



ECOLE
POLYTECHNIQUE
DE BRUXELLES

1

Distributed Edge-to-Cloud Mobility Data Stream Processing

Thesis presented by Mariana MACHADO GARCEZ DUARTE

in fulfilment of the requirements of the PhD Degree in Engineering and
Technology ("Doctorat en Sciences de l'ingénieur et technologie")
Année académique 2025-2026

Supervisor : Professor Mahmoud SAKR

Co-supervisor : Professor Esteban ZIMANYI

Computer and Decision Engineering Department

Thesis jury :

Prof. Dr. Jan Tobias Mühlberg (Université libre de Bruxelles)

Prof. Dr. Mahmoud Sakr (Université libre de Bruxelles)

Prof. Dr. Esteban Zimányi (Université libre de Bruxelles)

Dr. Moaad Yacoubi (Bruxelles Mobilité (SPRB))

Prof. Dr. Eleni Tzirita Zacharatou (Hasso Plattner Institute (HPI), Universität Potsdam)

Abstract

The propagation of Internet-of-Things (IoT) sensors aboard moving objects has led to continuous spatiotemporal data streams that demand on-device and low-latency analysis. The underlying systems for processing this streaming data, however, are ill-prepared. On the one hand, standard stream processing engines lack support for spatiotemporal operations. On the other hand, existing libraries for spatiotemporal data are optimized for historical data rather than real-time processing.

Spatiotemporal streaming workloads are intrinsically more demanding than relational event processing. Spatial predicates require geometry-specific algorithms rather than scalar comparisons, while spatiotemporal reasoning adds temporal constraints to these computations. Moreover, trajectory construction requires ordering, buffering, and interpolation under tight latency budgets, and motion features (e.g., speed, heading, stops) require sub-trajectory processing, which increases CPU, memory, and synchronization costs.

To address these challenges, two systems were implemented: MobilityFlink and MobilityNebula. MobilityFlink, an extension of Flink for spatiotemporal stream processing in the cloud, served as the initial prototype to validate spatiotemporal stream operators within a well-established cloud engine. Building on these insights, MobilityNebula was designed by migrating to NebulaStream to leverage distributed edge-to-cloud execution and support continuous, low-latency processing in constrained, intermittently connected deployments. MobilityNebula is a mobility stream processing system designed for the edge-to-cloud continuum. We introduce a window-aware trajectory type system, a lift-combine-lower aggregation pattern for trajectory construction at stream rate, and a lightweight operator integration that enables Moving Objects Database (MOD) trajectory semantics to operate on constrained edge devices. MobilityNebula was evaluated by deploying the system on fog and edge devices and ingesting data from the Belgian Railway Operator/Société Nationale des Chemins de fer Belges (SNCB/NMBS) trains, executing real-time mobility queries such as brake system monitoring and high-risk zone proximity monitoring directly at the edge. In our evaluation, MobilityNebula sustained near real-time latencies for queries while ingesting a 20k events/s stream on an edge device such as a Raspberry Pi 5.

Additionally, we survey outlier detection as it is an essential step in data preprocessing that ensures the integrity and validity of data analyses. We focus on outlier points within individual trajectories, i.e., points that deviate significantly within a single trajectory. We experiment with ten open-source libraries to comprehensively evaluate available tools, comparing their efficiency and accuracy in identifying and cleaning outliers. The experiment considers the libraries as they are offered to end users, with real-world applicability. We compare existing outlier detection libraries and introduce a method for establishing ground-truth. Furthermore, we use our findings to implement Kalman filter into MobilityNebula.

Acknowledgments

I dedicate this thesis to Bisa Acy, my great-grandmother. Without you, I would not be writing these pages.

I also dedicate it to my grandparents, who inspired me and supported my decision to follow an academic path.

To my mother, who has always stood by me, no matter how different my ideas sounded. I treasure the countless conversations we shared about this thesis and about life during my journeys home from the lab.

I am deeply grateful to Rafael, whose patience and support are invaluable. Most of all, I thank him for his willingness to move so that I could pursue my dreams.

I also thank Dwi Nugroho for his help and support in implementing in earlier versions of NebulaStream.

Nevertheless, I owe special thanks to my advisors, Esteban Zimányi and Mahmoud Sakr, for their guidance, both academically and personally, and for always pushing me forward.

Contents

Abstract	i
Acknowledgments	iii
1 Introduction	1
1.1 Context and Motivation	2
1.2 Problem Statement	3
1.3 Objectives	4
1.4 Contributions	5
1.5 Thesis Structure	7
1.6 Disclaimer	7
2 Related Work	9
2.1 Data Stream Processing Systems	9
2.2 Systems for Managing Spatiotemporal Historical Data	11
2.3 Systems for Managing Spatiotemporal Streaming Data	12
2.4 Chapter Summary	14
3 Background	17
3.1 Data Stream Processing Systems	17
3.1.1 Data Stream	18
3.1.2 Concept of Time	18
3.1.3 Windows	18
3.1.4 Out-of-Order Data	20
3.1.5 Stream Query Languages	22
3.1.6 Directed Acyclic Graph Dataflow	24
3.1.7 Transactional Stream Processing	25
3.1.8 Just-in-Time Query Compilation	26
3.1.9 Consistency and Fault Recovery	28
3.1.10 Common Architectures	29
3.1.11 Stream Operators	31
3.2 Mobility Data	34
3.2.1 Spatiotemporal Data and Mobility Data	34
3.2.2 Mobility Continuous Queries	35
3.2.3 Outlier Detection	36
3.2.4 Lifted Operations	40
3.3 Chapter Summary	43

4	Distributed Edge-to-Cloud Mobility Data Stream Processing	45
4.1	Formal Model	46
4.1.1	Abstract Model	46
4.1.2	Discrete Model	46
4.2	Representative Systems	48
4.2.1	MobilityFlink	49
4.2.2	MobilityNebula	52
4.3	Spatiotemporal Stream Operations	54
4.4	Implementations	55
4.4.1	Implementation in MobilityFlink	55
4.4.2	Implementation in MobilityNebula	57
4.5	Real-Time Queries	60
4.5.1	Point-Based Queries	60
4.5.2	Trajectory-Based Queries	63
4.6	Performance Measurement	65
4.6.1	Distributed Setting	65
4.6.2	Single-Node Setting	68
4.6.3	Mobility Computation Breakdown	72
4.6.4	Geospatial vs. Non-Geospatial Operator Latency	73
4.6.5	Parameter Sensitivity	75
4.6.6	Network Volatility Stress Evaluations	77
4.7	Chapter Summary	77
5	Trajectory Processing Applications	81
5.1	SNCB Use Case	81
5.1.1	Context	82
5.1.2	Edge Devices	83
5.1.3	Implementation	84
5.1.4	Query Visualizations	85
5.1.5	Programmable Logic Controller (PLC) Logic Verification	85
5.1.6	Deployment Lessons Learned	88
5.2	Outlier Detection	90
5.2.1	Static Outlier Detection	90
5.2.2	Streaming Outlier Detection	93
5.3	Real-Time Outlier Detection	94
5.3.1	Theory and Implementation	94
5.3.2	Integration and Workflow	96
5.3.3	Outlier Detection in Historical Data and Real-Time Data	97
5.4	Experimental Evaluation	97
5.4.1	Constructing The Ground-Truth	97
5.4.2	Experimental Study	104
5.4.3	Interpreting Extended Kalman Filter (EKF) Results	112
5.5	Trajectory Functions	115
5.5.1	Implementation in Mobility Engine Open Source (MEOS)	116
5.5.2	Implementation in Python Binding for MEOS (PyMEOS)	118
5.5.3	Metric Extraction	124
5.5.4	Area of Interest Heatmap	124
5.5.5	Close Cars Identification	125
5.6	Chapter Summary	126

6	Conclusions and Future Work	127
6.1	Summary of Contributions	128
6.2	Future Work	129

List of Figures

1.1	Research gap addressed by this thesis, positioning MobilityFlink and Mobility-Nebula across stream processing, spatiotemporal analytics, and edge-to-cloud deployment	3
3.1	Turning-point insertion for a lifted distance computation when one argument is temporal, adding an intermediate point $q@t$	42
3.2	Two temporal points are synchronized by restricting them to a common time period and aligning instants for comparison	42
3.3	Turning points $p@t$ and $q@t$ are added after synchronization to capture the time of minimum distance between two moving segments	43
4.1	Sliding and tumbling windowing strategies for constructing <i>stinstant</i> and <i>stsequence</i> values from points	49
4.2	Overview of MobilityFlink, showing the integration of MEOS via Java Binding for MEOS (JMEOS) and the Kafka-Flink execution pipeline	50
4.3	Containerized MobilityFlink test setup, showing Kafka ingestion, Flink processing, and the mobility pipeline dataflow	51
4.4	High-level architecture of MobilityNebula, showing MEOS integration in Nebula-Stream and the query dataflow	53
4.5	Fog-edge deployment results for Queries 1–6, reporting throughput, events/s, selectivity, latency, CPU, memory, and network usage	67
4.6	Single-node edge-constrained evaluation across Queries 3 and 6–13, showing throughput, CPU usage, latency, and memory usage under varying resource caps	69
4.7	Single-node edge-constrained evaluation across Queries 3 and 6–13, showing latency/memory, CPU usage and success rate under varying resource caps	70
4.8	Throughput sensitivity to window size and slide interval across Queries 3 and 6–13 (sliding vs. tumbling variants)	76
4.9	Network-volatility stress-test results under controlled jitter (J), burst loss (B), outage (O), and combined scenarios, reporting per-query p50 end-to-end latency and sustained input throughput.	78
4.10	Network-volatility stress-test results under controlled jitter (J), burst loss (B), outage (O), and combined scenarios, reporting per-query average CPU utilization and average memory consumption.	79
5.1	GPS trajectories of six monitored SNCB trains over six months, as observed from onboard GPS readings	82
5.2	Previous SNCB diagnostic architecture, in which onboard measurements are transmitted to the cloud without edge-side processing	83

5.3	The SNCB M6 double-deck rolling stock used in the deployment and evaluation of MobilityNebula	84
5.4	The nROK 1030 onboard edge device used to execute MobilityNebula queries on trains	85
5.5	SNCB use-case architecture, showing onboard edge processing on the nROK device and MQTT-based alert delivery to the central monitoring backend	86
5.6	Dashboard visualizations of point-based streaming queries on SNCB data, including alerting, noise/speed monitoring, weather-based zones, and brake diagnostics . .	87
5.7	Examples of trajectory anomalies targeted by the ground-truth procedure, showing deviations in speed and/or bearing	98
5.8	Visualization of the AIS dataset before applying the ground-truth filtering procedure	100
5.9	Visualization of the AIS dataset after applying the ground-truth filtering procedure	100
5.10	Visualization of the OpenSky dataset before applying the ground-truth filtering procedure	101
5.11	Visualization of the Jakarta Grab-Posisi GPS dataset before applying the ground-truth filtering procedure	101
5.12	Visualization of the Singapore Grab-Posisi GPS dataset before applying the ground-truth filtering procedure	102
5.13	Sensitivity of AIS ground-truth to the speed and bearing thresholds (tS, tH) . .	104
5.14	Sensitivity of Jakarta ground-truth to different speed and bearing threshold configurations (tS, tH)	105
5.15	Runtime comparison of outlier-detection libraries across the Jakarta, Singapore, OpenSky, and AIS datasets	106
5.16	Aggregated runtime across all datasets for the evaluated Python libraries	107
5.17	Aggregated runtime across all datasets for the evaluated R libraries	107
5.18	Precision of each library measured against the ground-truth across the four datasets	108
5.19	Recall of each library measured against the ground-truth across the four datasets	109
5.20	F1 score of each library measured against the ground-truth across the four datasets	110
5.21	F0.5 score of each library measured against the ground-truth (precision-weighted)	111
5.22	F2 score of each library measured against the ground-truth (recall-weighted) . .	111
5.23	Comparison of removed point patterns on AIS trajectories for MovingPandas and the MEOS EKF	113
5.24	Jakarta example comparing points rejected by the MEOS EKF against the ground-truth outlier labels	114
5.25	Visualization of the raw STIB trajectories used in the qualitative removal-pattern comparison	115
5.26	Overlay of STIB trajectories with points removed by MovingPandas (OutlierCleaner and KalmanSmootherCV) and the MEOS EKF	116
5.27	Spatial distribution of points removed by the two MovingPandas stages (OutlierCleaner and KalmanSmootherCV) on STIB	117
5.28	Counts of removed points per MovingPandas stage on the STIB dataset	118
5.29	Comparison of removed point patterns on STIB for MovingPandas KalmanSmootherCV and the MEOS EKF	119
5.30	Example trajectories from the Singapore Grab-Posisi dataset used for offline analytics	120
5.31	Trajectories overlaid with spatiotemporal bounding boxes for Singapore Airport and Gardens by the Bay	120
5.32	Spatial density heatmap of trips between Singapore Airport and Gardens by the Bay	121
5.33	Filtering outcomes for trip selection, showing trajectories that intersect none, one, or both areas of interest (with zoomed views)	122

5.34	Single-leg trip geometries segmented between Singapore Airport and Gardens by the Bay using the geometry-based method	123
5.35	Speed and bearing profiles of geometry-based single-leg trip segments	123
5.36	Speed and bearing profiles of stop-based single-leg trip segments	124
5.37	Trajectory-density heatmap of trips between Singapore Airport and the Gardens by the Bay area	125
5.38	Example proximity event where two vehicles are detected within 10 m of each other	125

List of Tables

2.1	Comparison of general stream processing systems by real-time support, distributed execution, spatiotemporal support (points/trajectories), and deployment scope (edge, cloud, edge-to-cloud)	11
2.2	Comparison of spatiotemporal systems for historical data by distributed execution, support for point and trajectory types, and deployment scope (edge, cloud, edge-to-cloud)	13
2.3	Comparison of spatiotemporal stream processing systems by real-time support, distributed execution, point/trajectory support, and deployment scope (edge, cloud, edge-to-cloud)	14
3.1	Summary of standard window types, indicating whether they overlap, how their bounds are defined, and whether their boundaries are deterministic or data-dependent	19
3.2	Comparison of windowing semantics and implementation support in Flink and NebulaStream	20
3.3	Progress signals used for out-of-order handling, summarizing who emits them, their scope and semantics, and how they trigger window finalization	22
3.4	Comparison of how Flink and NebulaStream track progress and handle out-of-order events under event-time semantics	23
3.5	Comparison of how Flink and NebulaStream represent and execute queries as a Directed Acyclic Graph (DAG) dataflow	25
3.6	Comparison of transactional stream processing capabilities in Flink and NebulaStream	27
3.7	Comparison of Just-in-Time (JIT) compilation strategies in Flink and NebulaStream	28
3.8	Comparison of consistency guarantees and fault-recovery mechanisms in Flink and NebulaStream	30
3.9	Comparison of the deployment architecture assumptions and execution roles of Flink and NebulaStream	32
3.10	Comparison of how Flink and NebulaStream expose and execute core stream operators, including extensibility and placement	35
4.1	Abstract type constructors in the MOD framework, showing each constructor, its signature, and its type category	47
4.2	Discrete type system used in this thesis, including MobilityDB temporal types and the proposed window-aware spatiotemporal types	48
4.3	Example of lifting raw GPS tuples (longitude, latitude, timestamp) into Mobility-Nebula encoding (<i>instant</i>)	49
4.4	Mapping of representative MEOS operations to MobilityNebula bindings (wrapper, logical, physical, parser)	55

4.5	Query coverage across systems, indicating which platforms can express each query without changing its semantics or modifying the system’s core code	71
4.6	Computation breakdown per query under 512 MB–2 GB caps, separating time spent in the MobilityNebula wrapper/MEOS call from operator-level costs	73
4.7	Geospatial operator classes, classified by class, the per-function call cost as the percentage of time spent in the MEOS function compared to the total query processing time, as measured by the per-function profiling of Sect. 4.6.3. Primary overhead source indicates where the overhead is more prominent: inside the MEOS call or at the operator-level	74
4.8	Geospatial vs. non-geospatial operator cost at 8 GB/4 vCPU	75
5.1	Overview of static trajectory outlier-detection libraries, listing each library, its detection technique, method category, and implementation language	92
5.2	Overview of streaming outlier-detection libraries and systems, listing each method, its category, and implementation language	94
5.3	Summary of the evaluated datasets, including source, total number of points, number of detected outliers, and outlier percentage.	103
5.4	Single-leg trip counts produced by geometry-based and stop-based segmentation	124

List of Acronyms

***k*-NN** *k*-Nearest Neighbors. [4](#), [13](#), [35](#), [40](#), [65](#), [71–77](#), [93](#), [94](#), [129](#), [130](#)

ACID Atomicity, Consistency, Isolation, and Durability. [25–27](#)

ADT Abstract Data Type. [12](#)

AR Autoregression. [37](#)

ARIMA Autoregressive Integrated Moving Average. [37](#), [93](#)

BIRCH Balanced Iterative Reducing and Clustering using Hierarchies. [38](#)

BP Brake Pressure. [61](#), [63](#), [66](#), [82–85](#), [89](#), [128](#)

CLARANS Clustering Large Applications on Randomized Search. [38](#)

CQL Continuous Query Language. [22](#)

CTE Common Table Expressions. [85](#)

CV Constant-Velocity. [95](#)

DAG Directed Acyclic Graph. [xiii](#), [17](#), [24](#), [25](#), [43](#)

DBSCAN Density-Based Spatial Clustering of Applications with Noise. [38](#), [90](#), [91](#), [94](#), [97](#)

DSP Data Stream Processing. [17](#)

EILOF Efficient Incremental Local Outlier Factor. [40](#)

EKF Extended Kalman Filter. [vi](#), [36](#), [71](#), [81](#), [94–97](#), [103–105](#), [111–115](#), [126](#), [128](#), [129](#)

GAB Acceleration-Bounded. [91](#)

GCEP Geospatial Complex Event Processing. [5](#), [62](#), [129](#)

GiST Generalized Search Tree. [12](#)

GOD Group Outlier Detection. [38](#)

GSB Speed-Bounded. [91](#)

- HF** Hampel Filter. [37](#), [91](#)
- ILOF** Incremental Local Outlier Factor. [40](#)
- IOD** Individual Outlier Detection. [38](#)
- IoT** Internet-of-Things. [i](#), [1](#), [5](#), [6](#), [9](#), [10](#), [13](#), [18–20](#), [26](#), [30](#), [32](#), [35](#), [36](#), [43](#), [52](#), [62](#), [127](#), [128](#)
- JIT** Just-in-Time. [xiii](#), [7](#), [10](#), [17](#), [26–28](#), [35](#), [130](#)
- JMEOS** Java Binding for MEOS. [ix](#), [46](#), [49–52](#), [57](#), [127](#)
- KF** Kalman Filter. [36](#), [37](#), [90](#), [92](#), [112–115](#)
- LGSB** Local Greedy Speed-Bounded. [91](#)
- LOF** Local Outlier Factor. [38](#), [40](#), [90](#), [97](#)
- LRD** Local Reachability Density. [90](#)
- MA** Moving Average. [37](#)
- MedAD** Median Absolute Deviation. [37](#)
- MenAD** Mean Absolute Deviation. [113](#)
- MEOS** Mobility Engine Open Source. [vi](#), [ix](#), [x](#), [xiii](#), [xiv](#), [2](#), [4–7](#), [10](#), [12–15](#), [34](#), [40–43](#), [45](#), [46](#), [49–57](#), [59–61](#), [72–75](#), [77](#), [81](#), [94–97](#), [104](#), [105](#), [111–116](#), [119](#), [124–130](#)
- MOA** Massive Online Analysis. [93](#), [94](#)
- MOD** Moving Objects Database. [i](#), [6](#), [11](#), [37](#), [45–47](#), [52](#), [77](#), [128](#)
- NIS** Normalized Innovation Squared. [95](#), [97](#)
- PCA** Principal Component Analysis. [93](#)
- PF** Particle Filter. [36](#), [37](#)
- PLC** Programmable Logic Controller. [vi](#), [6](#), [7](#), [82](#), [83](#), [85](#), [86](#), [88](#), [89](#), [129](#)
- PyMEOS** Python Binding for MEOS. [vi](#), [12](#), [81](#), [115](#), [118](#), [121](#), [124](#), [126](#)
- PySAD** Python Streaming Anomaly Detection. [93](#), [94](#)
- R k -NN** Reverse k -Nearest Neighbors. [40](#)
- RDDs** Resilient Distributed Datasets. [13](#)
- SAMOA** Scalable Advanced Massive Online Analysis. [93](#), [94](#)
- SGAB** Smart Greedy Acceleration-Bounded. [91](#)
- SGSB** Smart Greedy Speed-Bounded. [91](#)

SNCB/NMBS Belgian Railway Operator/Société Nationale des Chemins de fer Belges. [i](#), [2](#), [5–7](#), [60](#), [68](#), [79](#), [81–86](#), [88](#), [89](#), [126–129](#), [132](#)

SP-GiST Space Partitioning Generalized Search Tree. [12](#)

SRA Stream Relational Algebra. [22](#)

TSP Transactional Stream Processing. [25–27](#)

UDFs User-Defined Functions. [2](#), [5](#), [35](#), [51](#)

WLOF Weighted Local Outlier Factor. [38](#)

Chapter 1

Introduction

The growing use of IoT sensors in moving objects, such as ships and trains, generates large amounts of streaming data that include spatiotemporal information. These data streams can support applications including real-time accident detection and maintenance prediction. Nevertheless, achieving efficient in situ spatiotemporal processing poses challenges due to high data rates, varying network conditions, and the limited CPU and memory capabilities of edge devices [78].

As moving objects are increasingly equipped with sensors, the demand for continuous spatial and temporal reasoning increases correspondingly, creating requirements that traditional streaming systems were not designed to meet. Beyond architectural limitations, spatiotemporal data introduce intrinsic computational challenges that make them significantly more demanding than relational data [90]. Spatial predicates require geometry-specific algorithms rather than scalar comparisons, and spatiotemporal predicates require the addition of the space dimension in these computations. Trajectory construction requires ordering, buffering, and interpolation under tight latency constraints. Motion features, such as speed, acceleration, heading, and stop detection, require sub-trajectory processing and stateful computation, intensifying CPU, memory, and synchronization costs. These properties fundamentally differentiate mobility analytics from the stateless or scalar workloads for which most streaming engines were initially designed.

General stream processing frameworks, such as Kafka Streams [64] and Flink [16], are optimized for high-throughput event streams in cloud environments. However, while these systems can be applied to IoT data pipelines, they are not tailored for resource-constrained edge deployments and intermittent connectivity. In addition, these systems often lack spatiotemporal functionality, requiring users to implement such operations. In contrast, existing spatiotemporal systems primarily target historical datasets and are not designed to cope with real-time workloads characterized by continuous streams, low-latency requirements, and fluctuating processing demands [90]. Recent research and system development have begun to address trajectory processing within streaming pipelines. However, this topic is still emerging. Integrating trajectory construction at a stream rate, continuous low-latency querying, and operations within the strict CPU and memory constraints of edge devices remains insufficiently addressed by current approaches.

To address these challenges, two complementary systems were developed: MobilityFlink¹ and MobilityNebula.² Rather than constructing a new stream processing engine from scratch, we follow a systems-integration approach, leveraging established platforms and extending them with mobility-aware abstractions and operations. This engineering decision enhances robustness and reproducibility, while focusing research efforts on the core technical challenge: reconciling the

¹<https://github.com/MobilityDB/MobilityFlink>

²<https://github.com/MobilityDB/MobilityNebula>

mismatch between streaming execution models and spatiotemporal trajectory semantics, and implementing window-aware types and operators that operate efficiently under edge constraints. MobilityFlink integrates Flink with MEOS operators to enable spatiotemporal stream processing in the cloud. Initially, MobilityFlink served as a prototype to validate spatiotemporal stream operators within a mature cloud engine. Building on these insights, MobilityNebula was designed to utilize NebulaStream for distributed edge-to-cloud execution and continuous low-latency processing in constrained and intermittently connected environments. MobilityNebula is an integrated system for spatiotemporal stream processing across the edge-to-cloud continuum. It is based on the distributed NebulaStream platform [117] and the MEOS library for spatiotemporal data processing [125], supporting both point-based and trajectory-based functions on constrained edge and fog devices. Extending spatiotemporal processing from a database context to an unbounded stream requires more than systems integration. Specifically, spatiotemporal types and operations must become stream-aware, associating these types with window bounds and enabling trajectories to be constructed and queried at stream rate. Furthermore, mobility functions originating in the database domain must be mapped and translated into streaming operators that compose with windowing, grouping, and incremental execution.

We have implemented our data model and executed our queries in MobilityFlink and MobilityNebula. MobilityFlink serves as a proof of concept, while MobilityNebula has been tested in a real-world setting for the SNCB/NMBS use case. We evaluated the use of MobilityFlink and conclude that it would be unsuitable, as we aim to reduce data transfer and latency while processing data on an edge device. In MobilityNebula, we perform queries directly on an edge device, maintaining low latency and enabling on-demand data gathering, even when connectivity is unstable. While generic engines can represent similar spatiotemporal predicates as User-Defined Functions (UDFs), MobilityNebula’s edge-to-cloud streaming architecture provides near real-time response times by eliminating the cloud round-trip. The system reduces bandwidth requirements by transmitting only windowed trajectories and alerts rather than raw measurements.

The distributed design of our system is a result of the inherent characteristics of mobility data. Information is produced simultaneously by multiple devices, including onboard sensors and monitoring stations, across a large geographic area. Processing these data streams on a single-node would create a computational bottleneck and would limit scalability. Instead, distributing the computation across multiple nodes allows queries to be executed near where the data is generated, while still enabling coordination across the network. This distributed setup reduces latency, balances the workload among heterogeneous devices, and enables the combination of local insights into a global view of system behavior.

The *edge-to-cloud* continuum reflects how computation can be dynamically placed along the architecture, from constrained edge devices to fog and cloud nodes, depending on device capabilities, network conditions, and data locality. For example, running a query directly on an edge node avoids transmitting raw data over the network and leverages local processing for real-time analysis. Conversely, when a broader context or historical correlation is required, computation can be offloaded to fog or cloud nodes. This flexibility provides a trade-off between latency, accuracy, and communication costs. It also enhances privacy by performing filtering and aggregation at the source, thereby sharing only derived or summarized data with the cloud.

1.1 Context and Motivation

Real-time spatiotemporal data has become central to modern mobility applications, including autonomous fleet coordination, railway maintenance monitoring, and maritime navigation. These workloads pose unique challenges as they generate large volumes of irregularly sampled obser-

vations, require immediate reaction to anomalous behavior, and are deployed on heterogeneous hardware ranging from embedded sensors to fog nodes and cloud servers.

On the one hand, traditional streaming systems focus on throughput and scalability but lack awareness of spatial and temporal semantics. On the other hand, spatiotemporal databases provide mobility operators but cannot sustain continuous low-latency ingestion. This contrast shows a significant need for systems that effectively integrate the responsiveness inherent in distributed streaming engines and the expressiveness offered by spatiotemporal models.

Processing mobility data at the network edge introduces additional constraints. Onboard devices must operate with limited CPU, memory, and energy budgets, while maintaining predictable latency even under unstable network conditions. Sending all raw measurements to the cloud is often impractical due to limited bandwidth and latency overheads. Instead, performing filtering, aggregation, and event detection near the data source ensures timeliness and resilience. This motivates the need for a mobility-aware distributed system capable of dynamically adjusting computation across edge, fog, and cloud layers based on device capabilities, network stability, and data locality.

Figure 1.1 illustrates the positioning of our proposed systems. MobilityFlink lies at the intersection of stream processing and spatiotemporal analytics in the cloud. MobilityNebula covers the intersection of low-latency stream processing and spatiotemporal data analytics across the edge-to-cloud continuum, addressing a gap not filled by existing systems.

