Platforms for Real-time Moving Object Location Stream Processing

SHÉRAZADE GADHOUMI
Abstract

Boarder security is usually based on observing and analyzing the movement of Moving Point Objects (MPOs): vehicle, boats, pedestrian or aircraft for example. This movement analysis can directly be made by an operator observing the MPOs in real-time, but the process is time-consuming and approximate. This is why the states of each MPO (ID, location, speed, direction) are sensed in real-time using Global Navigation Satellite System (GNSS), Automatic Identification System (AIS) and radar sensing, thus creating a stream of MPO states. This research work proposes and carries out (1) a method for detecting four different moving point patterns based on this input stream (2) a comparison between three possible implementations of the moving point pattern detectors based on three different Data Stream Management Systems (DSMS). Moving point patterns can be divided in two groups: (1) individual location patterns are based on the analysis of the successive states of one MPO, (2) set-based relative motion patterns are based on the analysis of the relative motion of groups of MPOs within a set. This research focuses on detecting four moving point patterns: (1) the geofence pattern consists of one MPO entering or exiting one of the predefined areas called geofences, (2) the track pattern consists of one MPO following the same direction for a given number of time steps and satisfying a given spatial constraint, (3) the flock pattern consists of a group of geographically close MPOs following the same direction, (4) the leadership pattern consists of a track pattern with the corresponding MPO anticipating the direction of geographically close MPOs at the last time step. The two first patterns are individual location patterns, while the others are set-based relative motion patterns. This research work proposes a method for detecting geofence patterns based on the update of a table storing the last sensed state of each MPO. The approach used for detecting track, flock and leadership patterns is based on the update of a REMO matrix (RElative MOtion matrix) where rows correspond to MPOs, columns to time steps and cells record the direction of movement. For the detection of flock patterns a simple but effective probabilistic grid-based approach is proposed in order to detect clusters of MPOs within the MPOs following the same direction: (1) the Filtering phase partitions the study area into square-shaped cells according to the dimension of the spatial constraint and selects spatially contiguous grid cells called candidate areas that potentially contain flock patterns (2) for each candidate area, the Refinement phase generates disks of the size of the spatial constraint within the selected area until one disk contains enough MPOs, so that the corresponding MPOs are considered to build a flock pattern. The pattern detectors are implemented on three DSMSs presenting different characteristics: Esri ArcGIS GeoEvent Extension for Server (GeoEvent Ext.), a workflow-based technology that ingests each MPO state separately, Apache Spark Streaming (Spark), a MapReduce-based technology that processes the input stream in batches in a highly-parallel processing framework and Apache Flink (Flink), a hybrid technology that ingests the states separately but offers several MapReduce semantics. GeoEvent Ext. only lends itself for a nature implementation of the geofence detector, while the other DSMSs
accommodate the implementation of all detectors. Therefore, the *geofence*, *track*, *flock* and *leadership* pattern detectors are implemented on *Spark* and *Flink*, and empirically evaluated in terms of scalability in time/space based on the variation of parameters characterizing the patterns and/or the input stream. The results of the empirical evaluation shows that the implementation on *Flink* uses globally less computer resources than the one on *Spark*. Moreover, the program based on *Flink* is less sensitive to the variability of parameters describing either the input stream or the patterns to be detected.
Acknowledgments

I would like to thank Győző Gidófalvi and Andreas Degwerth for their valuable and constructive suggestions.

This master thesis has been carried out at Airbus Defense and Space, Friedrichshafen.
# Contents

1 **Introduction** 7  
1.1 General Context 7  
1.2 Description of the problem and its importance 7  
1.3 Objectives 9  
1.4 Limitations and Delimitation 9  
1.5 Road map 9  

2 **Related Work** 10  
2.1 *Data Stream Management Systems vs. Database Management Systems* 10  
2.2 Data Processing in DSMS 11  
2.3 Challenges for DSMS 12  
2.3.1 Continuous Query Definition 12  
2.3.2 Near Real-time Output and Memory Requirements 14  
2.3.3 Scalability 14  
2.3.4 Processing Reliability and Failover Management 16  
2.3.5 Seamless Integration of Stored Data 16  

3 **Overview of Related Technology** 17  
3.1 General Overview of DSMSs 17  
3.1.1 *Workflow-based Technologies* 18  
3.1.2 *MapReduce-based Technologies* 19  
3.1.3 *Hybrid Technologies* 21  
3.2 Choice of DSMSs 22  

4 **Preliminaries and Problem Statement** 22  
4.1 Preliminary Definitions 22  
4.1.1 Moving Point Objects 22  
4.1.2 *Tumbling Time Windows* (TTW) 23  
4.1.3 Geofence Pattern 23  
4.1.4 REMO Patterns 24  
4.2 Problem Statement 27  

5 **Methodology** 28  
5.1 Overview of Research 28  
5.2 Geofence Pattern Detection 29  
5.2.1 Ingestion of MPO states sent via TCP Socket 30  
5.2.2 Point-in-polygon problem 30  
5.2.3 Transmission of Alerts via UDP Socket 31  
5.3 REMO Patterns Detection 31  
5.3.1 Spatial constraints 32  
5.3.2 *Flock* Pattern Detection: *Filtering-Refinement* approach 32  
5.3.3 General Computational Steps 37
### 6 Three Implementations of the Target Streaming Application

6.1 *GeoEvent Ext.* Solution ........................................................................... 38
6.1.1 A Three Machine Architecture ......................................................... 38
6.1.2 Actual Implementation ....................................................................... 39
6.1.3 Prerequisites ....................................................................................... 41
6.2 Spark Solution ......................................................................................... 41
6.2.1 Used Hardware .................................................................................... 42
6.2.2 Using *Elasticsearch* to store GeoJSONs ............................................. 42
6.2.3 Implementation of the *Geofence* Detector ........................................... 43
6.2.4 Implementation of the REMO Detectors .............................................. 45
6.2.5 Tuning Spark Solution ...................................................................... 48
6.3 Flink Solution ........................................................................................ 49
6.3.1 Implementation of the *Geofence* Detector ........................................... 50
6.3.2 Implementation of the REMO Detectors .............................................. 50
6.3.3 Tuning Flink Solution ...................................................................... 52

### 7 Empirical Evaluation

7.1 Evaluation of the SRDisks Algorithm .................................................... 53
7.1.1 Description of the Experiment ............................................................ 53
7.1.2 Results and Analysis ....................................................................... 54
7.2 Evaluation of the Real-time Detection of Patterns .................................. 55
7.2.1 Simulation Data ............................................................................... 55
7.2.2 Evaluation Metrics .......................................................................... 56
7.2.3 Description of the Experiment ............................................................ 57
7.2.4 Results and Analysis ....................................................................... 62

### 8 Discussion

8.1 Installation ............................................................................................ 69
8.2 Continuous Query Definition ............................................................... 70
8.3 Implementation of the Spark and Flink Solutions .................................... 70
8.4 Extrapolation to real-world data sets and scenarios .................................. 72

### 9 Conclusion and Future Work

73

### 10 References

74

### 11 Appendix

11.1 Detailed Results of the Evaluation of the SRDisks Algorithm ............... 79
11.2 Detailed Results of the Evaluation of the Real-time Detection of Patterns . 80
11.2.1 Geofence Detector ....................................................................... 80
11.2.2 Track Detector ............................................................................ 82
11.2.3 Flock Detector ............................................................................ 84
11.2.4 Leadership Detector .................................................................... 86
List of Figures

1. Abstract reference architecture of a DSMS .......................................................... 12
2. Aurora's GUI ........................................................................................................ 14
3. Simple example of stream partitioning .............................................................. 15
4. Simple example of stream pipelining ................................................................. 16
5. Failover management strategies. .......................................................................... 17
6. Example of a tumbling time window (TTW) ........................................................ 23
7. Example of REMO matrix generation ................................................................. 25
8. REMO patterns. ..................................................................................................... 26
9. Overlaps in REMO patterns. ................................................................................ 27
10. Methodology of the research ............................................................................... 29
11. Geofence detector: Computational steps for processing a batch collected during a tumbling window instance ................................................................. 30
12. Filtering phase of the flock detection: Grid and candidate areas ......................... 33
13. Filtering phase of the flock detection: Candidate areas and corresponding semi-random disks .......................................................... 35
14. Refinement phase ............................................................................................... 36
15. Computational steps of the REMO detectors ...................................................... 37
16. Implementation of the geofence detector with GeoEvent Ext. ............................. 40
17. Definition of the filter EnteringTrack using GeoEvent Ext.'s GUI ....................... 41
18. Command line to index a zone in ES ................................................................... 43
19. Implementation of the geofence detector using Spark ...................................... 44
20. Implementation of the REMO detectors using Spark ........................................ 46
21. Implementation of the geofence detector using Flink ........................................ 50
22. Implementation of the REMO detectors using Flink .......................................... 51
23. SRDisks algorithm: Effect of the maximum number of semi-randomly generated disks $n_{test}$ within each candidate area ($S_2$ and $S_3$) .................. 54
24. GeoJSON representing an MPO state ............................................................... 56
25. Geofence detector: Effect of the number of geofences ...................................... 62
26. Geofence detector: Effect of the number of considered MPOs within each TTW instance ........................................................................................................ 63
27. Track detector: Effect of the constance parameter ............................................. 64
28. Track detector: Effect of the number of considered MPOs within each TTW instance ........................................................................................................ 65
29. Flock detector: Effect of the spatial constraint .................................................... 66
30. Flock detector: Effect of the distribution of the MPOs ......................................... 67
31. Flock detector: Effect of the number of considered MPOs within each TTW instance ........................................................................................................ 67
32. Leadership detector: Effect of the 2nd spatial constraint ..................................... 68
33. Leadership detector: Effect of the number of ingested MPOs during each TTW instance ........................................................................................................ 69
List of Tables

1 Differences between DBMS and DSMS .................................................. 11
2 Used hardware for GeoEvent Ext. solution ........................................... 38
3 Used hardware for Spark and Flink solution ......................................... 42
4 Concordance between building blocks of the REMO detector and Spark operations .......................................................... 47
5 Correspondance between building blocks of the REMO detectors and Flink transformations .................................................. 52
6 Parameters and values used for the experiments on the geofence detector . 58
7 Parameters and values used for the experiments on the track detector .... 59
8 Parameters and values used for the experiment on the flock detector ....... 60
9 Parameters and values used for the experiment on the leadership detector 61
10 Detection accuracy (%) of the SRDisks algorithm on candidate area $S_2$ with different maximum number of tests $n_{test}$ and constance parameter $n_{min}$ .................................................. 79
11 Detection accuracy (%) of the SRDisks algorithm on candidate area $S_4$ with different maximum number of tests $n_{test}$ and constance parameter $n_{min}$ .................................................. 79
12 Results of the experiments on Spark’s geofence detector ....................... 80
13 Results of the experiments on Flink’s geofence detector ....................... 81
14 Results of the experiments on Spark’s track detector ............................ 82
15 Results of the experiments on Flink’s track detector ............................. 83
16 Results of the experiments on Spark’s flock detector ............................ 84
17 Results of the experiments on Flink’s flock detector ............................. 85
18 Results of the experiments on Spark’s leadership detector ..................... 86
19 Results of the experiments on Flink’s leadership detector ..................... 87
# List of Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airbus D&amp;S</td>
<td>Airbus Defense and Space</td>
</tr>
<tr>
<td>AIS</td>
<td>Automatic Identification System</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>DSMS</td>
<td>Data Stream Management System</td>
</tr>
<tr>
<td>Flink</td>
<td>Apache Flink</td>
</tr>
<tr>
<td>ES</td>
<td>Elasticsearch</td>
</tr>
<tr>
<td>GeoEvent Ext.</td>
<td>Esri ArcGIS GeoEvent Extension for Server</td>
</tr>
<tr>
<td>GIS</td>
<td>Geographic Information System</td>
</tr>
<tr>
<td>GNSS</td>
<td>Global Navigation Satellite System</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>MPO</td>
<td>Moving Point Object</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>REMO patterns</td>
<td>RELative MOtion patterns</td>
</tr>
<tr>
<td>Spark</td>
<td>Apache Spark Streaming</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TTW</td>
<td>Tumbling Time Window</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 General Context

More and more applications are taking advantage of daily generated data. This is due to the development of new sensors which have led to an easier and more comprehensive data collection and to the emergence of processing tools aiming at continuously processing the collected data [18]. The Internet of Things (IoT), which consists of ensuring the communication between different physical objects by seamlessly integrating them into the information network [30], has put to use those tools and accompanied their development. Continuous processing of real-time collected data has actually a lot of applications: finance (fraud detection), telecommunication (alarm messages), online applications (e.g. user’s data analysis to personalize web searches), sensor applications (e.g. environmental monitoring), health care (e.g. monitoring of the patient’s health) [18, 28], etc.

The analysis of real-time collected data is therefore crucial. The on-the-fly generated data can be represented by an unbounded timestamped sequence of data items subsequently termed data stream [18]. Those data items can have any type of data field types: String, Double, Integer, Geometry, etc.

Several data streams contain locational data, as tracking applications are common in the aerospace industry (departure and landing of aircrafts) [23], in the shipping industry and in homeland security applications for example. For those specific application domains, data streams contain a spatial field storing the location of the object of interest based on a given location tracking method. The location tracking method is usually chosen depending on the type of the object of interest and on its environment. For example, if it is possible to embed receivers on the target object, a common method is to periodically sense its position with the help of GNSS (Global Navigation Satellite System) receivers. GNSS is usually used for terrestrial applications. For maritime applications, AIS (Automatic Identification System) [51], a maritime safety tool based on electronic message exchange between vessels, AIS shore stations and satellites can be used as a tracking system. Indeed, AIS onboard transceivers transmitting, inter alia, identity, location, speed and route of the vessel. When the target object does not embed any receiver, location tracking is done through radar-based sensing (Radio Detection and Ranging), which consists of inferring target parameters (location, velocity, direction, etc.) based on the characteristics of the reflection of the emitted electromagnetic energy [22].

1.2 Description of the problem and its importance

Airbus Defense and Space (thereafter abbreviated Airbus D&S) is a division of Airbus Group specialized in military aircraft, space systems, and communications, intelligence and security. One of the current project of the division consists of developing homeland

---

1The current operational GNSSs are US’s Global Positioning System (GPS) and Russia’s GLObal Navigation Satellite System (GLONASS) that respectively involve a satellites’ period of revolution of 11h 58min and 11h 15min [27]
security tools. The goal is to control the boarders by analyzing parameters about the incoming and leaving Moving Point Objects within given buffer zones. Moving Point Objects (thereafter called MPO) are any type of elements moving in the area of a given border that can be represented by a point: vehicles, boats, pedestrians or aircraft for example. For each MPO, different parameters are sensed by radar, GPS or AIS: location, speed, direction, type, etc. All the parameters describing the situation of a given MPO at a given timestamp are called the MPO state. Every MPO is identified by a unique number (ID).

This homeland security project developed by Airbus Defense and Space is supposed to be fitted to any type of border by gathering all the information about the MPOs moving at a specific border. However, the collection of MPO states has not been tested yet. This is why a program simulating a data stream according to the format of the collected data is used as a basis of the research. This simulation program generates continuously the states of a chosen number of MPOs at a chosen data rate. The real situation is nevertheless slightly different from the simulation program, as the simulation program generates the states of a constant number of MPOs in the study area, while MPOs can enter and exit the study area in reality.

This real-time moving point object location stream is analyzed in order to detect on-the-fly moving point patterns. Moving point patterns are usually divided in two groups: set-based relative motion patterns and individual location patterns. Set-based relative motion patterns are detected by analyzing the motion of groups of MPOs in a set, while individual location patterns are based on the analysis of the states of a single MPO. Airbus D&S wishes to inform control stations through real-time notifications when such moving point patterns are detected.

The monitoring of the movement of MPOs close to the border is more crucial than the one of MPOs that are farther. This is why the area around a given border is divided into buffer zones. A pattern of interest is to know whenever an MPO enters/exists one of the buffer zones. This pattern is called geofence pattern (individual location pattern). However, the distance to the border is not the only crucial parameter. Indeed, the moving direction is also important, as MPOs heading towards a given border have to drive more attention than those moving along the border. The track pattern (individual location pattern) is used to identify MPOs that follow the same direction for a given time period. Another parameter to take into account is the relation of MPOs to each other (set-based relative motion). Indeed, MPOs moving independently from each other should drive less attention than those linked to each other. Thus, close MPOs following the same direction should be identified (flock pattern). Another relation between MPOs could be featured by an MPO leading the movement of other MPOs (leadership pattern).

The aim of this research is to implement and compare different real-time moving point location stream platforms for the purpose of detecting in real-time geofence, track, flock and leadership patterns.
1.3 Objectives

The objectives of this research are:

- the selection of three stream processing platforms that could be used to program the real-time detection of geofence, track, flock and leadership patterns

- the actual implementation on each stream processing platform of a program analyzing a stream of MPO states to detect in near real-time geofence, track, flock and leadership patterns

- the comparison of the three implementations against each other based on the assessment of their behavior towards changes of parameters defining either the input stream of MPO states (input rate and distribution of MPOs) or the patterns (number of concerned MPOs and size of the spatial constraint) to be detected.

1.4 Limitations and Delimitation

The comparison between different data stream processing platforms is done among architectures that can address the needs of the aforementioned homeland security project. The processing platforms to compare thus all handle real-time ingestion and processing of data streams representing point moving objects.

Concerning GIS applications, Esri tools are widely used in the industry, as their products represent more than 40% of the GIS market [47]. For real-time data stream ingestion and processing, Esri developed the ArcGIS GeoEvent Extension for Server (thereafter called GeoEvent Ext. for simplification purpose). Different out-of-the-box connectors exist to ingest, process and produce the most common data streams. Esri also provides a Software Development Kit in order to allow the user to directly program specific connectors for a given data feed, specific output data streams or specific processing. GeoEvent Ext. has therefore the advantage to present a big amount of processing tools, connectors to input sources and outputs as well as a plasticity allowing the user to implement new connectors or processing tools. GeoEvent Ext. can also be linked to other Esri products to allow the storage and visualization of data feed. Airbus D&S being an Esri partner, it is quite natural to use those products to process the point data objects provided by the simulation program. However, a lot of other data stream processing platforms have been developed in the last years, some of them being free and open-source. This research aims at comparing different data stream platforms -including GeoEvent Ext.- for the purpose of detecting given moving point patterns.

1.5 Road map

The research is organized as follows. Section 2 offers a general view of the stream processing platforms by introducing their characteristics and the challenges they have to

---

2 Esri (Environmental Systems Research Institute) is a 1969 founded company that provides Geographic Information Systems softwares, management and storing systems.
address. The different technical approach adopted by stream processing platforms are then explained and compared using a classification, and the choice of three stream processing platforms is rationalized in Section 3. Thereafter, Section 4 defines the set-based relative motion patterns and individual location patterns that one wants to detect using the selected stream processing platforms. Section 5.1 highlights the methodology of the research. Then, Section 5 describes the general method used to implement the applications aiming at detecting the selected moving point patterns. Their actual implementation on the three selected stream processing platforms is subsequently clarified in Section 6. Section 7 empirically evaluates two of the three implementations and examines the obtained results. Eventually, Section 8 reviews the points of comparison between the three implementations.

2 Related Work

2.1 Data Stream Management Systems vs. Database Management Systems

Before the development of data stream management tools, Database Management Systems (DBMS) were widely applied. In DBMSs, data is stored in a secondary storage device in order to be processed by specified queries in the main memory [18]. Indexes are used by the queries in order not to have to go through all the database items [48]. The queries are optimized by the query optimizer -called by the compiler- in order to enhance data retrieval performance [53]. The optimized queries are then evaluated only once against the current database instance. If data is updated and a new database instance is subsequently generated, the user has to perform the same query again to obtain the new result. Therefore, every time the relations between data items of the database or data attributes are updated, the indexing structure or query optimization also has to be updated. Thus, frequent data updates lead to delay in response time [18]. As databases can hardly be updated continuously, DBMSs are usually used to process less frequently updated data with dynamic queries.

DBMS is not the appropriate tool to process data streams. Indeed, data streams can be considered as continuously updated data, which means that a given query would have to be evaluated each time a new data item would be added to the data stream. In order to process data streams, it is not possible to consider evaluating dynamic queries on persistent data, as it is the case in DBMSs. A new paradigm is therefore considered to process data streams with Data Stream Management Systems (DSMS): persistent queries are continuously evaluated on dynamic data [28]. Table 1 [19, 28] sums up the implying consequences of this paradigm shift by presenting the differences between DBMS and DSMS.

Due to the fact that DSMS process data streams in one sequential pass, the query plan cannot be fixed: it is adjusted to the changes in data. Furthermore, since many continuous queries can run at the same time, the optimization has to be done across all the queries.
<table>
<thead>
<tr>
<th></th>
<th>DBMS</th>
<th>DSMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type</td>
<td>persistent relations</td>
<td>unbounded data set</td>
</tr>
<tr>
<td>Data reliability</td>
<td>high</td>
<td>middle</td>
</tr>
<tr>
<td>Latency</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Processing model</td>
<td>query-driven</td>
<td>data-driven</td>
</tr>
<tr>
<td>Queries</td>
<td>one-time</td>
<td>continuous</td>
</tr>
<tr>
<td>Query plans</td>
<td>fixed</td>
<td>adaptive</td>
</tr>
<tr>
<td>Query optimization</td>
<td>one query</td>
<td>multi-query</td>
</tr>
<tr>
<td>Query answers</td>
<td>exact</td>
<td>exact or approximate</td>
</tr>
<tr>
<td>Time notion</td>
<td>not necessary</td>
<td>inherent</td>
</tr>
<tr>
<td>Update rates</td>
<td>low</td>
<td>high</td>
</tr>
</tbody>
</table>

Table 1: Differences between DBMS and DSMS [19, 28]

[28]. Another point to take into account is that, as data are ingested in real-time, values can be unreliable (due to sensor errors or network delays). Thus, if the incoming data is presumed unreliable, query answers have to be approximate.

