmenden Ereignissen auftreten.

- *Nicht-deterministische entspannte Kontiguität*: Lockert weiter die Kontiguität, wodurch zusätzliche Übereinstimmungen ermöglicht und einige übereinstimmende Ereignisse ignoriert werden können.

Das definierte Muster kann nun unter der Verwendung von `CEP.pattern` mit der Angabe des eingehenden Streams angewendet werden.

```
val input : DataStream[Event] = ...

val partitionedInput = input.keyBy(event => event.getId)

val pattern = Pattern.begin[Event]("start")
  .next("middle").where(_.getName == "error")
  .followedBy("end").where(_.getName == "critical")

val patternStream = CEP.pattern(partitionedInput, pattern)

val alerts = patternStream.select(createAlert(_))
```

Abb. 6. Prinzip des Complex Event Processing

Der in Abbildung 6 gezeigte Scala-Codeausschnitt zeigt ein FlinkCEP Muster. Dieses findet Anwendung in folgendem Szenario: Eine Produktionsstraße besitzt Maschinen, welche Events bei einer Zustandsänderung erzeugen. Eine Maschine kann dabei einen der Zustände *working*, *error* sowie *critical* annehmen. Sollte ein *error*-Zustand auftreten, probiert die zugehörige Maschine diesen selbständig aufzulösen. Ist das Beheben des *error*-Zustands möglich, wechselt die Maschine anschließend in den *working*-Zustand. Ist dies nicht möglich, geht die Maschine in den Zustand *critical* über. Wechselt eine Maschine ihren Zustand von *error* zu *critical*, so ist ein menschliches Eingreifen notwendig. Das dabei in Abbildung 6 definierte Pattern erkennt diesen Zustandsübergang und erzeugt ein Benachrichtigungsevent.

III. Corona-Verstosserkennung

Folgender Abschnitt behandelt den Aufbau, sowie die Implementierung eines auf CEP basierenden Cluster-Algorithmus zur Corona-Verstoßerkennung. Ziel ist es dabei, in einem kontinuierlichen Datenstrom an Positionsereignissen Cluster zu erkennen. Die Erkennung eines Clusters dient dazu, Personengruppen auszumachen und im Falle eines Corona-Verstoßes eine Meldung der Gruppenposition als Event auszugeben.

Dabei sieht der Aufbau wie in Abbildung 7 dargestellt aus. Einzelne Bestandteile des Aufbaus werden in den folgenden Sektionen erörtert.

A. Gruppen-Positionsdaten-Generator

Da kein Zugang zu realen Positionsdaten bestand, welche bspw. durch Computer Vision erfassbar wären, wurde zu Beginn der Arbeit ein Datengenerator entwickelt. Diese platziert zufällig n Punkte in einem durch $[x_{min}, x_{max}]$, $[y_{min}, y_{max}]$ begrenzten Raum (siehe Abbildung 7). Dabei wird jeder Punkt bei seiner Initiierung mit einer Geschwindigkeit in

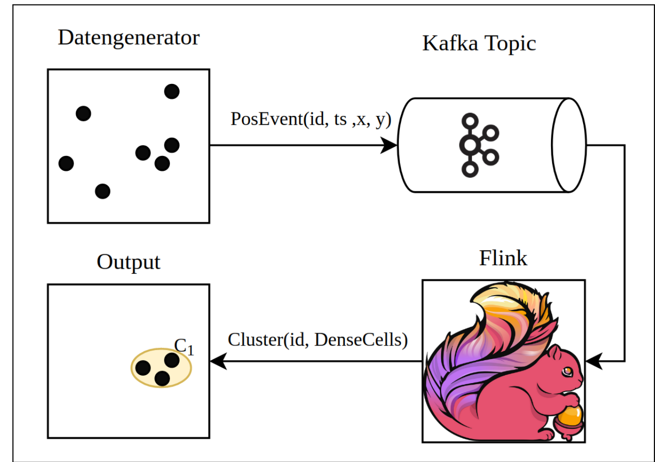

Abb. 7. Aufbau des Proof of Concept

eine willkürliche Richtung bewegt. Jeder Punkt der Simulation besitzt Kollisionseigenschaften. Trifft er gegen den Rand des Raumes oder einen anderen Punkt, prallt er ohne Geschwindigkeitsverlust ab. Dieses Verhalten ermöglicht eine kontinuierliche Bildung sowie Auflösung zufälliger Cluster.