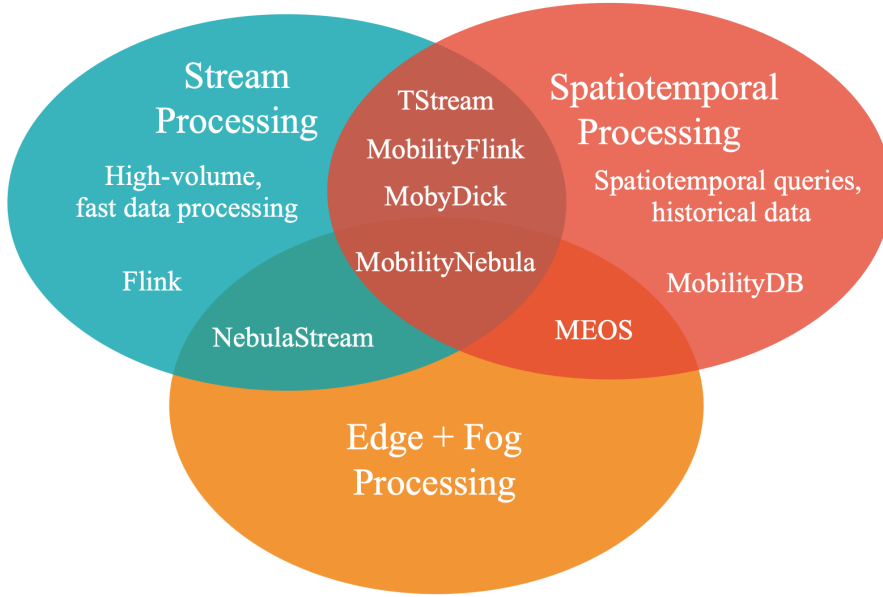


Figure 1.1: Research gap addressed by this thesis, positioning MobilityFlink and MobilityNebula across stream processing, spatiotemporal analytics, and edge-to-cloud deployment

1.2 Problem Statement

Despite ongoing advancements in streaming and spatiotemporal data management, no existing system satisfies the combined requirements of:

- **Expressive mobility analytics**, including geometric predicates, trajectory creation, and

movement-based reasoning.

- **Trajectory construction at stream rate**, building and analyzing window-bounded trajectories in real-time while preserving the semantics of discrete moving objects and their window bounds.
- **Low-latency mobility operators on constrained hardware**, implementing distance, containment, intersection, and k -Nearest Neighbors (k -NN) predicates over windowed trajectories as streaming operators that fit within the CPU and memory budgets of edge devices.
- **Reduced communication overhead**, by in-network filtering and aggregation rather than shipping raw data.
- **Data quality safeguards**, including online detection and treatment of noisy or faulty mobility measurements.

This thesis addresses the following problem:

How can we design data models, operators, and a distributed system that enable expressive, real-time spatiotemporal analytics for mobility data streams on heterogeneous hardware across the edge-to-cloud continuum?

This question drives the design and implementation of MobilityFlink and MobilityNebula, the development of window-aware spatiotemporal data types, and the integration of data-quality techniques necessary for reliable mobility processing. The main approach we take in this work is to integrate existing systems, that combined together converge to our goal and fulfills the design requirements. The main technical challenge of this work is the model mismatch between stream processing systems, e.g., NebulaStream, and spatiotemporal trajectory processing systems, e.g., MEOS.

1.3 Objectives

The goal of this thesis is to enable low-latency spatiotemporal processing and analysis for mobility data streams across the edge-to-cloud continuum. We build models and tools that support mobility processing on heterogeneous hardware in a distributed deployment. This objective is divided into the following research questions.

Research Questions

1. RQ 1 - Data Model

How can we define abstract and discrete window-aware spatiotemporal types that enable mobility operations in streaming environments? This question concerns the foundations of representing mobility data in a system that processes information incrementally. We require an abstract data model that captures the semantics of movement, such as points, trajectories, windows, and temporal bounds, and a discrete streaming data model that materializes these concepts as atomic units in a continuous pipeline. The challenge is twofold as abstract types must define behavior for spatiotemporal objects, while discrete types must encode these objects efficiently so they can be created and processed at stream runtime. This includes handling ordering, interpolating, windowing, and dealing with dynamic arrival patterns. The resulting model must therefore be expressive for complex mobility reasoning while being light to operate under edge and fog resource constraints.

2. RQ 2 - System Integration

How can general streaming engines be extended with mobility awareness so that spatiotemporal queries run end-to-end with low overhead? This question addresses the systems challenge of embedding spatiotemporal logic into engines built initially for relational or event-based processing. Mobility workloads require geometry operations, trajectory assembly, stateful windowing, and motion-based predicates that are fundamentally more complex than scalar filters or joins. We aim to incorporate these capabilities into existing architectures without sacrificing throughput or latency. This requires exploring operator design, dataflow schedule, state management, and language bindings for mobility-aware UDFs. Ultimately, we must show that a general-purpose streaming engine can support mobility semantics without restructuring its core.

3. RQ 3 - Expressiveness for Real Use Cases

Can real-world mobility operations, such as geofencing, Geospatial Complex Event Processing (GCEP), trajectory construction, stop detection, speed/bearing analytics, and railway diagnostics, be expressed as continuous queries and executed on constrained edge devices? This question concerns the applicability of the proposed models and systems to operational mobility scenarios. Real deployments require more than simple spatial predicates as operators must support dynamic windows, event correlation, temporal joins, sub-trajectory extraction, anomaly detection, and safety-critical logic. The challenge is to demonstrate that a complete pipeline from raw sensor readings to detection of mobility events can be expressed and executed continuously. This includes validating expressiveness for railway operations SNCB/NMBS, transport analytics, and generic IoT mobility applications. The question ensures that our framework is not only theoretically expressive but also practically usable.

4. RQ 4 - Data Quality and Outliers

Which trajectory outlier detection methods are most effective and practical for mobility datasets, and how can such methods be incorporated into real-time pipelines? Mobility data is notoriously noisy and inconsistent, with errors arising from GPS drift, sensor desynchronization, and network-induced fluctuation. This question investigates both the methodological challenge of evaluating trajectory cleaning techniques across heterogeneous datasets and the systems challenge of incorporating such methods into streaming environments. We assess the trade-offs of classical distance-based, clustering-based, and incremental anomaly detection methods. In addition, we quantify accuracy through a ground-truth, and suggest which methods are suitable for real-time deployments.

1.4 Contributions

This thesis provides the main contributions, listed below:

- We introduce the *stinstant* model, a discrete model for point streaming instances that encodes each data point and its window as a single atomic type for streaming analytics.
- We introduce the *stsequence* and *stsequenceset* models, discrete models for trajectory streaming sequences and sequence sets that encode each trajectory and its window bounds as a single atomic type for streaming analytics.
- We present MobilityFlink, a prototype that integrates MEOS operations and our discrete models (*stinstant*, *stsequence*, *stsequenceset*) into Flink to support mobility queries in the cloud.

- We design and implement MobilityNebula, a mobility stream processing system that supports Moving Objects Database (MOD) trajectory stream processing under tight CPU and memory budgets on edge devices and can perform real-time spatiotemporal analytics across distributed IoT nodes on edge, fog, and cloud. MobilityNebula provides a window-aware trajectory type system, a lift–combine–lower aggregation pattern for trajectory construction, and an operator integration that keeps MEOS temporal geometries and passes them by reference between NebulaStream operators while running on constrained edge nodes.
- We evaluate a real-world use case SNCB/NMBS, implementing geofencing and geospatial complex event queries on edge devices installed on trains. We translate and validate the Programmable Logic Controller (PLC) diagnostic logic into declarative streaming queries using MEOS and NebulaStream operators, and show functional equivalence between the PLC and SQL implementation and our trajectory-based streaming queries.

MobilityNebula research and development was conducted in close collaboration with the NebulaStream team at Technische Universität Berlin. Several resulting publications emerged from this partnership and therefore include co-authors from the joint work.

The following publications, listed chronologically, collectively form the foundation of this thesis:

1. M. M. G. Duarte and M. Sakr. “Outlier detection and cleaning in trajectories: A benchmark of existing tools.” In *Proceedings of the Workshops of the 26th International Conference on Extending Database Technology (EDBT ’23)*, 2023.
2. E. Zimányi, M. M. G. Duarte, and V. Diví. “MEOS: An open-source library for mobility data management.” In *Proceedings of the 27th International Conference on Extending Database Technology (EDBT ’24)*, 2024.
3. M. M. G. Duarte and M. Sakr. “An experimental study of existing tools for outlier detection and cleaning in trajectories.” *GeoInformatica*, 2025.
4. M. M. G. Duarte, D. Nugroho, G. Tod, E. Bevernage, P. Moelans, E. Tas, E. Zimányi, M. Sakr, S. Zeuch, and V. Markl. “Mobility stream processing on NebulaStream and MEOS.” In *Companion of the 2025 International Conference on Management of Data (SIGMOD/PODS ’25)*, 2025.
5. M. M. G. Duarte, D. P. A. Nugroho, G. Tod, E. Bevernage, P. Moelans, E. Saerens, E. Zimányi, M. Sakr, and S. Zeuch. “Mobility data stream processing beyond the cloud.” In *Proceedings of the 33rd International Conference on Advances in Geographic Information Systems (SIGSPATIAL ’25)*, 2025.
6. M. M. G. Duarte, D. P. A. Nugroho, G. Tod, E. Bevernage, P. Moelans, E. Saerens, E. Zimányi, M. Sakr, and S. Zeuch. “Mobility trajectory data stream processing beyond the cloud.” *Revision submitted to the 29th International Conference on Extending Database Technology (EDBT ’26)*.

In addition to the main papers, I have also co-authored the following publication, as part of the Horizon EU project MobiSpaces, and it is not included in this thesis.

1. C. Doukeridis, G. Santipantakis, N. Koutroumanis, G. Makridis, V. Koukos, G. Theodoropoulos, Y. Theodoridis, D. Kyriazis, P. Kranas, D. Burgos,

R. Jimenez-Peris, M. M. G. Duarte, M. Sakr, E. Zimányi, A. Graser, C. Heistracher, K. Torp, I. Chrysakis, T. Orphanoudakis, E. Kapassa, M. Touloupou, J. Neises, P. Petrou, S. Karagiorgou, R. Catelli, D. Messina, M. Compagnucci, and M. Falsetta. “MobiSpaces: An architecture for energy-efficient data spaces for mobility data.” In *Proceedings of the 2023 IEEE International Conference on Big Data (BigData '23)*, 2023.

1.5 Thesis Structure

The remainder of the thesis is organized as follows:

- **Chapter 2: Related Work**
This chapter surveys general stream processing systems, spatiotemporal systems, and spatiotemporal streaming systems.
- **Chapter 3: Background**
This chapter introduces fundamentals of data stream processing systems, query languages, direct acyclic graph execution, transactional stream processing, Just-in-Time (JIT) compilation, delivery guarantees and recovery, architectures, core operators, and the mobility foundations.
- **Chapter 4: Distributed Edge-to-Cloud Mobility Data Stream Processing**
This chapter formally describes the main contributions of this work - MobilityNebula system. It formalizes the abstract and discrete models with windowed temporal types. It presents MobilityFlink and MobilityNebula, defines the spatiotemporal operator set, and demonstrates implementations on real-time point and trajectory queries. Evaluates execution on edge setups and under edge-constrained resources.
- **Chapter 5: Trajectory Processing Applications**
This chapter explains the main use case used to validate the proposed system. It describes the SNCB/NMBS railway deployment devices and architecture on trains, query deployment at the edge, visual dashboards, and PLC validation. The chapter also presents our work before trajectory stream processing. We later used our findings to implement data cleaning in MobilityNebula. The chapter introduces a ground-truth construction method for validating trajectory quality and then evaluates ten state-of-the-art static trajectory outlier-detection libraries across four datasets. It then demonstrates how MEOS supports offline mobility analytics, including trajectory segmentation, metric extraction, heatmap generation, and proximity detection.
- **Chapter 6: Conclusions and Future Work**
This chapter summarizes contributions and findings, then outlines future work directions.

1.6 Disclaimer

We would like to acknowledge that this thesis is written by combining new material and using part or complete content from the papers listed in Sec. 1.4. We used standard writing-support tools such as Grammarly to improve readability.

Chapter 2

Related Work

Moving objects, such as trains, ships, and cars, are increasingly equipped with IoT sensors that generate continuous streams of mobility data. Processing these streams in real-time is valuable for applications such as maintenance monitoring, emergency event detection, and routing. However, processing mobility data streams in real-time on resource-constrained edge devices requires a combination of mobility semantics, continuous low-latency ingestion, and lightweight execution. These are requirements that no existing system jointly satisfies. General stream processing engines are optimized for scalar and relational workloads and lack support for mobility semantics, such as trajectory creation and spatial predicates. Spatiotemporal database systems implement expressive mobility operators, but are built for batch and historical workloads and cannot sustain continuous ingestion with low latency.

In this chapter, we survey prior work in continuous data processing and spatiotemporal data management, focusing on three domains: stream processing systems, spatiotemporal systems, and spatiotemporal stream processing systems. First, we examine general data stream processing frameworks, focusing on scalability, low-latency execution, and their suitability for emerging edge-to-cloud deployments. Then, we survey spatiotemporal systems and libraries designed to manage and analyze historical mobility data. Afterward, we review existing stream-based systems that incorporate spatiotemporal operators. Throughout the chapter, we emphasize the limitations of existing systems, specifically their lack of support for spatiotemporal analytics across resource-constrained edge-to-cloud deployments, to motivate our proposed approach.

2.1 Data Stream Processing Systems

The authors of the survey [32] review several stream processing systems that support continuous data handling. Early stream engines such as STREAM [37] and Aurora [1] introduced continuous SQL queries and sliding windows within database systems, enabling real-time filtering and aggregation. Building on the MapReduce paradigm [23], systems such as Apache Flink [16], Spark Streaming [115], Google Dataflow [65], and Storm¹ scaled-out stream processing to large clusters in the cloud. Apache Kafka Streams [64] decouples producers and consumers, creating durable, replayable queues that form the backbone of many modern pipelines, as exemplified by Uber’s real-time architecture [34].

In addition, other approaches include distributed databases such as RisingWave [106], LeanX-

¹<http://hortonworks.com/hadoop/storm/>

cale [52], Amazon Kinesis,² and Apache Samza.³

RisingWave [106] is a cloud streaming database that unifies real-time processing, SQL querying, and long-term storage. RisingWave decouples compute and storage with stateless compute nodes that perform incremental joins, aggregations, and windowed queries with exactly-once semantics. At the same time, states and results persist in object storage using the Apache Iceberg format.⁴ RisingWave unifies continuous SQL execution, elastic scaling, and durable state management, eliminating the need for separate stream engines with external storage.

LeanXcale [52] is a distributed, transactional SQL database and provides linear horizontal scalability across clusters. The database architecture combines a log-structured key-value storage layer with a SQL execution engine, enabling a combination of historical and streaming data. By decoupling storage and compute and using fine-grained sharding, LeanXcale scales performance as nodes are added.

Amazon Kinesis is an AWS service suite for ingesting and processing large-scale real-time data streams. Flink applications, implemented in Java or Scala, can be deployed via the Managed Service for Apache Flink, which automates resource provisioning, scaling, and maintenance.

Apache Samza is a stream processing framework designed for stateful applications. The framework integrates with messaging systems such as Apache Kafka, Amazon Kinesis, and HDFS, and supports deployment on YARN or as a standalone library. Samza has been employed at LinkedIn for production-scale, stateful stream processing [79].

More recently, the growth of IoT sensor deployments and the emergence of heterogeneous network architectures have motivated the development of NebulaStream [117],⁵ a distributed stream processing engine designed for the edge-to-cloud continuum. The system delivers performance superior to large-scale engines such as Kafka Streams [64] and Apache Flink [16], yet consumes far fewer resources, making it well-suited to devices with limited CPU or memory capacities [117]. NebulaStream can adapt to varying workloads [17], such as unexpected variations in the frequency of data measurement readings. The system minimizes the dependency on reliable or continuous network connections by processing data at the edge. This strategy reduces latency since events do not have to be transmitted to a cloud environment for processing. At the same time, the system can scale the execution across multiple nodes in the cloud, fog, and edge when available. NebulaStream uses Nautilus, a hybrid query execution engine that combines interpreted and compiled execution, employing a trace-based Just-in-Time (JIT) compiler [40]. NebulaStream execution engine features a versatile plugin architecture that allows for the addition of functionalities, facilitating the incorporation of third-party libraries [40]. The plugin system loads and registers specialized components dynamically via CMake. The system includes an expression framework that enables users to create custom operators and functions using inheritance and composition. This framework also allows for defining operators at runtime through dynamic registration, facilitating the incorporation of domain-specific operator logic, such as invoking MEOS functions [40, 24].

Table 2.1 summarizes systems by four capabilities: (1) real-time processing, (2) distributed execution, (3) spatiotemporal types (points and trajectories), and (4) deployment scope (edge, cloud, edge-to-cloud continuum). Cloud stream engines (STREAM, Aurora, Flink, Spark Streaming, Storm, Kafka Streams, RisingWave, LeanXcale, Kinesis, Samza) concentrate on the first two columns: they provide real-time, distributed processing in the cloud, but none of the surveyed systems integrates spatiotemporal support with edge-to-cloud deployment. In addition, among surveyed systems, NebulaStream is the only entry that marks edge, cloud, and

²<https://aws.amazon.com/kinesis/>

³<https://samza.apache.org>

⁴<https://iceberg.apache.org>

⁵<https://nebula.stream>

Table 2.1: Comparison of general stream processing systems by real-time support, distributed execution, spatiotemporal support (points/trajectories), and deployment scope (edge, cloud, edge-to-cloud)

Name	Real-time	Distrib.	Spatiotemporal		Deployment		
			Point	Traj.	Edge	Cloud	Edge-to-Cloud
STREAM	✓	–	–	–	–	✓	–
Aurora	✓	–	–	–	–	✓	–
Flink	✓	✓	–	–	–	✓	–
Spark Streaming	✓	✓	–	–	–	✓	–
Storm	✓	✓	–	–	–	✓	–
Kafka Streams	✓	✓	–	–	–	✓	–
RisingWave	✓	✓	–	–	–	✓	–
LeanXcale	✓	✓	–	–	–	✓	–
Kinesis	✓	✓	–	–	–	✓	–
Samza	✓	✓	–	–	–	✓	–
NebulaStream	✓	✓	–	–	✓	✓	✓

the edge-to-cloud continuum, but it does not implement spatiotemporal modeling.

2.2 Systems for Managing Spatiotemporal Historical Data

Database extensions such as SECONDO [41] and Parallel SECONDO [70] introduce moving object support for point and trajectory data representation into a generic Database Management System, offloading parallel execution to Berkeley DB [82]⁶ and Hadoop [107].⁷ SECONDO utilizes an abstract model that captures movement semantics and implements the discrete model as a *sliced representation* from the Moving Objects Database (MOD) framework defined by the authors in [43].

HadoopTrajectory [7] enhances Hadoop with a global grid and 3D-R tree indexing to prune spatiotemporal data before distribution, outperforming Distributed SECONDO on spatial filters.

ST-Hadoop [5] builds on SpatialHadoop [26] to offer spatiotemporal queries across a cluster. The system augments the grid-and-tree indexes with time support so that new data can be appended without repartitioning existing blocks. ST-Hadoop’s data model adds a temporal attribute to each **point** and introduces an **interval** types to represent time ranges. ST-Hadoop enables distributed range and join queries that consider both space and time.

Hermes [87] integrates trajectory support into Oracle 10g by implementing an object-relational database extension with moving-object types and functions. Hermes defines new PL/SQL types, such as those for moving points and moving polygons, alongside methods to compute entry and exit points, traveled distance, and trajectory similarity. Built in C++ on top of the Oracle Spatial option and the TAU Temporal Literal Library, Hermes exposes these capabilities through Oracle’s extensibility API, allowing users to issue SQL queries over moving objects.

TrajSpark [121] reduces repartitioning overhead by combining dynamic spatial partitions with a time-decay scoring mechanism. The system divides the area into a grid, computes new point

⁶<https://www.oracle.com/database/technologies/related/berkeleydb.html>

⁷<https://hadoop.apache.org>

counts per cell, and groups cells into equally loaded partitions. The system applies a decay factor to past counts, incorporates counts from the current batch, and computes the Jensen-Shannon distance [80] between the updated and previous distributions. When this distance exceeds a preset threshold, TrajSpark rebuilds its partitions using the latest decayed counts.

MobilityDB [90]⁸ is an open-source extension of PostgreSQL and PostGIS designed to manage spatiotemporal data efficiently. The system extends PostgreSQL and PostGIS type system with Abstract Data Type (ADT) to represent moving objects and their trajectories. MobilityDB implements a wide range of temporal and spatiotemporal types, such as `tgeometry` for moving geometries, `tgeography` for moving geographies, and other temporal types such as `tint`, `tfloat`, `tbool`, and `ttext`, which represent temporal integers, floats, booleans, and text, respectively. These temporal types are constructed from pairs of base types and time types, allowing for the continuous mapping of values over time [127, 8]. The core functionality of MobilityDB includes the ability to store and query moving object trajectories through SQL efficiently. MobilityDB supports spatiotemporal indexing by extending the Generalized Search Tree (GiST) and Space Partitioning Generalized Search Tree (SP-GiST) index access methods, as MGIST and MSP-GiST [93]. The sequence representation model used by MobilityDB enables linear interpolation between consecutive timestamps, providing a way to handle continuous data representations [9, 126, 127]. Furthermore, MobilityDB supports distributed query processing using Citus,⁹ a PostgreSQL extension for horizontal scalability. Citus enables MobilityDB to handle large-scale data sets by distributing data and queries across multiple nodes [9, 8].

MEOS¹⁰ is a C library designed to manipulate and manage temporal and spatiotemporal datasets, focusing on mobility data. MEOS can run on both edge and cloud environments, enabling spatiotemporal processing in resource-constrained applications as well as in cloud backends. MEOS architecture separates storage from processing, enhancing flexibility when managing data. MEOS can be integrated into various programming environments and languages, including Python and SQL, with the PyMEOS library¹¹ and MobilityDB. PyMEOS enhances functionalities by enabling data visualization and integrating data frames, which support advanced analytics and visualization of mobility data. MEOS was demonstrated using real-world datasets, including Singapore’s Grab-Posisi GPS trajectory dataset [48]. This demonstration exemplified MEOS capabilities in trajectory manipulation, bounding box creation, and trip metric extraction, including distance, speed, and bearing calculations, as described in [125].

Table 2.2 compares systems by four capabilities: (1) real-time processing, (2) distributed execution, (3) spatiotemporal types (points and trajectories), and (4) deployment scope (edge, cloud, edge-to-cloud continuum). Spatiotemporal databases and Hadoop extensions (SECONDO/Parallel SECONDO, HadoopTrajectory, ST-Hadoop, Hermes, TrajSpark, MobilityDB) support point and trajectory processing, but they are not built for low-latency streaming and target cloud deployments. MEOS implements spatiotemporal types that can be deployed either at the edge or in the cloud, yet it is neither a distributed system nor a streaming engine. None of the surveyed spatiotemporal systems natively supports continuous stream processing over unbounded streams.

2.3 Systems for Managing Spatiotemporal Streaming Data

Uber’s real-time analytics stack [34] federates Kafka [64], Flink [16], Pinot [50], and Presto [95] to ingest readings and produce geo-aware dashboards, but it lacks trajectory types and operators.

Sedona [113] extends Apache Spark with distributed spatial data types and operators, adding

⁸<https://mobilitydb.com>

⁹<https://www.citusdata.com/>

¹⁰<https://libmeos.org>

¹¹<https://github.com/MobilityDB/PyMEOS>

Table 2.2: Comparison of spatiotemporal systems for historical data by distributed execution, support for point and trajectory types, and deployment scope (edge, cloud, edge-to-cloud)

Name	Real-time	Distrib.	Spatiotemporal		Deployment		
			Point	Traj.	Edge	Cloud	Edge-to-Cloud
SECONDO	–	✓	✓	✓	–	✓	–
HadoopTrajectory	–	✓	✓	✓	–	✓	–
ST-Hadoop	–	✓	✓	✓	–	✓	–
Hermes	–	–	✓	✓	–	✓	–
TrajSpark	–	✓	✓	✓	–	✓	–
MobilityDB	–	✓	✓	✓	–	✓	–
MEOS	–	–	✓	✓	✓	✓	–

Spatial Resilient Distributed Datasets (RDDs), spatial indexing, and spatial joins to Spark SQL for batch and micro-batch workloads. Sedona supports common geometry types (e.g., points, polygons, and line strings) and enables trajectory analytics by representing movement paths as line-string geometries and applying spatial predicates and joins over them.

GeoFlink [97] adds spatial types and a grid index to Flink’s streaming engine, enabling continuous range queries, k -Nearest Neighbors (k -NN) searches, and spatial joins over static geometries. TStream [96] builds on GeoFlink by introducing window-bounded grid indexes and trajectory operators. This extension allows TStream to handle real-time path queries over moving objects, complementing GeoFlink’s support for points, polygons, and line strings with windowed trajectory processing.

Tornado [71] layers an augmented grid and FAST keyword index on Storm¹² to process spatial-keyword streams with adaptive load balancing.

GeoEkuiper [47] enriches a lightweight IoT engine with spatial SQL, supporting range, join, k -NN, and predicate queries at the edge.

MobyDick [36] is a Flink extension that adds support for spatiotemporal data. The extension defines types such as `mobilePoint`, `temporalPoint`, and `temporalPolygon` as Scala classes on the DataStream API. Each object uses a *sliced representation* model, where linear motion segments cover discrete time intervals. The library provides operators for spatial distance, nearest-approach detection, and trajectory queries, such as total distance traveled. It runs on event-time semantics to handle out-of-order events correctly. MobilityFlink draws on the MobyDick type system but replaces its sliced based view with MEOS *sequence representation* model, where objects appear as ordered instants or intervals with optional interpolation semantics.

Table 2.3 relates spatiotemporal streaming systems along four dimensions: real-time processing, distributed execution, support for point and trajectory types, and deployment scope (edge, cloud, edge-to-cloud continuum). The table reveals several trends. Spatial libraries that extend stream engines (Sedona, GeoFlink, TStream, Tornado) add streaming spatial queries and, in some cases, trajectory support, but they do not provide a full mobility-aware spatiotemporal type system with temporal trajectory semantics (e.g., moving-object temporal types) across the edge-to-cloud continuum. GeoEkuiper brings spatial processing to the edge but lacks a distributed cloud backend. MobyDick and MobilityFlink demonstrate that spatiotemporal types and operators can be implemented in a streaming system, yet their deployments are cloud-centric.

The table highlights a gap among surveyed systems. These systems implement either streaming

¹²<http://hortonworks.com/hadoop/storm/>

Table 2.3: Comparison of spatiotemporal stream processing systems by real-time support, distributed execution, point/trajectory support, and deployment scope (edge, cloud, edge-to-cloud)

Name	Real-time	Distrib.	Spatiotemporal		Deployment		
			Point	Traj.	Edge	Cloud	Edge-to-Cloud
Uber	✓	✓	–	–	–	✓	–
Sedona	✓	✓	✓	✓	–	✓	–
GeoFlink	✓	✓	✓	–	–	✓	–
TStream	✓	✓	✓	✓	–	✓	–
Tornado	✓	✓	✓	–	–	✓	–
GeoEKuiper	✓	–	✓	–	✓	–	–
MobyDick	✓	✓	✓	✓	–	✓	–
MobilityFlink	✓	✓	✓	✓	–	✓	–
MobilityNebula	✓	✓	✓	✓	✓	✓	✓

scale in the cloud or spatiotemporal support, but not both across the edge-to-cloud continuum. MobilityNebula is positioned to fill that gap by combining low-latency streaming with point and trajectory types and explicit support for edge-to-cloud execution.

2.4 Chapter Summary

In this chapter, we have surveyed three categories of systems that are relevant to continuous mobility analytics: general-purpose stream processing engines, spatiotemporal database systems and libraries, and spatiotemporal streaming systems. Cloud-centric stream processors such as STREAM, Aurora, Flink, Spark Streaming, Storm, Kafka Streams, RisingWave, LeanXcale, Kinesis, and Samza provide scalable, low-latency dataflow execution and fault tolerance except for NebulaStream, which explicitly targets heterogeneous edge-to-cloud deployments. However, these engines do not support trajectory types and operators.