As an illustration, if a DBMS is used for data stream processing, the data stream has first to be stored in a database in order to be then processed via queries. This process generates two I/O per data item [18]. If a DSMS is used for the same processing, data processing is performed without storing the data stream, which leads to no I/O and thus a lower latency [18, 49]. If needed, the data stream can be partially or totally stored. However, in a DSMS, data storage is totally independent from stream processing.

### 2.2 Data Processing in DSMS

**Processing architecture of DSMS** The structure of a DSMS is shown on Figure 1. A buffer first ingests the incoming data stream, possibly filtering data by using an input monitor. Secondly, the recent data items of the input data stream are temporarily stored into the working storage for specific processings requiring previous data items. The local storage is used to store metadata updates about the input data stream. The continuous queries are written into the repository, converted into a query plan and applied on the input data stream by the query processor. The query processor eventually delivers the streaming output(s).

**Processing types in DSMS** Continuous queries in DSMS can either handle each incoming data item separately (stateless operators) or perform processing over several incoming data items (stateful operators). In the second case, the range of the concerned data items over which the output is calculated has to be specified, as data stream have the particularity to be unbounded. This is done by setting a stream window.

Different window-based processing exist [19, 28], each of them being either based on a time interval -time-based window- or a number of tuples -count-based window:
Figure 1: Abstract reference architecture of a DSMS [28]

- **Sliding window**: The window’s width is fixed, but the starting and ending points move, replacing old data items by newer ones. The window instances overlap in time.

- **Tumbling window**: The characteristics are similar to sliding windows, except that window instances are disjoint in time (the elements collected during two consecutive window instances are different from each other)

- **Landmark window**: The window has a fixed starting point and a forwards moving ending point, increasing thus the windows width.

### 2.3 Challenges for DSMS

Particular challenges are faced by DSMS technologies, mainly due to the unbounded nature of data streams. The efforts put in addressing problems arisen by data streams are crucial for the performance of the DSMS. The main challenges for DSMSs are presented and explained in this section. For each difficulty, different technological approaches are briefly presented.

#### 2.3.1 Continuous Query Definition

A very straightforward requirement of DSMS is to present a user-friendly interface for defining streaming applications. This can be done by three ways, each of them being presented in this section.

**SQL-like languages**  SQL is a very natural solution concerning data stream processing as it is the main language of DBMS [49, 34]. SQL has the great advantage of being a high-level language that is based on very straightforward primitives. Complex queries can be created by combining those primitives. However, SQL is not fitted for data stream
processing, as it cannot describe time/count-based windows nor handle stateful operators (cf. Paragraph 2.2). Hence, in order to handle these stream-specific requirements, several DSMSs use an extended and revised version of SQL. For example, GSQL [28] is an SQL-like syntax that has been developed for the DSMS Gigascope. It is a “pure stream query language”, as it can only input and output streams. This is however not the case for CQL (Continuous Query Language) for example, the SQL-like syntax of the DSMS STREAM that can support both streams and relations [8]. An example of CQL language is presented in Code 1. In this example, different point moving objects identified by their ID are considered. Their altitudes are sensed at different observation times, generating an incoming stream. An output stream containing only the objects that are located below 1000 meters is thereafter created.

**Graphical Box and arrows language** A solution that seems more user-friendly is to allow the user to build a streaming application by building a network of boxes -representing the operators- and arrows -representing stream data transmission [34], as shown on Figure 2. The box-and-arrow is then translated into the underlying language and optimized [14].

**Implementation-oriented approaches** This approach consists of allowing the user to implement directly the operators using a defined programming language such as Java, Python or Scala for example [34]. This means that no declarative interface is provided.
2.3.2 Near Real-time Output and Memory Requirements

One of the most important challenges of DSMS is to be able to provide answers in real-time. As input data is continuously ingested and processed, this means that the computation time of data elements has to be as low as possible [10]. Indeed, if the computation time is too high, the data stream processing algorithm will not be qualified to cope with data stream pace, resulting in bottlenecks delaying the output generation. In order to ensure low computation latency, one of the requirements is to build algorithms that access main memory without accessing the disk, as disk access is more time consuming than main memory access. A second requirement to achieve low computation latency is to optimize the execution path in order to ensure “minimum ratio overhead to useful work” [49].

2.3.3 Scalability

Data streams can be highly unpredictable, which means that DSMSs have to be able to deal with suddenly large volume of incoming data without compromising data stream processing. In other words, DSMSs have to ensure scalability. In order to achieve this, data processing is distributed across different processors and/or machines [49]. Indeed, processing reliability is increased by redundancy across nodes (cf. Section 2.3.4) and load distribution entails better performance and scalability [45]. Usually, two strategies are used to ensure data distribution across processors and/or machines [45, 34]:

- **Stream partitioning:** Stream partitioning consists of subdividing input streams into partitions in order to process all the partitions in a parallel way [26]. Each input tuple of a given input stream is redirected to a processing node of a set of predefined
nodes. After processing, the tuples are merged into an output stream. The main disadvantage of this approach is that its implementation can be difficult, for example in case of window-based operators [45]. An example of stream partitioning is shown on Figure 3. The considered input stream contains the positions of objects, their ID and the ID of their type (cars, boats, etc.). The goal is to perform a left join between the input stream and a table (table $T$) based on the type ID ($\text{type\_id}$) in order to generate an output stream containing the object’s type. In this example, the input stream is partitioned in two sub-streams, each one being independently joined to table $T$. The sub-streams are then merged while preserving the order of the data.

- **Stream pipelining**: Stream pipelining consists of dividing a complex query node into very simple query nodes that are assigned different processors and/or machines. Stream pipelining can be limited by the existence of stateful operators (cf. Paragraph 2.2). Based on the same streaming application as in the example of stream partitioning, a simple example of stream pipelining is shown in Figure 4: The former table $T$, on which the left join has to be done, is divided in two sub-tables, namely table $T_a$ and Table $T_b$. This table division is aimed to divide the left join on table $T$ in two operations: left join on table $T_a$ and left join on table $T_b$. 

![Figure 3: Simple example of stream partitioning](image)
2.3.4 Processing Reliability and Failover Management

Ensuring processing reliability is crucial in DSMS. Indeed, lost ingested data cannot be recovered since streams are processed in real-time without storage requirements. In order to handle hardware failures during run-time, failover management strategies have to be applied [49, 45]. Three main strategies are usually used to tackle failures: simple standby, checkpointing and hot standby. Those strategies consist of maintaining a secondary system on a different computer. For the simple standby strategy, the secondary system is only activated when the primary one fails, which means that it can only be used for stateless operators as the state of the primary system is lost. To handle stateful operators, a checkpointing strategy can be adopted: it consists of periodically synchronizing the states of the secondary and primary systems. In case of failure of the primary system, the application can failover to the secondary system using the last synchronized state. The last strategy called hot standby consists of keeping both primary and secondary systems active by sending a copy of the input data stream to the secondary system. The different strategies are presented on Figure 5.

2.3.5 Seamless Integration of Stored Data

Even if DSMSs’ crux is to process data streams in real-time without involving storage costs (cf. Section 2.1), it must be noted that access to historical data can be needed for several applications [49]. Indeed, very common applications aim at comparing live with historical data or at testing algorithms over historical data before using it on real-time data. This could be achieved by including an interface to a DBMS. However, as this option leads to high latency, a better option is to integrate a storage part in the operating system space of the application. The latter option is based on the possibility to use the same continuous query definition for both live and historical data.
Figure 5: Failover management strategies. For each failover management strategy, the primary and secondary systems (or nodes) are respectively represented by the letters \( N \) and \( S \). The solid arrows represent permanent data streams, while the dashed arrows represent temporary streams.

3 Overview of Related Technology

DSMSs have gained importance due to the huge development of big data which has led to the implementation of different DSMS-like technologies. First, a general overview of the development of DSMS technologies is drawn through a rough evaluation against the previously stated challenges (cf. Section 2.3). The choice of focusing on specific DSMSs for this work is eventually rationalized.

3.1 General Overview of DSMSs

Different technologies have been developed to process streams. Some of them were especially developed to address data stream processing in special fields while other platforms aimed at processing any kind of data stream. Data stream processing technologies can be clustered into three groups, depending on the underlying technologies. This classification is introduced in [34, 21]. A first group gathers all the technologies based on workflow processing, while the second group is based on MapReduce paradigm. A third group is introduced for technologies taking advantage of both paradigms. However, across those groups, similar characteristics such as clustering possibilities or levels of continuous query languages can be identified.
3.1.1 Workflow-based Technologies

**Definition** A workflow-based DSMS technology ingests each data item of the input stream separately. The processing of the stream is based on the definition a Directed Acyclic Graph (DAG) with vertices and edges respectively representing operators and data flows. This representation takes advantage of two patterns [45], namely stream pipelining and stream partitioning. The first refers to the division of operators into consecutive subtasks which are assigned an exclusive process [34]. The latter consists of splitting a stream into partitions and distributing them among independent operators. Workflow-based DSMS technologies were historically the first ones.

**Overview** Aurora [15] is a 2002 DSMS developed in the frame of a collaboration between Brandeis University, Brown University and Massachusetts Institute of Technology initially for sensor-network monitoring. It supports three types of applications that are real-time monitoring, archival and spanning applications (applications using both real-time and archival data). To create those applications, the user interacts with a GUI (Graphical User Interface) —which relies on SQuAL (Stream Query Algebra) algebra—with the purpose of creating a workflow with boxes representing operators on streams and arrows representing streams (cf. Figure 2). By combining primitive operations (among eight provided stateful and stateless primitives ³), it is possible to create custom operators. Query optimization is done dynamically during run-time by applying predefined local rules (reordering, combination of boxes, etc.) on portions of the un-optimized workflow created by the user. By taking advantage of run time statistics, local rules can generate a more optimized workflow. Moreover, Aurora is based on the definition of different Quality of Service (QoS) graphs, each of them representing the utility of the result against a chosen quality metric (result delay, percentage of tuples delivered or output values) [1]. In case of an overload situation, which is detected by continuously evaluating the current processing load of the network, the system proceeds to load scheduling and load shedding by optimizing the aggregate QoS. This leads to a minimum aggregate utility loss (minimum loss of accuracy for the outputs). Aurora also handles unreliable data by enabling timeout and slack parameters for windowed operations. Indeed, by setting those parameters in an appropriate way, slow and/or out-of-order data can be processed [1].

The main disadvantage of Aurora is that it is designed only for small-scale applications as it cannot be distributed across different machines.

This has been enhanced in Borealis [2, 14], the distributed multi-processor version of Aurora, whose development started in 2005. Furthermore, Borealis allows changing data and operator attributes during run-time. Enabling clustering on Borealis has led to a better scalability and availability of the system. Besides supporting clustering, Borealis allows changing data and operator attributes during run-time, which is of interest for

³Aurora provides seven different operators [1] that can be gathered in two groups. Windowed operators “operate on sets of consecutive tuples from a stream” (cf. Paragraph “Processing types in DSMS” in Section 2.2): Slide, Tumble, Latch (Latch is the stateful equivalent of Tumble operator: the state is maintained between window calculations). The second group is composed of operators that “act on a single tuple at a time”: Filter, Drop, Map, GroupBy, Join.
specific domains (for example in finance). The Borealis project is however no longer active and has stopped in 2008 with the release of a Borealis software running on Linux x86-based computers.

STREAM (STanford stReam datA Manager) is a DSMS project (2003-2006) launched by Stanford University. STREAM is a more general purpose DSMS that aims at providing both DBMS and DSMS functionalities [18]. It is based on the creation of an SQL-like language named CQL that includes streams, relations (set of tuples) and time windows. Indeed, in order to process streams, STREAM first convert streams into relations using stream-to-relation operators. Using the same logic, the generation of output streams follows a relation-to-stream operator. One of the advantages of STREAM over Aurora is that load scheduling is handled by a “Chain” scheduling algorithm, which leads to a better memory management [10, 9, 34]. Indeed, this algorithm minimizes the runtime memory usage for backlog buffering. However, “Chain” scheduling leads to high latencies in some overload situations [11], which is why new algorithms aiming at both minimizing latency and memory usage were developed. STREAM has also developed new approaches for DSMS such as resource sharing for common portions of different query plans, resource management and query approximations in case of many rapid incoming streams and many continuous queries [39]. However, like Aurora, STREAM can only be used for small-scale applications as the system do not support clustering.

Aurora, Borealis and STREAM are DSMS research prototypes that have paved the way for new commercial workflow-based DSMS such as ArcGIS GeoEvent Extension for Server. ArcGIS GeoEvent Extension for Server (GeoEvent Ext.) is a commercial DSMS based on graphical box and arrows language used for processing spatial data. The user accesses a web-platform called ArcGIS GeoEvent Manager to create a particular stream application using the out-of-the-box input connectors, filters, processors and output connectors. It is also possible to program its own elements using the GeoEvent SDK. Concerning windows, GeoEvent Ext. is limited: it can only handle event-by-event processing. GeoEvent Ext. interprets each input stream using a schema called GeoEvent Definition that can be either created by the user or automatically generated by the system. The big advantage of ArcGIS GeoEvent Extension is that it is quite easy to take advantage of Esri environment in order to manage and visualize (Portal for ArcGIS), and store processed spatial data (ArcGIS Spatiotemporal Big Data Store). ArcGIS Spatiotemporal Big Data Store is actually a non-SQL database based on an Elasticsearch search engine [50].

3.1.2 MapReduce-based Technologies

Definition MapReduce is a highly-parallel processing framework based on two core functions, namely Map and Reduce, in order to process disk-based big data that are stored

---

4 It is not possible to create time/count-based windows on GeoEvent Ext.

5 Elasticsearch is an open-source distributed, scalable and near real-time search and analytics engine based on Apache Lucene, a “full-featured Information Retrieval library” [24, 33].
in a distributed file system like HDFS\textsuperscript{6} [34, 45, 54]. MapReduce paradigm processes batches of input data. In this model, input data is represented as key/values pairs. The Map operator processes the tuples in order to generate intermediate key/values tuples, which are grouped according to their keys. The Reduce function then processes all the values corresponding to a given key and produces a set of values. MapReduce paradigm is initially aimed to process big data materialized inputs by materializing intermediate results. It has been recently applied to process data stream processing systems.

Due to the definition of the MapReduce paradigm, it is not possible for MapReduce-based DSMS to process data as they enter. Indeed, MapReduce considers unordered sets of data items as input and output. For stream processing, a compromise has been found by dividing the input stream into small data batches corresponding to very small time windows and generating an output stream by sending small data batches. Within each data batch, the order of the data item according to their ingestion time is not kept.

Overview The Hadoop Online Prototype (HOP) [20] paved the way for MapReduce-based data stream platforms. Indeed, while the original MapReduce framework is based on data materialization at each map/reduce task, HOP proposed a new MapReduce architecture enabling data pipelining. It handles continuous queries by periodically involving a new MapReduce job over each new incoming set of data [20, 12, 3] (batch). This approach works well when a single MapReduce job needs to be continuously evaluated against incoming data. However, for more complex workflows of MapReduce jobs, a job scheduling strategy has to be adopted. Continuous-MapReduce Framework (C-MR), a parallelized stream processing engine developed by Brown University [12], has developed such a strategy. Like HOP, C-MR provides window specifications on reduce operations. A next-generation distributed version of C-MR is Spark Streaming [34].

Apache Spark [7, 55] is an open-source big data analysis tool. Spark Streaming is a distributed extension of the core Spark API that can process in real-time live streams data. The underlying paradigm of Spark Streaming is to consider data stream processing as micro-batch processing. Spark Streaming (Spark) is based on an implementation-based approach, letting the user choosing between Scala, Java, Python and R to program the wanted application. Data can be ingested from many sources like Kafka, Flume, Kinesis, (advanced sources) or TCP/UDP sockets (basic sources), and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Spark receives live input data streams and discretizes them into batches, which are then processed by the Spark engine to generate a discretized stream. Discretized streams are actually composed of Resilient Distributed Datasets (RDD, Spark’s abstraction of an immutable distributed dataset). Thus, any operation applied on a discretized stream is converted and applied on each RDD. Each created RDD is linked with a lineage graph accounting for its generation process. The crucial point in Spark is to choose the appropriate batch size, as this parameter is controlling the whole RDD processing stage. Indeed, if the chosen batch interval is lower than the required processing time, bottle-

\textsuperscript{6}Hadoop Distributed File System is a Java-based scalable distributed file system.
necks occur, leading to delays concerning the output. A limitation of this batch-based approach is that windows can only be based on time units and not on event counts.

3.1.3 Hybrid Technologies

Definition Some DSMS technologies take advantage of workflow-based and MapReduce paradigms. In this section, workflow-based streaming platforms considering incoming data as key/values pair (like in MapReduce framework) are presented.

Overview Esc [46] is a hybrid cloud stream computing engine which is based on defining applications through a Directed Acyclic Graph. The strength of Esc is to automatically handle load scheduling and fault tolerance while adapting to the data arrival rate and to changing computational needs. One of the limitations of Esc is that the virtual machines over which Esc can be executed have to be homogeneous in terms of performance (CPU, memory capacity, etc.). Moreover, Esc does not provide any primitive operators for expressing operations: the user only has to follow a pattern concerning the input types and number when programming the operators.

Apache Flink [4, 16] is an open-source platform that handles distributed batch and stream processing. Flink was originally developed by Data Artisan. It relies on a workflow-based architecture but incoming data are converted into key/values pairs like in the MapReduce-based technologies. Like Spark, Flink is based on an implementation-oriented approach, letting the user programming data stream applications either in Java or Scala. Its core paradigm differs however from Spark, as it considers batch processing as a sub-part of data stream processing. Indeed, Flink’s core engine is a streaming dataflow engine. Two core APIs, DataSet and DataStream APIs, are respectively used for batch and stream processing. They create both runtime programs that can be executed by the streaming dataflow engine. Flink supports different data stream sources and sinks such as sockets, text files, message queue connectors (Apache Kafka, RabbitMQ). Custom sources and sinks can also be implemented. Flink can handle all types of windows, count-based or time-based, as the system is not based on batch processing. Concerning processing reliability, Flink is based on the Asynchronous Barrier Snapshotting mechanism, which consists of injecting checkpoint barriers that contains snapshot IDs. Each operator waits for all the checkpoint barriers from its inputs before writing its state to a buffer. In case of node failure, recovery is then be performed using the previous states. This mechanism corresponds to the checkpointing strategy presented in Section 2.3.4. Flink can also handle unordered incoming data: watermarks are originated by the source to delimit ordered data. Stateless operators then just forward those watermarks, while stateful operators compute operations based on them before forwarding them -for example time window operations.

21
3.2 Choice of DSMSs

The purpose of this work is to compare different DSMSs for a given spatiotemporal application. Different reasons have led to the selection of some DSMS technologies. A first idea is to select at least one real-time stream processing platform in each aforementioned group (cf. Section 3.1), so that the qualities of each DSMS group can be truly appreciated. The chosen platforms must be able to handle spatiotemporal data, originally or with the addition of a geospatial API. All real-time stream processing platforms based on an implemented oriented approach (cf. Section 2.3.1) using Java can for example make use of Esri’s geospatial API [25]. Another criteria is the scalability of the chosen systems. As the incoming data stream can be in the real case (not with the simulation program) unpredictable in terms of data incoming rate and quantity, it seems appropriate to choose a DSMS that can be clustered over different machines. Furthermore, new popular technologies have been preferred to real-time streaming platforms projects that have wound down. This work therefore focuses on three real-time streaming platforms:

- Esri ArcGIS GeoEvent Extension (GeoEvent Ext. for short)
- Apache Spark Streaming (Spark for short)
- Apache Flink (Flink for short)

4 Preliminaries and Problem Statement

In this section, useful definitions to sate the problem are highlighted. Then, the problem is clarified. For all the definitions, one defines the time domain \( \mathbb{T} \) as a totally ordered set of the space of non-zero natural numbers \( \mathbb{N}^+ \).

4.1 Preliminary Definitions

4.1.1 Moving Point Objects

Let \( o \) be a Moving Point Object (thereafter called MPO) whose state \( s_o(t) \) at refresh time \( t \in \mathbb{T} \) is defined by a tuple as follows:

\[
s_o(t) = (p_o(t), m_o(t), im_o)
\]

where \( p_o(t) \) represents the WGS84\(^7\) latitude \( \lambda \) and longitude \( \phi \) of the object \( o \) at time \( t \), \( m_o(t) = (t, v_o(t), d_o(t)) \) represents the refresh time \( t \), the speed \( v_o(t) \) and the direction \( d_o(t) \) of \( o \) and \( im_o \) is a tuple of immutable parameters of the object \( o \).

The trajectory of the MPO \( o \) during the time interval \([t_0,t_n]\) is then defined as a tuple of positions \( (p_o(t_0),p_o(t_1),...,p_o(t_n)) \), with \( t_0 < t_1 < .. < t_n \).

\(^7\)The World Geodetic System 1984 [43] is an Earth-centered Earth-fixed reference system developed and defined by the US Department of Defense for GPS use. Its ellipsoid is characterized by a semi-major axis of 6 378 137 m, a flattening of 1/298.257 223 563. Further characteristics are: a nominal mean angular velocity of 7 292 115 \( 10^{-11} \) rad/s and a geocentric gravitational constant of 3 986 004.418 \( 10^8 \) m\(^3\)/s\(^2\).
4.1.2 Tumbling Time Windows (TTW)

Each instance of a time-based window is defined by a time interval \([w_e, w_s]\) over which the elements of a stream are grouped into a finite set. Tumbling time window (TTW in short) instances never overlap in time. Let \(w_{len} \in \mathbb{T}\) represents the length of a TTW \(ttw\) and let \(t_0 (t_0 \in \mathbb{T})\) be the time when the stream starts being ingested. An instance of \(ttw\) is a time interval \([t_0 + (p - 1) * w_{len}, t_0 + p * w_{len}]\) such that \(p \in \mathbb{N}^+\). TTWs are applied on a stream of MPO states (moving point object location stream). Figure 6 shows the sets of states collected by two consecutive TTW instances on a moving point object location stream considering three different MPOs \(O_0, O_1, O_2\).

4.1.3 Geofence Pattern

Monitoring a border is usually done by defining buffer zones associated with different warning levels and adapting the inspection to the buffer zone. It is thus important to be alerted whenever an MPO enters/exits one of the buffer zones. This corresponds with identifying a geofence pattern.