Da im Laufe der Arbeit festgestellt wurde, dass die Willkürlichkeit des Generators ungeeignet ist, um Szenarien, wie das gezielte Bilden oder Auflösen von Gruppen zu testen, wurde ein weiterer Generator nach diesen Anforderungen umgesetzt. Ein Canvas auf einer HTML-Seite erlaubt die Erzeugung eines Punktes, durch Klicken in freie Bereiche. Dieser farbige Punkt repräsentiert eine Person, die sich im Raum befindet. Punkte können im Raum durch einfaches drag-and-drop bewegt werden. Eine andere Möglichkeit, Clusterbewegungen zu simulieren, besteht darin, alle Punkte, mithilfe von 4 Knöpfen, gleichzeitig in eine Richtung zu bewegen (siehe Abbildung 8). Um die Erzeugung von Clustern zu vereinfachen, wurden zusätzlich Linien in das Canvas gezeichnet, welche den Raum in die einzelnen Zellen des Gittern unterteilen.

Die Positionsdaten aller Punkte werden in einem Intervall von $\frac{1}{2}$ Sekunden im JSON-Format an ein Python-Backend gesendet. Dieses leitet die übergebenen Positionsdaten an das Messaging-System weiter.

B. Messaging-System

Kafka ist ein ursprünglich von LinkedIn entwickeltes, verteiltes Publish-Subscribe Messaging-System, welches sich gerade durch seine enorme Skalierbarkeit sowie hohen Durchsatz auszeichnet [14]. Zur Übermittlung der generierten Positionsdaten senden beide Simulationsumgebungen, mittels des *KafkaProducer* [15] der *kafka-python* Bibliothek, kontinuierlich in einem Zeitintervall Δ *LocationEvent* $\langle id, ts, x, y \rangle$ Events bestehend aus einer Identifikationsnummer id, einem Zeitstempel ts und den XY-Koordinaten x, y an ein Kafka-Topic (siehe Abbildung 7).

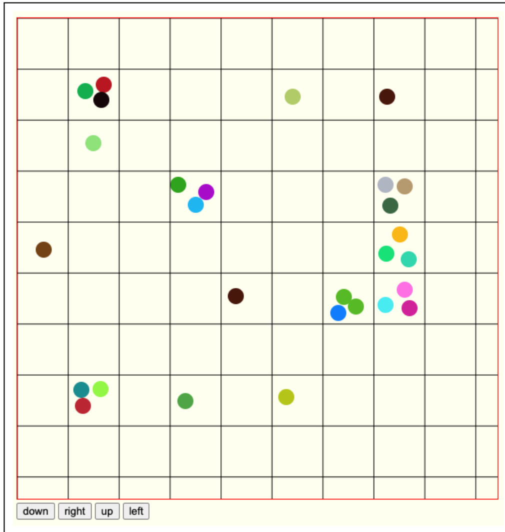

Abb. 8. HTML-Seite zur Generierung von Positionsdaten

C. Clustererkennung mittels FlinkCEP

Die Erkennung einzelner Personengruppen kann mittels eines dichtebasiertes Clustering-Algorithmus erzielt werden. Wie in Kapitel II. Abschnitt C. beschrieben, eignen sich hier, zur Erkennung solcher Cluster, dichtebasierte Gitter-Clustering-Algorithmen. Um ein kontinuierliches Ermitteln der Cluster in nahezu Echtzeit zu ermöglichen, wird ein Clustering-Algorithmus mittels Complex Event Processing durch die FlinkCEP Bibliothek implementiert.

Basierend auf der Arbeit von Roriz et al. [4] zur Implementierung eines CEP Cluster-Algorithmus in *ESPER* [16] wurde in dieser Arbeit eine abgewandelte, auf den Versuch zugeschnittene Form des Algorithmus in FlinkCEP implementiert.

Der in [4] beschriebene Algorithmus arbeitet grob in den folgenden Schritten.