Spatiotemporal systems such as SECONDO and Parallel SECONDO, HadoopTrajectory, ST-Hadoop, Hermes, TrajSpark, MobilityDB, and MEOS enrich database kernels and Hadoop-based stacks with moving-object models, trajectory representations, and spatiotemporal indexing. These systems excel at offline analysis of historical mobility datasets. Nevertheless, they are not designed to support data streams.

Spatiotemporal streaming systems such as Sedona, GeoFlink, TStream, Tornado, GeoEKuiper, MobyDick, and MobilityFlink demonstrate that spatial and, in some cases, trajectory operations can be embedded into streaming frameworks. However, existing solutions either focus on spatial point processing without full trajectory semantics, restrict execution to the cloud, or omit explicit support for edge-to-cloud deployments. GeoEKuiper operates at the edge but lacks a scalable distributed backend, whereas cloud-oriented systems, including TStream, assume more powerful infrastructure and do not address strict edge resource budgets.

Spatiotemporal trajectory processing in streaming engines remains an emerging research area, and current systems do not yet meet essential deployment requirements, such as edge execution, stream-rate trajectory construction, and continuous low-latency querying under strict CPU and memory constraints. Collectively, the tables demonstrate a significant gap: no current system provides (1) real-time spatiotemporal processing with trajectory types and mobility operators, (2) distributed execution, and (3) deployability across the edge-to-cloud continuum under stringent

CPU, memory, and network constraints. This thesis addresses this gap by integrating MEOS-based spatiotemporal types and operators into NebulaStream. Specifically, Chapter 4 introduces MobilityNebula, formalizes its data models and operator sets, and evaluates its ability to enable distributed, low-latency mobility stream processing on resource-constrained edge devices and fog nodes, while leveraging existing NebulaStream support for cloud deployment.

Chapter 3

Background

Designing a system for continuous spatiotemporal processing requires foundations from two traditionally separate areas: stream processing and mobility data management. Stream processing engines contribute to the execution model, including unbounded dataflows, windowing, event-time semantics, and fault recovery. In parallel, mobility systems contribute to the data model, with temporal types, trajectory representations, spatial predicates, and lifted operations.

We begin with Data Stream Processing (DSP) systems and abstractions, including streams, time semantics, windows, and out-of-order handling. We further explore query languages for streams, SQL dialects, and operator APIs. We explain Directed Acyclic Graph (DAG) execution models, transactional stream processing, and Just-in-Time (JIT) compilation techniques that support both consistency and performance in modern systems. Because our systems, MobilityFlink and MobilityNebula, are built on top of Flink and NebulaStream, respectively, we devote particular attention to these two engines. For each concept, such as windows, progress tracking, DAG dataflow, transactional guarantees, JIT compilation, consistency and fault recovery, architectures, and core operators, we describe how Flink and NebulaStream realize it and summarize their capabilities and differences in comparative tables. This comparative background clarifies the space in which our contributions operate and explains why MobilityFlink targets cloud-centric deployments while MobilityNebula focuses on the edge-to-cloud continuum. In addition, we survey the mechanisms for consistency and fault recovery. We introduce Lambda and Kappa architectures, the edge-to-cloud continuum, and stream operators. Furthermore, we transition to spatiotemporal systems and discuss spatiotemporal and mobility data. We survey outlier detection for this data type and explain lifted operations.

3.1 Data Stream Processing Systems

DSP systems, such as Flink [16], run continuous queries over unbounded data streams. They can accept out-of-order events and track progress in real-time. These systems define windows to limit the scope of computation. Additionally, DSP systems support stream query languages, enabling the execution of continuous queries. They ensure consistency and can recover state after failures through the use of checkpoints and transactional stream processing. Furthermore, DSP systems balance the load across workers and adapt the workload dynamically at runtime. Their set of operators includes filters, maps, joins, aggregations, and pattern detection for event processing.

3.1.1 Data Stream

A *data stream* is an unbounded flow of data elements produced by external sources such as IoT sensors and GPS devices [35, 20, 4]. Unlike static datasets that are fully available before querying, stream data arrives incrementally and requires immediate processing upon arrival.

3.1.2 Concept of Time

In data streams, there are two intrinsic temporal domains to consider: *event-time* and *processing-time* [35]. *Event-time* represents the timestamp at which the data was generated in the real world. *Processing-time*, in contrast, refers to the timestamp of the system clock at the moment when an element is processed.

3.1.3 Windows

Windows are abstractions used to process unbounded data by limiting the data processed. Several methods exist to define and manage these windows. The frequency and manner in which windows are updated and elements discarded are handled by window types and eviction policies [105]. Windows can be categorized by size (fixed or variable), overlap (overlapping or non-overlapping), and measurement criteria (element-count or time-based). Additionally, windows may be deterministic or non-deterministic, and can also be classified by context awareness into context-free, forward-context-free, and forward-context-aware types [105, 35]. Table 3.1 relates window types, if the window can overlap, how the window's bounds are defined, and whether boundaries are deterministic or data-dependent.

A *time-based window* groups stream elements whose timestamps fall within a fixed-duration interval. The window *size* represents a maximum period, for example, five minutes, after which the window advances according to its eviction policy.

A *tuple-based window* includes a fixed number of elements, independent of their timestamps. The window *size* is specified by element count, for example, 100 elements. Once this count is reached, the window finalizes according to its eviction policy.

A *sliding window* is defined by two parameters: its fixed *size* and a *slide* interval. With each slide interval, a new window of equal size is emitted, causing overlapping intervals that share common elements. Sliding windows are also *context-free*, but their overlap allows continuous temporal analysis.

A *tumbling window* is a special case of sliding windows, where *slide* is equal to *size*. These windows produce consecutive intervals of equal size without overlaps. Each window interval starts immediately after the previous one ends, resulting in distinct, independent batches. Thus, tumbling windows are *context-free*.

A *session window* groups elements into intervals of activity separated by periods of inactivity. Unlike tumbling or sliding windows, session windows neither overlap nor have fixed boundaries. A session ends after a specified inactivity period, known as the duration of a *gap*.

A *snapshot window* is a non-overlapping, time-based window with dynamically varying *duration* based on a defined *interval*. For example, each incoming element can have a field called *window length* that specifies the duration of the current interval. We can define a snapshot window whose duration is directly derived from this attribute. Snapshot windows are *forward-context-free*, meaning each snapshot depends exclusively on data within the current interval, independently of previous states.

A *threshold window*, also known as *predicate window*, opens or closes based on whether a specified attribute of incoming elements crosses a defined threshold. These windows are variable length, non-overlapping, and *forward-context-aware*, as their boundaries depend on runtime

Table 3.1: Summary of standard window types, indicating whether they overlap, how their bounds are defined, and whether their boundaries are deterministic or data-dependent

Type	Overlap	Bounds	Determinism
Tumbling	No	Fixed size	Deterministic
Sliding	Yes	Fixed size and slide	Deterministic
Session	No	Variable with inactivity closures	Data-dependent
Snapshot	No	Time-based	Deterministic
Threshold/Predicate	No	Variable opens/closes on predicate	Data-dependent
Landmark (lower)	No	Start fixed, ends on next landmark / punctuation	Deterministic start / data-dependent end
Landmark (upper)	No	End fixed at landmark, starts at first record	Deterministic end

evaluation rather than fixed intervals.

A *landmark window* fixes one edge of the window at a predetermined *landmark* timestamp and lets the other edge move with the stream. In its lower-bounded form, the window starts at the *landmark* and grows forward as new records arrive. It remains open until a punctuation or user event defines the next *landmark*, at which point the current window closes and a new one begins. Because its start time is known in advance, this variant is deterministic and context-free. In the upper-bounded form, the window opens on the first record and closes once the stream reaches the landmark. Its end time is fixed in advance, so the window has a constant duration and does not change after closing.

Systems Comparison

Flink implements the window abstractions described above through its `DataStream` and `Table` APIs. It is possible to specify a window assigner (for example, tumbling, sliding, or session), a time domain (event-time, ingestion-time, or processing-time), and optional triggers and evictors. Windows can be defined on keyed and non-keyed streams. For keyed windows, Flink maintains per-key, per-window state and performs incremental aggregation to avoid buffering all records in a window [3, 4, 16].

NebulaStream provides a similar set of window operators but is explicitly centered on event-time and IoT workloads. Its C++ and Java query APIs support time-based tumbling, threshold, and sliding windows parameterized by an event-time attribute, which are implemented using the Scotty window aggregation framework and its general stream-slicing model [101, 102, 116]. These window types can be applied to keyed or non-keyed streams, and the optimizer can insert pre-aggregation operators close to sources to distribute window aggregation across multiple nodes in the edge-to-cloud continuum. This design reduces network traffic while retaining the expressiveness of general windowing semantics [101, 102, 116].

Table 3.2 contrasts how Flink and NebulaStream implement the window abstractions introduced in this section. Flink exposes time-based and count-based window assigners over multiple time domains and focuses on per-key, per-window state within a homogeneous cluster. Overlapping windows are evaluated within an operator, with limited sharing via incremental combiners and optimizations such as panes. NebulaStream, in contrast, builds its windowing layer on Scotty’s general stream-slicing model [102], which represents diverse window types (for example, tumbling, sliding, session, landmark, and predicate windows) as unions of non-overlapping slices that are

shared across queries, measures, and time domains, and that automatically adapt to workload characteristics such as stream order and window measures. This slice abstraction decouples the logical definition of context-free and context-aware windows from their physical placement so that the same internal representation can be instantiated on sensors, fog nodes, and cloud resources. Jointly with Disco’s distributed window-merging strategies for unique, context-free, and context-aware windows [10], NebulaStream can create and close windows independently on many nodes and combine them along a tree topology, whereas Flink typically forms and evaluates windows after the records have been collected inside the cluster.

Table 3.2: Comparison of windowing semantics and implementation support in Flink and NebulaStream

Aspect	Flink	NebulaStream
Window types	Tumbling and sliding windows for time-based and count-based measures, and session windows for time-based measures, on keyed and non-keyed streams	Tumbling and sliding time-based windows and predicate-based threshold windows are expressed in the query APIs and implemented via the aggregation framework
Time domains	Event-time, ingestion-time, and processing-time. Configured per stream or operator, with event-time used for out-of-order data	Event-time and ingestion-time windows. Event-time windows are parameterized by explicit timestamp attributes extracted from records. Ingestion-time windows use the system record-creation timestamp
Non-time windows	Count-based windows and custom windows via user-defined assigners, supporting arbitrary partitioning of unbounded streams	General window semantics realized through slice-based aggregation (for example, threshold windows), although the NebulaStream API primarily exposes time-based windows
Keyed vs. non-keyed windows	Keyed windows use per-key state and partitioned parallelism. Non-keyed windows aggregate over the entire stream within a task	Keyed and non-keyed windows supported. Window aggregation is defined over arbitrary keys and can be distributed across devices along the edge-to-cloud continuum
Distribution of aggregation	Incremental combination within operators (for example, pre-aggregation in keyed windows). Windowed aggregates are computed inside cluster TaskManagers	Slice-based aggregation can be pushed to sensors and fog nodes, forwarding only partial aggregates towards the cloud and thereby reducing network traffic

3.1.4 Out-of-Order Data

Data originating from IoT devices often arrives late due to weak connectivity, device limits, network delays, or environmental interference. Late data is defined in *event-time*: a record is late

if its event timestamp is earlier than the timestamps of records already processed or currently in line for processing. To handle late arrivals, systems track event-time progress with measures of progress, and use them to decide if late data should be discarded or processed [2].

Measures of Progress

Measures of progress outline how far a stream has advanced according to an ordering attribute A , such as the event-time. It is a lower bound on unseen values of A among unprocessed tuples that increases over time [68]. Measures of progress help operators deal with out-of-order arrivals, close windows, and reset the state. Standard measures of progress include heartbeats, watermarks, slack, punctuations, and frontiers [32]. Table 3.3 relates progress signals with origin, their scope, semantics, and window finalization based on this measure.

Heartbeats H are control records emitted by the source that assert no future event will arrive with $A \leq H$. Operators treat H as a safe lower bound, and with this measure, can close windows, finalize results, and evict the state. Heartbeats advance logical time without sorting and therefore deal with out-of-order streams [2].

Watermarks are a system estimate of progress W such that the system expects no future event with $A \leq W$. Watermarks advance A progress and trigger window finalization. Watermarks can be global, per stream source, or per-key. Watermarks are the standard mechanism in modern engines such as Google Dataflow [65], Flink [16], and NebulaStream [102] to bound lateness and finalize computations [4, 2].

Slack, or *Allowed Lateness*, sets a grace period δ beyond the end of a window. An operator accepts inputs for a window until $W \geq \text{window_end} + \delta$. Within δ , engines emit updates or retractions. After δ , late data is dropped [32].

Punctuations are markers that assert that no future tuples satisfying a predicate, such as time or key, will arrive. Operators use them to release a predicate state early, to improve memory use and latency [32].

Frontiers are the set of minimal outstanding timestamps across inputs. It represents progress when there are multiple inputs or partitions. Any timestamp less than the frontier is complete. Therefore, operators can safely finalize results for those times [32].

Systems Comparison

Flink manages out-of-order data using watermarks as its primary mechanism for progress tracking. Sources periodically emit watermarks that serve as lower bounds for future event-time timestamps. These in-band markers are propagated and aggregated across parallel operators by taking the minimum watermark over all inputs [3, 4, 16]. When the watermark at a window operator advances past the end of a window, Flink treats it as if all on-time elements have arrived and emits the results. Allowed lateness and side outputs can be configured so that late elements within a bounded lateness window can update previously emitted results.

NebulaStream also uses watermarks to reason about progress in the presence of out-of-order processing and must propagate these signals across heterogeneous devices in the edge-to-cloud continuum [101, 102, 116]. Event-time (and ingestion-time) windows in NebulaStream are defined on explicit timestamp attributes, and the system builds on Scotty out-of-order aggregation model to finalize local window instances on sensors, gateways, and cloud workers once sufficient progress has been observed [101, 102]. Because devices may be intermittently connected, NebulaStream combines progress tracking with adaptive operator placement and fault-tolerance mechanisms, enabling applications to adjust how they finalize windows based on network conditions and correctness requirements [101, 116].

Table 3.4 summarizes how both engines track progress and cope with out-of-order arrivals.

Table 3.3: Progress signals used for out-of-order handling, summarizing who emits them, their scope and semantics, and how they trigger window finalization

Signal	Emitted by	Scope	Semantics	Window closes when
Heartbeats	Source	Stream / key	No more earlier events will arrive	Heartbeat is later than the window end
Watermarks	Processing System	Global / source / key	System assumes earlier events are done	Watermark moves past the window end
Slack	Processing System	Per operator	Grace period for late events	Incoming data is later than the grace period
Punctuations	Processing System	Predicate / key	No more tuples match the predicate	A punctuation that covers the window arrives
Frontiers	Processing System	Multiple inputs	All inputs have advanced beyond a time	Every input is later than the window end

Flink computes watermarks per operator as the minimum over its inputs and combines those with configurable lateness bounds and side outputs, providing a cluster-level view of event-time progress. NebulaStream also relies on watermarks, but aggregates them hierarchically across edge, fog, and cloud workers and couples progress signals with delivery guarantees and operator placement.

3.1.5 Stream Query Languages

Stream query languages extend standard query languages to unbounded inputs. They add windows, track event-time, and define measures of progress so operators can produce consistent results as data arrives. We outline three query languages: stream relational algebra, Flink query language, and NebulaStream query language.

Stream Relational Algebra and Continuous Query Language

Stream Relational Algebra (SRA) adapts relational operators to support continuous queries over streams [6]. The Continuous Query Language (CQL) implements SRA by adding windows to SQL [62]. Queries run continuously in event-time and join or aggregate over time-bounded views. Code 1 computes the average train speed over a 5-second sliding window from data stream **SNCB** with 1-second slide (Line 3). It filters tuples whose composite **noise** score ($0.5 \times \text{speed} + 0.5 \times \text{BP}$) > 2 (Line 4), then outputs per-window averages (Line 2) as an insertion stream via **ISTREAM**.

Code 1. CQL query for average train speed over a sliding window

```

1 SELECT ISTREAM
2   AVG(speed) AS avg_speed
3 FROM SNCB [RANGE 5 SECOND SLIDE 1 SECOND]
4 WHERE (0.5 * speed + 0.5 *BP) > 2;
```


Table 3.4: Comparison of how Flink and NebulaStream track progress and handle out-of-order events under event-time semantics

Aspect	Flink	NebulaStream
Progress mechanism	Watermarks propagated with the dataflow. Each operator maintains a watermark equal to the minimum over its inputs and uses it to drive window finalization	Watermarks and progress metadata propagated across distributed operators, aggregated into a global watermark that feeds into a slice-based window aggregation model
Time semantics	Event-time processing is the default for out-of-order tolerant computations. Processing and ingestion-time are also supported	Event-time and ingestion-time supported. Event-time windows use explicit timestamp attributes from sources, while ingestion-time windows rely on system assigned timestamps
Handling out-of-order records	Records arriving after the watermark are considered late. Allowed lateness is configurable and determine whether a late records trigger updates or are dropped	Disorder introduced by parallel and distributed execution is handled by buffering records into slices. Window aggregates are emitted once global progress guarantees that no earlier timestamps are pending
Scope of progress aggregation	Cluster-level aggregation across TaskManagers. Progress is computed over all partitions of each input stream	Progress is aggregated using OriginId, SequenceNumber, ChunkNumber, and watermark timestamps, yielding a global notion of progress across intermediate devices in an edge-to-cloud deployment
Trade-off controls	User-configurable watermark strategies and lateness bounds balance latency and completeness of results	Deployments can trade off latency, bandwidth, and robustness by choosing window parameters and where to place windowed operators along the edge-to-cloud topology

Flink Query Language

Flink implements the DataStream API, stateful streaming operators, watermarks, windowing, and temporal joins. Code 2 sketches the same computation as Code 1 in Flink DataStream API. The code filters records by the noise score (Line 3), groups them into 5-second sliding event-time windows with 1-second slide (Line 4), and aggregates average speed (Line 5).

Code 2. Flink DataStream API equivalent of the CQL average speed query

```

1 DataStream<SNCBEvent> sncb = env.addSource(sncbSource);
2
3 sncb.filter(e -> 0.5 * e.speed + 0.5 * e.BP > 2)
4 .windowAll(SlidingEventTimeWindows.of(Time.seconds(5), Time.seconds(1)))
5 .aggregate(new AverageSpeed());

```

NebulaStream Query Language

NebulaStream implements a polyglot query engine with support for multiple query languages, including C++ operator API and SQL-like language for heterogeneous deployments. The optimizer places operators across edge, fog, and cloud and uses measures of progress to handle out-of-order data. Code 3 expresses the same query as Code 1 by deriving a **noise** attribute (Line 2), filtering on **noise** > 2 (Line 3), applying a 5-second sliding event-time window with 1-second slide (Line 4), and averaging speed (Line 5).

Code 3. NebulaStream query language equivalent of the CQL average speed query

```

1 Query::from(SNCB)
2   .map(Attribute(noise) = Attribute(speed) * 0.5 + Attribute(BP) * 0.5)
3   .filter(Attribute(noise) > 2)
4   .window(SlidingWindow::of(EventTime(Attribute(ts)), Seconds(5), Seconds(1)))
5   .apply(Avg(Attribute(speed)));

```

3.1.6 Directed Acyclic Graph Dataflow

Stream processing systems such as Google Dataflow, Flink, and NebulaStream represent query computations as DAG. In this representation, each query component, such as a filter, join, or window, is represented as a node, and edges illustrate how data flows between them. The system builds a high-level operator graph to represent the query’s logical plan. Following this, the optimizer improves this plan by merging operators, defining join and aggregation strategies, and reordering tasks. The outcome is a physical plan that executes operations in parallel as subtasks, enabling independent processing of input partitions. These subtasks exchange records using techniques such as partitioning, shuffling, or broadcasting [3].

Systems Comparison

Flink translates the operator graph into several internal representations. Program workflows are first translated into a logical graph, represented as a StreamGraph for streaming applications, and subsequently transformed into a JobGraph, enabling the chaining of operators into coordinated tasks for efficient execution. At deployment time, the JobManager derives an ExecutionGraph with one vertex per parallel subtask. The scheduler assigns execution vertices to TaskManagers, considering edges as pipelined or blocking, along with existing operator chains. This approach reduces serialization and scheduling overhead while maintaining the data-parallel semantics of the original DAG [3, 4, 16]. For instance, a map operator can expand into several subtasks, each responsible for a fraction of the data. The operators are connected in a pipelined execution, allowing records to flow continuously through the graph and reducing overall latency.

NebulaStream follows a similar multi-stage compilation pipeline and also incorporates knowledge of the physical network topology and device capabilities across the edge-to-cloud continuum. Queries are compiled into execution graphs that utilize diverse resources. The coordinator creates a global logical plan, which can be optimized using rules for query merging and placement strategies. This plan is then divided into execution fragments for individual nodes, which are deployed to sensors, gateways, and cloud workers. The optimizer considers device capacity when placing these operators, as well as the network topology [19]. In-network processing enables the engine to push filters and pre-aggregations toward the sources while keeping more complex joins and analytics in the fog or cloud (see operator placement in Sect. 3.1.11).

Table 3.5 compares how Flink and NebulaStream materialize DAG dataflows. Flink refines user programs into JobGraphs and ExecutionGraphs that target a homogeneous collection of

Table 3.5: Comparison of how Flink and NebulaStream represent and execute queries as a DAG dataflow

Aspect	Flink	NebulaStream
Graph layers	User program $\rightarrow$ StreamGraph $\rightarrow$ JobGraph $\rightarrow$ Execution-Graph with one vertex per parallel subtask	Logical query plan $\rightarrow$ global query plan $\rightarrow$ global execution plan $\rightarrow$ per-node execution fragments deployed across edge, fog, and cloud
Optimization focus	Operator fusion, join and aggregation strategies, selection of shuffle patterns, and parallelism levels to exploit cluster resources efficiently	Query merging into shared query plans, operator placement across sensors, fog, and cloud, and reuse of common subgraphs across queries to minimize communication and resource use
Operator chaining/merging	Chains compatible operators into a single task to reduce overhead and improve pipelining while respecting data-parallel boundaries	Merges semantically equivalent query fragments into shared query plans and places shared operators so that multiple queries can reuse intermediate results
Placement decisions	Placement is primarily within a homogeneous cluster of TaskManagers under the control of the JobManager and resource manager	Placement explicitly considers heterogeneous devices and network links along the topology

TaskManagers and exploits operator chaining to minimize overhead within the cluster. NebulaStream splits the global plan into per-node fragments that are deployed on sensors, gateways, and cloud servers.

3.1.7 Transactional Stream Processing

While many modern stream processors provide transactional building blocks, only a subset of systems are explicitly designed as full Transactional Stream Processing (TSP) engines. TSP engines unite continuous, low-latency ingestion and transformation of stream processing with Atomicity, Consistency, Isolation, and Durability (ACID) guarantees of traditional transactions [119, 44]. Early stream engines based on publish-subscribe and data stream processing systems handled high-volume, real-time data but lacked support for maintaining a consistent state. TSP systems were developed by embedding ACID guarantees within the dataflow to prevent anomalies and to reconcile distributed states across failures [119].

A TSP system comprises several components: transactions, operators, schedulers, and storage [119]. Streaming events trigger transactions in TSP systems and can be executed individually or in batches. The system ensures that these transactions adhere to the ACID properties. Operators in TSP systems process incoming data and perform operations such as filtering, aggregation, transformation, and joins. The scheduler manages the execution order of these operators to

ensure that transactions are processed correctly and efficiently, handling out-of-order events and coordinating distributed execution. The storage component of TSP systems deals with both temporary and permanent states. In-memory storage is used for fast access to mutable states and intermediate results, while persistent storage ensures durability and recovery in case of failures. TSP systems implement transactions using different approaches, including unified, embedded, and state transactions. Unified transactions integrate stream processing operations into a transactional framework, treating both operations uniformly. Embedded transactions incorporate transaction processing into the stream processing pipeline, allowing real-time processing with transactional guarantees. State transactions manage shared mutable states through transactions, ensuring consistency and isolation across the dataflow pipeline [119, 65].

TSP systems must provide strong delivery guarantees to handle failures and ensure data integrity. Standard guarantees include *at-most-once*, *at-least-once*, and *exactly-once* processing, with *exactly-once* being the most desirable for ensuring each event is processed precisely once. To achieve these guarantees, TSP systems employ fault-tolerance mechanisms such as state replication, periodic checkpointing, and log-based recovery, enabling them to recover consistently after crashes or network partitions [119].

A TSP system manages several interrelated components: a continuous stream of incoming events, continuous queries defined by a windowing strategy, such as time-based, count-based, or threshold-driven, and aggregation functions, such as sum, average, count, min, and max, that operate within each window. By executing each windowed aggregation inside a transactional context, TSP systems ensure that results reflect a consistent snapshot of the stream even under concurrent execution and failure conditions [119].

Systems Comparison

Flink provides *exactly-once* consistency for operator state via distributed snapshots and integrates with transactional sinks. Checkpoints establish globally consistent recovery points for the dataflow. When a checkpoint succeeds, all operators have persisted a consistent view of their state, and two-phase-commit sinks either commit or roll back their external writes based on the checkpoint outcome [3, 4, 16]. This mechanism approximates state transactions at the granularity of checkpoints and sink commits, while isolation between distinct streaming jobs remains at the level of independent dataflows [4, 119].

NebulaStream emphasizes snapshot-based state management and fault tolerance rather than exposing a general-purpose ACID transaction interface. The platform focuses on coordinating consistent processing across heterogeneous devices in the edge-to-cloud continuum by combining stateful operators, window aggregation, and fault-tolerance mechanisms [101, 102].

Table 3.6 highlights the different positioning of Flink and NebulaStream with respect to transactional stream processing. Flink delivers *exactly-once* state via periodic checkpoints and coordinates transactional sinks at checkpoint boundaries. NebulaStream, in contrast, is being extended with a transaction-aware engine to orchestrate consistent processing across operators and devices, aiming to achieve ACID-like guarantees in highly volatile IoT settings.

3.1.8 Just-in-Time Query Compilation

JIT query compilation creates code at runtime and avoids the execution overhead of the interpreted code during query evaluation. In databases, JIT compilation replaces interpreted query operations with specialized code for the current query. For example, instead of using a statistical function to evaluate a predicate such as `WHERE a.col = 3`, the system generates a routine tailored to the predicate and column type. Both PostgreSQL and NebulaStream use JIT in their execution engines, but have different objectives.

Table 3.6: Comparison of transactional stream processing capabilities in Flink and NebulaStream

Aspect	Flink	NebulaStream
Primary consistency mechanism	Distributed checkpoints and exactly-once operator state. Two-phase-commit or idempotent sinks are required for end-to-end guarantees	Snapshot-based state management combined with window aggregation and operator replication in a heterogeneous edge-to-cloud deployment
Transaction granularity	Implicit, checkpoint-level state transactions	Consistency is managed at the level of operator state and query fragments
Isolation across queries	Isolation is provided by separate jobs and their checkpointing	Queries share resources and may reuse operators, but transactional isolation between arbitrary query combinations has not been a primary focus in the literature
Integration with external storage	Transactional or idempotent sinks are needed to avoid duplicates on recovery	Integrates with various brokers and sinks. Research emphasizes reducing data movement and managing state at the edge rather than full transactional integration with external databases
Position with respect to TSP	Provides strong processing guarantees and building blocks discussed in the TSP survey, but is not a dedicated TSP system	Shares many architectural concerns with transactional stream processors (for example, state management and fault tolerance across a topology), but publications do not yet present NebulaStream as a full ACID TSP engine

PostgreSQL introduced JIT [39] compilation to improve the performance of CPU-intensive queries. The system uses a cost model to decide when JIT should be enabled. For example, in expression evaluation, rather than walking expression trees row by row, PostgreSQL generates code specialized to the query’s conditions and evaluates them in tight loops.