The geofence pattern is an individual location pattern based on the real-time detection of MPOs entering/exiting predefined zones termed geofences.

A geofence is a zone of the Earth described by a polygon such that:

\[
GF_A = \{p_1, ... , p_n\}
\]

with \(p_{i,i \in [1, n]}\) being the locations of the polygon’s edges.

Considering the aforementioned TTW \(ttws\), object \(o\) is respectively defined to have entered/have exited a geofence \(GF_A\) during time period \([t_0 + (p - 1) * w_{len}, t_0 + p * w_{len}]\) \((p \in \mathbb{N}^+, w_{len} being the length of the tumbling window, \(t_0\) being the time when the input stream starts being sensed) if:

- \(ttw(t_0 + (p - 1) * w_{len})\) contains a state of \(o\) indicating that its position is outside / inside \(GF_A\).
• $t tw(t_0 + p \cdot w_{len})$ contains a state of $o$ indicating that its position is inside / outside $GF_A$

Therefore, if the object $o$ has exited and entered again $GF_A$ during time period $(t_0 + (p - 1) \cdot w_{len}, t_0 + p \cdot w_{len}]$ without being sensed outside $GF_A$, $o$ is assumed to have stayed in $GF_A$. Indeed, MPO $o$ is defined to have stayed within $GF_A$ during time period $(t_0 + (p - 1) \cdot w_{len}, t_0 + p \cdot w_{len}]$ if both $t tw(t_0 + p \cdot w_{len})$ and $t tw(t_0 + (p - 1) \cdot w_{len})$ contain a state of $o$ indicating that its position is inside $GF_A$.

4.1.4 REMO Patterns

Separately monitoring the location of MPOs -like for the geofence pattern- is not enough for border security. Indeed, one of the goal of border security is to anticipate the behavior of MPOs relatively to each other in order to set up the right response. For example, MPOs heading to the border could be of more interest than those just moving along it. This is why it is important to evaluate the moving direction of MPOs. MPOs that are linked together should also drive more attention, as they are potentially more crucial than MPOs moving independently. MPOs can be linked to each other in several manners, for example when a group of MPOs moves in the same direction or when an MPO seems to lead other MPOs. All of those tasks can be solved by identifying REMO patterns.

REMO patterns (RElative MOtion [35, 36]) are moving point patterns -mostly set-based relative motion patterns- that are based on the analysis of the motion of MPOs over space and time. Indeed, the time is discretized according the considered TTW instances: the MPO states collected during the same TTW instance are considered to belong to the same time step (the end time of the considered TTW instance). The relative motion is analyzed by transforming the motion of MPOs to a REMO matrix featuring the chosen motion attribute in the REMO space: each column of the REMO matrix corresponds to a time step while each row corresponds to a different MPO. The motion of MPOs can be featured using different attributes such as speed, direction, acceleration, etc. This research work focuses on the direction attribute (also called route). The direction is represented as the motion azimuth ($0^\circ$ if the considered MPO moves towards the North, $180^\circ$ if it moves towards the South). The range of the motion azimuth ($[0^\circ, 360^\circ]$) is separated into eight directional sets. One considers that two MPO states correspond to the same direction if they belong to the same directional set. The REMO matrix is then analyzed and given spatial constraints are checked for the purpose of detecting three given spatially constrained REMO patterns: track, flock and leadership patterns. Generic motion patterns are only based on the analysis of the REMO matrix, as explained in Paragraph “Generic motion patterns”. Spatially constrained REMO patterns are generic motion patterns that also satisfy a given spatial constraint (cf. Paragraph “Spatially constrained REMO patterns”).
Patrick Laube, Marc van Kreveld, Stephan Imfeld

pitch. In reverse, MPOs moving around in a circle may build a wonderful cluster but never be converging. In addition the process of convergence and the final cluster are in many cases sequential. Consider the lifelines of a swarm of bees. At sunset the bees move back to the hive from the surrounding meadows, showing a strong convergence pattern without building a spatial cluster. In the hive the bees wiggle around in a very dense cluster, but do not converge anymore. In short, even though convergence and clustering are often spatially and/or temporally tied up, there need not be a detectable relation in an individual data frame under investigation.

The Basic REMO–Analysis Concept

The basic idea of the analysis concept is to compare the motion attributes of point objects over space and time, and thus to relate one object’s motion to the motion of all others (Laube and Imfeld 2002). Suitable geospatial lifeline data consist of a set of MPOs each featuring a list of fixes. The REMO concept (RElative MOtion) is based on two key features: First, a transformation of the lifeline data to a REMO matrix featuring motion attributes (i.e. speed, change of speed or motion azimuth). Second, matching of formalized patterns on the matrix (Fig. 1).

Two simple examples illustrate the above definitions: Let the geospatial lifelines in Fig. 1a be the tracks of four GPS-collared deer. The deer O1 moving with a constant motion azimuth of 45° during an interval t2 to t5, i.e. four discrete time steps of length \( \partial t \), is showing constance. In contrast, four deer performing a motion azimuth of 45° at the same time show concurrence.

![Figure 1](image)

**Figure 1.** The geospatial lifelines of four MPOs (a) are used to derive in regular intervals the motion azimuth (b). In the REMO analysis matrix (c) generic motion patterns are matched (d).

**Generic motion patterns** Figure 7 illustrates the process of transforming the motion direction of MPOs into a REMO matrix and detecting three generic motion patterns. The three considered generic motion patterns are:

- the *constance* pattern (*individual location* pattern): an MPO follows the same direction for \( r \) consecutive time steps (\( r \) is the *constance* parameter and is set beforehand)

- the *concurrence* pattern (*set-based relative motion* pattern): at least \( n_{\text{min}} \) different MPOs follow the same direction at a given time step (\( n_{\text{min}} \) is the *concurrence* parameter and is set beforehand)

- the *trend-setter* pattern (*set-based relative motion* pattern): an MPO moves according to a *constance* pattern until time step \( i \) and anticipates the motion of at least \( n_{\text{min}} \) MPOs (called “follower” MPOs) at time \( i \)
Figure 8: Illustration of the three spatially constrained REMO patterns. The arrows represent the direction of the considered MPOs, while the solid lines represent the motions of the MPOs. The gray circles and rectangles delineate for each pattern the area where the MPOs satisfy the spatial constraint. For the leadership pattern, a first spatial constraint concerns the locations of the leader MPO (whose trajectory is pictured by a thick line), while a second spatial constraint involves the last sensed locations of all MPOs. For the flock pattern, no trajectory is pictured as only the last situation picture matters for this pattern. If the parameters of the patterns have the values given in the three figures’ titles, the pictured MPOs satisfy the spatially constrained REMO patterns.

Spatially constrained REMO patterns  According to Tobler’s first law of geography, “Everything is related to everything else, but near things are more related than distant things.” This means that generic motion patterns can be more probably explained by the influence of certain MPOs if a proximity criteria is satisfied. Influence of different MPOs on each other’s motion is indeed bounded to a proximity criteria that can be expressed as spatial constraints. This is why focus is on REMO patterns fulfilling given spatial constraints. Spatial constraints can be of different sorts. The considered spatially constrained REMO patterns are illustrated on Figure 8 and explained below:

- The track pattern consists of the REMO constance pattern and maximal dimensions (width $\delta x_{max}$ and height $\delta y_{max}$ on Figure 8) concerning the minimum bounding rectangle (envelope) of the different MPOs’ different locations in time.

- The flock pattern consists of the REMO concurrence pattern as well as a maximum radius ($\delta r_{max}$) for the minimum bounding circle of the considered MPOs’ locations.

- The leadership pattern consists of the REMO trend-setter pattern with two spatial constraints: a maximal envelope (width $\delta x_{max}$, height $\delta y_{max}$) for the different locations of the leader MPO (similarly to the track pattern) and a circle of radius
Figure 9: Example of cases where an MPO is part of two patterns. In Figure 9(a), $MPO_A$ belongs to two different disks enclosing enough MPOs of the same direction at time $t$: $MPO_A$ is part of two flock patterns. In Figure 9(b), $MPO_A$ and $MPO_B$ follow each a track pattern associated with the same direction, are in each other’s neighborhood at the last time step and have enough MPOs following the same direction at this last time step: Thus, $MPO_A$ and $MPO_B$ are simultaneously leaders and followers of different leadership patterns.

Each time a new spatially constrained REMO pattern (thereafter called REMO pattern for simplification purpose) is detected, an alert is sent. At time $t$, an MPO can be part of different instances of leadership or flock patterns, as pictured on Figure 9. However, for flock patterns, in the situation where two flock patterns $Fl_A$ and $Fl_B$ are detected but $Fl_B$ involves all the MPOs that are part of $Fl_A$ plus other MPOs, only $Fl_B$ leads to an alert.

4.2 Problem Statement

The comparison of streaming platforms is drawn based on four pattern detection tasks: (1) a geofence detector, (2) a track detector, (3) a flock detector and (4) a leadership detector. Those four detection tasks can be gathered in two groups: (a) Geofence detector and (b) REMO detectors (track, flock and leadership detectors).

Both detectors take as input a stream of MPO states expressed in a GeoJSON format. The input stream is checked for geofence and REMO patterns based on a TTW processing whose length is set beforehand. New collected states during the last instance of the TTW are checked against geofence patterns if their refresh time is bigger than the last stored states of the considered MPOs. At each instance of the TTW, the REMO detector considers for each MPO the state corresponding to the biggest refresh time collected during the time window.
Thus, four detection tasks are implemented:

- the geofence detector is an individual location detector that determines to which predefined geofence an MPO characterized by its last sensed state belongs to (point-in-polygon problem) and whether it entered or exited one of the geofences. The geofences are stored beforehand.

- the track detector is an individual location detector that identifies REMO track patterns based on the user-defined constance parameter and spatial constraint.

- the flock detector is a set-based relative motion pattern detector that identifies REMO flock patterns based on the user-defined concurrence parameter and spatial constraint.

- the leadership detector is a set-based relative motion pattern detector that identifies leadership REMO patterns based on user-defined concurrence and constance parameters, and two spatial constraints.

This research work focuses on the implementation on three different real-time streaming platforms of the aforementioned streaming detection tasks. The chosen real-time streaming platforms (DSMS) are GeoEvent Ext., Spark and Flink. The objective is to draw a comparison between those three platforms against the streaming detection tasks.

5 Methodology

This section first explains how the research is organized. Then, the method for implementing the target streaming application is thoroughly explained. The primary focus is on solving the point-in-polygon problem concerning incoming MPOs against registered geofences, as well as determining whether an MPO enters/exits a geofence (geofence detector). The geofence detector is aimed to perform the aforementioned tasks for each batch of MPO states collected during an instance of a specified TTW. In a second part, the detection of specific REMO patterns is explained (REMO detectors). The REMO detectors are also based on TTW processing.

5.1 Overview of Research

The procedure used to carry this research is explained in this section. Airbus D&S’s needs are first translated into an application description and criteria about the DSMS platforms to use. Then, based on the cross-checking of those criteria and research made on DSMS technologies, three different DSMSs (GeoEvent Ext., Spark, Flink) are selected in order to implement the wanted application on each DSMS. After the evaluation of each application prototype based on evaluation criteria, those are compared against each other and benefits/disadvantages of each of them are highlighted. The whole process is illustrated on Figure 10.
5.2 Geofence Pattern Detection

The geofence detector (1) collects all the received MPO states during an instance of the TTW, (2) tags all the MPO states of the collected batch with the geofence they belong to (2) adds the whole transformed batch to the historical table (3) replaces the MPO states stored in the update table with the new transformed MPO states if they correspond to a bigger refresh time. The processing of a batch collected during a TTW instance is pictured on Figure 11. The description of the geofence detector is presented as a flowchart illustrating the computational steps on Figure 11. The different parts are further explained in the following subsections.
5.2.1 Ingestion of MPO states sent via TCP Socket

The MPO states (GeoJSONs) are received by the geofence detector via a TCP (Transmission Control Protocol) socket. TCP [32] is a connection-oriented protocol, which means that a connection between the server and the client is created before any data bit is sent. This connection is maintained until the data bits have been exchanged. TCP has the advantage to ensure that MPO states are received in the same order they have been sent. Moreover, TCP is high reliable as it offers guaranties of delivery by tracking the sent bits of data in order to prevent data loss and corruption. Each MPO state is processed when the whole batch of states received during the time tumbling window has been received.

5.2.2 Point-in-polygon problem

The geofence detector is based on three tables. A first table ($T_{GF}$) contains the geofences of the study area (GeoJSONs). $T_{GF}$ is a static table that is set before starting the application. A second table ($T_{update}$) contains the last sensed state of each MPO with a new field containing the ID of the geofence it belongs to (no value is associated with this field if the considered MPO does not belong to any field). $T_{update}$ is a dynamic continuously
updated table that is empty when starting the application. $T_{\text{hist}}$ stores all the tagged states (states containing the ID of the geofence the MPO belongs to) from the launch of the application, enabling the search for historical data: at every time step, the new MPO states are added to $T_{\text{hist}}$ while keeping the old states. By detecting to which geofence an MPO belongs to and comparing that with the geofence pointed by the last state of the considered MPO stored in $T_{\text{update}}$, one detects whether the MPO is entering/exiting/staying to/from/in one of the predefined geofences (according to the definitions of Section 4.1.3).

Let us consider the example of an object $o$. Three states are sensed for object $o$. At time $t_0$, object $o$ is in geofence $A$: $s_o(t_0)$ is sensed. $s_o(t_0)$ belongs to the TTW instance $i$. At time $t_1$ and $t_2$ ($t_2 > t_1 > t_0$), $o$ is in geofence $B$ generating the state $s_o(t_1)$ and $s_o(t_2)$. $s_o(t_1)$ and $s_o(t_2)$ belong to the TTW instance $i + 1$ (instance that is consecutive to the TTW instance $i$). At the end of the TTW instance $i$, $s_o(t_0)$ is collected: it is tagged with the ID of geofence $A$ and added to $T_{\text{hist}}$. Then, $s_o(t_0)$ is stored in $T_{\text{update}}$. At the end of the TTW instance $i + 1$, $s_o(t_1)$ and $s_o(t_2)$ are collected: they are tagged with the ID of geofence $B$ and stored in $T_{\text{hist}}$. Then, one searches for $o$’s last stored state in $T_{\text{update}}$ ($s_o(t_0)$): As $s_o(t_0)$ points to geofence $A$ and $s_o(t_2)$ (collected state that corresponds to the biggest refresh time) points to geofence $B$, an alert about $o$ entering $B$ and exiting $A$ is sent. Moreover, $s_o(t_0)$ is replaced by $s_o(t_2)$ in $T_{\text{update}}$.

5.2.3 Transmission of Alerts via UDP Socket

When an MPO enters or exits any geofence, an alert containing the MPO’s ID, its location and the considered geofence’s ID is sent via UDP socket. UDP [32] is a connectionless protocol, which means that no guaranty about the delivery of the data bits is offered. The data bits are sent without any acknowledgment and their order can be modified. However, this protocol is more efficient than TCP is terms of speed. Moreover, the alerts can be sent to many clients with the same UDP socket (whereas only one client is allowed for TCP as it is connection-secured), which is essential on an operator point of view. For the target application, some pattern loss due to communication failure is acceptable. Moreover, the order of the emitted alerts is less important than the ease of forwarding the alerts to multiple clients. Thus, UDP is preferred to TCP socket for sending the alerts.

5.3 REMO Patterns Detection

The REMO detectors aim at detecting specific REMO patterns within the received MPO states by adding a new column (corresponding to a new time tick) to the REMO matrix (thereafter termed $RM$) every time a new batch of states is collected during a TTW instance. For each TTW instance and for each MPO, only the MPO’s newest state (according to its refresh time) is considered.

Three REMO patterns are detected through the analysis of the updated REMO matrix:

---

8As $s_o(t_0)$ is the first sensed state, no state of $o$ is previously stored in $T_{\text{update}}$, so no alert is sent

9UDP stands for User Datagram Protocol.
• the track detector detects if an MPO belongs to the same direction set for \( r \) time steps (i.e. for \( r \) TTW instances, i.e. \( r \) last cells of \( RM \) on the row representing the motion of the considered MPO), \( r \) being the constance parameter, and has a trajectory (during those \( r \) time steps) that can be enclosed by an envelope whose dimension (height and width) are set beforehand.

• the flock detector detects if at least \( n_{min} \) \( \delta r_{max} \) being the concurrence parameter-
different MPOs belong to the same direction set at the same time tick (i.e. among the cells of the last added \( RM \) column) and can be enclosed by a circle whose maximum radius is \( \delta r_{max} \).

• the leadership detector identifies MPOs being part of a track pattern that have at least \( n_{min} - 1 \) MPOs belonging to the same direction set at the last time step and that can be enclosed by an envelope whose dimensions are set beforehand.

5.3.1 Spatial constraints

For detecting track patterns, one first detects constance patterns and then checks the spatial constraint. Concerning leadership patterns, one first detects track patterns, considers the associated MPO as a potential leader MPO, and checks if this MPO has enough MPOs following the same direction in its neighborhood at the last time step. For flock patterns, one first detects concurrence patterns by grouping MPOs according to the directional set they belong to. Then, within the created groups of MPOs, clusters of points that can be enclosed by a circle of dimension \( \delta r_{max} \) are considered: if enough MPOs lie within a circle, a flock pattern is detected. An alert is sent about the detected flock pattern if the involved MPOs are not a sub part of the MPOs involved by another flock pattern of the same time step.

The process of checking the spatial constraint associated with the flock pattern is thus less obvious than for track and leadership patterns as the spatial constraint is not associated with any reference MPO. The algorithm used to check this spatial constraint is clarified in Section 5.3.2.

5.3.2 Flock Pattern Detection: Filtering-Refinement approach

For the flock pattern, for each time step, one first generates groups of points following the same direction. Then, each group is checked against the spatial constraint. This spatial constraint is quite difficult to check as it is not based on a reference MPO (unlike the spatial constraints associated with the track and the leadership patterns). The approach proposed by [35] in order to check the spatial constraint (at least \( n_{min} \) MPOs have to be inside a disk of size \( \delta r_{max} \)) consists of (1) building an \( n_{min} \)-order Voronoi diagram\(^{10}\) (\( n_{min} \) being the parameter of concurrence) for each group of MPOs following the same direction.\(^{10}\) An \( n_{min} \)-order Voronoi diagram is a partition of the plane into cells based on given \( N \) points (\( N > n_{min} \)) called generators: a cell of the diagram consists of all the points of the space that have the same \( n_{min} \) closest generators among the \( N \) given generators.

\(^{10}\) An \( n_{min} \)-order Voronoi diagram is a partition of the plane into cells based on given \( N \) points (\( N > n_{min} \)) called generators: a cell of the diagram consists of all the points of the space that have the same \( n_{min} \) closest generators among the \( N \) given generators.
direction, (2) for each detected cluster of at least $n_{\text{min}}$ points (cell of the $n_{\text{min}}$-order Voronoi diagram), verifying that the minimum bounding circle has a lower radius than $\delta r_{\text{max}}$. However, generating Voronoi diagrams is quite complex to implement [42].

This is why another approach based on the DensityQuery algorithm presented in [31] is considered. However, [31] highlights an algorithm for detecting non-overlapping clusters, whereas the flock patterns can overlap each other for this research work. The DensityQuery algorithm is therefore altered to allow the detection of overlapping clusters of at least $n_{\text{min}}$ MPOs with a minimum bounding circle of maximal dimension $\delta r_{\text{max}}$. Disks of size $\delta r_{\text{max}}$ containing at least $n_{\text{min}}$ MPOs are called dense disks thereafter. The process consists of two phases: the Filtering (based on the DensityQuery algorithm of [31]) and the Refinement phases.

**Filtering phase** The Filtering phase consists of first partitioning the study area into square-shaped cells of size $a = 2r_{\text{max}}$. The goal of the Filtering phase is to detect cell combinations (called candidate areas) that could enclose dense disks based on the number of MPOs detected in each cell combination (cf. Figure 12). For each cell of the study area, three potential candidate areas are checked: $S_4$ consisting of the square of four cells having the current cell at its top left corner, $S_2$ consisting of the two top cells of $S_4$ and $S_3$ consisting of the two left cells of $S_4$. The different candidate areas are pictured on Figure 12: a grid of twelve cells is considered and the candidate areas directly associated with cell $(1, 1)$ are highlighted. If a considered cell $(i, j)$ is at the border of the study area, only its candidate areas that are totally inside the study area are considered. Indeed, checking a candidate area that is partially outside the study area would be wasteful as the part outside the study area is of no interest and the part inside the study area is already covered by one of the candidate areas of the neighborhood cells of cell $(i, j)$.
Algorithm 1: DensityQuery algorithm

Input: Radius $\delta r_{max}$, Threshold $n_{min}$ (parameter of concurrence)
Output: Set $flocks$

1 begin
2    Set $flocks$
3    Side of a cell $a \leftarrow 2r_{max}$
4    foreach cell $i$ of the study area do
5        $disk_i \leftarrow$ inscribed circle of cell $i$
6        $n_1 \leftarrow$ number of MPOs inside $disk_i$
7        if $n_1 \geq n_{min}$ then
8            $fl \leftarrow$ MPOs inside $disk_i$
9            CheckRedundant ($fl$, $flocks$)
10       $n_4 \leftarrow$ number of MPOs in $S_4$
11       if $n_4 \geq n_{min}$ then
12           $n_2 \leftarrow$ number of MPOs in $S_2$
13           if $n_2 \geq n_{min}$ then
14              invoke SRdisks ($S_2$, $\delta r_{max}$, $n_{min}$)
15              if flock $fl$ is found then CheckRedundant ($fl$, $flocks$)
16           $n_3 \leftarrow$ number of MPOs in $S_3$
17           if $n_3 \geq n_{min}$ then
18              invoke SRdisks ($S_3$, $\delta r_{max}$, $n_{min}$)
19              if flock $fl$ is found then CheckRedundant ($fl$, $flocks$)
20              invoke SRdisks ($S_4$, $\delta r_{max}$, $n_{min}$)
21              if flock $fl$ is found then CheckRedundant ($fl$, $flocks$)
22 end

The detailed algorithm of the Filtering phase is shown in Algorithm 1. For each cell $i$ of the study area, one considers its inscribed disk $^{11}$: if the number of MPOs detected within this disk is higher than $n_{min}$, a dense disk is detected (Lines 6–7). Then, the function CheckRedundant (Line 8) first ensures that the new detected flock $F_{iA}$ is not a sub-flock $^{12}$ of an already detected flock (redundancy check). If it is not the case, $F_{iA}$ is added to the set of $flocks$ that have been detected for the same time step, after having removed from this set the sub-flocks of $F_{iA}$. Then, the cell combination $S_4$ associated with cell $(i, j)$ is considered.