1) Zuweisen eingehender Position-Events zu entsprechenden Gitterzellen G_{ij}.
2) Erkennung dichter Zellen.
3) Erkennung einer aufgelösten dichten Zelle.
4) Zusammenfassung von aneinander angrenzenden dichten Zellen zu einem Cluster.

Zu Beginn müssen die vom Datengenerator produzierten Position-Events aus dem Kafka-Topic geladen werden. Hierzu kann der Flink eigene Kafka-Konsument *KafkaSource* verwendet werden. Dieser projiziert den aus der *KafkaSource* erhaltene JSON-String-Stream auf einen entsprechenden *LocationEvent*-Stream. Der entstandene *LocationEvent*-Stream wird nun in einem weiteren Schritt zu einem *LocationUpdateEvent*-Stream abgebildet. Dazu werden mit den in Kapitel II. Abschnitt C. vorgestellten Funktionen $i = \frac{x - x_{min}}{\epsilon\sqrt{2}}$ und $j = \frac{y - y_{min}}{\epsilon\sqrt{2}}$ die Indizes der Gitterzelle G_{ij} bestimmt. Der entstandene Datenstrom an *LocationUpdateEvent*-Events wird anhand der jeweiligen Indizes in einen *KeyedStream* partitioniert. Die Eigenschaft dieser Partitionen sorgt dafür, dass die zu einer Gitterzelle gehörigen *LocationUpdateEvent*-Events nicht auf mehrere Rechnerknoten verteilt werden und somit der Zustand des Automaten (Muster) nicht verteilt verwaltet werden muss. Der partitionierte *LocationUpdateEvent*-Stream kann nun weiter verwendet werden, um auf jeder Partition nach Mustern zu suchen.

Dichte Zellen werden durch ein Muster erkannt, welches in Abbildung 9 dargestellt ist. Dieses Muster wird auf die *LocationUpdateEvent* Events einer einzelnen Gitterzelle angewandt. In diesem Muster werden *LocationUpdateEvent* Events solange gesammelt, bis eine Anzahl einzigartiger Produzenten von *LocationUpdateEvent* Events $\geq minPts$ erreicht wurde. Gibt es eine vollständige übereinstimmung der Muster, wird ein ein *DenseCellEvent* $\langle LocationUpdateEvents, i, j\rangle$ gesendet.

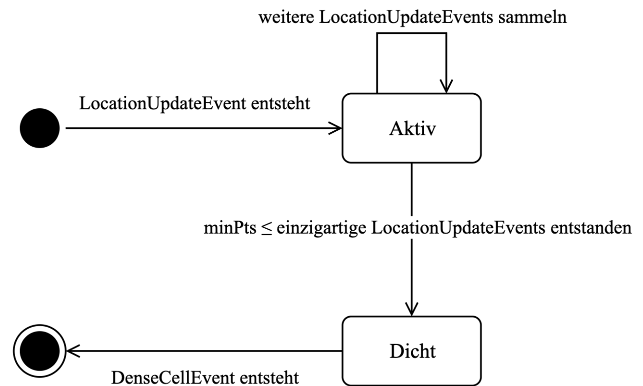

Abb. 9. Zustandsdiagramm der Entstehung einer dichten Zelle

Aufgrund der Natur sich bewegender Personen, kann eine vorab dichte Zelle den Schwellwert $minPts$ unterschreiten und dadurch ihre Dichte verlieren. Die Funktionsweise dieses Patterns ist dabei in Abbildung 10 aufgezeigt. Dabei verweilt das Pattern, solange *DenseCellEvent*-Events eintreffen, in seinem ersten, internen Zustand. Sollte innerhalb eines Zeitintervalls Δ kein weiteres *DensCellEvent* eintreffen, gilt die Zelle als aufgelöst. Ist dies der Fall, wird ein *DispersedCellEvent* ausgelöst.

Eine Einschränkung von FlinkCEP ist, dass ein Muster immer nur dann evaluiert wird, sobald ein neues Event eintrifft. Diese macht es unmöglich, das Nichteintreffen von Events als eigenes Event zu erkennen. Damit besitzt FlinkCEP nicht den in Kapitel II. Abschnitt D. gelisteten *Negierungs*-Operator. Ausbleibende Events zu erkennen, ist in dem Muster (siehe Abbildung 10) aber notwendig, sodass ein Workaround implementiert werden muss.