Systems Comparison

Flink primarily relies on ahead-of-time compilation of the engine but augments this with runtime code generation for user-defined operators. The planner generates specialized Java classes for expressions, projections, and aggregations, which are then compiled on-the-fly using lightweight Java compilers such as Janino and executed on the JVM. This approach inlines user logic, eliminates virtual method calls, and reduces boxing overhead, allowing the underlying JVM JIT to further optimize hot code paths while keeping compilation latency low [4, 16].

NebulaStream pushes JIT compilation further with Nautilus, a hybrid execution engine that combines interpretation and trace-based compilation [40]. Queries initially execute in an interpreter that records hot traces across operators. When a trace becomes sufficiently long-lived

or expensive, Nautilus lowers it to an intermediate representation and compiles it to native code using different backends, ranging from an in-process assembler to LLVM [40]. Short-running or rarely executed code paths remain interpreted. At the same time, compute-intensive sections benefit from optimized, hardware-conscious native code, enabling NebulaStream to adapt its execution strategy to both workload characteristics and device capabilities.

Table 3.7 contrasts the JIT compilation strategies of the two systems. Flink generates Java code for expressions and operators at planning time and relies on the JVM JIT to optimize hot paths, which keeps compilation latency small but limits hardware-specific specialization. NebulaStream Nautilus engine instead records hot traces across operators, lowers them to an intermediate representation, and compiles them to native code using different backends.

Table 3.7: Comparison of JIT compilation strategies in Flink and NebulaStream

Aspect	Flink	NebulaStream
Compilation unit	Generated Java classes for expressions, projections, and operators in the Table/SQL APIs, compiled and optimized by the JVM	Hot traces across operators captured by an interpreter and lowered to an intermediate representation for compilation
Compiler technology	Lightweight Java compilers plus JVM JIT optimization of the generated bytecode	Trace-based JIT compiler with interpreter, assembler backend, and an experimental LLVM backend for different latency/throughput trade-offs
Execution modes	Primarily compiled Java execution with JVM-level JIT	Hybrid model: interpretation for cold code paths and native compilation for hot traces within the same engine
Hardware specialization	Exploits JVM optimizations. Hardware-specific code generation is limited	Generates hardware-specific code tailored to different CPU architectures and, experimentally, for accelerators such as GPUs

3.1.9 Consistency and Fault Recovery

Consistency and fault recovery are necessary mechanisms for stream processing systems. To produce correct results despite failures, systems must define how often each event is delivered and how the state is recovered. Delivery guarantees specify if events may be lost, duplicated, or processed. Fault recovery complements these guarantees by providing mechanisms to restore operator states.

Delivery Guarantee

A delivery guarantee defines how a streaming system processes each event. *At-most-once* delivery states that an event is processed zero or one time, i.e., some data may be lost, but no duplicates appear. *At-least-once* delivery ensures that no data is lost, but events may be processed more than once by the system. *Exactly-once* delivery is the strongest guarantee, in which every event contributes to the results exactly-once, with no loss and no duplication [4]. Google Cloud

Dataflow [3] supports exactly-once processing for internal state. The system deduplicates records so that each event has an impact on the output only once. The system assigns sequence numbers to records and performs a streaming shuffle to remove repeated records.

Flink [16] offers exactly-once semantics for stateful stream processing. Flink achieves this by taking distributed snapshots across the execution graph when checkpoint barriers are reached. When an operator receives a barrier, it snapshots its state and records input positions.

Exactly-once engine state does not imply end-to-end exactly-once. Sink outputs must be written via transactional sinks (e.g., two-phase commit) or idempotent sinks to avoid duplicates on recovery, otherwise the guarantee holds only within the dataflow [3, 4, 16].

Recovery and Snapshot

Recovery is the ability to recover all of its states and proceed with execution after a failure without data loss. A standard technique employs *snapshot*, which is a fixed representation of the state and position of the input for the operator at a given point in time. Snapshots record the local state of operators, such as counters, aggregations, and window buffers, as well as placement in the current position of each input stream. They pair with progress measures to enable fault recovery without reprocessing finalized windows [32].

Checkpoint is a consistent snapshot of a dataflow that captures states and is written to durable storage so that, in the event of a failure, the job can resume from that snapshot without losing data or reprocessing events inconsistently. For example, Flink [16] relies on consistent global snapshots and restores the operator state, reverting to the last completed checkpoint. Events processed after the checkpoint but before the failure are reprocessed.

Systems Comparison

Flink exposes its consistency model through configurable checkpoints and savepoints. Applications select checkpoint intervals, state backends, and external storage for checkpoint data. During execution, the engine injects barriers into all input streams, aligns them at stateful operators, and persists operator state to durable storage when a checkpoint completes [4].

NebulaStream inherits snapshot-based techniques from modern dataflow engines but must apply them in more volatile, resource-constrained environments. Its design targets consistent processing across heterogeneous devices in the edge-to-cloud continuum and combines stateful operators with research-proposed dynamic operator migration and reconfiguration [17]. NebulaStream prioritizes critical operators or pipeline segments, reflecting the constraints imposed by heterogeneous devices and intermittent connectivity.

Table 3.8 compares the consistency and recovery mechanisms offered by Flink and NebulaStream. Flink provides exactly-once operator state through global checkpoints and exposes savepoints for controlled upgrades and rescaling in stable cluster environments. NebulaStream adopts snapshot-based techniques as well, but decouples delivery guarantees from where fault-tolerance mechanisms are implemented, protecting only critical operators when resources are scarce.

3.1.10 Common Architectures

Streaming systems are based on architectural designs that address latency, throughput, and accuracy. Over time, three main architectures have emerged: Lambda, Kappa, and the edge-to-cloud continuum.

Table 3.8: Comparison of consistency guarantees and fault-recovery mechanisms in Flink and NebulaStream

Aspect	Flink	NebulaStream
State consistency	Exactly-once semantics for operator state via consistent global checkpoints and barrier alignment	Combines window aggregation, stateful operators, and snapshot-based techniques
Recovery mechanism	Restores operator state from the latest completed checkpoint and replays input from recorded offsets	Uses snapshot-based recovery, aims to rely on reconfiguration and replication of operator fragments across workers
Snapshot granularity	Global checkpoints covering all stateful operators. Savepoints for operational life cycle management (upgrades, rescaling)	Experimental flexible placement of fault-tolerance mechanisms, potentially protecting only subsets of operators or specific parts of the edge-to-cloud pipeline
Environment assumptions	Relies on stable clusters with reliable storage systems	Volatile edge and fog devices with intermittent connectivity and limited resources. Fault tolerance must be aware of topology dynamics

Lambda Architecture

The Lambda architecture integrates batch and stream processing to support accurate and low-latency analytics [59]. The architecture maintains two parallel pipelines, a batch path that processes historical records in bulk, and a stream path that ingests and processes in real-time. At query time, outputs from both paths are merged to produce complete results. Nevertheless, Lambda architecture increases complexity, requiring the maintenance of two codebases for the same logic and keeping them synchronized across distinct systems [3].

Kappa Architecture

The Kappa architecture reduces complexity by removing the batch path and relies on a single pipeline for all processing [69]. Real-time updates and historical reprocessing are handled by replaying event logs through the stream processor. Systems such as Flink [16] support this model, as they include stateful operators that can consume real-time data as well as replayed data. However, the architecture assumes durable event logs are available and may be less efficient for some large-scale offline analytics.

Edge-to-Cloud Continuum

The edge-to-cloud continuum extends processing from the cloud to the edge [56]. This architecture distributes computation between IoT devices, fog, and cloud servers. Processing data at the edge reduces latency and bandwidth usage by filtering or aggregating streams locally, while the cloud provides large-scale computation, coordination, and long-term storage. Summarized data can be transmitted upstream for later analysis, balancing responsiveness with scalability. This ar-

chitecture introduces several challenges, including coordinating computation across heterogeneous environments and dynamically determining where queries should be executed. NebulaStream implements a modular pipeline across sensors, fog, and cloud, placing operators according to device capabilities and network conditions [116].

Systems Comparison

Flink is a cloud-focused cluster-based engine that can be used to implement Kappa architectures. A JobManager accepts jobs, builds the ExecutionGraph, coordinates checkpoints, and manages failure recovery, while TaskManagers execute operator subtasks and exchange records via network shuffles [16]. Durable log systems, such as Kafka, and persistent file systems, such as HDFS or object stores, provide the replayable history required for both real-time processing and batch-style reprocessing.

NebulaStream, by design, targets the edge-to-cloud continuum. A central coordinator maintains metadata about logical and physical sources, the network topology of workers, and running queries [116]. Workers can be deployed on sensors, gateways, or cloud servers and host local execution engines that run operator fragments compiled by Nautilus [40]. The architecture supports in-network processing, placing operators along the path from sensors to the cloud to enable filtering and pre-aggregation close to the data sources. Experimental adaptive resource management in NebulaStream design allows the system to react to topology or workload changes with minimal disruption to running queries [18].

Table 3.9 contrasts the architectural roles and deployment assumptions of the two engines. Flink assumes a cluster or managed cloud setting with JobManagers and TaskManagers orchestrating long-running streaming jobs over durable logs and distributed storage, which can support Kappa-style architectures based on replay. NebulaStream instead organizes a coordinator and a large number of workers that may run on sensors, gateways, or cloud machines, and explicitly targets in-network processing.

3.1.11 Stream Operators

Stream operators are the fundamental building blocks of streaming programming. They define how to change, reduce, or summarize incoming records. Operators range from simple, stateless transformations, such as filters and maps, to complex, stateful computations, including aggregations and joins. They form the nodes of the dataflow graph that a streaming engine compiles and executes.

Operator Placement

Operator placement is an optimization problem where the optimizer should decide on which node an operator will be executed. The objective is to guarantee expected latency, throughput, power consumption, and privacy under limited resources and unreliable connectivity. In such deployments, placement is dynamic due to a varying number of mobile, unreliable nodes, changing bandwidth, and the arrival of new query updates that may invalidate the previous mapping.

The ISQP approach in [17] views placement to be an iterative and concurrent optimization process. Instead of recomputing a full plan, it batches changes such as query additions or topology updates and identifies only the operators whose placement is no longer valid. It then creates *deltas* which are subgraphs bounded by fixed sources and sinks. Such deltas are re-optimized in parallel with the concurrency control of shared topology resources.

Table 3.9: Comparison of the deployment architecture assumptions and execution roles of Flink and NebulaStream

Aspect	Flink	NebulaStream
Primary deployment model	Cluster-based engine running in data centers or cloud services, often backed by Kafka and distributed file systems	Edge-to-cloud engine spanning sensors, gateways, and cloud servers, designed for heterogeneous IoT deployments
Control components	JobManager and ResourceManager coordinate job execution, resource allocation, and checkpointing	Central coordinator manages topology, data sources, and query optimization. Workers are lightweight executors
Workers	TaskManagers host operator subtasks and manage state backends. Each TaskManager offers a set of task slots	Workers run on heterogeneous hardware, including constrained edge devices, and execute operator fragments generated by Nautilus
Integration with logs/storage	Relies on durable logs and distributed file systems for replay and state backup	Integrates with message brokers and custom sinks, but emphasizes minimizing upstream data through in-network filtering and aggregation
Architectural style	Can realize Kappa-style architectures by using a single streaming pipeline for both real-time and reprocessing workloads	Implements an edge-to-cloud continuum architecture with modular pipelines optimized at each hop and adaptive operator placement

Stateful and Stateless Functions

Stateless functions, such as `filter`, handle each record separately and do not keep the operator state. Stateless code scales, as it can be executed in parallel and can be placed anywhere in the topology. In addition, *checkpoints* do not need to capture extra data for these functions.

Stateful functions keep data across records, read and update the operator state. Aggregation functions such as counters and averages, joins, and windowed operations are examples. Stateful code requires a placement strategy, as it should be located near its keys or sources to minimize network costs.

Filter, Map, and Flatmap

Stream processors have core stateless operators that modify the data before it reaches stateful operations, such as joins or aggregations. The `filter` operator keeps a record if a predicate evaluates to true. The `map` operator returns one modified record for each input record, providing a one-to-one relationship. This transformation is generalized by the `flatMap` operator, which can generate zero, one, or multiple output records from a single input.

Window Aggregation

Window aggregation is a necessary functionality for the computation of real-time data analysis and data management. Aggregation functions play an integral role in the calculation of aggregated

results derived over a particular time frame or specific events. Windowed aggregation functions are performed within predefined windows using different measure types, such as time and count. Many operations are made possible by such aggregations: summation, averaging, counting, and computation of the minimum and maximum values of the data.

The authors in the study [101], Scotty is introduced. The system is a framework designed for aggregation and implemented in NebulaStream. Scotty distinguishes itself by its ability to apply aggregate functions independently of the window type. In that sense, Scotty design aims to offer adaptable and scalable stream aggregation. This strategy guarantees that the same aggregation functions are always applied in a way that allows the development and maintenance of stream processing systems to be straightforward.

Aggregation Function

An aggregation function accesses all elements in the window or the dataflow and transforms them into a result. These functions are stateful as they keep state, update it as records arrive, and produce a final result. Examples of aggregation functions are sums, counts, and averages. These functions can be keyed, windowed, or global. Keyed aggregation keeps one partial per key, windowed aggregation keeps one partial per (key, window), and global aggregation keeps one partial for all records.

NebulaStream defines its aggregation by splitting it into three functions, namely **Lift**, **Combine**, and **Lower**. This approach supports parallel execution and allows the engine to perform incremental, distributed, and reusable aggregation independently of where (sensor/fog/cloud) and how (global/keyed/windowed) the result is computed [17, 40]. In addition, it enables incremental updates. As data is input, the whole aggregation does not need to be fully recomputed. An example is the average aggregator, which NebulaStream expresses as three stages:

$$\begin{aligned} \text{Lift}(\text{record}) &\rightarrow \text{partial} \\ \text{Combine}(\text{partial}_A, \text{partial}_B) &\rightarrow \text{partial} \\ \text{Lower}(\text{partial}) &\rightarrow \text{result} \end{aligned}$$

In **Lift**, one input record is mapped to an initial partial state. For example for computing average, we use $\text{lift}(x) = \{\text{sum} : x, \text{count} : 1\}$. In **Combine**, two partial states are merged. For average, we use $\{s_1, c_1\} \oplus \{s_2, c_2\} = \{s_1 + s_2, c_1 + c_2\}$. Finally, **Lower** produces the final output value from the partial state. We then use $\text{lower}(\{s, c\}) = s/c$.

Flink also implements aggregation functions and exposes windowed aggregation as a window operator that takes an aggregate function and, optionally, a process window function. The Flink aggregate function also separates the accumulator, update, merge, and finalize, but the API is presented via window contexts.

Systems Comparison

Flink implements stream operators as instances that wrap user-defined functions, such as **MapFunction**, **FlatMapFunction**, or **ProcessFunction**. Each parallel operator subtask runs within a task occupying a slot on a TaskManager and processes records one by one while interacting with a pluggable state backend that stores keyed and operator state [16]. Stateless operators are often chained together within the same task to reduce serialization and scheduling overhead. Stateful operators, including windowed aggregations and joins, rely on operator state and timers to manage per-key buffers and emit results when progress permits, using the checkpointing mechanisms to preserve state across failures [4].

NebulaStream implements its operators through an extensible C++ execution engine with a plugin architecture [40]. Core operators such as filters, maps, joins, and window aggregations are expressed in an expression framework that models attribute access and predicate logic [24, 40]. New domain-specific operators can be implemented by extending these base classes and registering them dynamically, for example, to expose MEOS functions in MobilityNebula [24] (see Sect. 4.2.2). Nautilus then interprets or compiles the resulting operator pipelines to native code, and the NES optimizer decides where to place each operator instance along the pipeline [40].

Table 3.10 summarizes how stream operators are exposed and executed. Flink implements user-defined function interfaces and Table/SQL operators. NebulaStream offers operators via an extensible C++ execution engine and an expression framework that supports dynamically registered, domain-specific operators, which Nautilus can interpret or compile to native code and which the optimizer places across sensors, fog, and cloud.

The architectural differences in window implementation (see Sect. 3.1.3) are also reflected in how aggregation functions are expressed and executed in both systems. On the one hand, Flink, windowed aggregation is implemented by user-defined functions that operate on the state of a single window instance, either incrementally (for example, via `ReduceFunction` or `AggregateFunction`) or as full-window operators (for example, `ProcessWindowFunction`) that can inspect all records in a window at once. Aggregation proceeds in one logical stage per operator and relies on the state backend for scalability and fault tolerance. NebulaStream, on the other hand, utilizes the **Lift**, **Combine**, and **Lower** interface, which is derived from general stream slicing [102]. In this approach, aggregation functions are structured in a decomposed format that enables incremental updates to slice aggregates. When slices are combined, the aggregates are merged, and the model leverages properties such as invertibility and commutativity whenever possible. Disco [10] distributes these partial aggregates across the edge-to-cloud topology, sending compact aggregate states for distributive and algebraic functions. Therefore, NebulaStream aggregation layer is designed to share work across overlapping windows and push aggregation into the network. In contrast, Flink primarily optimizes aggregation within a centralized cluster, where each window is aggregated locally per operator.

3.2 Mobility Data

Spatiotemporal systems handle data from moving objects and allow queries over this data type. This section provides the foundations for spatiotemporal data management, outlier detection, and lifted operations, enabling the large-scale processing of such data.

3.2.1 Spatiotemporal Data and Mobility Data

Spatiotemporal data commonly originates from measurements taken by sensors in moving objects. It integrates spatial attributes, latitude and longitude, with temporal information, timestamps. Efficiently processing this data on a large scale necessitates models that can effectively represent and manage spatial and temporal dimensions [90].

A *spatiotemporal object* represents an object whose spatial position changes over time, as captured by sensors such as GPS. In this thesis, we implement point objects and trajectory objects that trace continuous paths over time.

Mobility data is spatiotemporal data for moving objects related to the mobility domain. We assume the path between observations is interpolable and that the object keeps a fixed shape and size. For example, data from cars, aircraft, ships, and trains is considered mobility data. Phenomena with changing extent or behavior, such as storms, are out of the scope of this thesis.

Table 3.10: Comparison of how Flink and NebulaStream expose and execute core stream operators, including extensibility and placement

Aspect	Flink	NebulaStream
Operator API	User-defined functions (for example, <code>map</code> , <code>flatMap</code> , <code>process</code>) and declarative operators in the Table/SQL APIs	C++ and Java query APIs with operators such as <code>map</code> , <code>filter</code> , <code>join</code> , and window operators expressed via an expression framework
State management	Keyed and operator state backed by configurable state backends. Integrated with checkpoints for exactly-once recovery	Operator state managed by the NES Engine. Fault-tolerance and placement strategies are tuned for IoT deployments across edge, fog, and cloud
Extensibility	User-Defined Functions (UDFs) and custom operators implemented via function interfaces and rich functions. Libraries can wrap external systems	Plugin architecture and expression framework allow dynamic registration of new operators and functions
Execution	Operators run inside TaskManagers as chained tasks connected by back pressure-aware channels	Operators are executed by local engines on workers. Nautilus can interpret or JIT-compile operator pipelines to native code
Placement	Placement within a homogeneous cluster. Operators are scheduled based on slot availability and configured parallelism	Placement across sensors, fog, and cloud nodes. The optimizer considers latency, bandwidth, and resource constraints when deciding where each operator runs

3.2.2 Mobility Continuous Queries

Mobility data streams are continuous flows of spatial information capturing movements, events, and changes of spatial objects over time [37]. Characterized by their real-time nature, these streams integrate data from diverse sources, such as GPS receivers, mobile devices, and sensors. Effective management of mobility streams requires addressing challenges related to data integration, spatial awareness, continuous query processing, and visualization, all of which are critical for extracting actionable insights [12]. Applications leveraging mobility streams include real-time urban planning, environmental monitoring, transportation management, and emergency response systems, where timely spatiotemporal data analysis significantly improves decision-making and operational responsiveness.

Unlike generic event streams, mobility streams pair *event-time* with *location*. Mobility streams present both domain-specific challenges and opportunities [12]. This data type can have a path-dependent state in many predicates that rely on the entire trajectory history, such as average speed. Mobility streams can require expensive operators, such as calculating distance and defining *k*-Nearest Neighbors (*k*-NN), which necessitate non-trivial geometric calculations that are challenging to maintain at stream rates. When events are dependent but not identically distributed, it is essential to consider spatial autocorrelation and locality. Nearby observations exhibit a correlation that can be exploited through the use of bounding boxes or grid cells.

Trajectories have continuity constraints and must respect temporal ordering and feasible motion, such as maximum speed, which enables the safe discarding of impossible moves but also may require preprocessing or buffering of the sequence. In particular, while stream systems can accept late data, trajectory construction must explicitly account for late arrivals and cannot assume strictly ordered input.

A mobility continuous query is initiated once on a system, logically persists, and continuously operates on the data until explicitly terminated [35]. This type of query enables users to receive real-time updates and new results from the system without repeatedly reissuing the same query. Unlike traditional queries in databases, which are executed once and provide results based on the data available at the time of execution. The key distinction is that continuous queries are not limited to the data available at the time of execution [35].

3.2.3 Outlier Detection

Communication signals are widely used to collect mobility data on various systems. However, the data collected through this system often needs to be revised due to various factors, including faulty installation, message collisions, signal deficiencies caused by limitations in the communication equipment, transmission delays, or environmental issues. Detecting and correcting these errors is crucial for obtaining correct results.

In the last few decades, the IoT has led to an unprecedented accumulation of data collected daily from various sensors. Researchers have noted that approximately 60% of the data mining process is devoted to data preparation, emphasizing the critical need for data cleaning [49].

Outliers Outliers are points that significantly deviate from the others. In the context of trajectory and spatiotemporal data, outliers could represent an abrupt and uncharacteristic change in direction, speed, or heading. Outlier detection is an essential step in data preprocessing to ensure the integrity and validity of data analyses [25].

Statistic-Based Methods

The essential concept of these techniques for identifying outliers relies on their alignment with the statistical distribution model. These methods include the Kalman Filter (KF). KF is a well-established method used to smooth point series. This algorithm estimates missing points based on previously observed values that might have measurement errors. In [104], the authors mention that the advantage of the KF and its derivatives is their recursive aspect, which can be used in real-time. It is also widely used due to its capability to provide accurate estimations and prediction results.

The KF is restricted to linear system models, which limits its applicability in multiple real-world scenarios where system dynamics and measurements may be nonlinear. To address this limitation, the Extended Kalman Filter (EKF) extends the capabilities of the Kalman filter by linearizing the nonlinear system model around the current estimate [53, 58]. The EKF is a recursive and real-time estimation method that predicts the system's evolution, calculates the expected observation, and then adjusts the prediction based on the actual measurement.

The EKF operates under the assumption of a nonlinear process model. Due to the nonlinearity of the system, an exact optimal solution in the mean-square sense is not attainable. Instead, the EKF approximates the evolution of the state by linearizing the functions around the current estimate. The Jacobian matrices used in this process capture the local linear dynamics and the sensitivities of the observations. Subsequently, the filter employs a discrete predictor-corrector recursion to refine its estimates.

Particle Filter (PF) use a set of randomly generated particles to represent the possible states of the system and update the particles based on observed data. In contrast to KF, they are not

restricted to the Gaussian distribution of errors, which makes them applicable to a broader range of noisy data. PF can, however, be computationally intensive and thus not commonly implemented in trajectory libraries. Additionally, as with KF, PF is sensitive to the first measurement in the trajectory, and their accuracy can be reduced if the first point is an outlier [123, 67, 61].

The Hampel Filter (HF) detects and replaces outliers in trajectories with estimates via the Hampel identifier. The HF uses the median and the Median Absolute Deviation (MedAD) as robust estimators, assuming that most observations lie within $k * MedAD$ of the median [85]. For each trajectory, the method calculates the median of a sliding window and adjacent points on each side of the trajectory. The HF estimates the standard deviation of each point about its window median using the median absolute deviation. If a measurement differs from the median by more than the threshold, the filter replaces the sample with the median.

Autoregressive Integrated Moving Average (ARIMA) is a statistic-based method for modeling and forecasting time series. The method incorporates temporal dependence by combining two components: Autoregression (AR) and Moving Average (MA). The model depends on its past values, applies differencing to remove non-stationary trends, and depends on past errors. ARIMA is used for a single series with evenly spaced data in time.

Sliding Window Methods

The foundation of sliding window algorithms is calculating the distance between different data points in their neighborhood. An outlier is a data point that is significantly distant from its neighbors. When detecting an outlier inside a trajectory, the method compares each point in the trajectory to its neighbors and selects the points substantially further away than expected. Methods based on the mean or median filters replace points compared to the measurements done at preceding points in time. These algorithms are practical and straightforward for detecting single outliers. Nonetheless, these techniques depend on the number of predecessors compared to the mean. Multiple successive outlier points can affect the accuracy of the outcome trajectory.

The authors in [60] review a series of sliding window algorithms. They develop an optimized cell-based algorithm that outperforms the existing methods. The cell-based methodology is built on partitioning the multidimensional data space into a structured grid of cells. The dimensions of these cells are contingent on the distance threshold established for detecting outliers. Subsequently, each cell within the grid is searched to identify potential outliers. In experiments, the method exhibits scalability with large datasets. However, the computational cost escalates significantly as the dimension increases. The optimization of the cell-based methodology entails a strategic approach to minimize the number of cells that need examination. This is accomplished by maintaining a register of active cells containing at least one object. In addition, only these active cells are subject to examination.

The authors in [114] tackle identifying outliers within extensive trajectory streams. The paper introduces a taxonomy of neighbor-based trajectory outlier definitions. The authors evaluate their work on two real-world datasets, the Beijing Taxi trajectory data and the Ground Moving Target Indicator data stream. The authors also utilize the synthetic dataset Moving Objects Database (MOD) generated by a network-based moving object generator [14] using real road networks. The Minimal Examination framework is central to their approach. It embodies three optimization principles, such as point neighbor-based outliers, trajectory neighbor-based outliers, and synchronized neighbor-based outliers that exploit spatiotemporal and predictability properties of neighbor evidence to reduce detection costs significantly. The framework has algorithms that detect the outliers based on these classes of new outlier semantics. The testing shows that Minimal Examination is scalable and can handle up to one million moving objects per second on a desktop.

The authors in [15] introduce a method for outlier detection in trajectory data streams,

proposing two algorithms: trajectory outlier detection on trajectory data streams and an approximate trajectory outlier detection algorithm, which aims to reduce detection costs through a space-approximate approach. The effectiveness of these methods is validated through experiments with both real and synthetic data, demonstrating their capability to maintain accuracy while reducing the computational load.

The authors in [98] present a framework, RUTOD, developed for the real-time detection of outliers in urban traffic data streams. Utilizing Flink, RUTOD integrates Individual Outlier Detection (IOD) and Group Outlier Detection (GOD) to process streaming data. It employs a street-based grouping method for data preprocessing, feature-based outlier detection for IOD, and a density-based Local Outlier Factor (LOF) algorithm. The LOF algorithm [13] calculates a score for each data point based on the density of its local neighborhood compared to its neighbors' neighborhoods. Data points with a high LOF score are considered outliers. The processing of incoming trajectory streams is organized within each time window. Experiments indicate superior performance in throughput and latency compared to existing methods while maintaining comparable accuracy. When there are duplicates in the data points, the Weighted Local Outlier Factor (WLOF) algorithm can be used, as implemented for example in [99].

Clustering-Based Methods

Clustering-based techniques leverage standard clustering algorithms to distinguish outliers in the data. These techniques leverage the data inherent structure and density characteristics to differentiate outliers from regular data points. In these methods, data points that do not belong to or lie close to large or dense clusters are considered outliers.