If $S_4$ has more than $n_{min}$ MPOs, one checks if there are more than $n_{min}$ MPOs within $S_2$. If it the case, it means that $S_2$ can contain dense disks. Thus, $S_2$ is processed by the algorithm SRdisks of the Refinement phase. If SRdisks produces an answer, which means that a flock is detected, this detected flock is checked for redundancy and

$^{11}$The dimension $\delta r_{max}$ of the disks is set by the user. The dimension $a$ of the square-shaped cells is derived from this value: $a = 2r_{max}$.

$^{12}$A sub-flock $b$ of a flock pattern $a$ is a detected flock that only involves MPOs that are part of flock $a$. 

34
Figure 13: Candidate areas and corresponding semi-random disks: For each candidate area, the gray lines or square represent the zone of the centers of the semi-random disks. $a/2$ and $\delta r_{\text{max}}$ are equal.

potentially added to the set flocks using function CheckRedundant. Using the same process as for $S_2$, cell combination $S_3$ is analyzed (Lines 17–19). Eventually, $S_4$ is checked as a whole for dense disks using algorithm SRdisks.

**Refinement phase** The Refinement phase -implemented by the SRDisks algorithm analyzes each candidate area selected by the Filtering phase (DensityQuery algorithm) by generating maximum $n_{\text{test}}$ semi-random disks ($n_{\text{test}}$ is set by the user from the beginning) of radius $\delta r_{\text{max}}$ that are totally inside the candidate area (cf. gray areas in Figure 13): the centers of the semi-random generated disks lie further than $a/2 = r_{\text{max}}$ ($a$ being the size of the side of a cell) from the boarders of the candidate areas. Every time the SRDisks algorithm is invoked by the DensityQuery algorithm for a specific candidate area, a disk -whose center lies within the gray area highlighted in Figure 13- is generated. If the number of MPOs that are within the considered disk is strictly lower than $n_{\text{min}}$, a new disk is generated, and so on until a disk contains more than $n_{\text{min}}$ MPOs (which means that the disk is dense) or until $n_{\text{test}}$ disks are generated. If the $n_{\text{test}}$ generated disk of the candidate area is not dense, no flock is detected and thus, no answer is produced. An example of dense disks detected by the Refinement phase is illustrated for each candidate area on Figure 14.

The SRDisks algorithm of the Refinement phase is probabilistic. Thus, it can miss flock patterns in the candidate area: the detection accuracy of the algorithm depends on $n_{\text{min}}$ and $n_{\text{test}}$. Tests about the detection accuracy of the SRDisks algorithm can be found in Section 7.1.

Using the Filtering and Refinement phases, the whole study area covered by the grid is checked for potential flocks.

Indeed, any current cell at grid coordinates $(i, j)$ can be part of two candidate areas $S_2$ generated by cells $(i, j)$ and $(i, j - 1)$, two candidate areas $S_3$ generated by cells $(i - 1, j)$ and $(i, j)$, and four square candidate areas $S_4$ generated by cells $(i - 1, j - 1)$,
Figure 14: Result of the Refinement phase for each candidate area: MPOs are represented by arrows while the semi-random generated disk that led to the detection of the flock is represented in gray. The MPOs that are part of the detected flock are represented as solid forms, those that are considered to be “outsiders” as empty forms $(i - 1, j)$, $(i, j - 1)$ and $(i, j)$. These candidate areas are all checked during the coarse of Algorithm 1. Furthermore, any disk of size $\delta r_{\text{max}}$ that overlaps with the current cell at grid coordinates $(i, j)$ is inside at least one of the aforementioned candidate areas generated by the cell and its neighbors, hence Algorithm 1 correctly checks all candidates areas and identifies all dense disks.
5.3.3 General Computational Steps

The REMO detectors ingest the lastly sensed MPO states and send alerts via UDP socket when spatially constrained REMO patterns (track, leadership and flock patterns) are detected. The REMO detectors are based on TTW processing: each batch of MPO states collected during an instance of the TTW is processed using the same algorithm, as highlighted in Figure 15. Indeed, incoming MPO states are not processed one by one but considered as part of a batch of states collected during an instance of the specified TTW. After collecting a whole TTW’s batch of MPO states, only the newest state (state corresponding to the biggest refresh time of an MPO) of each MPO is considered. Those states are assigned to a directional set (motion azimuth range is divided in eight sets). Then, a first thread (thread 4.a on Figure 15) consists of identifying flock patterns by (1) grouping MPOs belonging to the same directional set in order to detect concurrence patterns within this batch (step 4.a.1.), (2) filtering those that satisfy the spatial constraint (step 4.a.2.) using the parameter $n_{min}$ and (3) sending alerts about the detected flock patterns (step 4.a.3.). A second thread (thread 4.b on Figure 15) adds the whole batch to the REMO matrix ($RM$ on Figure 15) and keeps the size of $RM$ along the time dimension strictly below $r + 1$, with $r$ being the parameter of the generic motion pattern constance pattern.
Operating System | Virtual processor | RAM | Number of cores
---|---|---|---
Windows Server 2012 | Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz 2.00 GHz x 4 | 8.00 GB | 4

Table 2: Characteristics of the Real-time Server, Big Data Server, GIS Application Server machines

(cf. Section 4.1.4) used for detecting track and leadership patterns. Constance patterns are detected and filtered by the spatial constraint associated with the track pattern (step 4.b.3). While alerts about the detected track patterns are being sent, the MPOs that are part of track patterns are checked for a second spatial constraint: one filters those that have enough MPOs (more than \( n_{min} \)) that belong to the same directional set at the last time step (step 4.b.a.1). Eventually alerts about the detected leadership patterns are sent.

6 Three Implementations of the Target Streaming Application

The target streaming application is implemented on three different DSMSs: GeoEvent Ext., Spark and Flink. The streaming applications based on those DSMSs are thereafter respectively called GeoEvent Ext., Spark and Flink solutions. Concerning the geofence detector part of the target streaming application, the storage of MPOs is performed using the ArcGIS Spatiotemporal Big Data Store in relation with the ArcGIS GeoEvent Extension for Server for GeoEvent Ext. solution, whereas Elasticsearch is used for similar purpose for Spark and Flink solutions. One must however notice that those storage possibilities are equivalent, as ArcGIS Spatiotemporal Big Data Store is based on Elasticsearch. Thus, no comparison is drawn about the storage means of the different solutions.

6.1 GeoEvent Ext. Solution

GeoEvent Ext. solution is based on a three machine site, which is presented in a first part. Then, the actual implementation using box-and-arrows language is explained. Eventually, pre-required steps for running the application GeoEvent Ext. solution are outlined.

6.1.1 A Three Machine Architecture

GeoEvent Ext. is an extension of ArcGIS for Server that allows the user to process real-time data streams and produce new data streams or other relevant outputs -such as notifications- in the frame of a GIS application. If the ingested data items have to be stored, ArcGIS Spatiotemporal Big Data Store (NoSQL database based on Elasticsearch) can be used. In this case, three different hosts are at least needed [37, 52]. For

---

\(^{13}\)The ArcGIS Spatiotemporal Big Data Store is an Esri product that enables the user to store stream elements in the frame of a streaming application.
this purpose, three virtual machines are used. The first machine—called thereafter Real-
Time Server machine—hosts ArcGIS for Server and GeoEvent Ext., two software packages
aiming at ingesting and processing input streams. The Big Data Server machine hosts
ArcGIS Spatiotemporal Big Data Store, a search engine-like software used for storing
outputs of GeoEvent Ext. The third machine thereafter called GIS Application Server
machine hosts possible desktop, web and mobile clients of the application, as well as a
software called Portal for ArcGIS that can be used for visualizing observation data stored
by ArcGIS Spatiotemporal Big Data Store. In order to increase the storage capacity and
the write throughput, it is also possible to run ArcGIS Spatiotemporal Big Data Store
across multiple Big Data Server machines.

Table 2 presents the hardware used to build this three machine site.

### 6.1.2 Actual Implementation

This section presents only the implementation on GeoEvent Ext. of the geofence detector.
The REMO detectors can indeed not be implemented using Esri software. This is due to
the fact that the REMO detectors are intrinsically bounded to the the definition of TTW
instances\textsuperscript{14}, which are not possible to implement using GeoEvent Ext. GeoEvent Ext.
proposes only a limited choice of processing items that do not include window-based
processing elements. Furthermore, even if it is possible to implement new processing
elements and add them to GeoEvent Ext., their structure is based on the generation of one
output data item for each ingested input data item\textsuperscript{15}.

Concerning the geofence pattern, its description in Section 5.2 is based on TTW pro-
cessing: By comparing the location associated with the last ingested state of an MPO
in a TTW instance \(i\) with the location associated with the last ingested state of MPO
in TTW instance \(i + 1\), one determines whether an MPO has stayed in/entered/exited
a geofence. However, the definition of TTW instances is very cumbersome in GeoEvent
Ext. (as explained earlier). This is why the geofence detector is implemented on Geo-
Event Ext. based only on a comparison between the locations associated with the last
two ingested states of each MPO without making use of TTW instances: for each new
received MPO state, its associated location is directly compared to the last sensed loca-
tion of the same MPO, alerting thus directly whether the considered MPO entered/exited
a geofence.

The graphical box-and-arrow implementation of the geofence detector on GeoEvent
Ext. is shown on Figure 16.

\textsuperscript{14}The MPO states that are sensed within the same TTW instance are considered to be associated to
the same time step. TTW instances allow thus to group MPO states that were sensed “simultaneously”,
enabling the update of the REMO matrix (Figure 7): each time a TTW instance is processed, a new column
associated with the new sensed “simultaneous” MPO states is added to the REMO matrix.

\textsuperscript{15}More explanation about why the REMO detectors cannot be implemented on GeoEvent Ext. can be
found in Section 8.2.
Figure 16: Implementation of the geofence detector with GeoEvent Ext.

The input connector Tracks (green rectangle) ingests MPO states received as GeoJSONs through TCP socket and translates them into GeoEvents, an internal representation of the object understood by GeoEvent Ext. [13]. The ingested stream is then processed by three threads in parallel using:

- **Filters** (yellow diamonds on Figure 16): Components that filter out GeoEvents according to their attributes or spatial relationships with stored geofences.

- **Processors** (yellow rectangles on Figure 16): Components that enrich GeoEvents (by calculating new attributes based on other attributes for example), alter their attributes’ values according to mathematical/lexical expressions or transform their spatial attributes according to given spatial transformations (convex hull, envelope, etc.).

Both used filters (EnteringTrack, LeavingTrack) are spatial filters related to the geofences stored beforehand. Indeed, geofences are stored in GeoEvent Ext. under the name “zone” and the ID of the geofence. For instance, in order to filter MPOs that are entering the geofences, one uses a graphical user interface that allows a reference to geometrical objects stored in ArcGIS GeoEvent Extension for Server, as shown on Figure 17. The other spatial filter LeavingTrack is defined using the same GUI.
The first thread consists of adding a field “zone” to the MPO states received through the input connector Tracks and filling this new field with the geofence’s ID the MPOs belong to (processor GeoTagger-geofence). The MPO states are eventually stored into the Spatiotemporal Big Data Store, either as historical data using the output connector AllStates (all the MPO states that have been received from the launch of the application are stored there) or as current data using LastUpdatedStates (only the last sensed state of the MPOs are kept in the table).

The second thread filters the MPOs entering any geofence (filter EnteringTrack), tags them with the appropriate geofence’s ID (processor GeoTagger-EnteringZone) and sends the corresponding altered GeoJSON with header “EnteringTrack” as an alert via UDP socket (output connector AlertUdp).

The last thread filters the MPOs exiting any geofence (filter LeavingTrack) and sends the corresponding GeoJSON to the same UDP socket. One must notice that it is not possible to tag exiting MPOs with the geofence that has been left as ArcGIS GeoEvent Extension offers no processor performing such a task.

### 6.1.3 Prerequisites

Before running the application, the geofences have to be stored in ArcGIS GeoEvent Extension as geofences. This is achieved with the help of a small ArcGIS GeoEvent Extension application which ingests GeoJSONs (MultiPolygons) that represents geofences from a TCP socket (sent via a Java program) and stores them into ArcGIS Spatiotemporal Big Data Store. Once the geofences are stored, they can be declared as geofences and thus used in other applications on ArcGIS GeoEvent Extension. A possible alternative is to create a map of geofences on an Esri software - such as ArcMap - and import this map as ArcGIS map package on GeoEvent Ext.

### 6.2 Spark Solution

The implementation based on the MapReduce-based DSMS (cf. Section 3.1.2) Spark is presented in this section. The geofence detector and REMO detectors of Spark solution are implemented using the Java API of Spark. The program needs the following input parameters: the host/port of the TCP socket used for receiving the input stream as well as the host/port of the UDP socket used for sending alerts. The specific input parame-
ters and requirements of each detector are explained later. The program is based on the
discretization of the input stream into small batches according to Spark batch interval.
First, each program part consists of creating a DStream out of the MPO states received
per TCP socket and the chosen Spark batch interval. Indeed, a DStream consists of a
continuous serie of RDDs (Resilient Distributed Dataset), Spark’s abstraction of an im-
mutable, distributed dataset [7]. A DStream can be transformed into a new DStream
by applying MapReduce operations on its underlying RDDs. The applied MapReduce
operations are either self-sufficient operations (for example the GroupByKey MapReduce
operations) or operations based on a user-defined function (for example in the case of a
Transform MapReduce operation which consists of transforming each incoming RDDs
according to a user-defined function). Output operators (also called Sinks) are eventually
used in order for external systems to consume the processed streams. In order to imple-
ment an application using Spark, one first selects the MapReduce operators to use and
then implement the functions needed by those operators.

Sections 6.2.1 and 6.2.2 concern both the Spark and the Flink solutions. Indeed, both
solutions share the same hardware environment and the same search engine for storing
processed tagged MPO states. Sections 6.2.3 and 6.2.4 focus on the actual implementa-
tion. Eventually, parameters of the Spark solution are highlighted in Section 6.2.5.

### 6.2.1 Used Hardware

A virtual machine based on a CentOS 7 Linux distribution is used for running Spark
and Flink solutions. This choice has been adopted because Elasticsearch is primarily
developed to run on Unix, which means that all the documentation is written for a Linux
environment. Table 3 sums up the characteristics of the used hardware for both Spark
and Flink solutions. For the implementation, only one virtual machine has been used.
However, one must remark that both Flink and Spark can be run across a cluster of
machines. Elasticsearch, the search engine used for both solutions (cf. Section 6.2.2)
can also be clustered.

### 6.2.2 Using Elasticsearch to store GeoJSONs

Search engines are systems aimed to retrieve information from a collection of documents
based on the user’s request. Elasticsearch (thereafter called ES) is an open-source real-
time full-text search engine [24] based on Apache Lucene library (a text search engine
written in Java) [29]. ES can handle application scaling by distributing processing across
different nodes. The nodes’ health is constantly evaluated, leading to data re-balancing in
case of failure. ES’s data is stored as structured JSON format and data can be requested

<table>
<thead>
<tr>
<th>Linux Distribution</th>
<th>Virtual processor</th>
<th>RAM</th>
<th>Number of cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>CentOS 7</td>
<td>Model Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz x 4 (vCPUs)</td>
<td>8.00 GB</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3: Used hardware for Spark and Flink solution
and queried using a RESTful API based on JSON over HTTP or a Java API. For this research work, the Java API was used for storing and retrieving data.

Basic units of information expressed in JSON \(^{16}\) are termed documents in ES. They are organized using some specific rules. The first rule is that documents that are indexed under the same index must share common characteristics (logical category chosen by the user). A second rule is that documents that have the same index and type must share identical or very similar JSON schemas (same fields and types associated with each field). Thus, what one traditionally calls “table” can be translated for ES as a set of documents sharing the same index and type. Within a specific index and type, documents are identified by their ID. Therefore, each document is uniquely described by its tuple (index, type, id). The geofence detector part of the implementation takes advantage of ES for the following storage needs: (1) the index/type zone/poly stores all the geofences before launching the application, (2) the index/type spark/allStates stores all the tagged MPOs’ (historical table \(T_{\text{hist}}\) of Section 5.2.2) when the application is running, and (3) spark/lastUpdatedStates stores the last sensed tagged MPO states (update table \(T_{\text{update}}\) of Section 5.2.2) when the application is running. Flink solution uses ES in an equivalent way (the used index is flink). In order to store the predefined geofences before launching the application, ES’s REST API can be used, as shown on Figure 18. The REMO detectors do not use ES as no storage table is needed by the user (the only outputs are the UDP alerts).

### 6.2.3 Implementation of the Geofence Detector

The geofence detector is implemented using only two Spark operators, namely a socket-TextStream (type of Source Operator ingesting data via a socket) generating the DStream out of the data received via TCP socket, a foreachRDD primitive (generic output operator enabling the processing of RDDs according to a user-defined function) operator processing the MPO states, storing them in ES and sending alerts about entering and exiting MPOs using the implemented function AssignToZone. Those general steps are featured on Figure 19.

---

\(^{16}\)As GeoJSON is a specific JSON format, documents can be directly expressed in GeoJSON format.
The core of the *geofence* detector is thus the function *AssignToZone*. Its pseudo-code is explained in Algorithm 2. A prerequisite for this function is that the geofences be stored in ES. For each MPO state $s$ of the received RDD (Line 3), one searches for the last stored state ($\textit{oldState}$) associated with this MPO in ES type $\textit{lastUpdatedStates}$ (Line 6).

Then, one checks if the concerned MPO belongs to one of the geofences. If it the case (Line 9), one tags its new state with the ID of the geofence (Line 11). After checking that $s$ corresponds to a bigger refresh time than $\textit{oldState}$ (Line 12), the geofence pointed by $\textit{oldState}$ is compared with the one the considered MPO currently belongs to. This comparison leads to alerts being sent (Lines 14–16). Eventually, the altered state $s$ replaces $\textit{oldState}$ under ES type $\textit{lastUpdatedStates}$.

If the current MPO does not belong to any *geofence* (zone) and its new ingested state corresponds to a bigger refresh time than its last stored state under $\textit{lastUpdatedStates}$ (Line 18), one checks whether this last stored state shows that the MPO was in a geofence before and sends an alert according to that (Lines 19–20). The new state of the MPO replaces the old one under the index $\textit{lastUpdatedStates}$.

Eventually, the altered state $s$ of the current MPO is stored under the type $\textit{allStates}$.

---

17 *Spark* creates batches of stream elements that are not time-ordered
Algorithm 2: AssignToZone

Input: RDD batch
begin
List zones ← stored geofences in ES
foreach MPO state $s$ in batch do
  Boolean $isInZone = false$
  Integer $mID ← s.ID$
  oldState ← document with ID equal to $MPO_ID$ stored under type
  lastUpdatedStates
  foreach geofence $z$ in zones do
    Integer $zoneID ← z.ID$
    if $s.geometry$ inside $z.geometry$ then
      $isInZone = true$
      $s.zone ← zoneID$
    if $s.refreshTime > oldState.refreshTime$ then
      if oldState.zone different from $s.zone$ then
        Send alert about entering MPO $mID$ in geofence $zoneID$
      if oldState.zone points to the ID of a zone then
        Send alert about exiting MPO $mID$ from geofence $oldState.zone$
      Replace oldState with $s$ under type lastUpdatedStates
  if $isInZone = false$ and $s.refreshTime > oldState.refreshTime$ then
    if oldState.zone points to the ID of a zone then
      Send alert about exiting MPO $mID$ from geofence
    Replace oldState with $s$ under type lastUpdatedStates
  Store $s$ under type allStates
end

6.2.4 Implementation of the REMO Detectors

The REMO detectors detect three different spatially constrained REMO patterns that are the track, flock and leadership patterns. Spark batch interval is used for creating TTW instances. The end time of each TTW instance corresponds to the time step used for generating new columns of the $RM$ matrix. Figure 20 shows the Spark operators used for programming. The MPO states are first received as a batch through TCP socket from the source operator SocketTextStream. The second step is to keep only the states of the batch that correspond to the biggest refresh time of each MPO. This is done by transforming the received RDD with the function LastStates in the frame of the Transform operator\textsuperscript{18}.

\textsuperscript{18}The Transform operator enables to transform an RDD according to a user-defined function.
Three threads based on the DStream generated by the transform operator are used to detect the different spatially constrained REMO patterns.

**Flock pattern** Thread a) aims at detecting flock patterns. The received MPO states (received as a batch from the transform) are first mapped to a key/value pair with the ID of the directional set the considered MPO belongs to as the key and the received MPO state as the value using the function DirAsKey in the operator MapToPair. Pairs that have the same key are grouped together (groupByKey operator). Then, the groups containing more than $n_{\text{min}}$ elements ($n_{\text{min}}$ being the parameter of concurrency) are filtered using the function FilterConcurrence within Spark operator filter. Eventually, each forwarded group is processed independently using the function DetectFlocks of the foreachRDD operator. The function DetectFlocks takes as input an RDD representing a set of groups of key/value pairs (MPO states belonging to the same constance pattern). Each group is processed by the DensityQuery algorithm presented in Algorithm 1. Based on the set of detected flocks produced by the DensityQuery algorithm, alerts are sent.