Der Workaround sieht vor, zu jedem *DenseCellEvent* verzögert ein *PingEvent* auszulösen. Hiermit wird erreicht, dass das Muster evaluiert wird, obwohl ein *PingEvent* nicht relevant für das Muster ist. Die Verzögerung des *PingEvent* sollte etwas größer als Δ sein, damit das Muster immer

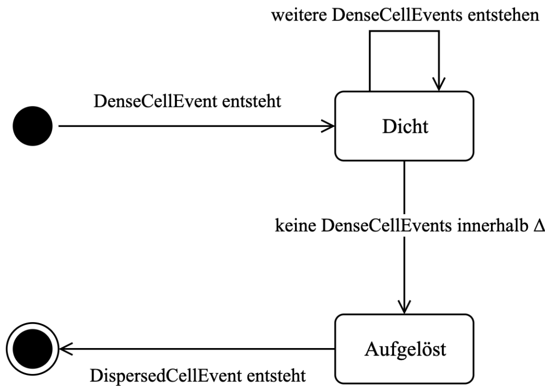

Abb. 10. Zustandsdiagramm einer sich auflösenden Zelle

dann evaluiert wird, sobald die Auflösung einer dichten Zelle möglich ist.

Der entstehende Datenstrom an *DenseCellEvent*-Events und *DispersedCellEvent*-Events wird nun durch `union` zu einem Datenstrom vereint. Dies ermöglicht eine Betrachtung aller Events des Gitters G und wird benötigt, um aneinander angrenzende dichte Zellen zu einem gemeinsamen Cluster zu verschmelzen. Die Events aus dem vereinten Datenstrom werden an ein State-Objekt übergeben, welche diese Events verarbeitet. Von diesem State-Objekt werden die ermittelten Cluster, mit ihrer Identifikationsnummer und Indizes der zugehörigen Zellen, ausgegeben. Damit die Events des vereinten Datenstroms nicht auf verschiedene Rechnerknoten verteilt werden können, wird der Datenstrom mit einem statischen Wert partitioniert, sodass nur eine Partition entstehen kann. Vorteil dieses Workarounds ist, dass nun sichergestellt ist, dass alle *DenseCellEvent*-Events und *DispersedCellEvent*-Events in einem einzigen State-Objekt verarbeitet werden. Da das State-Objekt Hash-Tabellen verwendet, ist die Komplexität für alle Operationen $O(1)$.

Die Ausgabe des State-Objekts kann nun verwendet werden, um Corona-Verstöße zu erkennen und dementsprechend auf sie zu reagieren. Alle an das Proof of Concept gestellten Anforderungen sind somit erfüllt.

D. Limitationen

Durch das Verwenden gitterbasierter Clustering-Algorithmen sind schnelle Aussagen zu Clustern möglich. Diese erkennen jedoch im Vergleich zum DBSCAN-Algorithmus nicht alle Cluster. Dies resultiert aus dem Blind-Spot-Problem, wie in Abbildung 11 dargestellt. Auch wenn hier ein Mensch einfach einen Cluster identifizieren kann, so wäre keine der Zellen bei einer $minPts$ von 4 dicht.

Eine Lösung des Blind-Spot-Problems wird von Roriz et al. [4] durch heuristische Methoden präsentiert.

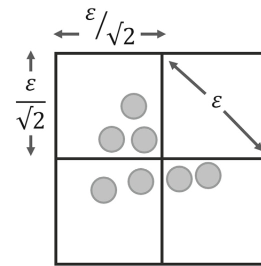

Abb. 11. Probleme mit Gittern

IV. Fazit

A. Community

Die Community um FlinkCEP besteht aus einer kleinen von Experten geprägten Gruppe. Dies und die geringe Anzahl an Anleitungen und Tutorials erschweren den Einstieg in die Verwendung von FlinkCEP. Des Weiteren musste aufgrund des Fehlens eines $Negierungs$-Operators ein Workaround implementiert werden. Bei der Suche nach einem geeigneten Workaround wurde zwar ein am 7. August 2017 erstellter Pull-Request [17] zur Erweiterung der Bibliothek um eine $Negierungs$-Funktionalität gefunden, dieser fand jedoch aufgrund ungenügender Reviews keinen Weg in FlinkCEP.