One such algorithm that employs this approach is Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [28]. DBSCAN is a density-based clustering algorithm that identifies clusters as high-density regions separated by regions of lower density. It is particularly effective in detecting outliers as it does not require the specification of the number of clusters, and it can discover clusters of arbitrary shape, unlike many other clustering algorithms. Outliers in DBSCAN are identified as points that do not belong to any cluster.

Another algorithm that utilizes clustering-based techniques for outlier detection is Clustering Large Applications on Randomized Search (CLARANS) [77]. CLARANS is an algorithm for medoids (representative objects) based clustering, a partitioning-based clustering variant. The algorithm identifies outliers as points far from the medoids of any clusters. CLARANS has the advantage of being more robust to noise and outliers than other partitioning-based clustering algorithms.

Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH) [120] is a hierarchical clustering algorithm that can incrementally and dynamically cluster incoming multi-dimensional data points to produce the best quality clustering with the memory available and time constraints. In the context of outlier detection, BIRCH identifies outliers while building the Clustering Feature tree, a data structure used to summarize the information of data points in the dataset. Data points that do not fit well into the structure of the Clustering Feature tree are considered outliers.

K-Means [51] is a partitioning method that divides N observations into K clusters, each belonging to the nearest mean cluster. The algorithm operates iteratively and randomly selects K data points from the dataset. These points act as the initial partitions of the clusters. A Euclidean distance to each centroid is computed for the remaining data points, and the point is assigned to its nearest centroid. Once all points have been assigned to clusters, the partitions of the clusters are recalculated. This process continues until the partitions no longer move significantly or a set number of iterations is reached. K-means is an efficient and straightforward method, particularly suitable for large datasets. However, the algorithm has few limitations with complex geometrically

distributed data. It is also sensitive to the initial choice of partitions and can fall into local minima. Furthermore, the number of clusters K needs to be specified beforehand, which can be a drawback if the data does not suggest a precise number of clusters.

Clustering-based methods could work with long trajectories with possibly sparse or irregularly sampled locations. For example, in [27], the algorithm partitions a long trajectory into line segments and summarizes those segments without affecting the spatial properties of the original trajectory. In addition, in [122] the authors propose the Crowdsourcing Trajectory Outlier Detection, which employs an adaptive spatial clustering algorithm to remove location offset points in trajectory sequences. Additionally, the paper introduces a 6-dimensional movement feature vector for each trajectory point. It utilizes a Temporal Convolutional Network Autoencoder with a Squeeze-and-Excitation channel attention mechanism to enhance detection.

Graph-Based Methods

Graph-based techniques are a set of methodologies that leverage the structure of graphs to capture the relationships between interconnected data points. These techniques are effective in identifying outliers. Points are constituted as nodes, and their relationships are represented as edges. The weight of an edge typically represents the similarity or distance between two data points. The graph thus formed encapsulates the data structure, and the properties of this graph are used to identify outliers. Outliers are often identified as nodes that have few connections or are connected to other nodes with weak relationships. In other words, outliers are typically data points that do not fit well into the graph structure.

One common graph-based technique for outlier detection is the Spectral Clustering algorithm [76]. This algorithm uses the eigenvalues of the Laplacian matrix of the graph, a matrix representation of the graph, to identify clusters of nodes. Outliers are often found as nodes that do not belong to these clusters.

Heuristic-Based Methods

Heuristic-based techniques focus more on detection than correction. For instance, [123] does not replace outlier points with estimated values but instead removes them from the trajectory. Common heuristics are based on speed with the idea that if the speed change rate is significantly higher than a given threshold and a proportion of the points in the entire trajectory, the point is removed. These methods have the advantage of not introducing any estimated values into the trajectory, but they can lead to significant data loss.

In [22], a new method category based on physical movement properties is introduced, such as speed (Optimal Speed-bounded) and acceleration (Optimal Acceleration-bounded). This method defines limits on the minimum and maximum allowed values for these properties and uses them to determine whether a point in the trajectory is consistent with the model. The limits are minimum v - and maximum speed $v+$, minimum a -, and maximum acceleration $a+$. It follows the definition that from one point to its successor, there should always be inside $[v-, v+]$ and $[a-, a+]$. In addition, a trajectory $T = \langle p_1, \dots, p_n \rangle$ is consistent with the model if and only if there exists at least one point in a path such that the measurement coincides with the point and the speed and acceleration are inside the speed and acceleration bounds. The method defines a reachable region as a cone, i.e., given the physical boundaries, reaching the cone from point p_i to p_{i+1} is possible. In contrast, it is not necessarily possible to construct a trajectory from the concatenation of two consistent sub-trajectories: the concatenation $\langle p_1, \dots, p_n = q_1, \dots, q_m \rangle$ of two consistent subsequences $T = \langle p_1, \dots, p_n \rangle$ and $U = \langle q_1, \dots, q_m \rangle$ with $p_n = q_1$ is not necessarily consistent. Joining these sub-trajectories can reproduce inconsistent points, especially when considering an acceleration-bound model. The speed of two points can infer two accelerations for

the same position. The model is called concatenable if it is possible to join both sub-trajectories, respecting the bounds.

We have implemented in MEOS an open-source C library for mobility analytics [90], which utilizes a heuristic method for outlier detection by comparing maximum and minimum speeds.

Online Methods

Online methods are streaming methods that take into account the characteristics of systems that work with unbounded data. Traditional batch methods such as LOF compute a density-based score for each point by comparing its local density to that of its neighbors. LOF cannot meet the latency requirements of high-rate streams because it recomputes the k -neighbor graphs on every update.

The Incremental Local Outlier Factor (ILOF) algorithm, proposed by [88], addresses this issue by updating only those reachability distances and density estimates that are affected by each new point. ILOF achieves the same detection accuracy as repeated static LOF while reducing computation, but its cost grows with stream volume and it remains sensitive to the choice of k . The ILOF extends the LOF method into an online setting by maintaining density-based outlier scores as new observations arrive. Rather than recomputing LOF over the entire dataset for each insertion, ILOF updates only the k -NN and Reverse k -Nearest Neighbors (Rk-NN) relationships affected by the new point. When new points arrive, the algorithm computes their distances to existing instances, updates the neighbor lists for both directions, and identifies the subset of points whose local reachability distances and outlier scores require revision. It then recalculates reachability distances from each new point to its neighbors and from reverse neighbors back to the new point, propagates updates through the affected neighborhoods, and finally updates the local reachability densities and LOF scores only for those points that have changed. This incremental strategy achieves the same detection accuracy as rerunning the static LOF after each insertion but with far lower computational cost, since each update impacts only a small, locality-bounded subset of the data rather than the entire dataset.

The authors in [46] propose the Efficient Incremental Local Outlier Factor (EILOF) algorithm to manage the overhead of continuously recomputing LOF scores on streaming data. Unlike the standard ILOF, which updates reachability distances and densities for every point affected by each new record, EILOF computes LOF only for the incoming points and leaves existing scores unchanged. This append-only update reduces computation and memory pressure, yet retains detection accuracy under noise. In simulation and on real-world datasets, EILOF matches or outperforms ILOF in F1 score as the stream grows, while significantly reducing processing-time by orders of magnitude.

Memory-bounded variants, such as MiLOF, cap storage by summarizing older points, trading off some accuracy to control resource use. More recent work in streaming outlier detection has explored sliding window Isolation Forests and grid-based density methods. However, these often lack formal guarantees or struggle with dynamic, heterogeneous mobility traces.

3.2.4 Lifted Operations

Lifting is a concept in mathematics and computer science in which a function defined in a specific domain is extended to a more complex or higher-dimensional domain. In the context of MEOS [90], lifting refers to extending traditional operations such as arithmetic, comparison, and spatial functions to handle spatiotemporal data types. Lifted operations are essential, since they allow existing operations from PostgreSQL, PostGIS, and GEOS [21]¹ libraries to be used

¹<https://libgeos.org>

with spatiotemporal data. In addition, by lifting existing functions, MEOS can leverage the robust and well-tested functionalities of PostgreSQL, PostGIS, and GEOS, ensuring reliability and consistency in operations.

When an operation is lifted, it is adapted to accept temporal arguments, such as temporal points and temporal intervals, and return results that are also temporal. For example, traditional arithmetic operations such as addition and multiplication can be lifted to work on temporal numbers. This means that instead of adding two static numbers, the operation can add two temporal sequences of numbers, producing a temporal sequence of sums.

Comparison operations, such as equality and greater-than, can be lifted to work on temporal data. For instance, one can check if one moving object was ever at the same location as another object over time. Spatial functions such as distance and intersection from PostGIS can be lifted to calculate distances and intersections over time, not just at a single point.

Lifting extends traditional static functions into the temporal domain. This process involves adapting a static function to accept temporal types as arguments and return results as temporal types. For instance, basic arithmetic operators addition (+), subtraction (-), multiplication (*), and division (/) typically require each operand to be either an integer or a float and return results of the same type. When lifted, these operators can work where one or both operands are temporal numbers, resulting in a temporal number as the outcome. Similarly, lifting the standard distance function, commonly implemented in PostGIS as the `ST_Distance` function, enables it to accept either temporal or static geometries as inputs and produce a temporal float that quantifies the distance over time.

MEOS incorporates a comprehensive infrastructure that supports lifting various functions and operators applied to temporal types, including arithmetic, boolean, comparison operators, and spatial relationship functions. To lift a function, one must consider the number of arguments the function accepts. Unary functions, which include operations such as converting degrees on temporal floats or adjusting precision on temporal points, take a single argument. Binary functions, which include arithmetic, comparison operators, such as = and <, and spatial functions, such as `tintersects`, require two arguments. In binary functions, the argument types can combine a temporal and a base type, such as adding a temporal float to a static float or two temporal types. This combination entails synchronization of the temporal instants before applying the function.

Some functions may require additional parameters, such as spatial functions that require parameters to define proximity or containment. The result type, such as integer, float, boolean, can vary based on the function and whether the output is linear or stepwise. Additionally, whether the function is symmetric, such as addition, or not symmetric, such as subtraction, affects how arguments are handled.

For functions involving two temporal types or a temporal and a base type, it may be necessary to add turning points at crossings or local maxima. Lifting a function, such as the temporal distance operator, starts with defining the base function to be lifted, for instance the `ST_Distance` in PostGIS for geometrical points.

We utilize the temporal distance operator defined by the distance function to demonstrate the methodology for function lifting. PostGIS provides this function with the signature `ST_Distance(geo, geo):float`, implemented to handle arguments of types geometry and geography. For example, suppose we need to calculate the distance between one static point and one temporal point. We can create additional functions, such as `distance_tgeo_point(tgeo, point):tfloat`, suitable for both geometric and geographic point data. This approach demonstrates the need to extend base functionality through lifting to accommodate spatial computations.

Once the base function is identified for lifting, in this case, `ST_Distance`, it becomes necessary to define additional functions to manage data where one of the arguments is temporal or all

arguments are temporal. We focus on the first scenario, where one argument is temporal, and the other is a base type. In this case, it is necessary to introduce a method for adding an instant between the distances, known as a turning point (see Fig. 3.1). The method facilitates extending non-temporal spatial functions into the temporal domain.

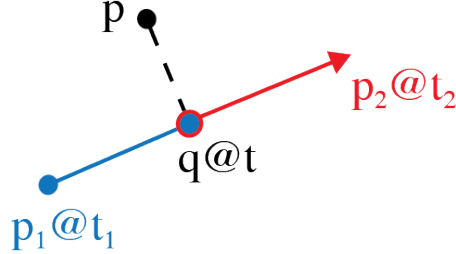


Figure 3.1: Turning-point insertion for a lifted distance computation when one argument is temporal, adding an intermediate point $q@t$

The turning point is added at timestamp t when the distance between point p and a segment defined by the points p_1 and p_2 reaches its minimum. To effectively calculate this, the sequence is represented by three distinct instants: $ST_Distance(p_1, p)@t_1$, $ST_Distance(p, q)@t$, and $ST_Distance(p_2, p)@t_2$. In MEOS, a specialized function named `tgeo_geo_min_dist_at_timestamptz` is utilized to return the distance and timestamp at which a temporal geometry segment and a point are at the minimum distance.

When both arguments are temporal points, a different approach is required. The initial step involves synchronizing the two temporal points to ensure their movements are aligned for comparison. This synchronization allows for a meaningful computation of distance considering both points' temporal evolution. Such synchronization often involves interpolating the positions of the points at equivalent timestamps or adjusting the temporal resolution of one or both points to match (see Fig. 3.2).

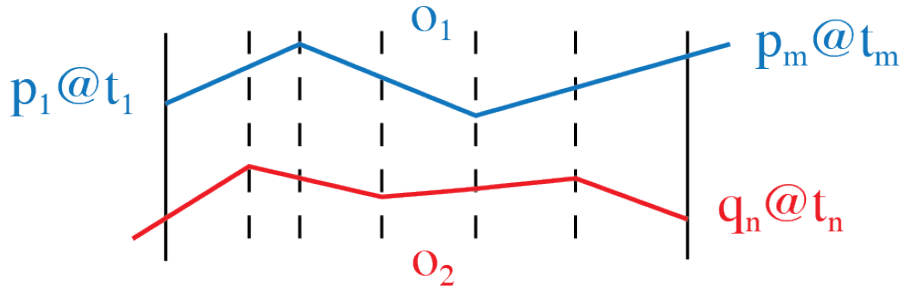


Figure 3.2: Two temporal points are synchronized by restricting them to a common time period and aligning instants for comparison

Synchronization involves checking the alignment of two temporal points. This is accomplished by first restricting the two arguments to their common period and filtering out periods that do not overlap between the two points. This step guarantees that the analysis focuses solely on

the time intervals where both temporal points have defined values. Following this, MEOS adds intermediate points to both arguments where necessary. The next step involves the addition of turning points to the synchronized segments. Figure 3.3 shows how to add turning points to a pair of synchronized segments. Here we have two objects o_1 and o_2 . The first object moves from $p_1@t_1$ to $p_2@t_2$, while the second object moves from $p_3@t_1$ to $p_1@t_2$. The location of the two objects defines the turning point at time t , at which point the distance between the two objects is minimal. Therefore, the result of the temporal function for this segment would be composed of three instants: $[ST_Distance(p_1, p_3)@t_1, ST_Distance(p, q)@t, ST_Distance(p_1, p_2)@t_2]$.

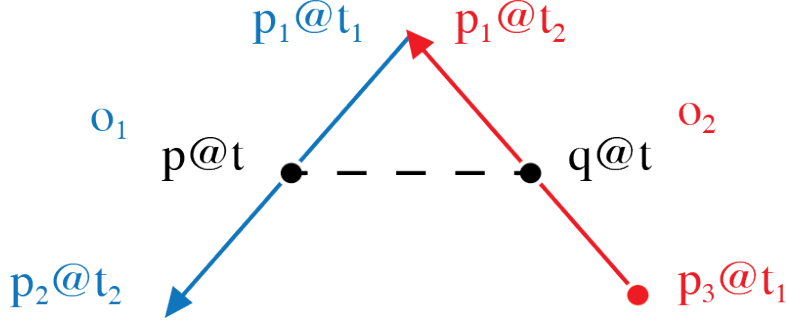


Figure 3.3: Turning points $p@t$ and $q@t$ are added after synchronization to capture the time of minimum distance between two moving segments

The lifted operators introduced above are instantiated under event-time semantics through the window-aware, sequence-based types *stinstant*, *stsequence*, and *stsequenceset*, enabling trajectory analytics to run over unbounded data streams. In our implementation, we use MEOS lifted operations for temporal and spatiotemporal types and expose them inside the streaming engines in MobilityNebula and MobilityFlink.

3.3 Chapter Summary

In this chapter, we introduced the background of stream processing and mobility data. We defined the stream abstraction, event-time, processing-time, and windowing. We then explained why out-of-order arrivals are intrinsic to mobility and IoT streams, and summarized the progress mechanisms used to manage them, including watermarks and allowed lateness.

We explain stream query languages and operators, DAG-based execution models, and the runtime mechanisms that provide correctness and performance. Throughout the chapter, we contrasted how Flink and NebulaStream realize these concepts, clarifying the implications for deployment: Flink typically assumes a homogeneous cloud setting, whereas NebulaStream explicitly targets heterogeneous resources and operator placement across the edge-to-cloud continuum.

We introduced mobility data and spatiotemporal streams. We summarized spatiotemporal data types and mobility continuous queries, reviewed outlier-detection approaches for static and streaming trajectory data, and explained lifted operations as the mechanism used by MEOS to extend standard arithmetic and spatial functions to temporal and spatiotemporal types. These concepts provide the foundation for Chapter 4, where we formalize a window-aware, sequence-based spatiotemporal model and map mobility operators onto streaming run times.

Chapter 4

Distributed Edge-to-Cloud Mobility Data Stream Processing

Continuous spatiotemporal queries over mobility streams require data types and operators that neither general stream processing engines nor spatiotemporal databases provide on their own. Stream engines deliver the execution model, including windowing, event-time processing, and distributed scheduling, but treat every record as a flat tuple without spatial or temporal structure. Spatiotemporal systems offer trajectory representations and geometric predicates, but assume finite and historical datasets and cannot sustain unbounded ingestion with low latency. Bridging this gap requires a formal model that embeds mobility semantics into the streaming execution pipeline, along with an architecture that can run the resulting operators on hardware ranging from edge devices to cloud nodes.

This chapter formally describes the main contributions of this work - MobilityNebula system. It presents an approach to implementing distributed mobility data streams processing across the edge-to-cloud continuum. We target scenarios where sensors on moving objects emit continuous readings that must be processed with low latency and limited resources. Our goal is to provide mobility operations within a streaming system that can function on constrained edge devices and scale as needed. These operations include geofencing, proximity, trajectory creation, and derived metrics.

We first formalize the data model implemented in our system. Building on the Moving Objects Database (MOD) framework, we adopt the abstract model and a discrete, sequence-based representation, then extend it with window-aware temporal types (*stinstant*, *stsequence*, *stsequenceset*).

Instead of building a new stream processing engine from scratch, we adopt a systems-integration approach. By leveraging established streaming runtimes for execution, scheduling, and distribution, we introduce the necessary mobility layer for spatiotemporal stream processing. This engineering decision enhances robustness and reproducibility, enabling us to concentrate research efforts on addressing the core technical challenge of reconciling streaming execution models with spatiotemporal trajectory semantics through window-aware types, trajectory construction, and operator mappings. We instantiate this approach on two representative systems (Sect. 4.2): *MobilityFlink*¹ and *MobilityNebula*.² We first implemented the proposed window-aware discrete model in MobilityFlink, an open-source geospatial trajectory stream processing prototype built on Apache Flink. MobilityFlink exposes MEOS spatiotemporal data types and operations inside

¹<https://github.com/MobilityDB/MobilityFlink>

²<https://github.com/MobilityDB/MobilityNebula>

Flink via JMEOS, an intermediate layer that bridges Java operators and the MEOS C API. The system supports continuous queries over spatiotemporal point and trajectory streams, including spatial filtering and distance computations, in a cluster-based cloud setting.

MobilityNebula introduces the same spatiotemporal capabilities into NebulaStream and enables operator placement at the edge, in the fog, or in the cloud. Next, we define the spatiotemporal operator set (map, filters, joins, aggregations, and trajectory analytics). We specify event-time semantics for windowed processing, including watermark-driven window finalization and our policy for handling late events.

This chapter evaluates the systems and methods we propose from a systems perspective. We assess MobilityNebula under real deployment conditions, focusing on its ability to sustain real-time spatiotemporal processing across edge and fog devices. We measure throughput, query selectivity, latency, CPU and memory usage, and network traffic for the continuous queries introduced.

We examine how operator complexity and output selectivity affect resource consumption and performance. We study the behavior of MobilityNebula under strict resource caps and compare it against related spatial streaming systems to understand the limits of edge deployability and robustness in constrained environments. We stress-test MobilityNebula under network volatility (jitter, bursts, and outages) to characterize robustness under edge connectivity disruptions.

4.1 Formal Model

The data types used in this thesis follow the model introduced in the MOD framework [42]. This framework defines two layers: an abstract model that captures movement semantics and a discrete model that implements it.

We adopt the MOD model with the abstract model from [43, 31] and the discrete model from MobilityDB [90]. The streaming extensions to support real-time data are inspired by the model introduced in [36].

4.1.1 Abstract Model

The abstract model proposed in [42, 43] introduces a type system for representing spatiotemporal data. This system includes constructors designed to represent spatial, temporal, and base data types, enabling their combination to model the evolution of values over time. For example, a *moving point* represents a spatial location whose position continuously changes through time, while a *range* captures time intervals or numeric values. This conceptual foundation effectively describes object movements over time, although it does not define methods for data storage or query execution.

Table 4.1 summarizes the abstract type constructors for the type system, adapted from [42]. Each type is composed of a constructor and a category. Depending on the data type, it can also have a signature input. For example, applying the *moving* constructor to a spatial data type and a base type yields a temporal object whose spatial coordinates evolve continuously with time.

4.1.2 Discrete Model

The discrete model implements the abstract model by defining explicit strategies for storing, updating, and querying spatiotemporal data. The discrete model decomposes the abstract *moving* constructor into orthogonal dimensions of value domain and temporal representation. Two representations are discussed in the literature: the *sliced representation* and the *sequence representation*.

Table 4.1: Abstract type constructors in the MOD framework, showing each constructor, its signature, and its type category

Type Constructor	Signature	Type Category
<i>integer, float, text, bool</i>		$\rightarrow \text{BASE}$
<i>geometry, geography</i>		$\rightarrow \text{SPATIAL}$
<i>instant, interval</i>		$\rightarrow \text{TIME}$
<i>moving</i>	$\text{BASE} \cup \text{SPATIAL}$	$\rightarrow \text{TEMPORAL}$
<i>range</i>	$\text{BASE} \cup \text{TIME}$	$\rightarrow \text{RANGE}$

The *sliced representation*, as implemented in SECONDO [41] and MobyDick [36], decomposes a trajectory into a series of units. Each unit stores the segment between two consecutive observations together with an interval. Although this representation is conceptually straightforward, it inadvertently duplicates boundary events, thereby contributing to elevated storage requirements and increased I/O costs.

This thesis is based on the MobilityDB type model called *sequence representation* [90]. The *sequence representation* models the trajectory as a chain of instants, storing every observation once and introducing a split only when there is a temporal gap. This method eliminates repeated bounds embedded in sliced units. In addition, the sequence design inherently incorporates temporal continuity within the type. As a result, operations can eliminate the continuity checks that sliced algorithms require. MobilityDB implements the following constructors: *instant*, *tsequence*, and *tsequenceset*.

In the abstract MOD model, the constructor *moving* captures the semantics of a value that evolves over time, for either base or spatial domains. The discrete model refines this notion by introducing explicit *representations* for temporal values—*instant*, *tsequence*, and *tsequenceset*—which materialize *moving* objects as sampled observations (instants) and their organization into continuous segments (sequences) and sets of segments (sequence sets). These representations apply uniformly to both base-typed and spatial-typed temporal values (e.g., *tint* and *tgeometry*).

To support continuous data ingestion, we extend this sequence representation with the concept of *temporal windows* [36]. This extension results in *windowed spatiotemporal sequences* (see Fig. 4.1), where each spatiotemporal sequence is coupled with its associated window. Consequently, we introduce the *stinstant*, *stsequence*, and *stsequenceset* types, binding each observed spatial point or sequence to its corresponding time interval. Table 4.2 summarizes the type system, function signature, and category. For example, we define a *stsequence* by applying a window to a temporal type, such as a *tsequence* of points.

Example

Consider the GPS measurements in Table 4.3 to demonstrate the discrete model in practice. Each longitude, latitude, and timestamp tuple is lifted into a single value, following the *instant* and *tsequence* discrete models [90]. For example, a record with coordinates (4.3411, 50.8358) at a timestamp 2024-09-22 22:29:03 becomes

`Point(4.3411 50.8358)@2024-09-22 22:29:03`

³In MobilityDB, *instant*, *tsequence*, and *tsequenceset* are representations (subtypes) of temporal types over base domains (e.g., *tint*) and spatial domains (e.g., *tgeometry*).

Table 4.2: Discrete type system used in this thesis, including MobilityDB temporal types and the proposed window-aware spatiotemporal types

Type Constructor	Signature	Type Category
<i>integer, float, text, bool</i>		$\rightarrow$ <i>BASE</i>
<i>geometry, geography</i>		$\rightarrow$ <i>SPATIAL</i>
<i>timestampz, tstzset, tstzspan, tstzspanset</i>		$\rightarrow$ <i>TIME</i>
<i>tinstant, tsequence, tsequenceset</i> ³	$(\text{BASE} \cup \text{SPATIAL}) \times \text{TIME}$	$\rightarrow$ <i>TEMPORAL</i>
<i>tbool, tint, tfloat, ttext</i>	$\text{BASE} \times \text{TIME}$	$\rightarrow$ <i>TEMPORAL</i>
<i>tgeometry, tgeography, tgeompoint, tgeogpoint</i>	$\text{SPATIAL} \times \text{TIME}$	$\rightarrow$ <i>TEMPORAL</i>
<i>now, unbounded, range</i>		$\rightarrow$ <i>WINDOW</i>
<i>stinstant, stsequence, stsequenceset</i>	$\text{TEMPORAL} \times \text{WINDOW}$	$\rightarrow$ <i>STEMPORAL</i>
<i>stbool, stint, stfloat, sttext</i>	$(\text{BASE} \times \text{TIME}) \times \text{WINDOW}$	$\rightarrow$ <i>STEMPORAL</i>
<i>stgeometry, stgeography, stgeompoint, stgeogpoint</i>	$(\text{SPATIAL} \times \text{TIME}) \times \text{WINDOW}$	$\rightarrow$ <i>STEMPORAL</i>

Each input record is represented as a single *tinstant*, while a *tsequence* consists of multiple *tinstant*. For example, *tinstants* such as

```
{Point(4.3411 50.8358)@2024-09-22 22:29:03,
 Point(4.3431 50.8500)@2024-09-22 22:29:13,
 Point(4.3410 50.8359)@2024-09-22 22:30:13}
```

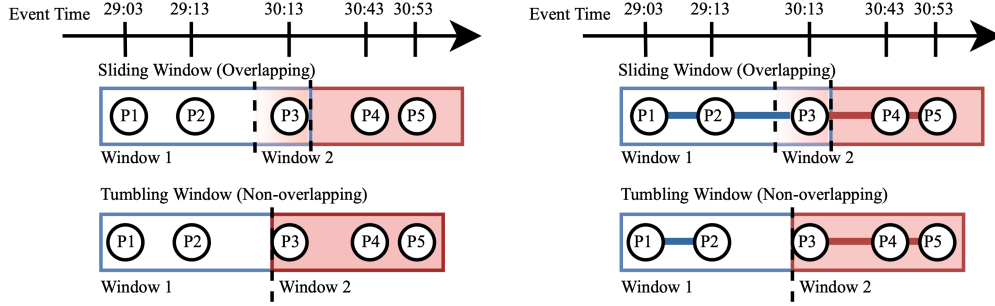
Figure 4.1 shows how applying different windowing schemes on these records produces *stinstants* and *stsequence* values. We demonstrate this with time-based sliding window and tumbling window strategies. The left side of Fig. 4.1 shows the creation of *stinstants*. In the sliding-window strategy, overlapping intervals capture a subset of *tinstants*: Window 1 contains the first three *stinstants*. Window 2 contains an overlapping *stinstant* with Window 1 and the last two *tinstants*. The tumbling window partitions the stream into non-overlapping intervals, grouping batches of *stinstants* into separate groups with no overlap. The right side of Fig. 4.1 shows how each window concatenates its *tinstant* to create *stsequences*. In the sliding-window strategy, Window 1 contains the sequence (p_1, p_2, p_3) and Window 2 contains (p_3, p_4, p_5) . The instant p_3 belongs to both sequences because the windows overlap. However, within each *stsequence* every observation appears exactly once and no edge crosses a window boundary. In the tumbling-window strategy, Window 1 produces (p_1, p_2) and Window 2 produces (p_3, p_4, p_5) , with no shared instants across windows.