**Track pattern** Thread b) detects track patterns. The MPO states are first mapped to a key/value pair with the following tuple as key: $(mID, dID)$, with $mID$ and $dID$ being respectively the ID of the MPO and the ID of the directional set it belongs to. This is done by the function IDandDirAsKey in the MapToPair operator. Then, the generated RDD of tuple is processed using the transformToPair operator that is based on the function DetectConstance aiming at detecting constance patterns. DetectConstance is based on the update of $RM$, the REMO matrix truncated to the last $r$ time steps ($r$ being the parameter of concurrency). $RM$ is implemented as an ArrayBlockingQueue (“a bounded blocking queue backed by an array” according to Oracle’s documentation) containing RDDs. Every time a new RDD is received, it is
Table 4: Concordance between Spark operations and the building blocks used to describe the REMO detector (cf. Figure 15)

<table>
<thead>
<tr>
<th>Thread</th>
<th>Spark operation</th>
<th>Corresponding building block(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a)</td>
<td>-socketTextStream</td>
<td>1. Collect MPO states received during TTW instance via TCP socket</td>
</tr>
<tr>
<td></td>
<td>-transform(LastStates)</td>
<td>2. Keep in the batch only the last state of each MPO</td>
</tr>
<tr>
<td></td>
<td>-mapToPair(DirAsKey)</td>
<td>3. Assign each MPO state of the batch to a directional set</td>
</tr>
<tr>
<td></td>
<td>-groupByKey</td>
<td>4.a.1. Identify instantaneous group motion patterns</td>
</tr>
<tr>
<td></td>
<td>-filter(FilterConcurrence)</td>
<td>4.a.2. Verify the spatial constraint (without reference object)</td>
</tr>
<tr>
<td></td>
<td>-foreachRDD(DetectFlocks)</td>
<td>4.a.3. Send alert via UDP socket about detected flock patterns</td>
</tr>
<tr>
<td>b)</td>
<td>-mapToPair(IDandDirAsKey)</td>
<td>3. Assign each MPO state of the batch to a directional set</td>
</tr>
<tr>
<td></td>
<td>-transformToPair(DetectConstance)</td>
<td>4.b.1 Add the batch to RM as a new column and keep size along time dimension below $r + 1$</td>
</tr>
<tr>
<td></td>
<td>-filter(SpCstrnt)</td>
<td>4.b.2. Identify temporal motion patterns</td>
</tr>
<tr>
<td></td>
<td>-foreachRDD(SendAlerts)</td>
<td>4.b.3. Verify the $1^{st}$ spatial constraint (with reference object location)</td>
</tr>
<tr>
<td></td>
<td>-foreachRDD(SendAlerts)</td>
<td>4.b.3.a.2. Send alert via UDP socket about detected track patterns</td>
</tr>
<tr>
<td>c)</td>
<td>-cogroup</td>
<td>4.b.a.1. Verify the spatial constraint (with reference object location)</td>
</tr>
<tr>
<td></td>
<td>-foreachRDD(DetectLeaderships)</td>
<td>4.b.3.a.2. Send alert via UDP Socket about detected Leadership patterns</td>
</tr>
</tbody>
</table>

added to $RM$ and the oldest stored RDD in $RM$ is removed. Storing directly RDDs in the ArrayBlockingQueue allows the processing of the stored RDDs using the MapReduce operations proposed by Spark. After having added the new RDD to $RM$, all the RDDs of $RM$ are grouped into a single RDD called $RM_{RDD}$. The key/value pairs of $RM_{RDD}$ are then grouped according to their keys. The groups that contains more than $r$ pairs (parameter of constance pattern) are forwarded to the output: those groups are sets of MPO states, each of them describing the evolution of an MPO following a constance pattern. The operator transformToPair thus generates an RDD containing those groups. The third step of thread b) consists of filtering the groups of MPO states that satisfy the spatial constraint using the operator filter and the function SpCstrnt: the dimensions of the envelope of the locations of the considered MPO are checked. If the spatial constraint is satisfied, the group of MPO states associated with the considered MPO is forwarded to the output.

Eventually, using the foreachRDD operator, those groups of pairs are used for sending the appropriate alerts based on the function SendAlerts.
Leadership pattern Thread c) deals with the detection of leadership patterns. In order to do that, the detected track patterns are grouped with the RDD containing all the new ingested states (keyed by ID and ID of directional set) using Spark’s operator cogroup. This approach of grouping two stream partitions in order to detect leadership patterns is used because it is not possible to access RM as it is “owned” by the function DetectConstance. The output RDD is then processed by the function DetectLeadership of the operator foreachRDD. The newly ingested states are first grouped according to their directional set. Then, for each detected track pattern, one checks if the group the considered MPO belongs to has more than \( n_{\min} \) pairs. If it is the case, one counts the MPOs of the group that also lie within the rectangle (its dimensions are given by the user) having the considered MPO at its center. If there are more than \( n_{\min} \) MPOs, an alert is sent.

For clarifying purposes, Table 4 shows each Spark operator (Figure 20) used for implementing the REMO detectors on Spark and associates it with one or more theoretical building blocks of the REMO detectors. Those building blocks were highlighted in Section 5.3.3 and illustrated in Figure 15.

6.2.5 Tuning Spark Solution

The key concept of Spark [7] is that incoming data is ingested by so-called Receivers that generate discretized streams (DStreams) by chunking the incoming data into batches (RDDs) and store them in Spark’s memory for processing. Spark processes one batch at a time with no pipelining (cf. Section 2.3.3) among batches. Each discretized stream (set of RDDs) is associated with a unique Receiver. Receivers also partition each batch into blocks, according to the following relation:

\[
N_{\text{block}} = \frac{t_{\text{batch}}}{t_{\text{block}}}
\]

with \( N_{\text{block}} \), \( t_{\text{batch}} \) and \( t_{\text{block}} \) being respectively the number of blocks per batch per Receiver, Spark batch interval and Spark block interval.

Processing is then performed by Spark’s engine which runs \( N_{\text{block}} \) short tasks per batch per Receiver. Those tasks are distributed among the available cores. However, as each Receiver is run as a long running task assigned to one core, one has to provide enough cores to run Spark application: if there are more Receivers than available cores, the input streams will be ingested but not processed. One must also notice that at the end of each operation, the blocks are gathered again into RDDs, so that operators input and output RDDs. This means that Spark’s engine waits for the slowest block before recreating the RDD. Furthermore, it is possible to set the number of parallel tasks \( N_{\text{partition}} \) for each distributed shuffle operation. It is also possible to explicitly repartition the blocks created by the Receivers in a similar way as for MapReduce operators using the operator repartition\( (N_{\text{partition}}) \) (\( N_{\text{partition}} \) parallel tasks distributed among the cores).

\[19\] An object can only be accessed by one Spark’s operator in order to prevent that resources be damaged by problems due to resource sharing while pipelining the stream.
Number of worker threads to run the application In the case of a local deployment, one must specify the number of threads to be used for running the tasks. Spark uses one worker thread to run the receiver (the socketTextStream) and at least one worker thread to process the stream. Thus, more than one worker thread has to be allocated to Spark solution.

Spark batch interval One of the crucial parameter of any streaming application on Spark is the batch interval (which corresponds to the length of a TTW). It should be greater than the processing time of each batch in order to limit overheads. If one batch has a processing delay that is higher than Spark batch interval, the consecutive batch is put in the queue during a time period called scheduling delay. This is due to the fact that Spark considers (default behavior) only one batch to be processed at each point of time. Therefore, if only one batch is processed at a time, the system is stable if no batch is queuing (this is equivalent to having a scheduling delay close to null or having an average processing delay that is lower than Spark batch interval). However, the processing delay depends from the amount of records in the considered batch and the processing task.

Level of parallelism (Number of parallel tasks \( N_{\text{partition}} \)) In order to use efficiently the resources and overcome bottlenecks, the number of parallel tasks has to be set to the number of available cores. In the case of this research work, the running environment is composed of four cores. As one core is already used by the receiver, the maximum value for the level of parallelism is 3, which also the chosen value. This default level of parallelism is used for the Spark operations that involve partitions of the stream.

6.3 Flink Solution

Flink solution is implemented in the same hardware environment as Spark solution (cf. Table 3) and also shares the same search engine Elasticsearch (cf. Section 6.2.2).

Flink is a hybrid DSMS. It is indeed a workflow-based technology that can consider elements of the stream as key/value pairs (like in MapReduce DSMSs). Flink is however not based on any abstraction of a data set, as incoming events are ingested one by one.

Flink is based on two building blocks: streams representing intermediate results and transformations (equivalent of the operators in Spark) representing operations on one or more streams that generates one ore more streams. Collecting elements received during a specific time window is also done using a transformation, namely the windowAll transformation, as opposed to Spark which directly considers batches of incoming elements when ingesting them. The windowAll transformation thus decides when the collected elements can be processed and removed from the current window instance. Then, in order to process window-collected elements, Flink offers the apply transformation, which

\(^{20}\text{This behavior can be changed, but the stability of the system is harder to analyze and it can yet, according to Spark documentation, lead to misuse of sharing of resources.}\)
allows to “apply a general function to the window as a whole” [5] called *window function*. The `WindowAll` and `apply` transformations are thus linked, as the *window function* is used to process the collected elements once the system decides that a window instance is ready to be processed based on a trigger. This means that the processing of window-collected elements cannot be split into many successive *Flink transformations* -which would lead to weird behaviors as the functions involved by the transformations would not be triggered by the `windowAll` transformation- but has to be done using one `apply` transformation. However, many `apply` transformations can be parallelized in order to simultaneously process copies of the window-collected elements according to different algorithms.

*Flink* converts the user program into a *streaming dataflow* composed of streams and transformation operators. Parallel processing in *Flink* dataflows is further explained in paragraph 6.3.3. Furthermore, *Flink* uses *pipelining* to process the records of the input stream. *Flink* is based on a *Lazy Evaluation*, which means that first “each operation is created and added to the program’s plan” [4] and then the operations are executed according to the execution plan.

Section 6.3.1 clarifies the implementation of the *geofence* detector using *Flink*. Then, Section 6.3.2 focuses on the implementation of the REMO detectors. Eventually, important tuning parameters are pointed out.

### 6.3.1 Implementation of the Geofence Detector

The implementation of the *geofence* detector using *Flink* is quite similar to the implementation using *Spark*, as pictured on Figure 21. *Flink’s* source operator just connects to the TCP socket without creating any batch of events. Thus, the main difference between the implementations of the *geofence* detector in *Flink* and *Spark* is that one has to create an explicit TTW in *Flink* in order to process the MPO states in batches, whereas *Spark* directly considers batches of incoming elements. Each instance of the TTW collects the MPO states as a list (which is the equivalent of the collected RDD in *Spark*). *Flink’s* `AssignToZone` function differs from *Spark’s* equivalent in that it inputs a list of MPO states collected by the TTW instead of an RDD.

### 6.3.2 Implementation of the REMO Detectors

*Flink* being based on event-per-event processing while *Spark* directly considers batches of incoming events in a *MapReduce* environment, it is not possible to directly transpose *Spark’s* operators into *Flink’s* transformations. *Flink’s* transformations used for implementing the REMO detectors are illustrated on Figure 22.

*Flink’s* main difference with *Spark* is that one has to explicitly invoke a transformation
for collecting the MPO states received during each TTW instance. After having ingested an MPO state using the Source operator `socketTextStream`, the received states are thus collected based on a TTW whose length is set by the user using a the `windowAll` transformation. The output of the `windowAll` transformation is a `WindowedStream` containing lists of states, each list corresponding to the states collected during a TTW instance.

Eventually, the collected states are processed using two parallel `apply` transformations: a first one aiming at detecting `flock` patterns, and a second aiming at detecting `tracks` and `leadership` patterns.

**Flock pattern**  The input list (states collected during a TTW instance) of states is processed using the function `DetectFlocks` in the frame of the `apply` transformation. The `DetectFlocks` function (1) selects only the last state of each MPO, (2) groups the states corresponding to the same directional set under the same key of `HashMap`, (3) processes each group of states corresponding to the same directional set using the `DensityQuery` algorithm presented in Algorithm 1 and sends UDP alerts when `flocks` are detected.

**Track and leadership patterns**  As for Spark solution, the detection of `track` and `leadership` patterns is based on the update of `RM`, the REMO matrix truncated to its last `r` time steps (`r` being the parameter of `constance`). `RM` is also implemented as a Java `ArrayBlockingQueue`. However, instead of storing RDDs (as it is the case in Spark solution), lists of tuples are stored in this `ArrayBlockingQueue`. Thread b)

On each list of states generated by a TTW instance, the function `DetectTracksAndLeaderships` is applied using Flink’s the `apply` transformation. It is necessary to detect `track` and `leadership` patterns within the same transformation because (1)`RM` can be accessed only by one transformation (as in Spark) (2) the processing of elements collected by a `windowAll` transformation has to be done using an `apply` transformation and not split into many transformations (cf. Introduction of Section 6.3).

This is why the function `DetectTracksAndLeaderships` is quite complex. It can be separated into eight parts: (1) select the last state of each MPO (2) map each state to a tuple of three elements namely `(mID, dID, s)` with `mID`, `dID` and `s` being respectively the ID of the considered MPO, the ID of the directional set the MPO belongs
Table 5: Correspondance between building blocks of the REMO detectors (cf Figure 15) and Flink transformations

<table>
<thead>
<tr>
<th>Flink transformation</th>
<th>Corresponding building block(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-socketTextStream</td>
<td>1. Collect MPOs’ states received during TTW instance via TCP socket</td>
</tr>
<tr>
<td>-windowAll</td>
<td>2. Keep in the batch only the last state of each MPO</td>
</tr>
<tr>
<td>(TumblingProcessingTimeWindow)</td>
<td></td>
</tr>
<tr>
<td>-apply (LastStates)</td>
<td>3. Assign each MPO state of the batch to a directional set</td>
</tr>
<tr>
<td></td>
<td>4.a.1. Identify instantaneous group motion patterns</td>
</tr>
<tr>
<td></td>
<td>4.a.2. Verify the spatial constraint (without reference object)</td>
</tr>
<tr>
<td></td>
<td>4.a.3. Send alert via UDP socket about detected flock patterns</td>
</tr>
<tr>
<td>-apply (DetectFlocks)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. Assign each MPO’s state of the batch to a directional set</td>
</tr>
<tr>
<td></td>
<td>4.b.1. Add the batch to RM as a new column and keep size along time dimension below $r + 1$</td>
</tr>
<tr>
<td></td>
<td>4.b.2. Identify temporal motion patterns</td>
</tr>
<tr>
<td></td>
<td>4.b.3. Verify the 1st spatial constraint (with reference object location)</td>
</tr>
<tr>
<td></td>
<td>4.b.3.b. Send alert via UDP socket about detected track patterns</td>
</tr>
<tr>
<td></td>
<td>4.b.3.a.1. Verify the 2nd spatial constraint (with reference object location)</td>
</tr>
<tr>
<td></td>
<td>4.b.3.a.1. Send alert via UDP Socket about detected leadership patterns</td>
</tr>
<tr>
<td>-apply (DetectTracksAndLeadership)</td>
<td></td>
</tr>
</tbody>
</table>

6.3.3 Tuning Flink Solution

As Flink is based on parallel processing, streams and operators are respectively partitioned into stream partitions and subtasks. The parallelism of an operator is the number of threads it can execute in parallel. To achieve optimal performance, it is necessary to balance the load among these threads. This can be done through various tuning techniques, such as adjusting the number of subtasks or the size of the window.

The Flink transformations used for implementing the REMO detectors can be associated with one or more building blocks used to describe the REMO detectors (cf. Figure 15). This is highlighted in Table 5.
of subtasks of this operator. It is possible to set a global parallelism at the application level, which means that all the operators involving partitioning would have the same parallelism in the application. Streams can either link two operators in a one-to-one pattern \(^{21}\) or a redistributing pattern \(^{22}\).

7 Empirical Evaluation

This section introduces the experiments performed on the geofence and REMO detectors. Section 7.1 focuses on the accuracy of the SRDisks algorithm used for detecting flock patterns within the candidate areas selected by the SRDisks algorithm of the Refinement phase (cf. Section 7.1.2). The real-time detection of patterns is evaluated in Section 7.2: the effect of several parameters -characterizing either the input stream or the patterns to be detected- on the computer resources used by the chosen program (geofence detector or one of the REMO detectors) is evaluated. Only the implementations on Flink and Spark are evaluated and compared, as GeoEvent Ext. solution does not share the same hardware environment and as the REMO detectors could not be implemented on GeoEvent Ext. due to the lack of support for creating time windows (cf. Section 8.2).

7.1 Evaluation of the SRDisks Algorithm

As explained in Section 21, the SRDisks algorithm is only probabilistically correct, as the disks used for detecting flock patterns within candidate areas selected by the DensityQuery algorithm are semi-randomly generated. In order to assess the detection accuracy of the algorithm, a random flock pattern is generated inside a given candidate area and tested against the SRDisks algorithm. This is further explained in Section 7.1.1. The results of this experiment are analyzed in Section 7.1.2.

7.1.1 Description of the Experiment

This experiment aims at estimating the probability of correctness of the SRDisks algorithm of the Refinement phase used for detecting flocks (cf. Section 21). In order to do that, one generates 1000 times a random flock pattern according to the same constance parameter \(n_{\text{min}}\) within the same candidate area (S2 or S4). Flock patterns are generated as follows: (1) a flock center is selected from the admissible locations according to a uniform distribution (cf. Figure 13), (2) \(n_{\text{min}} - 1\) MPOs are uniformly placed within \(\delta r_{\text{max}}\) distance from the flock center, \(\delta r_{\text{max}}\) being the disk radius associated with the flock spatial constraint. \(\delta r_{\text{max}}\) is fixed for all the generated flock patterns. Then, for each of the 1000 generated flock pattern, the SRDisks algorithm is invoked with the same maximum number of tests \(n_{\text{test}}\) (which corresponds to the maximum number of

\(^{21}\)One-to-one pattern: Each operator subtask outputs one sub-stream that is ingested by one operator subtask, preserving thus the same stream partitions for the upstream and downstream operators.

\(^{22}\)Redistributing pattern: operator subtasks generate more than one stream partition, changing thus the partitions of the stream between upstream and downstream operators.
Figure 23: SRDisks algorithm: Effect of the maximum number of semi-randomly generated disks $n_{\text{test}}$ within each candidate area (S2 and S3)

generated disks). By counting the number of cases –out of the 1000 cases- for which the SRDisks algorithm detects the flock pattern, the detection accuracy associated with the chosen $n_{\text{min}}$ and $n_{\text{test}}$ is calculated. The variation of $n_{\text{test}}$ produces a curve representing the detection accuracy for a given $n_{\text{min}}$.

Those detection accuracy curves are produced for two candidate areas: S2 and S4 (cf. Figure 12 of Section 5.3.2). As S3 covers the same area than S2, the results obtained for S2 are supposed to be the same for S3.

7.1.2 Results and Analysis

Figure 23 shows the evolution of the detection accuracy as a function of the maximum number of tests performed by SRDisks algorithm for the S2 and the S4 candidate areas. Each curve corresponds to the detection of a flock pattern associated with a different parameter of constance $n_{\text{min}}$. The raw results of the experiment can be found in Appendix 11.1. As expected, the higher $n_{\text{test}}$, the better the accuracy of the flock detection for both candidate areas, as the generation of more disks leads to a higher probability to find a disk that actually contains a flock pattern. One could intuitively think that less tests (lower $n_{\text{test}}$) are needed for detecting pronounced flock patterns (characterized by a larger $n_{\text{min}}$). However, Figure 23 shows the contrary: the larger $n_{\text{min}}$, the more tests are needed to detect the corresponding flock pattern. A possible explanation is that a larger $n_{\text{min}}$ value is associated with a more pronounced circular shape of the corresponding flock pattern, leading to individual points becoming more important for defining that shape. This means that there exists fewer disks that contain the corresponding $n_{\text{min}}$ MPOs. Thus, more semi-random generated disks (higher $n_{\text{test}}$) are needed for detecting the corresponding flock pattern.

For fixed $n_{\text{min}}$ and $n_{\text{test}}$, the detection accuracy associated with candidate area S2 is better than the one associated with S4. This is due to the fact that S4 covers an area that
is twice as big as the one covered by $S2$. Thus, $n_{\text{min}}$ should be chosen according to the results of $S4$.

### 7.2 Evaluation of the Real-time Detection of Patterns

This section presents the experiments that have been performed on the previously described Spark and Flink solutions. The GeoEvent Ext. solution is not compared with the other solutions, as it is not based on the same hardware environment and as the REMO detectors are not implemented on GeoEvent Ext.

Each experiment corresponds to a set of tests, each of them associated with the processing of a given input stream (generated by a simulation program) by a given detector according to a given setting.

For each test, a simulation program generates a stream of MPO states simulating the movement of a given number of MPOs according to a constant speed and direction initially set for each MPO. This is further explained in Section 7.2.1. The chosen simulation stream is then processed by a detector according to a user-defined setting, as explained in Section 7.2.3. While the simulation stream is being processed during fourteen minutes, different system performance metrics are sensed every two seconds (cf. Section 7.2.2). Then, for each system metric, the mean value ($\mu$) and standard deviation ($\sigma$) over the fourteen minutes are calculated, thus enabling a comparison between given settings, as shown in Section 7.2.4. Further descriptive (median, minimum and maximum) of the evolution of the sensed metrics can be found in Appendix 11.2.

#### 7.2.1 Simulation Data

For all the experiments, a stream of MPO states is generated by a simulation program.

The states generated by the simulation program are expressed in GeoJSON format, according to the structure pictured in Figure 24. The geometry field holds the geographical location of the MPO while the properties field holds the characteristics of the MPO. The location of each MPO is expressed according in WGS84 (cf. Section 4).
The simulation program simulates the movement of MPOs by continuously calculating the following locations of the MPOs based on their initial states and sending the updated states according to a rate set by the user. It takes as input a file containing the initial state of each considered MPO. The program considers the first MPO state, calculates its new location based on the initial speed (between 0 and 500 km/hour) and direction, generates a new state by just changing the location and sends it via TCP socket. After having waited for $\Delta t$ milliseconds ($\Delta t$ is set by the user and determines the input rate for the geofence and REMO detectors), the program does the same with the second MPO state of the input file, and so on until the end of the input file. Then, the whole process is repeated with the last updated states instead of the initial states, and so on, until the user stops the program.

The MPOs are thus considered to have a constant speed and direction. The user can nevertheless control the number of considered MPOs (by changing the number of considered MPOs in the input file), their initial states as well as the rate at which the MPO states are sent.