B. Performance

Durch den Einsatz von CEP zur Mustererkennung von dichten Zellen ist kein Windowing erforderlich. Das hat zur Folge, dass *DenseCellEvent*-Events schneller erkannt werden können. Außerdem kann durch die Partitionierung der Event-Streams eine hohe Skalierbarkeit erreicht werden. Dies zeigt eine hohe Eignung des Complex Event Processings für Anwendungen mit räumlichem Clustering.

References

[1] M. Rieke, L. Bigagli, S. Herle, S. Jirka, A. Kotsev, T. Liebig, C. Malewski, T. Paschke, and C. Stasch, “Geospatial IoT—the need for event-driven architectures in contemporary spatial data infrastructures,” vol. 7, no. 10, p. 385. [Online]. Available: http://www.mdpi.com/2220-9964/7/10/385

[2] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, “A survey on the evolution of stream processing systems.” [Online]. Available: http://arxiv.org/abs/2008.00842

[3] J. Bruns, F. Micklich, J. Kutterer, A. Abecker, and P. Zehnder, “Spatial operators for complex event processing,” vol. 1, pp. 107–123. [Online]. Available: https://hw.oeaw.ac.at?arp=0x003c13fd

[4] M. Roriz Junior, B. Olivieri, and M. Endler, “DG2cep: a near real-time on-line algorithm for detecting spatial clusters large data streams through complex event processing,” vol. 10, no. 1, p. 8. [Online]. Available: https://jisajournal.springeropen.com/articles/10.1186/s13174-019-0107-x

[5] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “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*, ser. KDD’96. AAAI Press, pp. 226–231. [Online]. Available: https://dl.acm.org/doi/10.5555/3001460.3001507

[6] C. C. Aggarwal, P. S. Yu, J. Han, and J. Wang, “A framework for clustering evolving data streams,” in *Proceedings 2003 VLDB Conference*. Elsevier, pp. 81–92. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B9780127224428500161

[7] A. Amini, T. Y. Wah, and H. Saboohi, "On density-based data streams clustering algorithms: A survey," vol. 29, no. 1, pp. 116–141. [Online]. Available: http://link.springer.com/10.1007/s11390-014-1416-y

[8] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, and M. Garofalakis, "Complex event recognition in the big data era: a survey," vol. 29, no. 1, pp. 313–352. [Online]. Available: http://link.springer.com/10.1007/s00778-019-00557-w

[9] E. Alevizos, A. Skarlatidis, A. Artikis, and G. Paliouras, "Probabilistic complex event recognition: A survey," vol. 50, no. 5, pp. 1–31. [Online]. Available: https://dl.acm.org/doi/10.1145/3117809

[10] U. Hedtstück, *Complex Event Processing: Verarbeitung von Ereignismustern in Datenströmen*. Springer Berlin Heidelberg. [Online]. Available: http://link.springer.com/10.1007/978-3-662-61576-8

[11] Apache Software Foundation. FlinkCEP - complex event processing for flink. [Online]. Available: //nightlies.apache.org/flink/flink-docs-release-1.14/docs/libs/cep/

[12] Apache Software Foundation. Welcome to the apache software foundation! [Online]. Available: https://apache.org/

[13] Apache Software Foundation. Apache flink: Stateful computations over data streams. [Online]. Available: https://flink.apache.org/

[14] Apache Software Foundation. Apache kafka. [Online]. Available: https://kafka.apache.org/

[15] D. Powers and D. Arthur. KafkaProducer — kafka-python 2.0.2-dev documentation. [Online]. Available: https://kafka-python.readthedocs.io/en/master/apidoc/KafkaProducer.html

[16] E. Inc. Complex event processing, streaming analytics, streaming SQL. [Online]. Available: https://www.espertech.com/

[17] [FLINK-7384] unify event and processing time handling in the AbstractKeyedCEPPatternOperator. - ASF JIRA. [Online]. Available: https://issues.apache.org/jira/browse/FLINK-7384