4.2 Representative Systems

To demonstrate the proposed window-aware mobility model without building a new streaming runtime from scratch, we adopt a systems-integration approach and realize the mobility layer on top of two representative systems. MobilityFlink serves as a cloud-oriented prototype that validates the data model, the formal type system, and the mapping of spatiotemporal operators onto a mature streaming engine. NebulaStream represents the edge-to-cloud setting and provides the placement and resource-awareness needed to execute the same spatiotemporal operators close

Table 4.3: Example of lifting raw GPS tuples (longitude, latitude, timestamp) into MobilityNebula encoding (*tinstant*)

Longitude	Latitude	Timestamp	MobilityNebula encoding (<i>tinstant</i>)
4.3411	50.8358	2024-09-22 22:29:03	Point(4.3411 50.8358) @2024-09-22 22:29:03
4.3431	50.85	2024-09-22 22:29:13	Point(4.3431 50.85) @2024-09-22 22:29:13
4.341	50.8359	2024-09-22 22:30:13	Point(4.341 50.8359) @2024-09-22 22:30:13
4.3413	50.8358	2024-09-22 22:30:43	Point(4.3413 50.8358) @2024-09-22 22:30:43
4.3409	50.8359	2024-09-22 22:30:53	Point(4.3409 50.8359) @2024-09-22 22:30:53

Figure 4.1: Sliding and tumbling windowing strategies for constructing *stinstant* and *stsequence* values from points

to data sources. The MobilityNebula extension brings the mobility types and operators into this environment. Together, these two systems ground the implementation choices made in the rest of the chapter and provide the basis for the operator definitions and performance evaluation that follow.

4.2.1 MobilityFlink

In this work, we have first implement the data model in MobilityFlink.⁴ MobilityFlink is an open-source geospatial trajectory data streaming platform based on Flink. It exposes the MEOS spatiotemporal data types and operations into Flink runtime via JMEOS, an intermediate layer between Java operators and the MEOS C++ library. MobilityFlink is designed to perform continuous queries over trajectory streams, including filtering and distance computations, on large, high-velocity datasets.

Figure 4.2 shows the MobilityFlink system overview. MobilityFlink integrates MEOS into Flink through multiple wrappers. JMEOS, a Java wrapper for MEOS, is composed of the outer wrappers, function wrappers, and Java Abstracted Foreign Function Layer (JNR-FFI). A stream

⁴<https://github.com/MobilityDB/MobilityFlink>

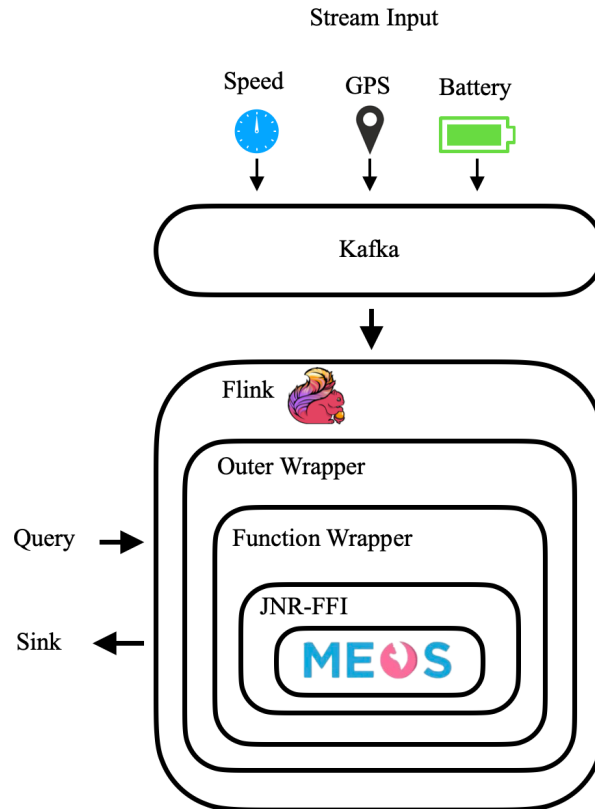


Figure 4.2: Overview of MobilityFlink, showing the integration of MEOS via JMEOS and the Kafka-Flink execution pipeline

of data measurements is produced and stored in Kafka. MobilityFlink consumes this Kafka stream and executes mobility queries on this data.

Adaptations and Implementations in MobilityFlink

MobilityFlink required an additional layer to reconcile Flink JVM-based operator runtime with MEOS C API. We use JMEOS as the bridging layer to convert arguments into MEOS format and return values from its operations in Java. This integration introduced deployment constraints, as the native MEOS shared library needed to be packaged and made accessible to Flink TaskManagers.

An adaptation we made was the representation and conversion of event-time to ensure that Flink records could be processed by MEOS. Flink uses timestamps as epoch milliseconds, whereas MEOS temporal constructors require formatted timestamps consistent with the MEOS input format. MobilityFlink introduces an explicit conversion step (from milliseconds to a timestamp string) and standardizes the time zone at initialization (e.g., `meos_initialize("UTC")`) to prevent inconsistencies arising from host-local defaults. We adopt MEOS point encoding (such as `POINT(lon lat)@ts`), and we use JMEOS constructors (such as `TGeomPointInst` and `TGeomPointSeq`) to parse these representations into temporal objects.

Translating MEOS functions into Flink operators required mapping static mobility semantics to Flink’s programming model, in which spatiotemporal operations are implemented as either

stateless User-Defined Functions (UDFs) (e.g., per-event filtering), or as stateful windowed operators (e.g., trajectory assembly and window-bounded analytics). In practice, point creation is implemented as a `MapFunction` that transforms raw tuples into temporal instants. Spatial and spatiotemporal predicates, such as intersection or containment into an `STBox`, are implemented as `FilterFunction` or as predicate checks within incremental aggregators. Trajectory construction is achieved through keyed event-time windows that materialize a `TGeomPointSeq` for each (key, window) pair. For example, we implement a Flink `AggregateFunction` whose method invokes the corresponding MEOS predicate via JMEOS, embedding spatiotemporal reasoning directly into the aggregator update.

The primary implementation challenges originated from a mismatch between the JMEOS research prototype and Flink’s operational requirements. First, JMEOS was not updated to MEOS version (1.3), necessitating iterative debugging of missing or outdated wrappers and validation of function signatures. During early implementation, this led to segmentation faults due to incorrect parameter number or improper null handling. Second, Flink may deliver events out-of-order under event-time semantics, whereas MEOS sequence constructors require consistent temporal ordering. To address this, watermark-driven window finalization was used, and explicit ordering was implemented at window finalization to ensure that each sequence is ordered by event-time, regardless of arrival order.

Containerized Testing Environment

To test the full MobilityFlink pipeline, we set up a containerized testing environment. The pipeline consists of three Docker images with Kafka, Flink, and MobilityDB, deployed as a single stack and connected internally via a Docker bridge network (see Fig. 4.3). The implementation and scripts are available in the MobilityFlink query repository.⁵

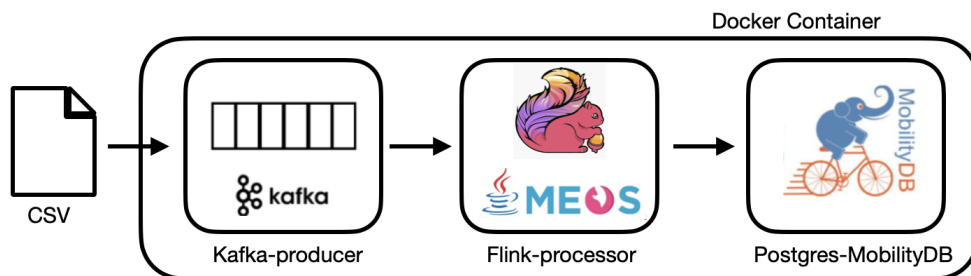


Figure 4.3: Containerized MobilityFlink test setup, showing Kafka ingestion, Flink processing, and the mobility pipeline dataflow

Data is generated by a Kafka producer that reads a CSV file and publishes records to a Kafka topic. Flink consumes directly from the Kafka topic, deserializes each message into the internal record representation, and then assigns the record based on event time using a watermark strategy. We introduced a wait script during startup to avoid race conditions between Kafka and Flink, using a readiness timer that delays launching the Flink job until Kafka starts producing tuples.

We exemplify this integration with a spatiotemporal aggregation query. We count the number of distinct vessels in a specified geographic area and period. We use spatial filtering with bounding boxes and temporal filtering. This query is implemented as a windowed aggregation over event-time. In our configuration, we use a ten s sliding window and compute the count by

⁵<https://github.com/marianaGarcez/FlikPointQueries/tree/main/2>

evaluating whether each point lies within the target spatiotemporal bounds, implemented as a predicate inside `CountAggregator()`. The accumulator update calls the MEOS intersection test (via `JMEOS`) to determine whether the incoming temporal point intersects the spatial bounding box.

While `MobilityFlink` provides a stable environment for validating correctness and integration overheads, it remains tailored to cluster-based deployments and does not directly address edge constraints, such as bounded CPU/memory footprints, reduced network round-trips, and continuous low-latency results. To meet these requirements, we designed `MobilityNebula` on top of `NebulaStream`, enabling distributed execution and operator placement along the edge-to-cloud continuum.

4.2.2 MobilityNebula

The implementation of `MobilityFlink` validated the feasibility of MEOS spatiotemporal operators in a cloud stream engine, but it did not meet our requirements for edge processing. In particular, our target scenario requires executing continuous spatiotemporal queries close to the data source under tight CPU and memory budgets, while sustaining low and predictable latency. This motivated *MobilityNebula*, which integrates MEOS into `NebulaStream` to enable distributed execution and explicit operator placement across edge, fog, and cloud nodes.

The goal of `MobilityNebula` is to support MOD trajectory stream processing under tight CPU and memory budgets on edge devices. To realize this, the system integrates MEOS, a C library for spatiotemporal data management, into `NebulaStream`, an edge-to-cloud streaming platform, so that trajectory types and operators can run as streaming operators on constrained hardware in IoT environments.

As illustrated in Fig. 4.4, `MobilityNebula` integrates MEOS into `NebulaStream` through function bindings that extend `NebulaStream` expressions and operators with spatiotemporal capabilities. `MobilityNebula` continuously executes mobility queries on this data before forwarding the results to downstream sinks for persistent storage and later offline analysis.

`MobilityNebula` incorporates the discrete spatiotemporal type system defined in Sect. 4.1 into `NebulaStream`'s execution framework [24]. The system exposes several MEOS functions as `NebulaStream` operators. These functions are invoked through a `NebulaStream` plugin that wraps the MEOS C API. Incoming records are arranged into *stinstant*, *stsequence*, or *stsequenceset* objects, which are passed to MEOS by reference pointers. The results are then emitted back into the data flow. Since MEOS functions appear as regular `NebulaStream` operators, queries can combine them with windowing and operators such as `map` and `filter`.

The main technical challenge is defining mobility data types and operators that can integrate MEOS spatiotemporal types and functions into `NebulaStream`'s low-latency engine while operating under strict resource constraints. `MobilityNebula` must integrate mobility data functions and operators into the streaming pipeline without incurring significant overhead. Additionally, it must not impact the dynamic distribution of operators across heterogeneous levels of the edge-to-cloud continuum.

In `MobilityNebula`, spatiotemporal *stsequence* and *stsequenceset* are produced by a trajectory aggregator (see Sec. 4.4.2 for implementation). For each key, such as each train, incoming records are assigned to windows and incrementally added into a window-bounded sequence using `NebulaStream`'s `lift`, `combine`, and `lower` functions. `Lift` creates a *tinstant* by combining longitude, latitude, and timestamp in one record. `Combine` merges two *tinstant*s by concatenating them. `Lower` creates the *stsequence*/*stsequenceset* and outputs it. `Lower` can also be used to extract metrics from the *stsequence*, such as the average speed or minimum speed.

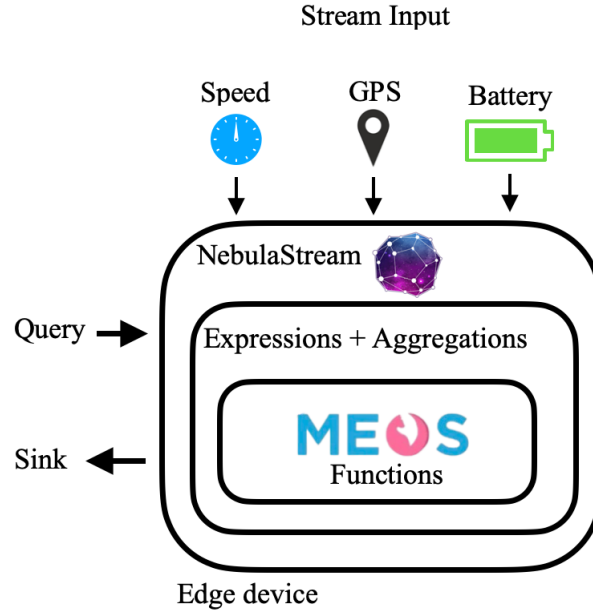


Figure 4.4: High-level architecture of MobilityNebula, showing MEOS integration in NebulaStream and the query dataflow

Adaptations and Implementations in MobilityNebula

Initial development of MobilityNebula was based on a NebulaStream version that separated operations into expressions, scalar functions, and aggregations. MEOS functions were mapped to these categories, enabling MEOS predicates to execute within map or filter operators and MEOS trajectory logic to operate within windowed aggregations. However, several mobility routines require more than two inputs. For instance, point construction requires at least longitude, latitude, and timestamp, and intersection functions require one or two points, resulting in three or six parameters. Since the NebulaStream logical function layer was originally designed for unary or binary functions, we have extended it to support higher-arity operators. The new signatures were integrated into the logical planning layer to enable type checking and compilation, and into the parser to accept these parameters and send them upstream.⁶

Build and deployment issues with the MobilityNebula were resolved. On macOS, we improved library path handling ensuring NebulaStream could find MEOS during development. In addition, to avoid making MEOS a core engine dependency, we packaged it as an optional plugin activated through a CMake option (e.g., `MEOS=ON`). This kept the NebulaStream default build unchanged while enabling MobilityNebula builds to use MEOS in several core components.

NebulaStream team later introduced a refactored code with a new interface that both made operator integration more uniform and moved the development workflow to Docker. We migrated our bindings to the new plugin API, updated operator registrations to match the new interfaces, and added MEOS to the development Docker image, ensuring reproducibility across machines. This refactoring reduced the amount of core code we needed to touch, simplified adding new MEOS operators, and, while beneficial, still required us to revalidate and reconstruct function signatures, build flags, and native library loading in the containerized runtime.

⁶This code is available in my branch of NebulaStream-private project

4.3 Spatiotemporal Stream Operations

Spatiotemporal stream operations specialize traditional streaming operators (*filter*, *map*, *window*, *aggregate*, *join*) to work on windowed spatiotemporal types, such as *stinstant*, *stsequence*, and *stsequenceset*, under event-time semantics. In order to expose MEOS spatiotemporal functionality as NebulaStream spatiotemporal stream operations, we implement in MobilityNebula a binding workflow across the engine's extension points. For each supported MEOS operations, we add a C++ wrapper that encapsulates the MEOS C API call and performs conversions, a logical function that integrates the operator into NebulaStream's logical plan (including typing and expression semantics), a physical function that implements the runtime execution path, and a parser entry that makes the function available in NebulaStream query language. Table 4.4 summarizes these additions for a representative subset of MEOS functions.

These layers define the operator set that MobilityNebula makes available to users when writing continuous queries. The resulting operator set combines stateless, point-level spatiotemporal predicates over *stinstant* values with stateful, windowed operators that assemble *stsequence* and *stsequenceset* objects and compute mobility-specific aggregates and analytics under event-time semantics. The following taxonomy groups the available operators by their role in a query plan from spatial and proximity checks to joins, trajectory construction, and post-aggregation analytics.

Mobility Operator Set

- **Map and derivation on data points (stateless)** Generates spatiotemporal records by computing derived features for each *stinstant*. For example, speed and heading.
- **Spatial range and geofence filter (stateless)** Retain or discard points depending on the spatial predicates with respect to static or dynamic regions. For example, `tgeo_at_stbox`, `eintersects_tgeo_geo`, `edwithin_tgeo_geo`, or nearest-approach distance (`nad_tgeo_stbox`). Common use: Location-based filtering prior to stateful operators.
- **Proximity and distance thresholding (stateless)** Keep points that are closer than a certain threshold from the geometry or moving object. It is used in range and proximity queries, for instance. Example usage: hazard-zone proximity.
- **Spatiotemporal joins (stateful)** Stream-static polygons joins (e.g., points $\bowtie$ polygons), and stream-stream joins (e.g., points $\bowtie$ weather) over a temporal window and spatial predicate (within/intersects/distance). Common usages: collision detection of one or two moving objects.
- **Trajectory assembly (stateful)** Collect object points of *stinstant* in a window into a *stsequence*.
- **Trajectory analytics (stateful)** On *stsequence*: length, average and maximum speed, stop and move segmentation, turn counts, bbox, similarity. Output can be scalars per window or derived trajectories.
- **Windowed aggregations (stateful)** Compute parameter **variation** and domain aggregation, such as average speed and length traveled, over *stinstant* groups and *stsequence*, grouped by object id and window.
- **Pattern and event detection (stateful)** Detect *enter/leave* events, *stay*, *crossing* of boundaries, or compound conditions. For example, high noise in a region, brake health, using Geospatial Complex Event Processing predicates over windowed inputs.

Table 4.4: Mapping of representative MEOS operations to MobilityNebula bindings (wrapper, logical, physical, parser)

Operation	Wrapper	Logical	Physical	Parser
edwithin_tgeo_geo	Geometry:: EdwithinStatic	EDWithin Geometry LogicalFunction	EDWithin Geometry PhysicalFunction	tedwithin_tgeo_geo
eintersects_tgeo_geo	Geometry:: EintersectsStatic	TIntersects Geometry LogicalFunction	TIntersects Geometry PhysicalFunction	eintersects_geometry
tgeo_at_stbox	Geometry:: AtStbox	TAt StBox LogicalFunction	TAt StBox PhysicalFunction	tgeo_at_stbox
nad_tgeo_stbox	Geometry:: NadStbox	NearestApproach Distance LogicalFunction	NearestApproach Distance PhysicalFunction	nad_tgeo_tgeo
variance_agg	N/A	VarAggregation LogicalFunction	VarAggregation PhysicalFunction	variance
tsequence	TSequence:: TSequence	TSequence Aggregation LogicalFunction	TSequence Aggregation PhysicalFunction	tsequence
ext_kalman_filter	TSequence:: EKF	TEKF Aggregation LogicalFunction	TEKF Aggregation PhysicalFunction	ext_kalman_filter
knn_agg	N/A	KnnAggregation LogicalFunction	KnnAggregation PhysicalFunction	knn_agg

- **Data cleaning (stateless/stateful)** Outlier filtering of points or trajectories, speed limits and Kalman Filter before aggregation.

4.4 Implementations

In this section, we show the implementation of two queries: filtering of data points based on distance from a fixed geometry and trajectory *stsequence* creation. We begin by demonstrating its implementation in MobilityFlink, and then proceed to MobilityNebula.

4.4.1 Implementation in MobilityFlink

Code 4. Distance-based filtering

```

1  DataStream<SNCB> source = rawStream
2  .map(new DeserializeAISDataMapFunction())
3  .assignTimestampsAndWatermarks(WatermarkStrategy.<SNCB>forBoundedOutOfOrderness(
       Duration.ofSeconds(10))
4  .withTimestampAssigner(new AISDataTimestampAssigner()))
5  .filter(new NadToStboxFilter(minLon, minLat, maxLon, maxLat, 3.0));
6
7  DataStream<TGeomPointSeq> trajectories = source
8  .map(new SNCBDataToTuple4MapFunction())
9  .keyBy(t -> t.f0)

```

```

10 .window(SlidingEventTimeWindows.of(Time.seconds(60), Time.seconds(3)))
11 .process(new TrajectoryWindowFunction());
12
13 trajectories.print();

```

The first part of Code 4 constructs the input stream and applies the distance-based filtering (Lines 1–5). Starting from the raw input stream, the `DeserializeAISDataMapFunction` deserializes each binary message into an `SNCB` record. We then attach event-time semantics by assigning timestamps and watermarks with a window of ten seconds, which tolerates slightly delayed updates. The `NadToStboxFilter` wraps the MEOS function `nad_tgeo_stbox` and computes, for every incoming spatiotemporal point, the nearest-approach distance to a predefined spatiotemporal box. Only records whose minimum distance is below three meters are retained.

The second part of Code 4 assembles trajectories from the filtered points (Lines 7–13). We first project each `SNCB` record into a tuple containing the object identifier, longitude, latitude, and timestamp. The stream is partitioned by identifier using `keyBy`, so that each key maintains its own independent state. Over each key, we then apply a sliding event-time window of 60 seconds with a slide of 3 seconds, which produces overlapping windows capturing recent movement. The `TrajectoryWindowFunction` (see Code 5) is invoked for every window to build a `TGeomPointSeq` trajectory from the points in that window. The resulting trajectories are emitted to the sink via `print()`.

Code 5. *stsequence* construction in Flink by concatenating encoded points per window

```

1 public class TrajectoryWindowFunction extends
2 ProcessWindowFunction<Tuple4<Integer, Double, Double, Long>, TGeomPointSeq, Integer,
   TimeWindow> {
3     @Override
4     public void process(Integer mmsiKey, Context context,
5         Iterable<Tuple4<Integer, Double, Double, Long>> elements,
6         Collector<TGeomPointSeq> out) {
7         StringBuilder trajbuffer = new StringBuilder();
8         trajbuffer.append("{");
9         for (Tuple4<Integer, Double, Double, Long> tuple : elements) {
10             String t = convertMillisToTimestamp(tuple.f3);
11             // Tuple: f0=mmsi, f1=lon, f2=lat, f3=timestamp
12             String str_pointbuffer = String.format("POINT(%f %f)@%s", tuple.f1, tuple.f2,
13                 t);
14             trajbuffer.append(", " + str_pointbuffer);
15         }
16         trajbuffer.append("}");
17         TGeomPointSeq trajectoryMEOS = new TGeomPointSeq(trajbuffer.toString());
18         out.collect(trajectoryMEOS);
19     }
20 }

```

Code 5 shows the implementation of `TrajectoryWindowFunction`, which creates spatiotemporal *stsequence* type from the points contained in a window. For each key-window pair, Flink provides an iterable collection of tuples (`mmsi`, `lon`, `lat`, `ts`). The function iterates over these tuples and, for every element, converts the millisecond timestamp into a textual representation and formats a point using the MEOS input syntax `POINT(lon lat)@timestamp`. All formatted points are appended to a buffer enclosed in braces, thereby constructing the textual representation of a temporal geometry sequence in the MEOS format.

Once the buffer contains the whole sequence, it is passed to the JMEOS wrapper constructor `TGeomPointSeq`, which parses the string into the corresponding MEOS object. The resulting `TGeomPointSeq` encodes a windowed trajectory (*stsequence*) with temporally ordered instants and is emitted via the collector. In this way, the window function bridges Flink native windowing abstraction and the sequence-based spatiotemporal representation used by MEOS.

4.4.2 Implementation in MobilityNebula

Code 6 shows a sliding-window query that selects points within three meters of a spatiotemporal box and prints any matches. Compared to the MobilityFlink implementation in Code 4, the MobilityNebula query exposes MEOS functions directly in the logical plan. In MobilityFlink, distance-based filtering is realized as a user-defined `NadToStboxFilter` operator applied after deserialization and watermark assignment, and trajectory construction requires an explicit windowed `ProcessWindowFunction`. In contrast, the MobilityNebula variant embeds the `nad_tgeo_stbox` call as an expression inside the `filter` operator and delegates window management to the `SlidingWindow` abstraction over event-time.

Code 6. Distance-based filtering

```
1 Query::from(SNCB)
2 .filter(nad_tgeo_stbox(Attribute("longitude"), Attribute("latitude"), Attribute("
   timestamp"))) < 3)
3 .window(SlidingWindow::of(EventTime(Attribute("timestamp")), Seconds(60), Seconds(3)))
4 .sink(PrintSinkDescriptor::create());
```

First, we specify the input stream in Code 7. In our deployment, SNCB refers to the real-time stream of spatiotemporal points from trains, including GPS coordinates, timestamps, speed, and pressure readings.

Code 7. Query source stream declaration

```
1 Query::from(SNCB)
```

We apply a distance-based filter (see Code 8). Here, we call the MEOS function `nad_tgeo_stbox`, passing the three attributes `longitude`, `latitude`, and `timestamp`. The function calculates the distance (in meters) between each incoming temporal point and a predefined spatiotemporal box. By using this expression, we retain only those points that are less than three meters away from the spatiotemporal box. Points outside this threshold are dropped and never enter the downstream processing.

Code 8. Distance to `stbox` predicate with a 3 m threshold

```
1 .filter(nad_tgeo_stbox(Attribute("longitude"), Attribute("latitude"), Attribute("
   timestamp"))) < 3)
```

We group the remaining points into overlapping time windows (see Code 9). We choose event-time so that each point timestamp defines its window assignment. We then specify a 60-second window duration with a three-second slide. Therefore, every three seconds, a new window is formed that overlaps the previous window by 57 seconds. All points whose `timestamp` lie within the current 60-second interval remain in that window until it closes.

Code 9. Sliding event-time window specification (60 s size, 3 s slide)

```
1 .window(SlidingWindow::of(EventTime(Attribute("timestamp")), Seconds(60), Seconds(3)))
```

We attach a print sink to emit the results whenever the window ends (see Code 10):

Code 10. Sink specification for printing

```
1 .sink(PrintSinkDescriptor::create());
```

Trajectory Creation

For creating trajectories in MobilityNebula, we utilize the NebulaStream aggregation function. NebulaStream divides the aggregation computation into lift, combine, and lower (see Sec. 3.1.11). In Code 11 we implement the lift function, where we create a record with **longitude**, **latitude**, and **timestamp** for each element in the window. The lift phase executes three physical field functions (**lonFunction**, **latFunction**, and **timestampFunction**) over the incoming **Nautilus::Record** and appends the resulting triple as a row into a **PagedVector** that stores a per-key, per-window aggregation state.

Code 11. *stsequence* construction in MobilityNebula (lift): append (**lon**, **lat**, **ts**) into per-window state

```
1 void TemporalSequenceAggregationPhysicalFunction::lift(
2     const nautilus::val<AggregationState*>& aggregationState, ExecutionContext&
3     executionContext, const Nautilus::Record& record)
4 {
5     const auto pagedVectorPtr = static_cast<nautilus::val<Nautilus::Interface::
6         PagedVector*>>(aggregationState);
7     ...
8     // Create a record with all three fields for temporal sequence
9     Record aggregateStateRecord({
10         {std::string(LonFieldName), lonValue},
11         {std::string(LatFieldName), latValue},
12         {std::string(TimestampFieldName), timestampValue}
13     });
14     const Nautilus::Interface::PagedVectorRef pagedVectorRef(pagedVectorPtr,
15         memProviderPagedVector);
16     pagedVectorRef.writeRecord(aggregateStateRecord, executionContext.
17         pipelineMemoryProvider.bufferProvider);
18 }
```

In Code 12, we implement the combine function. In this phase, we work with two-by-two aggregation states (e.g., produced by partial aggregation and merging across threads/operators). We then store the second state into the first by copying the content of one **PagedVector** into the other. In the implementation, **copyFrom** performs an append of all entries of the second vector into the first, resulting in a single logical multi set of (**lon**, **lat**, **ts**) values for the window.