### 7.2.2 Evaluation Metrics

Nmon for Linux\(^{23}\) [40] is an IBM software that enables the measurement of several system performance metrics at regular intervals during a user-defined time period. For all the experiments, the following Nmon metrics are sensed every two seconds for fourteen minutes while a given input stream generated by the simulation program is processed by the geofence detector or one of the REMO detectors: CPU usage, disk activity and active memory [41].

**CPU usage** CPU time is the time needed by the CPUs (1) to execute the code in user space and (2) to perform system calls to the kernel due to the program. Codes are executed in user space if no access to the kernel is needed. System calls to the kernel are

\(^{23}\)Nmon stands for “Nigel’s performance Monitor”
performed when the user space process has to get something from the system in order to control the process, to manage files and devices, to perform information maintenance or to communicate with other devices. CPU usage represents CPU time expressed as a percentage of one CPU capacity. Indeed, as CPU time is allocated in clock ticks (discrete time slices), CPU usage is the ratio between the number of clock ticks corresponding to CPU time and the total number of clock ticks since the last update. As the hardware environment is based on four CPUs, the value of CPU usage can be up to 400%.

CPU usage is expressed in percentage, so the mean ($\mu$) and standard deviations ($\sigma$) calculated over values of CPU usage sensed every two seconds during fourteen minutes are respectively expressed in percentage (%) and percentage point (p.p.).

**Disk activity** Disk activity is a percentage representing how busy the disk is. This activity is mainly due to IO operations (operations needed to write/read data on/from the disk). When performing IO operations the CPU is usually idle. The disk is also active when waiting for the read-and-write heads to reach the wanted sectors on the disk.

**Active memory** The active memory is the amount of RAM (in MB) that is actually being currently used by processes.

### 7.2.3 Description of the Experiment

For all the experiments, the TTW’s length is set to five seconds. This parameter has not been changed as increasing TTW’s length is equivalent to ingesting more MPO states within each TTW instance, which is already evaluated by considering more MPOs within the study area (cf. later in this section).

All the experiments consists of running one of the detectors several times with different settings during fourteen minutes on a given input stream generated by the simulation program. During the fourteen minutes the chosen detector is running, the system metrics described in Section 7.2.2 are sensed every two seconds.

Concerning the REMO detectors, their user-set spatial constraints are considered to be either “circular” (for the *flock* and the *leadership* patterns) or “square-shaped” (for the *track* and *leadership* patterns). Thus, when referring to a spatial constraint in this section, the given value corresponds to the radius of a circular spatial constraint or to the side of the square-shaped spatial constraint.

The simulated stream of MPOs states being based on WGS84 geographical locations, the spatial constraints used for the REMO patterns detectors are expressed in degrees\(^{24}\). This means that the “square-shaped” spatial constraints with reference object location expressed for the *track* and *leadership* patterns are actually distorted depending on the reference object location. This also means that the grid used for detecting clusters of MPOs for the *flock* pattern is based on “square-shaped” cells whose side are expressed in degrees. However, as one degree along one parallel and along one meridian do not represent the same geographical distance, and as those distances change with the considered

---

\(^{24}\)It is too heavy to embed a library converting coordinates in a *Flink* or *Spark* applications
area on the Earth, the cells considered for the grid do not have the same surface and their distance along parallels and meridians are not equal. A better alternative for creating the grid is to choose an equal area projection. However, as only small values are used for the spatial constraints (up to 0.2° for the side of the “square-shaped” constraints and of the cells used for the grid detecting flocks) in these experiments and as the considered geographical study area is quite limited (2°x2°), one can consider that a given spatial constraint involves approximately the same area for each considered location.

**Geofence detector** Two experiments are carried out on the geofence detector: one consists of evaluating the effect of the number of geofences (thereafter called “Effect of the number of geofences”), while the second focuses on the number of considered MPOs within each TTW instance (thereafter called “Effect of the number of considered MPOs within each TTW instance”). For both experiments, the same geographical study area is chosen. The initial locations of the MPOs are chosen according to a uniform distribution in the study area. Their initial speed and direction are chosen randomly and kept constant. The geofences are chosen such that they cover the whole study area. The parameters used for each experiment are summarized in Table 6: the default parameters are pictured in bold. The length of the used TTW is not changed as increasing the TTW length is equivalent to ingesting more MPO states within each TTW instance.

**Effect of the number of geofences** For this experiment, the stream generated by the simulation program consists of the states of 100 MPOs periodically sent. The rate of this stream is chosen such that during each time period of five seconds a new state is generated for each object, meaning that 100 new states are ingested during each TTW instance of five seconds (cf. Table 6: 100 considered MPOs within each TTW instance). This simulation data stream is processed four times (four times fourteen minutes of processing), the only difference between the four tests being the number of considered geofences (4, 16, 64 or 256 geofences).

**Effect of the number of considered MPOs within each TTW instance** For this experiment, the setting of the geofence detector (TTW length of five seconds and 64 geofences) is fixed while the characteristics of the input stream are changed: four different data streams are processed, each data stream corresponding to a different number of considered MPOs within one TTW instance. For example, when the number of considered MPOs within one TTW instance is set to 100, it means that the input file of the simulation program contains 100 states corresponding to 100 different MPOs and that the input rate

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTW’s length (sec.)</td>
<td>5</td>
</tr>
<tr>
<td>Number of geofences</td>
<td>4, 16, 64, 256</td>
</tr>
<tr>
<td>Number of considered MPOs within each TTW instance</td>
<td><strong>100, 200, 500</strong></td>
</tr>
</tbody>
</table>

Table 6: Parameters and values used for the experiments on the geofence detector
Table 7: Parameters and values used for the experiments on the track detector

of the states is set such that the geofence detector ingests 100 updated MPO states during each TTW instance (five seconds).

**Track detector** For the track detector, the effects of the constance parameter (cf. Section 4.1.4) and of the number of considered MPOs within each TTW instance are investigated. For both experiments, the initial locations of the MPOs are chosen according to a uniform distribution in a fixed study area. The parameters used for each experiment are highlighted in Table 7. It has been chosen to fix the dimension of the spatial constraint for all the experiments on the track detector. Indeed, varying the dimension of the spatial constraint should not lead to any significant changes in performances, as it is implemented in the track detector by a spatial filter on the selected constance patterns. Whether a constance pattern is filtered out or not by this spatial filter only determines if an alert is sent or not, which is not worth evaluating as this empirical evaluation focuses on the performance of the actual pattern detection and not on the performance of UDP messaging of alerts.

**Effect of the constance parameter** This experiment consists of processing a stream of states concerning 100 MPOs (for each MPO, a new updated state is ingested during each TTW instance, leading to a total of 100 ingested states for each TTW instance) with a track detector with three different values for the constance parameter. TTW’s length and spatial constraint are fixed for the three tests.

**Effect of the number of considered MPOs within each TTW instance** In a similar way to the experiment carried on the geofence detector (cf. Section 7.2.3), four different simulated streams, each of them corresponding to a different number of considered MPOs within each TTW instance, are processed by a track detector whose setting is fixed (TTW’s length, constance parameter and spatial constraint are kept to default values).

Table 7 sums up the parameters used for each experiment on track detector.

**Flock detector** For the flock detector, the effect of three parameters are highlighted: the spatial constraint (cf. Section 5.3.2), the distribution of the MPOs and the number of considered MPOs within each TTW instance. The spatial constraint seems particularly important as it controls the construction of the grid used for the Filtering phase. It has been chosen to vary the MPO distribution, as this parameter controls the candidate areas.
**Parameter** | **Setting**
--- | ---
TTW’s length (sec.) | 5
Study area (°) | 2x2
*Concurrence* parameter | 5
Spatial constraint (radius °) | 0.0025, 0.005, 0.02, 0.1
Maximum number of semi-random generated disks in the *Refinement* phase | 100
Number of considered MPOs within each TTW instance | 100, 200, 500, 1000
Number of MPO centers | 1, 5, 24

Table 8: Parameters and values used for the experiment on the *flock* detector

selected by the *Filtering* phase. Eventually, varying the number of considered MPOs within each TTW instance allows to draw conclusions about how the algorithm deals with different information load. The maximum number of semi-random generated disks in the *Refinement* phase is fixed for all the experiments on the *flock* detector, as its effect is already assessed in Section 7.1. The concurrence parameter is fixed to a value of 5 for all experiments on the *flock* detector in order to ensure that the *Refinement* phase is invoked and that *flocks* are detected. Setting the value of the maximum number of semi-random generated disks to 100 leads to a detection accuracy close to 100% for all the candidate areas according to Figure 23.

Unlike the experiments on the *geofence* detector and the *track* detector, the experiments on the *flock* detector are based on a simulated stream characterized by MPOs all having a speed of 100 km/hour and whose initial locations are not set according to a uniform distribution. Indeed, in order to set the MPOs’ initial locations, one selects $n_c$ “MPO centers”, associates $\frac{1}{n_c}$ share of the considered MPOs with each MPO centers and sets their location according to a normal distribution having its mean coordinates at the corresponding MPO center and a standard deviation of 0.005°. The motion direction of the MPOs associated with the same center is the same. Furthermore, the speed of all MPOs is set to 100 km/hour, so that the characteristics of the input distribution is kept.

The settings of the three experiments on the *flock* detector are summarized in Table 8.

**Effect of the spatial constraint** The experiment consists of considering an input stream based on the update of the states of 500 MPOs. The input rate is set so that 500 new states are ingested for all MPOs during each TTW instance (500 considered MPOs within each TTW instance). The 500 MPOs are associated with one MPO center and follow the same direction at the same speed. Four tests are carried out, each of them corresponding to a different disk radius associated with the *flock* spatial constraint, thus changing the characteristics of the grid associated with the *Filtering* phase (cf. Section

---

The experiments on the *flock* detector are based on an input stream describing the movement of initial *flocks*: the movement of the members of an initial flock is constant, so that the considered flock does not dissolve over time. The *flock* detector thus detects always the same MPOs as parts of *flock* patterns. A more challenging input stream could have been to consider an input stream that describes MPOs that steer to move towards the average position of geographically close MPOs, such as described by the *Boids* algorithm [44].
Table 9: Parameters and values used for the experiment on the leadership detector.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTW’s length (sec.)</td>
<td>5</td>
</tr>
<tr>
<td>Constance parameter</td>
<td>25</td>
</tr>
<tr>
<td>1\textsuperscript{st} spatial constraint (°)</td>
<td>0.35</td>
</tr>
<tr>
<td>Concurrency parameter</td>
<td>5</td>
</tr>
<tr>
<td>2\textsuperscript{nd} spatial constraint (radius (^{°}))</td>
<td>0.0025, 0.005, \textbf{0.02}, 0.1</td>
</tr>
<tr>
<td>Number of considered MPOs within each TTW instance</td>
<td>100, 200, \textbf{500}, 1000</td>
</tr>
<tr>
<td>Number of MPO centers</td>
<td>5</td>
</tr>
</tbody>
</table>

Effect of the distribution of the MPOs  For this experiment, the characteristics of the flock detector are fixed: fixed concurrency parameter, spatial constraint and maximum number of semi-random generated disks. This flock detector is used against three different streams, each of them characterized by 500 considered MPOs within each TTW instance. The only difference between the three streams is that different initial distribution are considered: those initial distributions are characterized by 1, 5 or 24 MPO centers.

Effect of the number of considered MPOs within each TTW instance  For this experiment, the load of the input stream varies while all the other parameters are kept constant. Three different input stream are considered, each of them corresponding to a given number of considered MPOs within each TTW instance. However, all the input streams have the same MPO initial distribution: all the initial MPOs’ locations are based on the same MPO center for the three the input streams.

Leadership detector  For the leadership detector, the experiments aim at observing the effects of the 2\textsuperscript{nd} spatial constraint (spatial constraint associated with the concurrency part of the leadership pattern, cf. Section 5.3.3) and of the number of considered MPOs within each TTW instance. The effects of the first spatial constraint and of the constance parameters are not analyzed as those are already highlighted by the experiments on the flock detector, the leadership detector being an extension of the flock detector.

As for the experiments on the flock detector, the input stream is based on MPOs, whose initial locations are set according to a normal distribution whose mean values are the coordinates of the chosen MPO centers. All the MPOs have a speed of 100 km/hour and all the MPOs associated with the same center follow the same direction.

The parameters of each experiment are presented in Table 9.

Effect of the 2\textsuperscript{nd} spatial constraint  One considers an input stream characterized by 500 considered MPOs within each TTW instance. The initial locations of the 500 MPO states are based on 5 MPO centers. Four tests are carried, each of them corresponding to a different size of the radius of the circular-shaped second spatial constraint.
Effect of the number of considered MPOs within each TTW instance For this test, only the simulated input stream changes for each of the four tests. Each test corresponds to a different number of considered MPOs within each TTW instance. However, all the input streams correspond to the same distribution, namely initial MPOs’ location set according to 5 MPO centers.

7.2.4 Results and Analysis

This section presents the results of the experiments using plots illustrating the evolution of a system metric’s statistic (mean $\mu$ and standard deviation $\sigma$) as a function of the considered parameter.

The full system metrics’ statistics (mean, median, standard deviation, minimum and maximum) can be found in Appendix 11.2.

**Geofence detector**

Effect of the number of geofences Figure 25 shows the evolution of the mean and standard deviation of the system metrics as a function of the number of geofences. For both Spark and Flink solutions, the average CPU usage increases linearly with the number of geofences according to the same rate (0.25 p.p/geofences). This is explained by the fact that when the number of geofences increases, the program goes through a bigger list of geofences when analyzing the location of each MPO. The variability of the CPU usage for both solutions is quite high, as their standard deviations (between 45 and 70 p.p. for Spark, around 15 p.p. for Flink) are significant in comparison to their associated mean values (between 56 and 119 % for Spark, between 18 and 80 % for Flink). Flink solution has generally a lower average CPU usage and a lower associated standard deviation than
Spark solution. Furthermore, the number of geofences does not really affect the standard deviation of the CPU usage associated with Flink solution, whereas standard deviation of Spark’s CPU usage increases with the number of geofences.

The lower and more stable Flink CPU usage has nevertheless its counterpart in a higher memory activity compared to Spark solution, as pictured in Figure 25(b). The memory activity increases slightly for both solutions with the number of geofences, which can be explained by the same argument as for the CPU usage.

The average disk activity for both solutions is approximately constant with the number of geofences, as increasing the number of geofences does not involve disk usage, the geofences being kept in the cache. Disk activity is however not close to null, as ES -like any other search engine- heavily uses disk in order to store the ES documents (cf. Section 6.2.2) that are not directly needed [38]. Disk activity is characterized for both solutions by a very high variability, the values of the mean and standard deviation being of the same order.

**Effect of the number of considered MPOs within each TTW instance**  The evolution of the statistics of each system metric as a function of the number of considered MPOs can be found in Figure 26. Increasing the number of considered MPOs within each TTW instance produces for both Spark and Flink solutions an increase in CPU usage, as more MPO states need to be checked for every time tumbling window instance.

However, the most notable growth concerns the average value of disk activity for both solutions, leading to a high disk activity for 500 considered MPOs (63% for Spark and 48% for Flink). This is explained by the more intense use of ES which has to fetch, for each ingested state during the last TTW instance, the old state of the corresponding MPO among all the stored states. The standard deviation of the disk activity however decreases with the number of considered MPOs, possibly because the disk is used for a longer time
for processing each batch of states collected during the TTW instance. The average value of the active memory increases slightly with the number of considered MPOs for Spark solution, which is explained by the fact that more states need to be processed for each TTW instance. On the opposite, Flink solution involves a stable average value for the active memory, but its value is always higher than the one involved by Flink solution.

Both solutions have also been tested with more than 500 considered MPOs within each TTW instance. However, both systems could not keep up with the input rate, leading to window-collected states waiting for being processed. This is probably due to a too high disk activity, as one has observed that the number of considered MPOs has a direct influence on disk activity (cf. Figure 26(b)). Indeed, it is usually admitted that a non critical disk activity is below 70 % [41].

**Track detector**

**Effect of the constance parameter** Figure 27 highlights the evolution of the statistics about the system metrics as a function of the value of the constance parameter. Notable differences between Spark and Flink solutions: while the value of the constance parameter has a really slight impact on Flink solution which is characterized by low CPU values (the average CPU value increases from 6.4 to 10.7 % when the constance parameter goes from 5 to 100, cf. Table 15), one observes a clear increase of the mean CPU

---

26 Please note the TTW has a length of five seconds, while the system metrics are sensed every two seconds.

27 In order analyze whether the system can keep up with the input rate, both Spark and Flink provides a web interface that allows the user to compare the number of ingested states (Flink)/batch of states (Spark) with the number of processed states/batches of states.
usage with the \textit{constance} parameter for \textit{Spark}. CPU usage, disk activity and active memory have globally a bigger variability (higher standard deviation) when \textit{Spark} solution is used. The average value of active memory increases a little bit with the \textit{constance} parameter for both \textit{Spark} solution. This is due to the fact that, when the \textit{constance} parameter increases, more memory is needed as the \textit{RM} matrix is bigger (the MPO states are stored during more TTW instances). Disk activity is globally low as ES is not used in the \textit{track} detector (as for all the REMO detectors), unlike in the \textit{geofence} detector.

\textbf{Effect of the number of considered MPOs within each TTW instance}  The evolution of the system metrics’ statistics as a function of the number of considered MPOs within each TTW instance is pictured in Figure 28. The number of considered MPOs affects both solutions: one observes that the mean CPU usage increases almost linearly with this parameter. However, the increase is a lot smaller for \textit{Flink} (0.01306 p.p./MPO rate with a coefficient of determination of 0.98) than for \textit{Spark} (0.1155 p.p./MPO rate with a coefficient of determination of 0.96). As more MPO states are received during each TTW instance, one expects the active memory to increase with the number of MPOs. This is observed for \textit{Spark} solution but not for \textit{Flink} solution, probably because the MPO states are so light (they are not memory-intensive) that there is practically no difference for the memory between ingesting 100 or 1000 MPO states\textsuperscript{28}. The average disk activity also increases with the number of MPOs for \textit{Spark} solution, probably for the same reasons as for the average active memory. Disk activity stays however very low for both solutions, which shows that the system supports the increase of the number of MPOs.

\textsuperscript{28}As for \textit{Flink} solution, the performance is maybe not measured for enough challenging streams. However, the number of ingested MPO states within each TTW instance is limited by the execution time of the simulation program, ie. the time needed by the simulation program to update the states of all the considered MPOs should be lower than the chosen TTW length.
**Flock detector**

**Effect of the spatial constraint**  Figure 29 shows the evolution of the mean and standard deviation of the system metrics as a function of the size of the radius determining the minimum bounding circle for MPOs being part of the same *flock* pattern. For both *Flink* and *Spark* solutions, the spatial constraint has an influence on the average CPU usage: when the radius of the minimum bounding circle increases, the average CPU usage decreases and the corresponding standard deviation decreases. This is explained by the fact that increasing the radius leads to the generation of a grid containing less cells, as the square-shaped cells have a side that is the double of the radius. Thus, as less cells are being checked for *flock* patterns (cf. Algorithm 1), less resources are needed. Both solutions are also characterized by practically no disk activity as the *flock* detector does not need to store states between consecutive TTW instances, which is the case of the *track* and *leadership* detectors that are based on the update of RM (REMO matrix, cf. Section 4.1.4). The evolution of the active memory does not show any pattern for both solutions, making it difficult to interpret.

**Effect of the distribution of the MPOs**  The evolution of the system metrics’ mean and standard deviation as a function of the number of centers is illustrated on Figure 30. Increasing the number of centers in the initial distribution of MPOs of the input stream has a slight impact on both solutions: the mean CPU usage and its corresponding standard deviation increase with the number of centers. Indeed, if the input distribution is characterized by more centers, the MPOs are more scattered, which leads to more candidate areas being selected by the DensityQuery algorithm of the Filtering phase (cf. Section 5.3.2) in order to be processed by the SRDisks algorithm of the Refinement phase. *Flink* uses globally less CPU than *Spark* solution. Active memory does not change.
with the distribution of the input MPOs.

Figure 30: Flock detector: Effect of the distribution of the MPOs

Figure 31: Flock detector: Effect of the number of considered MPOs within each TTW instance

**Effect of the number of considered MPOs within each TTW instance**  The evolution of the system metrics’ descriptive statistics as a function of the number of considered MPOs within each TTW instance can be found on Figure 31. The number of considered MPOs has a slight impact on Spark’s mean CPU usage and a very slight impact on Flink’s CPU usage: the mean CPU usage increases with the number of considered MPOs. Indeed, as the TTW instances contain more MPO states, one needs more
resources to collect and process them. The number of MPOs does not seem to have an impact on disk activity and active memory.

**Leadership detector**

**Effect of the 2\textsuperscript{nd} spatial constraint**  Figure 32 shows the evolution of the descriptive statistics as a function of the 2\textsuperscript{nd} spatial constraint. For both solutions, the descriptive statistics seems to be constant, which is surprising because a bigger 2\textsuperscript{nd} spatial constraint involves more possible follower MPOs. An evolution could maybe have been observed if more MPOs had been considered within each TTW instance, so that the number of MPOs selected by each spatial constraint would have been more different. While active memory is similar for both solutions, the mean CPU usage and its corresponding standard deviation are higher for Flink solution.

**Effect of the number of ingested MPOs during each TTW instance**  Concerning the evolution of the system metrics’ descriptive statistics with the number of considered MPOs within each TTW instance, the observed evolutions are similar to what was observed for the track detector. This is due to the fact the leadership detector is an extension of the track detector. The mean CPU values are slightly higher for the leadership than for the track detector, as more processing is needed in order to check the 2\textsuperscript{nd} spatial constraint.
Figure 33: Leadership detector: Effect of the number of ingested MPOs during each TTW instance

8 Discussion

This section consists of a comparison of the three platforms against several criteria: the installation of the DSMS, the continuous query definition, the implementation of the detectors (streaming application) on those DSMSs. Eventually, this section extrapolates the performance results on simulation data to real-time performance on real data based on a comparison of Flink and Spark solutions.

8.1 Installation

Installing GeoEvent Ext. in order to create a streaming application able to store information in the ArcGIS Spatiotemporal Big Data Store requires a lot of time, as a three machine architecture has to be created. This means all the Esri software needed to create such a streaming application must be linked through the three machines (cf. Section 6.1.1).