Code 12. *stsequence* construction in MobilityNebula (combine): merge partial window states by appending vectors

```
1 void TemporalSequenceAggregationPhysicalFunction::combine(
2     const nautilus::val<AggregationState*> aggregationState1,
3     const nautilus::val<AggregationState*> aggregationState2,
4     PipelineMemoryProvider&)
5 {
6     // Get paged vectors from the aggregation states
```

```

7   const auto memArea1 = static_cast<Nautilus::val<Nautilus::Interface::PagedVector*>>(
    aggregationState1);
8   const auto memArea2 = static_cast<Nautilus::val<Nautilus::Interface::PagedVector*>>(
    aggregationState2);
9
10  // Combine two paged vectors
11  Nautilus::invoke(+[] (Nautilus::Interface::PagedVector* vector1, const Nautilus::
    Interface::PagedVector* vector2) -> void { vector1->copyFrom(*vector2); },
    memArea1, memArea2);
12 }

```

In Code 13, we finalize the window and emit the trajectory. The lower phase first ensures that MEOS is initialized, then reads the accumulated `PagedVector` and retrieves the field indices for lon, lat, and timestamp. If the window is empty, the function directly returns the sentinel string `BINARY(0)`. Otherwise, it builds a temporal instants string in MEOS input syntax after sorting the window contents by event-time. This ordering is executed in `buildSortedTemporalInstantSetString`, which extracts all triples from the aggregation state and sorts them by timestamp, making the resulting trajectory independent of arrival order.

Next, the implementation parses the sorted instants string into a MEOS temporal object and converts it to a binary. For test compatibility with NebulaStream test environment, the operator returns a `BINARY(N)` string (where N is the WKB byte size) rather than returning the raw WKB bytes. The temporary MEOS objects and buffers are freed before returning.⁷

Code 13. *stsequence* construction in MobilityNebula (lower): sort by event-time and emit MEOS temporal sequence

```

1  Nautilus::Record TemporalSequenceAggregationPhysicalFunction::lower(
2      const Nautilus::val<AggregationState*> aggregationState, PipelineMemoryProvider&
    pipelineMemoryProvider)
3  { ...
4      MEOS::Meos::ensureMeosInitialized();
5      // Get aggregation state (PagedVector) and lon/lat/ts field indices
6      const auto pagedVectorPtr = static_cast<Nautilus::val<Nautilus::Interface::
    PagedVector*>>(aggregationState);
7      ...
8      if (numberOfEntries == Nautilus::val<size_t>(0)) {
9          // Empty window: return sentinel
10         ...
11         return resultRecord; // "BINARY(0)"
12     }
13     // Build a sorted MEOS temporal instants string "{Point(lon lat)@ts,...}"
14     auto trajectoryStr = Nautilus::invoke(
15         +[] (const Nautilus::Interface::PagedVector* pagedVector,
16             const MemoryLayout* layout,
17             uint64_t lonFieldIndex,
18             uint64_t latFieldIndex,
19             uint64_t tsFieldIndex) -> char*
20         {
21             return MeosTrajectoryDetail::buildSortedTemporalInstantSetString(
22                 pagedVector,
23                 layout,

```

⁷For complete implementation see <https://github.com/MobilityDB/MobilityNebula/blob/main/nas-physical-operators/src/Aggregation/Function/Meos/TemporalSequenceAggregationPhysicalFunction.cpp>

```

24         MeosTrajectoryDetail::TrajectoryFieldIndices{lonFieldIndex, latFieldIndex,
25             tsFieldIndex});
26     },
27     pagedVectorPtr,
28     nautilus::val<const MemoryLayout*>(memoryLayoutPtr),
29     nautilus::val<uint64_t>(lonIndexOpt.value()),
30     nautilus::val<uint64_t>(latIndexOpt.value()),
31     nautilus::val<uint64_t>(tsIndexOpt.value()));
32     // Parse with MEOS and convert to WKB, then return "BINARY(N)"
33     ...
}

```

Event-Time Semantics and Trajectory Aggregation

The windowed queries use event-time semantics, i.e., windows are defined over the timestamp(*ts*) attribute. In *NebulaStream*, each incoming tuple is assigned to windows based on its event timestamp, and an event-time watermark determines when a window is considered complete. Once the watermark has advanced beyond the window end time, that window is closed, and the corresponding aggregations are finalized. Late events that are received once this watermark crosses the end time of a window are not allowed.

We implement trajectories as a windowed aggregation. For each time window, the aggregator collects all data points into its internal state and, upon closing the window, sorts them by timestamp before constructing the MEOS input. Per (key and time) window aggregation state, we (1) extract all n points from the internal state into a vector in $O(n)$ time and $O(n)$ memory, (2) sort the vector by event-time in $O(n \log n)$ time, and (3) emit the MEOS input in a single pass in $O(n)$ time. MEOS then normalizes the temporal sequence and enforces a standardized time-ordered representation. If a point arrives out-of-order but before the watermark for its window, it is added to all windows whose interval contains its *ts*, and the final trajectory segment for that window is independent of arrival order. Once a window has been closed by the watermark, the trajectory for that interval is no longer modified.

4.5 Real-Time Queries

MobilityNebula processes mobility data streams across the edge, fog, and cloud layers. The system can be utilized in various scenarios, including ship routing management and public transportation monitoring. By pushing analytics to the edge, MobilityNebula enables the detection of emergency events, such as sudden braking or unscheduled stops, directly on edge devices located aboard trains. In this section, we present continuous queries implemented in MobilityNebula such as point-based queries with geofencing, geospatial complex event processing (see Sect. 4.5.1). In addition, we present trajectory-based queries, such as trajectory creation and trajectory smoothing (see Sect. 4.5.2).

4.5.1 Point-Based Queries

Using MEOS spatiotemporal operations [125], we implement six point-based queries in MobilityNebula that capture a part of the operational needs and challenges described by the SNCB/NMBS maintenance team. These queries were chosen because they reflect real-world tasks that SNCB/NMBS technicians perform when monitoring trains.

Geofencing

A geofence is a boundary that limits a location. It can be created dynamically in a radius from the center of the area or by setting the boundaries to perimeters, such as the edges of a neighborhood. Geofencing supports analysis by establishing and monitoring the relevant geographical boundaries within the operational area. Our integration enables four use cases in geofencing.

Query 1 performs location-based alert filtering. It identifies trains entering maintenance areas by first filtering for nonzero equipment codes (Lines 2 and 3). Instead of a fixed-duration window, the query employs a threshold window that opens when the `eintersects_tgeo_tgeo` predicate evaluates to true, i.e., when the train longitude (`long`), latitude (`lat`), and timestamp (`ts`) intersect the maintenance-zone polygon and closes as soon as the train leaves the monitored area. Within each threshold-bounded interval, the query computes the average speed using the `Avg` aggregation (Line 4), thereby capturing speed trends during maintenance periods and printing the results (Line 5).

Query 1. Location-based alert filtering

```

1 Query::from(SNCB)
2 .filter((Attribute("Code1") != 0 || Attribute("Code2") != 0))
3 .window(ThresholdWindow::of(eintersects_tgeo_tgeo( Attribute("long"), Attribute("lat"),
   Attribute("ts")) == 1))
4 .apply(Avg(Attribute("speed")))
5 .sink(PrintSinkDescriptor::create());

```

Query 2 performs train speed and noise monitoring. It applies the MEOS function `tgeo_at_stbox` to keep only records whose geographic coordinates and timestamps fall within a specified spatiotemporal box (Line 2). From these points, it retains entries with valid readings of speed and Brake Pressure (BP). Then it flags instances where the composite metric ($\text{speed} \times 0.5 + \text{BP} \times 0.5$) exceeds two units, indicating elevated noise levels (Lines 3–4). A 500-millisecond tumbling window on the event timestamp aggregates speed via `Avg`, producing periodic summaries of the train speed measurements (Lines 5–7).

Query 2. Train speed and noise monitoring

```

1 Query::from(SNCB)
2 .filter(tgeo_at_stbox(Attribute("long"), Attribute("lat"), Attribute("ts")) == 1 &&
   Attribute("speed") > 0 && Attribute("BP") > 0)
3 .map(Attribute("noise") = (Attribute("speed") * 0.5 + Attribute("BP") * 0.5))
4 .filter( Attribute("noise") > 2)
5 .window(TumblingWindow::of(EventTime(Attribute("ts")),Milliseconds(500)))
6 .apply(Avg(Attribute("speed")))
7 .sink(PrintSinkDescriptor::create());

```

Query 3 performs high-risk zone proximity monitoring. It identifies trains in proximity to dangerous zones by invoking `edwithin_tgeo_geo` on the SNCB stream against high-risk area polygons (Line 2). Any incoming temporal point within the specified distance of two meters in a ten-second tumbling window (Line 3) is immediately printed, generating real-time alerts for trains near hazardous locations (Line 4).

Query 3. High-risk zone proximity monitoring

```

1 Query::from(SNCB)
2 .filter(edwithin_tgeo_geo(Attribute("long"), Attribute("lat"), Attribute("ts"), 2) == 1)
3 .window(TumblingWindow::of(EventTime(Attribute("ts")),Seconds(10)))
4 .sink(PrintSinkDescriptor::create());

```


Query 4 performs severe weather monitoring. It joins a weather stream with the SNCB stream (Lines 2–3) in a 30-second sliding window with a slide rate of one second (Line 4). The query uses `edwithin_tgeo_geo` to correlate weather events, longitude, latitude, and timestamp with trains located within a given radius (Line 5). It then filters for adverse conditions (Lines 6–10). For records matching these predicates, a subsequent `map` operation assigns an `adjusted_speed_limit` value of 15, enabling dynamic speed recommendations under poor weather conditions (Lines 11–12).

Query 4. Severe weather monitoring

```

1 Query::from(SNCB)
2 .joinWith(Weather)
3 .where(Attribute("ts") == Attribute("weather$ts"))
4 .window(SlidingWindow::of(EventTime(Attribute("ts")), Seconds(30), Seconds(1)))
5 .filter((edwithin_tgeo_geo(Attribute("long"),Attribute("lat"),Attribute("ts"))==1)
6     &&(Attribute("temperature") < 12 ||
7     Attribute("temperature") > 16 ||
8     Attribute("snowfall") >= 0 ||
9     Attribute("wind_speed") > 15 ||
10    Attribute("precipitation") > 0.1 ))
11 .map(Attribute("adjusted_speed_limit") = 15)
12 .sink(PrintSinkDescriptor::create());

```

Geospatial Complex Event Processing

Geospatial Complex Event Processing (GCEP) is a paradigm that enables the detection and analysis of patterns and relationships within spatiotemporal data streams. GCEP facilitates the real-time processing of data generated by moving objects and enabling proactive decision-making. The GCEP is extended from the work presented in [124]. The system offers an alternative to existing GCEP systems, such as in [57], by integrating real-time spatiotemporal analytics and pushing down computation to IoT devices. We show four queries with GCEP.

Query 5 performs passenger counting. It evaluates the train occupancy by first using `map` to compute a `passenger_count` attribute as the average of four pressure sensors (T1, T2, F1, F2) (Line 2) and then filtering for above 3.15 (Line 3). It further filters the stream to a specific region via `tgeo_at_stbox` and checks if the train is moving. The query then suggests an adjusted temperature for air conditioning (Line 4). Finally, a 500-millisecond tumbling window on the event timestamp computes the average speed and outputs it (Lines 5–7).

Query 5. Passenger counting

```

1 Query::from(SNCB)
2 .map(Attribute("passenger_count") = (Attribute("T1") + Attribute("T2") + Attribute("F1")
3   + Attribute("F2")) / 4.0)
4 .filter(Attribute("passenger_count") > 3.15 || tgeo_at_stbox(Attribute("long"),Attribute
5   ("lat"), Attribute("ts")) == 1 && Attribute("speed") > 0)
6 .map(Attribute("adjusted_temp") = 20)
7 .window(TumblingWindow::of(EventTime(Attribute("ts")), Milliseconds(500))
8 .apply(Avg(Attribute("speed")))
9 .sink(PrintSinkDescriptor::create());

```

Query 6 performs brake-system monitoring. It first filters the SNCB stream to include only points that are not in a maintenance area, by applying the function `eintersects_tgeo_geo` (Line 2). It then applies a ten-second sliding window with a 100-millisecond step over the event

timestamp (Line 3). Within each window, the query computes the pressure variation for BP and FF using the `Variation` operator (Line 4), naming the results `varBP` and `varFF`. The query retains only those data points where `varBP` exceeds 0.6 bar, while `varFF` remains at or below 0.5 bar (Line 5), indicating a potential brake issue.

Query 6. Brake system monitoring

```

1 Query::from(SNCB)
2 .filter(eintersects_tgeo_geo(Attribute("long"), Attribute("lat"), Attribute("ts")) == 0)
3 .window(SlidingWindow::of(EventTime(Attribute("ts")), Seconds(10), Milliseconds(100)))
4 .apply(Variation(Attribute("BP")), Variation(Attribute("FF")))
5 .filter(Attribute("varBP") > 0.6 && Attribute("varFF") <= 0.5)
6 .sink(PrintSinkDescriptor::create());

```

4.5.2 Trajectory-Based Queries

Trajectory-based queries operate on windowed spatiotemporal sequences rather than individual data points. While point-based queries focus on instantaneous predicates and local conditions, trajectory-based queries reason about movement behavior over time, such as path construction, movement characteristics, similarity, and interaction between moving objects. In MobilityNebula, these queries are expressed over *stsequence* and *stsequenceset* types produced by windowed aggregation. This subsection presents a set of continuous trajectory queries that demonstrate how MobilityNebula constructs, filters, aggregates, and analyzes trajectories within temporal windows, enabling higher-level mobility analytics directly in the streaming engine.

Query 7 performs trajectory creation. It first applies a ten-second sliding window with a 100-millisecond step over the event timestamp (Line 2). Within each window, the query computes the trajectory by using attributes `longitude`, `latitude`, and `timestamp`, and outputs (Lines 3 and 4).

Query 7. Trajectory creation

```

1 Query::from(SNCB)
2 .window(SlidingWindow::of(EventTime(Attribute("ts")), Seconds(10), Milliseconds(100)))
3 .apply(temporal_sequence(Attribute("long"), Attribute("lat"), Attribute("ts")))
4 .sink(PrintSinkDescriptor::create());

```

Query 8 performs trajectory creation in a restricted space. It applies the function `tgeo_at_stbox` to keep records whose coordinates and timestamps fall within a specified spatiotemporal box (Line 2). It then applies a 20-second sliding window with a 200-millisecond step over the event timestamp (Line 3). Within each window, the query computes the trajectory by using attributes `longitude`, `latitude`, and `timestamp`, and outputs (Lines 4 and 5).

Query 8. Trajectory creation in a restricted space

```

1 Query::from(SNCB)
2 .filter(tgeo_at_stbox(Attribute("long"), Attribute("lat"), Attribute("ts")) == 1)
3 .window(SlidingWindow::of(EventTime(Attribute("ts")), Seconds(20), Milliseconds(200)))
4 .apply(temporal_sequence(Attribute("long"), Attribute("lat"), Attribute("ts")))
5 .sink(PrintSinkDescriptor::create());

```

Query 9 performs geofenced, windowed trajectory extraction with slow-movement alerts. It first keeps only records whose `longitude`, `latitude`, and `timestamp` lie inside a given polygon using `edwithin_tgeo_geo` (Line 2). It then groups by `device.id` and applies a 45-second sliding

window with a 5-second advance over the event timestamp (Lines 3–4). Within each window, it computes the trajectory and aggregates `avg_speed` and `min_speed` (Lines 5–7). It filters where `avg_speed` is less than 5 m/s or `min_speed` is less than 2 m/s (Line 8).

Query 9. Trajectory creation and slow-speed alert in a geofenced area

```

1 Query::from(SNCB)
2 .filter(edwithin_tgeo_geo(Attribute("long"), Attribute("lat"), Attribute("ts"), PolygonWKT
   ("SRID=4326;POLYGON((4.32 50.60, 4.32 50.72, 4.48 50.72, 4.48 50.60, 4.32 50.60))")
   ,1) == 1)
3 .groupBy(Attribute("device_id"))
4 .window(SlidingWindow::of(EventTime(Attribute("ts")), Seconds(45), Seconds(5)))
5 .apply(select(Attribute("device_id"),
6         temporal_sequence(Attribute("long"), Attribute("lat"), Attribute("ts")),
7         avg(Attribute("gps_speed")), min(Attribute("gps_speed"))))
8 .filter((Attribute("avg_speed") < 5) || (Attribute("min_speed") < 2))
9 .sink(PrintSinkDescriptor::create());

```

Query 10 computes a per-device positional divergence measure within each window. It performs a join on `device_id` (Line 2) and evaluates the join in ten-second tumbling windows. Here, `SNCB2` denotes a second logical view of the `SNCB` stream. For each pair, it computes the nearest approach distance between the two points (Line 3), retains only pairs where the left-side latitude is positive (`lat > 0.0`, Line 4), and emits the resulting `mindist` values.

Query 10. Positional divergence monitoring for a device

```

1 Query::from(SNCB)
2 .joinWith(SNCB2, Attribute("device_id") == Attribute("device_id2"))
3 .window(TumblingWindow::of(EventTime(Attribute("ts")), Seconds(10))) .apply(
   nearest_approach_distance(Attribute("lon"), Attribute("lat"), Attribute("ts"),
   Attribute("lon2"), Attribute("lat2"), Attribute("ts2")))
4 .filter(Attribute("lat") > 0.0)
5 .sink(PrintSinkDescriptor::create());

```

Query 11 computes the closest device pairs (top- k) per window. It performs a join, keeping only pairs with `device_id < device_id2` (Line 2). In each ten-second tumbling window (Line 3), it computes a distance score using `nearest_approach_distance` (Line 4). Then it retains the smallest k results per window via a top- k selection (Line 5), emitting them to the sink (Line 6). The `topK` operator performs a per-window global ranking and selection. Given all candidates result tuples in one window w with a numeric score attribute (here, `mindist` in meters), `topK` keeps only the k tuples with the smallest scores.

Query 11. Global closest device pairs (Top- k closest pairs)

```

1 Query::from(SNCB)
2 .joinWith(SNCB2, Attribute("device_id") < Attribute("device_id2"))
3 .window(TumblingWindow::of(EventTime(Attribute("ts")), Seconds(10)))
4 .apply(nearest_approach_distance(lon, lat, ts, lon2, lat2, ts2))
5 .apply(topK(mindist, 10))
6 .sink(PrintSinkDescriptor::create());

```

Query 12 performs trajectory smoothing with an extended Kalman filter. In each ten-seconds window on event-time (Line 2), it first constructs a temporally ordered trajectory from raw GPS measurements using `temporal_sequence` (Line 3). It then applies `temporal_ext_kalman_filter` with parameters `gate=3.0`, `q=0.01`, and `variance=1.0`. The final boolean `false` turns off fill

mode (Line 3) (see Sec. 5.3.2 for the implementation of Extended Kalman Filter). The cleaned trajectory is then emitted.

Query 12. Trajectory smoothing with extended Kalman filter

```

1 Query::from(SNCB)
2 .window(TumblingWindow::of(EventTime(Attribute("ts")), Seconds(10)))
3 .apply(temporal_ext_kalman_filter(temporal_sequence(Attribute("lon"), Attribute("lat"),
4   Attribute("ts")), 3.0, 0.01, 1.0, false))
5 .sink(PrintSinkDescriptor::create());

```

Query 13 performs a windowed k -Nearest Neighbors (k -NN). It forms pairs by joining the stream and excluding `device_id == device_id2` (Line 2). Within each ten-second tumbling window (Line 3), it computes pairwise distances using `nearest_approach_distance` (Line 4). It then groups by `device_id` and applies `knn_agg( mindist, device_id2, 3)` to retain the 3 k -NN (Lines 5–6), emitting a neighbor summary (Line 7). The `knn_agg` operator is a per-window, per-`device_id` aggregation that keeps the k -NN according to a pre-computed distance attribute.

Query 13. Windowed per-device k -NN join

```

1 Query::from(SNCB)
2 .joinWith(GPS2, Attribute("device_id") != Attribute("device_id2"))
3 .window(TumblingWindow::of(EventTime(Attribute("ts")), Seconds(10)))
4 .apply(nearest_approach_distance(Attribute("lon"), Attribute("lat"), Attribute("ts"),
5   Attribute("lon2"), Attribute("lat2"), Attribute("ts2")))
6 .groupBy(Attribute("device_id"))
7 .apply(knn_agg(mindist, Attribute("device_id2"), 3))
8 .sink(PrintSinkDescriptor::create());

```

4.6 Performance Measurement

This section evaluates the performance of MobilityNebula under realistic deployment conditions. Our aim is to assess whether the system can maintain continuous spatiotemporal processing when deployed across various edge, fog, and cloud devices. Additionally, we seek to quantify the impact of factors such as operator complexity, windowing behavior, and output selectivity on end-to-end performance. To characterize the operational limits of the system and to understand how MobilityNebula responds to varying resource budgets, thread parallelism, and network constraints, we measure throughput, events per second, query selectivity, latency, CPU and memory usage, and network traffic across multiple hardware configurations, ranging from embedded edge devices to more capable fog nodes.

4.6.1 Distributed Setting

We evaluated MobilityNebula in a fog-edge deployment. The NebulaStream coordinator ran on an NVIDIA Jetson AGX Orin, while a Raspberry Pi 4 hosted the worker and the client. We run six point-based queries from Sect. 4.5.1 (Queries 1–6).⁸

To evaluate the end-to-end performance, we measured theoretical throughput, sink throughput, events per second, query selectivity, query latency, CPU usage, memory usage, and network bandwidth in the fog-edge configuration for the six queries (see Fig. 4.5). Each component in the pipeline: the coordinator, the worker, and the Wi-Fi link between them contributes its

⁸This evaluation was done in NebulaStream-private previous version.

own processing throughput and bandwidth constraints. Theoretical throughput measurements are obtained at runtime via the NebulaStream E2E Benchmark. In the distributed setting, the sink throughput is determined by the slowest link, i.e., the minimum of NVIDIA throughput, Raspberry Pi 4 throughput, and network bandwidth.

Across the six queries, the sink throughput varied depending on the complexity of the operator and the selectivity (see Figs. 4.5a and 4.5c). Assuming a continuous payload is sent, each query theoretical data rate, measured by E2E Benchmark, remained at 4.8 MB/s. However, the actual data sent to the sink also depends on the selectivity. Queries with high emission rates (e.g., Query 5 at 32k events/s) sent nearly 4 MB/s upstream, while Queries 2 and 3 transmitted only 0.4 MB/s and 0.3 MB/s. Query 4, with a moderate selectivity, sent about 2.8 MB/s.

The events per second (see Fig. 4.5b) followed a similar pattern. Theoretical events per second, computed as a continuous ingestion rate processed by the worker, remained around 13k events per second for all queries. However, when measured at the sink, the delivered throughput exceeded theoretical values for low-selectivity queries. Queries 1 and 4 emitted 6.8 % and 4.1 % of incoming events, respectively, pushing around 20k events/s into the sink. However, Query 2 (11.3 % selectivity) and Query 3 (8.6 % selectivity) show moderate sink rates of around 8k and 10k events/s. Queries 5 and 6, with very low selectivity (2.6 % and 4.2 %), achieved nearly 32k and 20k events/s at the sink despite similar theoretical processing rates.

The latency measurements (see Fig. 4.5d) demonstrate how the number of worker threads affects the overall end-to-end latency. The latency of each query was measured using 8, 16, and 32 threads on the Raspberry Pi. It is possible to observe a trend that queries with heavier loads benefit most from additional parallelism. For Queries 3 and 4, both of which involve filtering and joins, latency decreases from roughly 50 ms (8 threads) to 37 ms (32 threads), as more threads enable concurrent evaluation of window buffers and predicates. Queries 5 and 6, which produce larger result sets, exhibit the most significant improvement, with their latencies decreasing from 90 ms with eight threads to 68 ms with 32 threads. In contrast, the simplest queries (Queries 1 and 2) exhibit only marginal changes: Query 2 latency improves from 37 ms (8 threads) to 23 ms (32 threads). In contrast, Query 1 latency increases slightly from 30 ms to approximately 37 ms. Overall, these results demonstrate that scaling the number of worker threads significantly reduces latency for complex or high-volume queries. In contrast, simple queries may see only modest gains or slight overhead when many threads contend for limited CPU resources.

The selectivity (see Fig. 4.5c) directly correlates with sink throughput and latency. Query 2, with the highest selectivity (11.3 %), incurs a greater downstream load compared to Queries 4 and 5, which emit only a small fraction of the incoming data (2.6 % and 4.1 %), and thus achieve lower latency. Query 6, with moderate selectivity (4.2 %), results in moderate latency and throughput.

Throughout all queries, CPU utilization, memory usage, and network traffic demonstrate how operator complexity and output selectivity impose varying demands on the distributed edge-fog topology (see Fig. 4.5e, 4.5f, and 4.5g).

CPU usage in the Raspberry Pi worker averaged between 55 % and 65 % for spatiotemporal filtering (Queries 1, 2, 3, and 5) but increased above 98 % for Query 4, reflecting its heavier join workload and larger intermediate state. Query 6, which computes two BP variations, observed moderate CPU usage around 53 %, indicating that although its logic is more complex, its selectivity means fewer tuples are processed downstream.

Memory usage remained below 30 MB for five of the queries, with Query 4 being an exception at approximately 92 MB. This increase is attributed to its larger window buffers and join state. Queries 1, 2, 3, and 5 consumed 29 MB of RAM. Query 6 used slightly more memory (30 MB) because tracking brake variations in parallel requires additional buffers for each window.

Network usage measured from the worker to the coordinator reveals how output selectivity

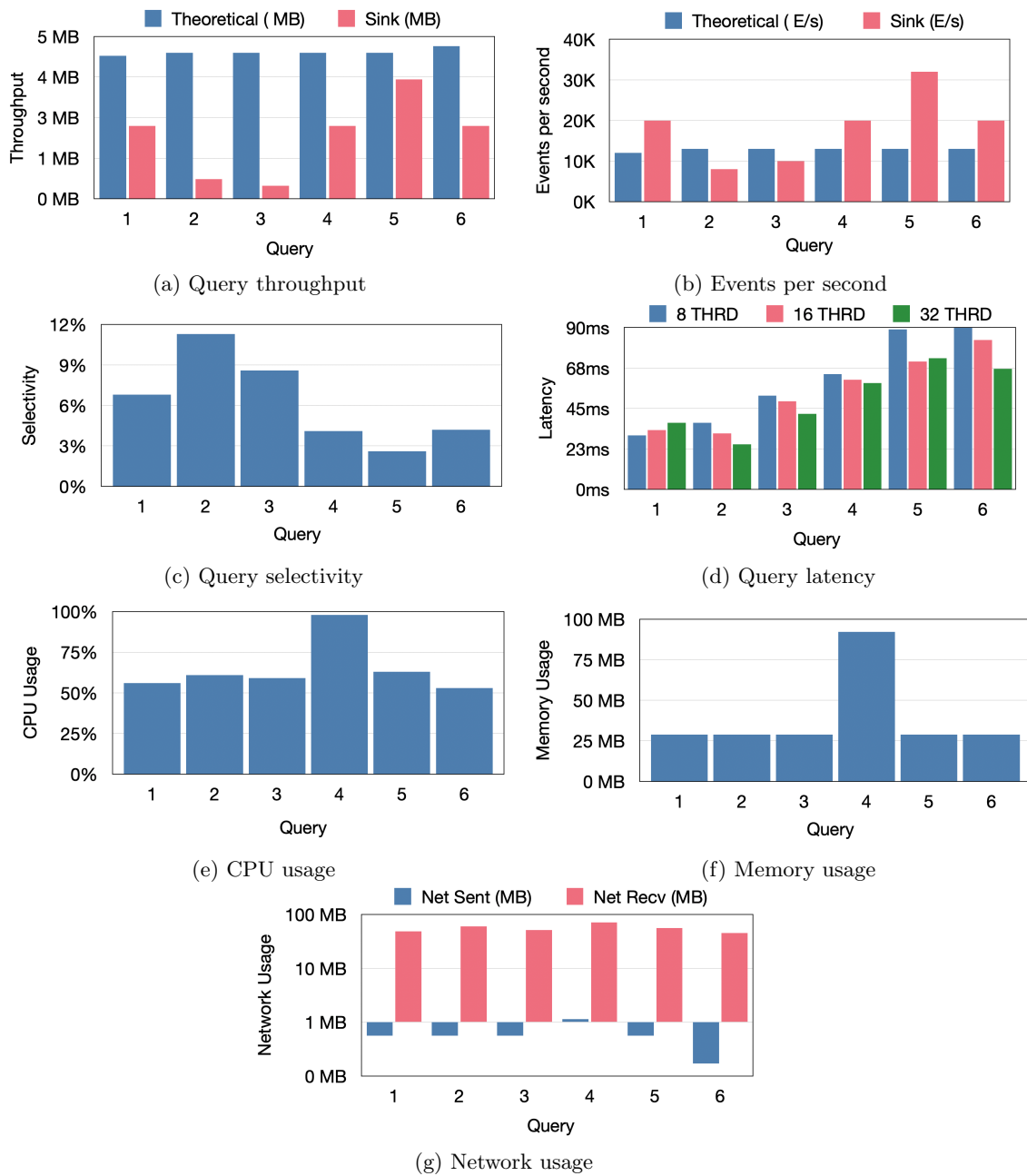


Figure 4.5: Fog-edge deployment results for Queries 1–6, reporting throughput, events/s, selectivity, latency, CPU, memory, and network usage

impacts network traffic. Queries 1, 2, 3, and 5 each sent about 0.56 MB of tuples per interval. Query 4, with moderate selectivity, transmitted 1.13 MB, while Query 6 sent only 0.17 MB, with a high deviation reflecting that the query only emits data when a brake fault is detected. Downstream acknowledgments, shown as the traffic received from the coordinator, remained under 0.071 MB for all queries.