Compared to installing GeoEvent Ext., installing Spark and Flink is quite easy: one only need to expand the open-source compressed folder to install the corresponding DSMSs. Then, in order to implement Java applications based on those DSMSs, the corresponding Java libraries need to be downloaded or referred to in a Maven Java project.

Thus, installing Spark/Flink is less time-consuming and also lighter than installing GeoEvent Ext. It is also important to note that an Esri license has to be bought in order to create an streaming application on GeoEvent Ext., while Spark/Flink are free and open-source.
8.2 Continuous Query Definition

*GeoEvent Ext.* is based on a graphical box-and-arrows language that is quite straightforward, making it very simple to create a streaming application. However, this simplicity has also its downside: the user has to choose between already out-of-the-box filters and processors (cf. Section 6.1.2) that can possibly not satisfy the needs of the streaming application. One of the big limitations of the proposed processors for example is that there exists no TTW processor, making it really cumbersome to implement the REMO detectors on *GeoEvent Ext.* An alternative could have been to implement the wanted processor using the *GeoEvent* SDK. However, the implementation of processors using the *GeoEvent* SDK is very codified and imposes to consider one output stream element for each received input element, making it difficult to implement a processor based on ingestion of several consecutive elements such as a TTW. Moreover, all the implementations’ examples of processors and filters using the *GeoEvent* SDK are based on element-per-element processing. Nevertheless, this should be possible as some out-of-the-box processors and filters are based on the comparison of ingested events, such as the *EnteringTrack* filter (cf. Section 6.1.2) that detects when MPOs enter the geofences. The code used for implementing this special spatial filter is unfortunately not exposed by *Esri.*

The code used for *Spark* (MapReduce-based technology) is less verbose than the one used for *Flink.* Indeed the MapReduce operators can directly be used to implement steps of the detectors without having to go through all the MPO states that are collected during each time TTW instance (cf. Figure 20). For *Flink,* one really has to directly consider the list of collected states. Moreover, the limitation due the fact that the states collected by the TTW have to be processed in one apply transformation (cf. Figure 22) can make the code a little bit less understandable than for *Spark.*

8.3 Implementation of the *Spark* and *Flink* Solutions

*Flink* and *Spark* do not belong to the same group of DSMSs. Indeed, *Flink* is a true streaming platform in the sense that *Flink*’s core engine is a streaming dataflow engine that considers each incoming element separately, while *Spark*’s core engine is a batch processing engine that processes each RDD (*Resilient Distributed Datasets*: batch of stream elements) separately.

*Spark’s batches versus Flink’s collection of input elements* This paradigm difference has a direct influence on the implementation. Indeed, each *Spark* RDD contains a collection of MPO states that arrived over the batch period (length of the TTW) but are not ordered according to the time they were ingested. Thus, even if the input stream of MPO states is ordered according to the refresh time of each state (cf. Section 5.2) -which should be the case as TCP connection ensures that the MPO states are received in the same order they are sent-, the batches created by *Spark* do not keep this order internally. This is why the *geofence* detector checks for each MPO state that the refresh time is bigger than the last stored state in the $T_{update}$ before inferring about a certain behavior of the
considered MPO (whether the MPO entered/exited a geofence). This step of filtering for each MPO only the state corresponding to a bigger refresh time than the last stored state in $T_{update}$ could have been removed from the implementation on Flink, as Flink keeps the order of the received MPO states.

**MapReduce operations on Flink**  
Flink being a hybrid DSMS, it considers each ingested element separately and offers at the same time some MapReduce operators. However, those MapReduce operators have not been used for the implementation of either the geofence or REMO detector. This is mainly due to the necessity of considering only the newest states of each TTW instance as part of the new column of the RM matrix. Indeed, if one considers this step as unnecessary -which could possibly lead to the detection of flock patterns involving different states of the same MPO-, it is possible to (1) map the input states to tuples containing the direction and the state using the $map(DirAndState)$ transformation (2) key the stream of tuples based on the direction using the $key(Dir)$ transformation, generating thus a stream partitioned according to the direction of the MPOs (i.e. maximum eight partitions) termed KeyedStream, (3) apply a window ($TumblingProcessingTimeWindow$) transformation grouping thus the states corresponding to the same direction together and collected during the same TTW instance, (4) eventually search for cluster of MPOs using the $apply(DetectFlocks)$, DetectFlocks being a function that applies the DensityQuery and the SRDisks (cf. Section 5.3.2) on the set of MPO states.

**Time windows on Spark and Flink**  
Spark relies on the division of the input stream in successive batches of elements (RDDs). This division in batches exists for every streaming application implemented on Spark, but it also directly corresponds to the definition of TTW instances. This is why the Spark batches were used as TTW instances for Spark solution. However, if another type of time window -such as time window instances that are overlapping in time- is needed, a special Spark operator is needed: $window(windowLength, slideInterval)$. As Spark considers the batches as basic units of the input stream, the length and slide parameters of the time window must be multiples of the batch interval of the source stream. This can be an issue for some streaming applications. On the opposite, Flink offers very flexible window operations, as the elements of the input stream are considered separately: it is possible to use sliding, tumbling and landmark windows in streaming applications implemented on Flink with no restriction on the windows’ parameters. Flink offers the possibility to use count windows which are defined like time windows but are based on the count of received elements and not on window time length and slide interval. Furthermore, Flink also manages session windows: session windows “start at individual points in time and end once there has been a

---

29 The newest states of each TTW instance are the states corresponding to the biggest refresh time of each MPO

30 the window transformation is different from the windowAll transformation -that is actually used for implementing the REMO detectors on Flink- in that the first one applies on a partitioned stream while the second applies on a regular stream
certain period of inactivity” [6] characterized by the session gap. Session windows could be an alternative to TTWs if all the sensors report the MPO states approximately at the same time, creating thus time gaps within the input stream. Flink can also deal with out of order data by creating window based on the time stamp of each input element.

8.4 Extrapolation to real-world data sets and scenarios

The empirical evaluation of section 7 is based on the processing of a simulated stream of MPO states. Based on this evaluation, one can extrapolate the measurements to real-time performance on real data.

The evaluation of the geofence pattern detector shows that for both Flink and Spark solutions, it is possible to process up to 500 MPO states per second with a density of 0.0125 objects/m² based on 64 geofences and one CPU core. Indeed, the further increase of the input rate leads to such a higher disk activity that the alerts are not sent in real-time anymore due to a too big processing delay.

The experiments on simulated data of the track pattern detector show that Flink has a better real-time performance than Spark. Indeed, the increase of the average CPU usage and average disk activity induced by the increase of input data rate or constance parameter is a lot more important for Spark than for Flink. Extrapolated to real data, one expects Flink is expected to scale better in terms of constance parameter and input rate.

Concerning the flock pattern detector, the experiments on simulated data for both Flink and Spark show that with the update of 500 MPO states every 5 seconds and 100 semi-random generated disks (cf. Table 8), one cannot consider more than 400 grid cells in the filtering phase if one uses only one CPU core. However, as shown by the second experiment on the flock detector, the more the MPO distribution resembles a uniform distribution (more MPO centers), the more computer resources are needed. Those results can be extrapolated to real data: if the locations of the MPOs are distributed in clusters, 500 MPO states updated every 5 seconds can be processed using Spark and Flink solutions with 100 semi-random generated disks using maximum 400 grid cells. As a rule of thumb, one can expect the maximum number of grid cells to increase as the number of MPO centers decreases.

For the leadership pattern detector, the increase in average CPU usage and disk activity due to a bigger input rate is a lot more important for Spark than for Flink. Thus, in the case of one CPU core, one expects Flink solution to have a maximum real-time performance on real data characterized by a bigger input rate than for Spark solution.

In summary, if the results on simulated data (Section 7.2.4) are extrapolated to real data, one expects Flink solution to have a globally lower CPU usage and disk activity than Spark solution -for the same pattern detector and same input stream. One expects however the active memory to be equivalent for both solutions. These characteristics show that the range of parameters characterizing either the input stream that can be processed or the patterns that can be detected is bigger for Flink than for Spark solution.
9 Conclusion and Future Work

Conclusion  This research proposed a method for detecting in real-time four specific moving point patterns based on the analysis of a stream of MPO states and compared their implementation on three different stream processing platforms. The moving point patterns the research focused on are: (1) the geofence pattern consisting of an MPO entering/exiting specific zones, (2) the track pattern consisting of an MPO following the same direction for a given number of time steps and satisfying a given spatial constraint, (3) the flock pattern consisting of a cluster of MPOs following the same direction at a certain time step, (4) the leadership pattern consisting of a track pattern with the associated MPO anticipating the direction of geographically close MPOs at the last considered time step. Track, flock and leadership patterns being based on the definition of time steps, their detection relies on the definition of TTWs. Three DSMSs were proposed for implementing the applications aiming at detecting those moving point patterns: (1) Esri ArcGIS GeoEvent Extension for Server, a workflow-based DSMS ingesting stream elements as soon as they are received, (2) Apache Spark Streaming, a MapReduce-based DSMS ingesting stream elements in batches, (3) Apache Flink, a hybrid DSMS considering each stream element separately but offering MapReduce operators. While the geofence detector was implemented on each DSMS, the other detectors were implemented only on Apache Spark and Apache Flink due to the cumbersomeness of implementing time windows on Esri ArcGIS GeoEvent Extension for Server. The experiments based on a simulated stream of MPO states of the implementations on Apache Spark and Apache Flink show that the detectors implemented on Apache Flink are more scalable and use less computer resources than the one implemented on Apache Spark.

Future work  All the experiments were based on an input stream characterized by a constant rate of MPO states. A possible future work could be to investigate how the detectors behave when there is a burst in the rate of MPO states of the input stream. Experiments could thus be performed using the detectors on an input stream characterized by an increasing or decreasing rate of MPOs’s states. The currently proposed implementations are based on the deployment of the DSMS on a single machine. However, the three DSMSs support a cluster deployment. Further research could be therefore done in order to analyze the behavior of each DSMS in the case of a cluster deployment. A proposed approach could be to perform the same experiments as the one proposed in this research and compare the computer resources and load of the different machines of the cluster. Another direction work could be to enhance the detectors. For example, the current flock detector sends two notifications even if the two detected flock patterns differs only by one MPO. An enhancement could be to let the user choose a minimum percentage of difference between the detected flock patterns detected for the same time step. One could also enhance the Refinement phase of the SRDisks algorithm used for the flock detection. Indeed, the proposed probabilistic approach has a counter intuitive behavior as denser flocks can only be achieved for a large number of semi-random generated disks (cf. Sec-
An enhancement could be to generate only disks that are normally distributed around the center of the MPO locations of the considered candidate area according to the standard deviation of those MPO locations. The leadership pattern detector could also be enhanced. Indeed, the current leadership pattern is based on the idea that an MPO following a track pattern anticipates the direction of MPOs that are within a certain distance from the leader MPO at the last considered time step. The spatial constraint is thus directly bounded to the location of the leader MPO. However, considering that follower MPOs could be influenced by the leader MPO only if they are at certain distance from it could be seen as too restrictive. Indeed, the leader MPO could be at the “boarder” of a cluster of MPOs at the last considered time step and still influence the direction of the cluster of MPOs. Thus, an enhancement could be to consider the leadership pattern as a union of the track and the flock patterns. The difficulty of its implementation resides in the synchronization of the detection of flock and track patterns.

10 References

References


11 Appendix

11.1 Detailed Results of the Evaluation of the SRDisks Algorithm

Table 10: Detection accuracy (%) of the SRDisks algorithm on candidate area $S_2$ with different maximum number of tests $n_{test}$ and constance parameter $n_{min}$

<table>
<thead>
<tr>
<th>$n_{min}$</th>
<th>$n_{test}$</th>
<th>1</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>74.51</td>
<td>95.63</td>
<td>99.4</td>
<td>99.97</td>
<td>99.81</td>
<td>99.76</td>
<td>99.76</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>32.57</td>
<td>89.68</td>
<td>95.64</td>
<td>99.07</td>
<td>99.21</td>
<td>99.47</td>
<td>99.66</td>
<td>99.7</td>
<td>99.86</td>
<td>99.93</td>
<td>99.94</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>17.72</td>
<td>80.36</td>
<td>89.08</td>
<td>93.73</td>
<td>96.97</td>
<td>97.06</td>
<td>97.45</td>
<td>97.89</td>
<td>99.19</td>
<td>99.69</td>
<td>99.94</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>4.85</td>
<td>27.95</td>
<td>53.8</td>
<td>67.33</td>
<td>72.54</td>
<td>82.21</td>
<td>84.28</td>
<td>85.18</td>
<td>87.58</td>
<td>88.49</td>
<td>89.94</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>2.63</td>
<td>19.44</td>
<td>36.05</td>
<td>49.67</td>
<td>54.52</td>
<td>62.33</td>
<td>66.7</td>
<td>66.7</td>
<td>72.76</td>
<td>73.55</td>
<td>77.02</td>
<td></td>
</tr>
</tbody>
</table>

Table 11: Detection accuracy (%) of the SRDisks algorithm on candidate area $S_4$ with different maximum number of tests $n_{test}$ and constance parameter $n_{min}$

<table>
<thead>
<tr>
<th>$n_{min}$</th>
<th>$n_{test}$</th>
<th>1</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>18.81</td>
<td>74.97</td>
<td>91.86</td>
<td>95.72</td>
<td>96.09</td>
<td>97.78</td>
<td>97.78</td>
<td>98.41</td>
<td>98.64</td>
<td>99.33</td>
<td>99.89</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5.85</td>
<td>45.36</td>
<td>69.06</td>
<td>72.78</td>
<td>78.5</td>
<td>80.32</td>
<td>86.01</td>
<td>86.86</td>
<td>89.58</td>
<td>90.22</td>
<td>95.38</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2.54</td>
<td>20.42</td>
<td>32.15</td>
<td>43.22</td>
<td>47.85</td>
<td>55.85</td>
<td>62.4</td>
<td>64.3</td>
<td>69.43</td>
<td>69.85</td>
<td>76.17</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>0.13</td>
<td>1.28</td>
<td>2.59</td>
<td>3.76</td>
<td>4.78</td>
<td>5.71</td>
<td>7.52</td>
<td>10.06</td>
<td>10.24</td>
<td>11.91</td>
<td>13.54</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0.03</td>
<td>0.51</td>
<td>0.64</td>
<td>1.01</td>
<td>1.39</td>
<td>1.73</td>
<td>1.89</td>
<td>2.85</td>
<td>2.88</td>
<td>3.32</td>
<td>3.63</td>
<td></td>
</tr>
</tbody>
</table>
11.2 Detailed Results of the Evaluation of the Real-time Detection of Patterns

11.2.1 Geofence Detector

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Value of the associated parameter</th>
<th>System metric</th>
<th>Mean</th>
<th>Med.</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 geofences</td>
<td>CPU us. (%)</td>
<td>56.5</td>
<td>52.0</td>
<td>45.5</td>
<td>1.5</td>
<td>177.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>25.3</td>
<td>21.8</td>
<td>22.3</td>
<td>0.0</td>
<td>72.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td>810.7</td>
<td>817.6</td>
<td>74.7</td>
<td>697.0</td>
<td>945.2</td>
<td></td>
</tr>
<tr>
<td>16 geofences</td>
<td>CPU us. (%)</td>
<td>60.5</td>
<td>54.0</td>
<td>47.6</td>
<td>1.5</td>
<td>192.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>25.1</td>
<td>22.8</td>
<td>21.7</td>
<td>0.0</td>
<td>71.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td>857.2</td>
<td>855.8</td>
<td>74.1</td>
<td>724.5</td>
<td>968.4</td>
<td></td>
</tr>
<tr>
<td>64 geofences</td>
<td>CPU us. (%)</td>
<td>71.4</td>
<td>69.9</td>
<td>52.9</td>
<td>1.5</td>
<td>192.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>25.4</td>
<td>24.3</td>
<td>20.6</td>
<td>0.0</td>
<td>64.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td>885.5</td>
<td>898.0</td>
<td>68.3</td>
<td>762.6</td>
<td>978.9</td>
<td></td>
</tr>
<tr>
<td>256 geofences</td>
<td>CPU us. (%)</td>
<td>118.2</td>
<td>121.9</td>
<td>68.4</td>
<td>2.5</td>
<td>256.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>28.8</td>
<td>29.2</td>
<td>18.1</td>
<td>0.0</td>
<td>58.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td>951.2</td>
<td>961.6</td>
<td>18.2</td>
<td>905.3</td>
<td>974.8</td>
<td></td>
</tr>
<tr>
<td>100 MPOs</td>
<td>CPU us. (%)</td>
<td>71.4</td>
<td>69.9</td>
<td>52.9</td>
<td>1.5</td>
<td>192.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>25.4</td>
<td>24.3</td>
<td>20.6</td>
<td>0.0</td>
<td>64.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td>885.5</td>
<td>898.0</td>
<td>68.3</td>
<td>762.6</td>
<td>978.9</td>
<td></td>
</tr>
<tr>
<td>200 MPOs</td>
<td>CPU us. (%)</td>
<td>140.2</td>
<td>145.6</td>
<td>36.1</td>
<td>68.4</td>
<td>263.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>50.7</td>
<td>55.4</td>
<td>12.9</td>
<td>0.0</td>
<td>68.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td>1103.4</td>
<td>1123.7</td>
<td>34.6</td>
<td>1020.2</td>
<td>1148.6</td>
<td></td>
</tr>
<tr>
<td>500 MPOs</td>
<td>CPU us. (%)</td>
<td>150.6</td>
<td>144.4</td>
<td>22.1</td>
<td>110.0</td>
<td>256.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>62.9</td>
<td>63.9</td>
<td>4.2</td>
<td>50.1</td>
<td>70.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td>1128.4</td>
<td>1134.6</td>
<td>27.3</td>
<td>1062.8</td>
<td>1172.4</td>
<td></td>
</tr>
</tbody>
</table>

Table 12: Results of the experiments on Spark’s geofence detector
<table>
<thead>
<tr>
<th>Experiment</th>
<th>Value of the associated parameter</th>
<th>System metric</th>
<th>Mean</th>
<th>Med.</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effect of the number of geofences</td>
<td>4 geofences</td>
<td>CPU us. (%)</td>
<td>18.3</td>
<td>16.4</td>
<td>15.8</td>
<td>0.5</td>
<td>139.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disk act. (%)</td>
<td>25.0</td>
<td>23.3</td>
<td>21.7</td>
<td>0.0</td>
<td>68.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Act. mem. (MB)</td>
<td>1129.8</td>
<td>1136.3</td>
<td>12.1</td>
<td>1107.0</td>
<td>1152.9</td>
</tr>
<tr>
<td></td>
<td>16 geofences</td>
<td>CPU us. (%)</td>
<td>21.5</td>
<td>20.3</td>
<td>15.6</td>
<td>0.5</td>
<td>119.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disk act. (%)</td>
<td>25.8</td>
<td>23.8</td>
<td>20.8</td>
<td>0.0</td>
<td>65.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Act. mem. (MB)</td>
<td>1172.2</td>
<td>1175.2</td>
<td>7.3</td>
<td>1158.4</td>
<td>1185.2</td>
</tr>
<tr>
<td></td>
<td>64 geofences</td>
<td>CPU us. (%)</td>
<td>35.2</td>
<td>35.7</td>
<td>20.7</td>
<td>1.0</td>
<td>118.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disk act. (%)</td>
<td>27.0</td>
<td>28.3</td>
<td>16.6</td>
<td>0.0</td>
<td>53.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Act. mem. (MB)</td>
<td>1257.8</td>
<td>1258.7</td>
<td>7.0</td>
<td>1234.9</td>
<td>1269.8</td>
</tr>
<tr>
<td></td>
<td>256 geofences</td>
<td>CPU us. (%)</td>
<td>79.7</td>
<td>78.2</td>
<td>15.6</td>
<td>1.0</td>
<td>134.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disk act. (%)</td>
<td>23.7</td>
<td>24.3</td>
<td>4.1</td>
<td>0.0</td>
<td>28.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Act. mem. (MB)</td>
<td>1198.5</td>
<td>1197.0</td>
<td>8.8</td>
<td>1185.1</td>
<td>1217.1</td>
</tr>
<tr>
<td>Effect of the number of considered MPOs within each TTW instance</td>
<td>100 MPOs</td>
<td>CPU us. (%)</td>
<td>35.2</td>
<td>35.7</td>
<td>20.7</td>
<td>1.0</td>
<td>118.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disk act. (%)</td>
<td>27.0</td>
<td>28.3</td>
<td>16.6</td>
<td>0.0</td>
<td>53.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Act. mem. (MB)</td>
<td>1257.8</td>
<td>1258.7</td>
<td>7.0</td>
<td>1234.9</td>
<td>1269.8</td>
</tr>
<tr>
<td></td>
<td>200 MPOs</td>
<td>CPU us. (%)</td>
<td>61.1</td>
<td>54.0</td>
<td>18.4</td>
<td>1.5</td>
<td>141.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disk act. (%)</td>
<td>46.5</td>
<td>48.0</td>
<td>7.9</td>
<td>0.0</td>
<td>54.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Act. mem. (MB)</td>
<td>1280.6</td>
<td>1277.8</td>
<td>9.7</td>
<td>1261.0</td>
<td>1303.5</td>
</tr>
<tr>
<td></td>
<td>500 MPOs</td>
<td>CPU us. (%)</td>
<td>65.9</td>
<td>54.9</td>
<td>22.0</td>
<td>2.0</td>
<td>176.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disk act. (%)</td>
<td>48.1</td>
<td>48.1</td>
<td>4.6</td>
<td>0.0</td>
<td>55.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Act. mem. (MB)</td>
<td>1190.3</td>
<td>1169.2</td>
<td>37.5</td>
<td>1140.9</td>
<td>1253.4</td>
</tr>
</tbody>
</table>

Table 13: Results of the experiments on *Flink*’s geofence detector
### 11.2.2 Track Detector

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Value of the associated parameter</th>
<th>System metric</th>
<th>Mean</th>
<th>Med.</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>CPU us.(%)</td>
<td>26.9</td>
<td>12.4</td>
<td>28.1</td>
<td>1.0</td>
<td>127.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>1.5</td>
<td>0.5</td>
<td>1.9</td>
<td>0.0</td>
<td>7.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>751.0</td>
<td>752.9</td>
<td>74.1</td>
<td>631.3</td>
<td>870.3</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>CPU us.(%)</td>
<td>136.2</td>
<td>139.1</td>
<td>74.4</td>
<td>1.5</td>
<td>281.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>7.5</td>
<td>5.9</td>
<td>6.9</td>
<td>0.0</td>
<td>31.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>782.6</td>
<td>768.5</td>
<td>66.9</td>
<td>564.3</td>
<td>924.2</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>CPU us.(%)</td>
<td>258.8</td>
<td>292.2</td>
<td>69.0</td>
<td>3.0</td>
<td>333.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>3.8</td>
<td>0.5</td>
<td>6.4</td>
<td>0.0</td>
<td>48.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>1015.6</td>
<td>1031.1</td>
<td>104.5</td>
<td>743.6</td>
<td>1203.6</td>
<td></td>
</tr>
<tr>
<td>100 MPOs</td>
<td>CPU us.(%)</td>
<td>69.5</td>
<td>57.4</td>
<td>64.0</td>
<td>1.0</td>
<td>267.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>4.5</td>
<td>1.0</td>
<td>5.5</td>
<td>0.0</td>
<td>25.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>703.4</td>
<td>713.9</td>
<td>43.2</td>
<td>517.9</td>
<td>743.5</td>
<td></td>
</tr>
<tr>
<td>200 MPOs</td>
<td>CPU us.(%)</td>
<td>89.8</td>
<td>84.1</td>
<td>72.9</td>
<td>1.0</td>
<td>263.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>4.9</td>
<td>1.0</td>
<td>6.1</td>
<td>0.0</td>
<td>27.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>736.5</td>
<td>740.2</td>
<td>51.3</td>
<td>542.4</td>
<td>801.3</td>
<td></td>
</tr>
<tr>
<td>500 MPOs</td>
<td>CPU us.(%)</td>
<td>136.2</td>
<td>139.1</td>
<td>74.4</td>
<td>1.5</td>
<td>281.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>7.5</td>
<td>5.9</td>
<td>6.9</td>
<td>0.0</td>
<td>31.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>782.6</td>
<td>768.5</td>
<td>66.9</td>
<td>564.3</td>
<td>924.2</td>
<td></td>
</tr>
<tr>
<td>1000 MPOs</td>
<td>CPU us.(%)</td>
<td>175.6</td>
<td>191.4</td>
<td>55.2</td>
<td>4.9</td>
<td>283.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>10.3</td>
<td>8.9</td>
<td>8.1</td>
<td>0.0</td>
<td>35.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>958.8</td>
<td>965.6</td>
<td>67.0</td>
<td>780.3</td>
<td>1088.7</td>
<td></td>
</tr>
</tbody>
</table>

Table 14: Results of the experiments on Spark’s track detector
<table>
<thead>
<tr>
<th>Experiment</th>
<th>Value of the associated parameter</th>
<th>System metric</th>
<th>Mean</th>
<th>Med.</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>CPU us.(%)</td>
<td>6.4</td>
<td>5.0</td>
<td>8.5</td>
<td>2.0</td>
<td>104.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disk act.(%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Act. mem.(MB)</td>
<td>887.3</td>
<td>891.2</td>
<td>13.3</td>
<td>843.9</td>
<td>902.4</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>CPU us.(%)</td>
<td>7.2</td>
<td>4.0</td>
<td>7.6</td>
<td>2.0</td>
<td>110.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disk act.(%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Act. mem.(MB)</td>
<td>804.2</td>
<td>804.8</td>
<td>3.8</td>
<td>794.1</td>
<td>809.6</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>CPU us.(%)</td>
<td>10.7</td>
<td>4.9</td>
<td>12.7</td>
<td>2.5</td>
<td>116.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disk act.(%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Act. mem.(MB)</td>
<td>913.6</td>
<td>910.9</td>
<td>3.8</td>
<td>909.4</td>
<td>919.5</td>
</tr>
<tr>
<td>100 MPOs</td>
<td></td>
<td>CPU us.(%)</td>
<td>3.5</td>
<td>2.5</td>
<td>4.6</td>
<td>1.0</td>
<td>71.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disk act.(%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.2</td>
<td>0.0</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Act. mem.(MB)</td>
<td>807.5</td>
<td>807.3</td>
<td>3.1</td>
<td>802.3</td>
<td>812.6</td>
</tr>
<tr>
<td>200 MPOs</td>
<td></td>
<td>CPU us.(%)</td>
<td>4.6</td>
<td>2.5</td>
<td>9.4</td>
<td>1.0</td>
<td>120.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disk act.(%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Act. mem.(MB)</td>
<td>791.9</td>
<td>790.7</td>
<td>4.3</td>
<td>784.3</td>
<td>799.6</td>
</tr>
<tr>
<td>500 MPOs</td>
<td></td>
<td>CPU us.(%)</td>
<td>7.2</td>
<td>4.0</td>
<td>7.6</td>
<td>2.0</td>
<td>110.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disk act.(%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Act. mem.(MB)</td>
<td>804.2</td>
<td>804.8</td>
<td>3.8</td>
<td>794.1</td>
<td>809.6</td>
</tr>
<tr>
<td>1000 MPOs</td>
<td></td>
<td>CPU us.(%)</td>
<td>15.3</td>
<td>6.5</td>
<td>14.7</td>
<td>3.0</td>
<td>136.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Disk act.(%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.2</td>
<td>0.0</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Act. mem.(MB)</td>
<td>856.0</td>
<td>855.2</td>
<td>3.0</td>
<td>850.5</td>
<td>861.2</td>
</tr>
</tbody>
</table>

Table 15: Results of the experiments on Flink’s track detector
### Effect of the spatial constraint

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Value of the associated parameter</th>
<th>System metric</th>
<th>Mean</th>
<th>Med.</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>.0025</td>
<td>CPU us.(%)</td>
<td>204.7</td>
<td>201.4</td>
<td>69.0</td>
<td>100.5</td>
<td>306.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>0.1</td>
<td>0.0</td>
<td>0.2</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>901.0</td>
<td>900.3</td>
<td>21.8</td>
<td>854.6</td>
<td>933.4</td>
<td></td>
</tr>
<tr>
<td>.005</td>
<td>CPU us.(%)</td>
<td>81.5</td>
<td>61.9</td>
<td>72.4</td>
<td>1.5</td>
<td>254.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>0.1</td>
<td>0.0</td>
<td>0.2</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>883.5</td>
<td>885.2</td>
<td>18.8</td>
<td>845.6</td>
<td>909.5</td>
<td></td>
</tr>
<tr>
<td>.02</td>
<td>CPU us.(%)</td>
<td>23.0</td>
<td>7.9</td>
<td>24.5</td>
<td>1.0</td>
<td>142.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>0.2</td>
<td>0.0</td>
<td>0.3</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>763.1</td>
<td>754.4</td>
<td>50.0</td>
<td>705.2</td>
<td>867.6</td>
<td></td>
</tr>
<tr>
<td>.1</td>
<td>CPU us.(%)</td>
<td>20.9</td>
<td>7.9</td>
<td>22.8</td>
<td>1.0</td>
<td>122.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>0.2</td>
<td>0.0</td>
<td>0.3</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>737.3</td>
<td>739.0</td>
<td>40.8</td>
<td>676.6</td>
<td>816.0</td>
<td></td>
</tr>
</tbody>
</table>

### Effect of the distribution of the MPOs

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Value of the associated parameter</th>
<th>System metric</th>
<th>Mean</th>
<th>Med.</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MPO centers</td>
<td>CPU us.(%)</td>
<td>17.5</td>
<td>6.4</td>
<td>18.3</td>
<td>1.0</td>
<td>103.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>0.2</td>
<td>0.0</td>
<td>0.3</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>698.9</td>
<td>704.8</td>
<td>29.7</td>
<td>641.4</td>
<td>749.8</td>
<td></td>
</tr>
<tr>
<td>5 MPO centers</td>
<td>CPU us.(%)</td>
<td>23.0</td>
<td>7.9</td>
<td>24.5</td>
<td>1.0</td>
<td>142.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>0.2</td>
<td>0.0</td>
<td>0.3</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>763.1</td>
<td>754.4</td>
<td>49.9</td>
<td>705.2</td>
<td>867.6</td>
<td></td>
</tr>
<tr>
<td>24 MPO centers</td>
<td>CPU us.(%)</td>
<td>24.7</td>
<td>8.4</td>
<td>25.4</td>
<td>1.0</td>
<td>110.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>0.2</td>
<td>0.0</td>
<td>0.3</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>825.4</td>
<td>825.1</td>
<td>49.5</td>
<td>756.6</td>
<td>891.9</td>
<td></td>
</tr>
</tbody>
</table>

### Effect of the number of considered MPOs within each TTW instance

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Value of the associated parameter</th>
<th>System metric</th>
<th>Mean</th>
<th>Med.</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 MPOs</td>
<td>CPU us.(%)</td>
<td>18.6</td>
<td>4.4</td>
<td>20.1</td>
<td>1.0</td>
<td>111.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>0.2</td>
<td>0.0</td>
<td>0.3</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>716.0</td>
<td>707.1</td>
<td>25.8</td>
<td>682.2</td>
<td>771.4</td>
<td></td>
</tr>
<tr>
<td>200 MPOs</td>
<td>CPU us.(%)</td>
<td>18.7</td>
<td>4.9</td>
<td>20.7</td>
<td>0.5</td>
<td>128.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>0.2</td>
<td>0.0</td>
<td>0.3</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>763.2</td>
<td>757.2</td>
<td>25.8</td>
<td>727.9</td>
<td>822.1</td>
<td></td>
</tr>
<tr>
<td>500 MPOs</td>
<td>CPU us.(%)</td>
<td>23.0</td>
<td>7.9</td>
<td>24.5</td>
<td>1.0</td>
<td>142.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>0.2</td>
<td>0.0</td>
<td>0.3</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>763.1</td>
<td>754.4</td>
<td>50.0</td>
<td>705.2</td>
<td>867.6</td>
<td></td>
</tr>
<tr>
<td>1000 MPOs</td>
<td>CPU us.(%)</td>
<td>29.1</td>
<td>20.8</td>
<td>29.2</td>
<td>1.0</td>
<td>137.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>0.2</td>
<td>0.0</td>
<td>0.3</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>846.1</td>
<td>876.4</td>
<td>59.8</td>
<td>737.6</td>
<td>917.7</td>
<td></td>
</tr>
</tbody>
</table>

Table 16: Results of the experiments on Spark’s flock detector
<table>
<thead>
<tr>
<th>Experiment</th>
<th>Value of the associated parameter</th>
<th>System metric</th>
<th>Mean</th>
<th>Med.</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU us. (%)</td>
<td></td>
<td>90.9</td>
<td>102.3</td>
<td>24.6</td>
<td>1.0</td>
<td>136.5</td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.3</td>
<td>0.0</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td></td>
<td>1027.8</td>
<td>1028.4</td>
<td>26.1</td>
<td>982.8</td>
<td>1072.9</td>
</tr>
<tr>
<td>.0025</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Effect of the spatial constraint</td>
<td>CPU us. (%)</td>
<td></td>
<td>70.0</td>
<td>74.4</td>
<td>30.5</td>
<td>2.0</td>
<td>135.2</td>
</tr>
<tr>
<td>.005</td>
<td>Disk act. (%)</td>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td></td>
<td>873.3</td>
<td>871.0</td>
<td>7.8</td>
<td>865.0</td>
<td>896.5</td>
</tr>
<tr>
<td>.02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CPU us. (%)</td>
<td></td>
<td>11.4</td>
<td>3.4</td>
<td>12.9</td>
<td>0.5</td>
<td>98.1</td>
</tr>
<tr>
<td>.1</td>
<td>Disk act. (%)</td>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.2</td>
<td>0.0</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td></td>
<td>1125.0</td>
<td>1125.4</td>
<td>3.4</td>
<td>1117.8</td>
<td>1130.3</td>
</tr>
<tr>
<td>1 MPO centers</td>
<td>CPU us. (%)</td>
<td></td>
<td>5.7</td>
<td>2.0</td>
<td>6.9</td>
<td>0.5</td>
<td>66.0</td>
</tr>
<tr>
<td>5 MPO centers</td>
<td>Disk act. (%)</td>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td></td>
<td>1094.7</td>
<td>1094.9</td>
<td>1.0</td>
<td>1092.2</td>
<td>1096.1</td>
</tr>
<tr>
<td>24 MPO centers</td>
<td>CPU us. (%)</td>
<td></td>
<td>11.4</td>
<td>3.4</td>
<td>12.9</td>
<td>0.5</td>
<td>98.1</td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.2</td>
<td>0.0</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td></td>
<td>1125.0</td>
<td>1125.4</td>
<td>3.4</td>
<td>1117.8</td>
<td>1130.3</td>
</tr>
<tr>
<td>Effect of the number of considered MPOs within each TTW instance</td>
<td>CPU us. (%)</td>
<td></td>
<td>9.4</td>
<td>3.0</td>
<td>9.8</td>
<td>0.5</td>
<td>71.8</td>
</tr>
<tr>
<td>100 MPOs</td>
<td>Disk act. (%)</td>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.2</td>
<td>0.0</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td></td>
<td>1116.9</td>
<td>1116.3</td>
<td>1.6</td>
<td>1113.8</td>
<td>1119.2</td>
</tr>
<tr>
<td>200 MPOs</td>
<td>CPU us. (%)</td>
<td></td>
<td>8.1</td>
<td>2.9</td>
<td>8.5</td>
<td>0.5</td>
<td>62.4</td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.2</td>
<td>0.0</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td></td>
<td>1121.8</td>
<td>1121.9</td>
<td>2.8</td>
<td>1116.7</td>
<td>1126.6</td>
</tr>
<tr>
<td>500 MPOs</td>
<td>CPU us. (%)</td>
<td></td>
<td>11.4</td>
<td>3.4</td>
<td>12.9</td>
<td>0.5</td>
<td>98.1</td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td></td>
<td>1125.0</td>
<td>1125.4</td>
<td>3.4</td>
<td>1117.8</td>
<td>1130.3</td>
</tr>
<tr>
<td>1000 MPOs</td>
<td>CPU us. (%)</td>
<td></td>
<td>11.9</td>
<td>4.0</td>
<td>12.4</td>
<td>1.0</td>
<td>80.2</td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>Act. mem. (MB)</td>
<td></td>
<td>1107.4</td>
<td>1107.5</td>
<td>3.0</td>
<td>1101.0</td>
<td>1112.4</td>
</tr>
</tbody>
</table>

Table 17: Results of the experiments on Flink’s flock detector
### 11.2.4 Leadership Detector

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Value of the associated parameter</th>
<th>System metric</th>
<th>Mean</th>
<th>Med.</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>.025</td>
<td>CPU us.(%)</td>
<td>144.0</td>
<td>145.5</td>
<td>85.1</td>
<td>1.5</td>
<td>311.9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>5.0</td>
<td>1.5</td>
<td>6.9</td>
<td>0.0</td>
<td>36.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>832.5</td>
<td>842.0</td>
<td>70.5</td>
<td>633.6</td>
<td>957.6</td>
<td></td>
</tr>
<tr>
<td>.005</td>
<td>CPU us.(%)</td>
<td>144.6</td>
<td>150.2</td>
<td>84.2</td>
<td>1.5</td>
<td>349.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>5.9</td>
<td>2.0</td>
<td>7.4</td>
<td>0.0</td>
<td>28.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>840.2</td>
<td>848.1</td>
<td>72.5</td>
<td>618.1</td>
<td>997.0</td>
<td></td>
</tr>
<tr>
<td>.02</td>
<td>CPU us.(%)</td>
<td>150.1</td>
<td>167.5</td>
<td>82.0</td>
<td>1.5</td>
<td>324.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>6.2</td>
<td>2.5</td>
<td>7.9</td>
<td>0.0</td>
<td>53.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>825.7</td>
<td>826.8</td>
<td>72.2</td>
<td>634.3</td>
<td>991.5</td>
<td></td>
</tr>
<tr>
<td>.1</td>
<td>CPU us.(%)</td>
<td>148.5</td>
<td>159.8</td>
<td>81.7</td>
<td>1.5</td>
<td>299.1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>6.6</td>
<td>3.5</td>
<td>7.8</td>
<td>0.0</td>
<td>54.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>840.6</td>
<td>830.8</td>
<td>76.6</td>
<td>615.5</td>
<td>1021.8</td>
<td></td>
</tr>
</tbody>
</table>

#### Effect of the 2\textsuperscript{nd} spatial constraint

#### Effect of the number of considered MPOs within each TTW instance

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Value of the associated parameter</th>
<th>System metric</th>
<th>Mean</th>
<th>Med.</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 MPOs</td>
<td>CPU us.(%)</td>
<td>75.2</td>
<td>58.5</td>
<td>70.8</td>
<td>1.0</td>
<td>268.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>3.2</td>
<td>0.0</td>
<td>5.0</td>
<td>0.0</td>
<td>22.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>707.1</td>
<td>713.9</td>
<td>41.7</td>
<td>562.0</td>
<td>774.6</td>
<td></td>
</tr>
<tr>
<td>200 MPOs</td>
<td>CPU us.(%)</td>
<td>104.1</td>
<td>101.0</td>
<td>80.5</td>
<td>1.5</td>
<td>287.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>5.7</td>
<td>2.5</td>
<td>7.1</td>
<td>0.0</td>
<td>28.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>775.2</td>
<td>778.8</td>
<td>56.6</td>
<td>578.2</td>
<td>864.0</td>
<td></td>
</tr>
<tr>
<td>500 MPOs</td>
<td>CPU us.(%)</td>
<td>150.1</td>
<td>167.5</td>
<td>82.0</td>
<td>1.5</td>
<td>324.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>6.2</td>
<td>2.5</td>
<td>7.9</td>
<td>0.0</td>
<td>53.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>825.7</td>
<td>826.8</td>
<td>72.2</td>
<td>634.3</td>
<td>991.5</td>
<td></td>
</tr>
<tr>
<td>1000 MPOs</td>
<td>CPU us.(%)</td>
<td>188.5</td>
<td>207.1</td>
<td>70.2</td>
<td>2.0</td>
<td>337.7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act.(%)</td>
<td>6.3</td>
<td>3.5</td>
<td>6.7</td>
<td>0.0</td>
<td>25.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>852.6</td>
<td>864.5</td>
<td>71.1</td>
<td>674.5</td>
<td>976.2</td>
<td></td>
</tr>
</tbody>
</table>

Table 18: Results of the experiments on Spark’s *leadership* detector
## Table 19: Results of the experiments on Flink’s leadership detector

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Value of the associated parameter</th>
<th>System metric</th>
<th>Mean</th>
<th>Med.</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU us. (%)</td>
<td>7.4</td>
<td>4.9</td>
<td>8.3</td>
<td>2.5</td>
<td>109.4</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>894.2</td>
<td>893.5</td>
<td>3.8</td>
<td>887.1</td>
<td>900.5</td>
<td></td>
</tr>
<tr>
<td>.025</td>
<td>CPU us. (%)</td>
<td>7.4</td>
<td>4.9</td>
<td>9.4</td>
<td>2.0</td>
<td>134.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.2</td>
<td>0.0</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>820.2</td>
<td>819.3</td>
<td>2.1</td>
<td>816.1</td>
<td>824.3</td>
<td></td>
</tr>
<tr>
<td>.005</td>
<td>CPU us. (%)</td>
<td>8.3</td>
<td>5.0</td>
<td>9.9</td>
<td>1.5</td>
<td>105.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>825.6</td>
<td>827.9</td>
<td>5.4</td>
<td>813.4</td>
<td>832.2</td>
<td></td>
</tr>
<tr>
<td>.02</td>
<td>CPU us. (%)</td>
<td>8.3</td>
<td>4.5</td>
<td>10.7</td>
<td>2.0</td>
<td>116.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.2</td>
<td>0.0</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>814.9</td>
<td>815.4</td>
<td>5.5</td>
<td>799.6</td>
<td>822.2</td>
<td></td>
</tr>
<tr>
<td>.1</td>
<td>CPU us. (%)</td>
<td>5.7</td>
<td>3.0</td>
<td>12.0</td>
<td>1.0</td>
<td>111.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>784.1</td>
<td>787.1</td>
<td>7.7</td>
<td>763.4</td>
<td>798.6</td>
<td></td>
</tr>
<tr>
<td>100 MPOs</td>
<td>CPU us. (%)</td>
<td>7.4</td>
<td>4.9</td>
<td>8.3</td>
<td>2.5</td>
<td>109.4</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>894.2</td>
<td>893.5</td>
<td>3.8</td>
<td>887.1</td>
<td>900.5</td>
<td></td>
</tr>
<tr>
<td>200 MPOs</td>
<td>CPU us. (%)</td>
<td>5.0</td>
<td>3.9</td>
<td>8.2</td>
<td>1.5</td>
<td>117.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>790.1</td>
<td>790.9</td>
<td>5.2</td>
<td>780.2</td>
<td>798.5</td>
<td></td>
</tr>
<tr>
<td>500 MPOs</td>
<td>CPU us. (%)</td>
<td>8.3</td>
<td>5.0</td>
<td>9.9</td>
<td>1.5</td>
<td>105.2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>1.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>825.6</td>
<td>827.9</td>
<td>5.4</td>
<td>813.4</td>
<td>832.2</td>
<td></td>
</tr>
<tr>
<td>1000 MPOs</td>
<td>CPU us. (%)</td>
<td>15.3</td>
<td>7.4</td>
<td>15.7</td>
<td>2.5</td>
<td>184.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disk act. (%)</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Act. mem.(MB)</td>
<td>862.7</td>
<td>864.6</td>
<td>4.9</td>
<td>851.5</td>
<td>868.7</td>
<td></td>
</tr>
</tbody>
</table>

**Effect of the $2^{nd}$ spatial constraint**

**Effect of the number of considered MPOs within each TTW instance**
Reports in Geodesy and Geographic Information Technology

The TRITA-GIT Series - ISSN 1653-5227