Despite operating on resource-constrained hardware, MobilityNebula sustained thousands of events per second in the distributed fog-edge setup, with sink events per second reaching 32k events/s on low-selectivity queries. By increasing worker thread counts from 8 to 32, queries with joins and large result sets experience a latency drop of up to 25 ms. Early filtering and dynamic thread-level parallelism enabled the system to balance latency and network usage, enabling real-time spatiotemporal processing under variable Wi-Fi bandwidth and heterogeneous compute capacities.

4.6.2 Single-Node Setting

In addition to the distributed fog-edge experiments, we evaluate MobilityNebula in a controlled single-node environment to isolate the effects of operator complexity, windowing, and output size on system performance.⁹ All components operate on a single machine, eliminating network variability and concentrating exclusively on the computational behavior of the engine. This setup also reflects the SNCB/NMBS use case, where processing can be conducted entirely on the onboard nROK device, without dependence on external nodes. To validate the edge deployability of MobilityNebula, we test our implementation under strict container resource caps in Docker.

We implement nine queries (see Sect. 4.5 Queries 3 and 6–13) in TStream [96],¹⁰ GeoEKuiper [47],¹¹ and MobyDick [36]¹² with the same TCP stream input workload of 20k events/s (see Table 4.5 for supported queries per system). We run each query for 30 seconds and measure CPU usage, memory usage, throughput, and latency. We test with multiple configurations, in a single-node core under three memory configurations: 512 MB, 2 GB and 8 GB of RAM. We utilize a configuration of 4 vCPU and 8 GB of RAM to simulate a Raspberry Pi 5, and 2 vCPU and 8 GB of RAM to simulate an nROK device. Furthermore, we evaluate the conditions under which the queries were either terminated or timed out without producing any sink output.

Table 4.5 maps each query to the systems capable of expressing it without altering semantics or requiring altering its core source code. For GeoEKuiper, Queries 10, 11, and 13 required code adaptations to construct a trajectory-like structure. We chose GeoEKuiper for these adaptations since it is edge-based, and it is the only non-Flink-based system.

TStream is executed in Flink’s local environment under the same Docker cgroup limits as MobilityNebula, with parallelism set to one and single-parallelism sinks. Checkpointing and external state backends are disabled, and the job runs with Flink defaults under the container budget. MobyDick is run as a single Flink job inside a Docker container with identical CPU/memory caps, using a host-driven TCP source. We do not add explicit checkpointing and we use single-parallelism sinks. GeoEKuiper runs a single `GeoEKuiper` instance under the same cgroup limits. The runner registers `sncb.stream` and the query rule via the CLI, connects to a local MQTT broker for input and output, and otherwise uses the engine defaults.

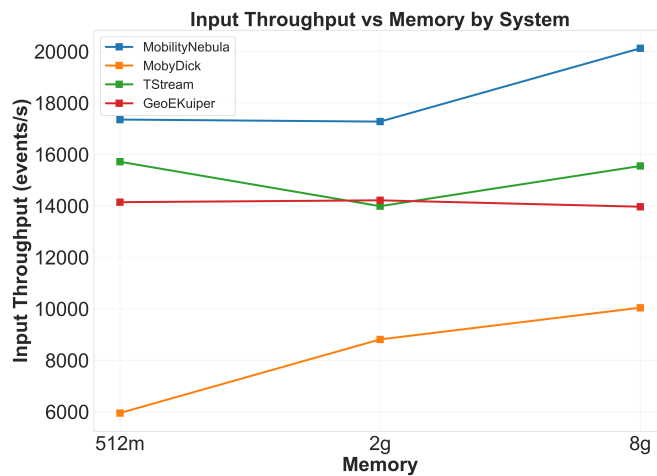
We classify a run as *successful* if the system produces sink output within the timeout and is not terminated (`oom_kill`). We distinguish input throughput (events/s accepted at the driver-to-system boundary) from sink throughput (window outputs/s) as defined by the authors in [54].

⁹We use the refactored NebulaStream available publicly at <https://github.com/nebulastream/nebulastream>.

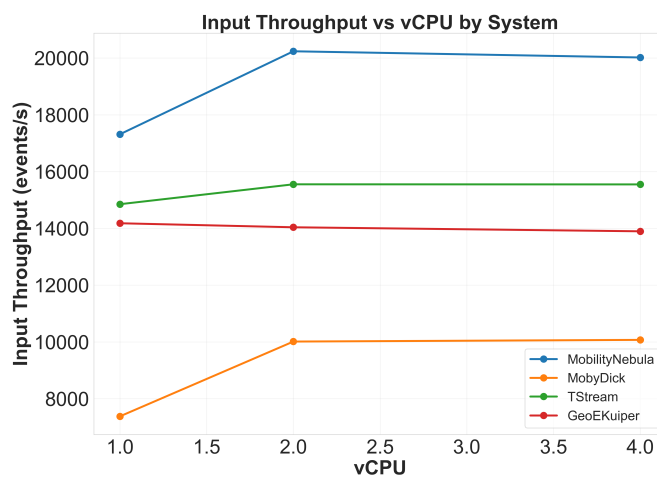
¹⁰<https://github.com/marianaGarcez/SpatialFlink>

¹¹<https://github.com/marianaGarcez/GeoEkuiper>

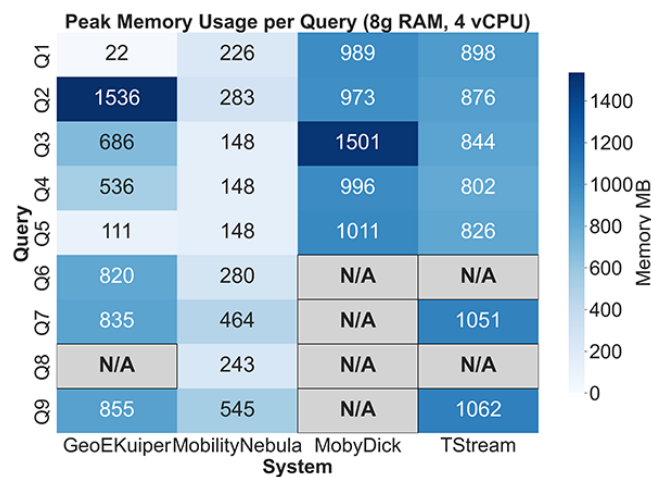
¹²<https://github.com/marianaGarcez/mobydick>



(a) Query throughput vs memory

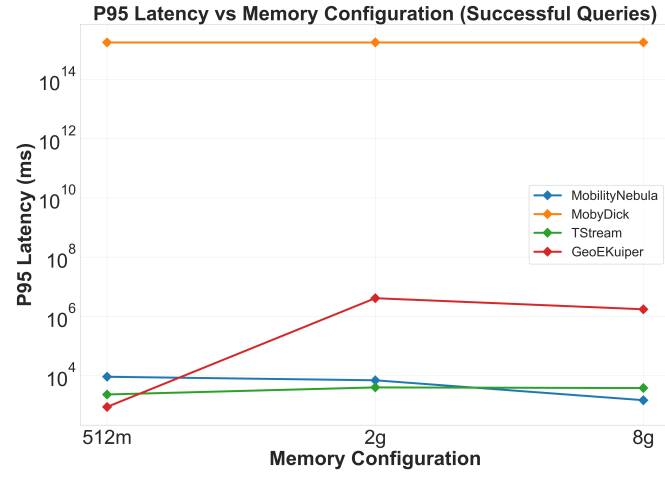


(b) Query throughput vs vCPU

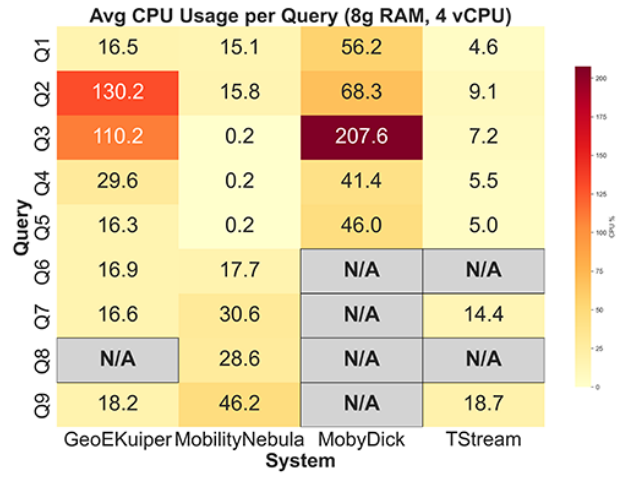


(c) Memory usage heat map

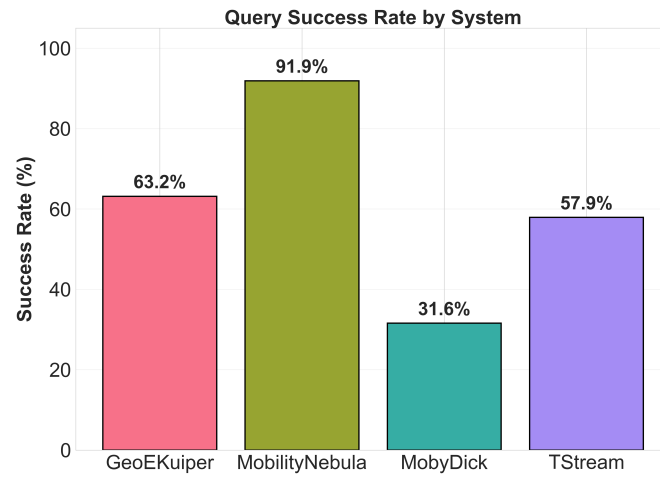
Figure 4.6: Single-node edge-constrained evaluation across Queries 3 and 6–13, showing throughput, CPU usage, latency, and memory usage under varying resource caps



(a) Latency vs memory



(b) CPU usage



(c) Success rate

Figure 4.7: Single-node edge-constrained evaluation across Queries 3 and 6–13, showing latency/memory, CPU usage and success rate under varying resource caps

Table 4.5: Query coverage across systems, indicating which platforms can express each query without changing its semantics or modifying the system’s core code

Query	MobilityNebula	MobyDick	TStream	GeoEKuiper
Q3	✓	✓	✓	✓
Q6	✓	✓	✓	✓
Q7	✓	✓	✓	✓
Q8	✓	✓	✓	✓
Q9	✓	✓	✓	✓
Q10	✓	–	–	✓
Q11	✓	–	✓	✓
Q12	✓	–	–	–
Q13	✓	–	✓	✓

Since our workloads are windowed and often selective, sink throughput is influenced by window emission frequency and selectivity, and therefore does not scale with the raw input rate. We measure input throughput, success rate, average CPU and peak memory under the nROK and Raspberry Pi 5 profiles, and event-time latency p95 across successful windows, defined as the host-side emission timestamp minus the corresponding window-end timestamp. Unsupported queries are excluded, and success rate and throughput are computed over attempted runs only. Figures 4.6 and 4.7 show the results as the mean of five runs.

Figure 4.6a shows the mean throughput in relation to memory usage. MobilityNebula scales its ingestion rate when at least 2 vCPUs are available, with throughput increasing from 17k events/s at 1 vCPU to 20k events/s at 2–4 vCPUs, indicating that operator execution and window aggregation constitute the primary bottlenecks under strict CPU constraints. MobyDick shows to be memory-bound, with throughput rising from 6k to 10k events/s as memory increases from 512 MB to 8 GB, indicating runtime pressure under limited memory. GeoEKuiper maintains stable throughput despite failures and tail latencies, suggesting that bottlenecks originate from memory usage limits and from query execution and window-finalization operations under load. TStream sustains near-target ingestion on its supported queries once memory is at least 2 GB, maintaining 20k events/s across Queries 3,6–9 and the heavier Queries 11 and 13 even at 1 vCPU. In contrast, under the 512 MB cap, it is largely non-viable as most queries are terminated by `oom_kill` (including Queries 3 and 8 and 11/13), indicating that its typical working set is closer to 0.7–1.1 GB on these workloads (see Fig. 4.6c).

Figures 4.6c and 4.7b report peak memory and average CPU usage per query under the 8 GB/4 vCPU profile. MobilityNebula remains within the cap memory even in peak memory for all queries, with peaks ranging from 148 MB (trajectory construction variants) to 545 MB (windowed k -NN join, Query 13). CPU utilization similarly stays within limits, with the most compute-intensive operators occurring in heavy workloads, such as Extended Kalman Filter (EKF) smoothing Query 12 and k -NN Queries 11–13. In contrast, the baselines show increased or more volatile resource footprints on trajectory-based workloads. GeoEKuiper peaks at 1,536 MB in Query 6 alongside elevated CPU (130.2%), and MobyDick peaks at 1,501 MB with very high CPU in Query 7 (207.6%). TStream and MobyDick exhibit higher memory footprints for join- and trajectory-based workloads (0.8–1.1 GB for TStream and 1.0–1.5 GB for MobyDick), leading to `oom_kill` or stall-induced timeouts.

Figure 4.7a reports p95 event-time latency versus memory. For each emitted window result, we record the emission timestamp, the window end timestamp, and the driver’s last successful

send time for that same window end to distinguish system-side delay from driver-side lag and backpressure. In MobilityNebula, these latencies show large tails in Queries 7 and 9 under 1 vCPU with 512 MB–2 GB: the system sustains 9–14k events/s and reaches p95 latencies of 15–24 s. At 8 GB/2 vCPU, throughput returns to 20k events/s with $p95 \leq 1.1$ s, consistent with reduced driver lag and more timely watermark-driven window finalization once ingestion approaches the target rate. In contrast, GeoEKuiper and MobyDick exhibit stall/timeout behavior under the same configuration, where windows fail to finalize within the timeout limit. MobilityNebula and TStream remain at p95 4.9 s and 3.75 s, respectively.

Some queries show high backpressure, as indicated by driver-side blocking and the drop from target injection to achieved input throughput, including zero post warm up ingestion in extreme cases (see Fig. 4.6a). This effect depends on query type and is strongest for state-heavy windowed operators (joins, k -NN, trajectory aggregation), where increased state and output volumes elevate buffering pressure and delay watermark-driven window finalization. We analyze the rate at which queries run and produce output, and define this as the success rate (see Fig. 4.7c). MobilityNebula achieves the highest success rate throughout, with a success rate of 91.9%; on all queries, the baselines achieve 73.3% (TStream), 70.6% (GeoEKuiper), and 54.5% (MobyDick) on supported queries. When counting unsupported queries as failures, this corresponds to 57.9%, 63.2%, and 31.6%.

4.6.3 Mobility Computation Breakdown

We perform a time breakdown for mobility queries using our driver-separation measurement setup. We measure host-side emission timestamp, the window end timestamp (event-time), and the driver’s last successful send time for that window end (wall clock). This procedure enables monitoring of post-admission processing, watermark-driven window finalization, and the capture of backpressure residuals dominated by sender blocking. In MobilityNebula, for Queries 7 and 9: under 512 MB–2 GB, throughput drops to 9–14k events/s and p95 rises to 15–24 s, whereas at 8 GB/2 vCPU throughput returns to 20k events/s with $p95 \leq 1.1$ s.

We measure the percentage of query time spent inside MobilityNebula MEOS wrapper and MEOS functions. These percentages correspond to the function body, a thin wrapper around the MEOS call in MobilityNebula and MEOS function computation. They therefore exclude operator-level work such as state updates, windowing, sorting, building the MEOS input, aggregation state, joins serialization, and downstream processing. Thus, `temporal_sequence` and `knn_agg` appear small because most of their cost lies outside the MEOS call. At 512 MB–2 GB, MEOS call time is still a secondary component for most queries, as for geometric predicates such as `edwithin_tgeo_geo` and `eintersects_tgeo_geo`, the MEOS call share is about 10–16%, while trajectory construction is nearly negligible inside the MEOS call (`temporal_sequence` on the order of 10^{-4} – $10^{-2}\%$). Distance queries place more work in proximity computation. For example, `nearest_approach_distance` reaches 11–14% MEOS-call share in the windowed per-device kNN join.

In addition, we measure the time rate of mobility operators in NebulaStream and report the query time spent inside each operator in `lift/combine/lower`, state updates, windowing, sorting, MEOS input construction, and any MEOS call. Under the same 512 MB–2 GB caps, most operators account for only a small fraction of end-to-end computation time (often 3–7%), consistent with overhead being dominated by runtime-level window finalization and backpressure. The main outlier is the k -NN workload as the operator-level share of `nearest_approach_distance` dominates, reaching 64–86% depending on the memory cap, confirming that this query is compute-bound in the distance operation rather than in the surrounding windowing logic.

Table 4.6 quantifies where time is spent under the 512 MB–2 GB resource caps by separating

Table 4.6: Computation breakdown per query under 512 MB–2 GB caps, separating time spent in the MobilityNebula wrapper/MEOS call from operator-level costs

Query	Function / Component	Wrapper / MEOS (%)	Operator- level (%)
Q3	edwithin_tgeo_geo	10.5841 / 10.3503	4.4688 / 4.5527
Q6	eintersects_tgeo_geo	13.4363 / 13.4841	5.2822 / 5.3026
Q7	temporal_sequence	0.0265 / 0.0102	7.3059 / 7.5427
Q8	tgeo_at_stbox	8.7877 / 9.6641	3.2108 / 3.3192
	temporal_sequence	0.0003 / 0.0003	1.4968 / 1.4541
Q9	edwithin_tgeo_geo	13.0312 / 15.7236	5.2962 / 5.4723
	temporal_sequence	0.0004 / 0.0004	1.1764 / 1.2483
Q10	nearest_approach_distance	0.0019 / 0.0018	85.6369 / 63.7801
Q11	nearest_approach_distance	7.1835 / 5.5241	6.1240 / 5.9721
Q12	temporal_ext_kalman_filter	3.5257 / 3.8194	3.0566 / 3.0308
	temporal_sequence	0.00003 / 0.00003	1.9370 / 1.9370
Q13	knn_agg	0.0943 / 0.0695	0.0295 / 0.0282
	nearest_approach_distance	13.8024 / 11.4642	3.8551 / 3.7323

the share inside MobilityNebula MEOS wrapper and the MEOS function body from the share inside NebulaStream mobility operators.

4.6.4 Geospatial vs. Non-Geospatial Operator Latency

To quantify the latency overhead introduced by the geospatial operations, we execute an experiment for three classes of operators with a representative geospatial query of each class and a scalar query structurally identical, in which each geospatial operator is replaced by a scalar equivalent, preserving source, schema, window, and grouping. We classify the queries into three operator classes based on the type of geospatial computation each uses. Predicates that filter tuples by spatial relation are classified as *predicate* class, aggregators that construct trajectories from points are classified as *trajectory* class, and pair distance computations that calculate proximity between moving objects are classified as *distance/k-NN* class. For the *predicate* class, Q6_G uses function `eintersects_tgeo_geo`, and has the same window as the scalar query Q6_S (see Query 14), which evaluates `gps_speed > 0.0`.

Query 14. Query Q6_S - Scalar predicate baseline for Q6

```

1 Query::from(SNCB)
2 .filter(Attribute("gps_speed") > 0.0)
3 .window(SlidingWindow::of(EventTime(Attribute("ts")), Seconds(10), Milliseconds(100)))
4 .apply(Variation(Attribute("BP")), Variation(Attribute("FF")))
5 .filter(Attribute("varBP") > 0.6 && Attribute("varFF") <= 0.5)
6 .sink(PrintSinkDescriptor::create());

```

For the *trajectory* class, we compared Q7_G that uses function `TEMPORAL_SEQUENCE` in a sliding window against Q7_S (see Query 15), which uses function `VARIANCE(gps_speed)` to compute the maximum speed variance.

Query 15. Query Q7_S- Scalar trajectory baseline for Q7

```

1 Query::from(SNCB)
2 .window(SlidingWindow::of(EventTime(Attribute("ts")), Seconds(10), Milliseconds(100)))
3 .apply(Variation(Attribute("gps_speed")))
4 .sink(PrintSinkDescriptor::create());

```

For class *distance/k-NN*, Q13_G computes the `nearestApproachDistance` by pair, joins with tumbling window of 10s, and calculates the `VARIANCE(mindist)` as an external aggregator. The scalar query Q13_S (see Query 16) computes the euclidean squared distance $(\Delta\text{lat})^2 + (\Delta\text{lon})^2$ with the same structure as the original query. We isolate the per-pair distance computation with the substitution of `KNN_AGG` by `VARIANCE(mindist)` from the original query Q13.

Query 16. Query Q13_S - Scalar distance baseline for Q13

```

1 Query::from(SNCB)
2 .joinWith(GPS2, Attribute("device_id") != Attribute("device_id2"))
3 .window(TumblingWindow::of(EventTime(Attribute("ts")), Seconds(10)))
4 .apply((Attribute("lat") - Attribute("lat2")) * (Attribute("lat") - Attribute("lat2")) +
        (Attribute("lon") - Attribute("lon2")) * (Attribute("lon") - Attribute("lon2")))
5 .groupBy(Attribute("device_id"))
6 .apply(Variation(Attribute("mindist")))
7 .sink(PrintSinkDescriptor::create());

```

The tests for the three classes use the single-node Docker setting from Sect. 4.6.2 at 4 vCPU/8 GB, TCP source, 30s warm-up + 90s measurement, and report driver-side processing latency as the mean of 5 runs. The trajectory and predicate classes run at 200 events/s, the distance/kNN class runs at 20 events/s, matching the query-specific input rates used in Sect. 4.6.6.

Table 4.7 compares the three operator classes by the MEOS call cost, which is the percentage of time spent inside the MEOS function in relation to the total query processing time, and the primary overhead source, which indicates if the cost is inside the MEOS call or in the stream operator-level, such as state management, sorting, and input construction. For predicates, the geometric test itself is the expensive part (10–16% inside MEOS). For trajectory construction, the MEOS call is negligible, but the surrounding operator computation dominates, especially for sorting the input. For distance/kNN, both the MEOS call (11–14%) and the operator-level processing (64–86%) contribute to the overhead.

Table 4.7: Geospatial operator classes, classified by class, the per-function call cost as the percentage of time spent in the MEOS function compared to the total query processing time, as measured by the per-function profiling of Sect. 4.6.3. Primary overhead source indicates where the overhead is more prominent: inside the MEOS call or at the operator-level

Class	Examples	MEOS call cost	Primary overhead source
Predicate	<code>edwithin</code> , <code>eintersects</code> , <code>eintersects_tgeo_geo</code>	10–16%	MEOS functions
Trajectory	<code>temporal_sequence</code>	10^{-4} – $10^{-2}\%$	Operator wrapping
Distance/kNN	<code>nearestApproachDistance</code> , <code>KNN_AGG</code>	11–14%	Operator processing

Table 4.8 summarizes the results for the trajectory, predicate, and distance/kNN class. For each pair we report the processing time latency at the 50th and 95th percentiles (proc-p50, proc-p95), defined as the time interval between the moment the driver sends the last tuple of a window and the moment the corresponding result appears at the sink, the average CPU utilization

and peak memory of the worker container during the measurement interval, and the Δ between the geospatial and scalar queries, expressed as absolute difference and relative percentage. For the predicate class, Q6_G adds 3.5 ms (+1.8%) in p50 and 3.6 ms (+1.1%) in p95, with similar CPU usage. For the trajectory class, Q7_G adds 7 ms (+3.4%) in p50 and 19 ms (+6.0%) in p95 over Q7_S, with additional 3.0 percentage points (pp) CPU usage (+21.6% relative) and 9 MB of additional memory peak. This result is significant since the detailing in Sect. 4.6.3 reports MEOS of `temporal_sequence` as small (10^{-4} – $10^{-2}\%$), the 6% of p95 latency that we measure are therefore overhead at the operator level, such as state management, sorting, and construction of the MEOS input, that the profile by function cannot capture. For the distance/ k -NN class, Q13_G adds 4.3 ms (+0.9%) in p50 and 65 ms (+0.5%) in p95, with 0.6 pp more CPU. At the streaming rate of 20 events/s with a 10 s tumbling window, the per-pair MEOS distance calculation cost is masked by the join state management, which dominates the 12 s absolute p95.

Table 4.8: Geospatial vs. non-geospatial operator cost at 8 GB/4 vCPU

Class	Query	Proc-p50 (ms)	Proc-p95 (ms)	CPU%	Mem peak (MB)
Predicate	Q6 _S (scalar)	197.5	311.1	12.1	385.8
	Q6 _G (geo)	201.0	314.7	12.2	391.2
	Δ	+3.5 (+1.8%)	+3.6 (+1.1%)	+0.1 pp	+5.4
Trajectory	Q7 _S (scalar)	204.6	319.8	13.9	396.4
	Q7 _G (geo)	211.6	339.0	16.9	405.4
	Δ	+7.0 (+3.4%)	+19.2 (+6.0%)	+3.0 pp	+9.0
Distance/ k NN	Q13 _S (scalar)	461.9	11912.9	43.7	176.6
	Q13 _G (geo)	466.2	11977.7	44.4	176.1
	Δ	+4.3 (+0.9%)	+64.8 (+0.5%)	+0.6 pp	−0.5

These three classes show that the participation by function in the MEOS call of Sect. 4.6.3, when considered alone, is not a reliable predictor of the end-to-end cost. The trajectory class has a small MEOS overhead but a measurable overhead of 6%, since most of the cost is outside the MEOS call. The predicate class accounts for 10–16% of MEOS calls, but has negligible overhead, since the cost per tuple is reduced in the windowing pipeline. The distance/ k NN class has a comparable 11–14% MEOS share but only 0.5% p95 overhead at the streaming rate since the per-pair calculation cost is masked by the join state management. No individual class adds more than 6% to end-to-end p95 latency, which is well within the margin required for the target edge use case, for edge processing with 1 Hz ingestion rate.

4.6.5 Parameter Sensitivity

We measure parameter sensitivity across the nine queries (Queries 3 and 6–13) by varying the window size and slide interval around the baselines in Sect. 4.5. Each query runs for 30 seconds under 4 vCPU/8 GB, using the TCP workload of 20k events/s. We test both the *sliding* and *tumbling* versions of each query. For originally tumbling queries (Queries 3, 10–13), we added a sliding variant at the baseline size with a 1 s slide, and for sliding queries (Queries 6–9), we added a tumbling variant at the baseline size. For tumbling queries (Queries 3, 10–13), we test sizes 5/10/20 s and for sliding queries (Queries 6–9), we sweep window sizes (Query 6: 5/7/10/20 s; Queries 7 and 8: 5/10/20 s; Query 9: 15/45/90 s) and slide intervals (Queries 6–8: 10 ms/100 ms/1 s; Query 9: 1/5/10 s).

Sink throughput is primarily determined by emission frequency and selectivity, rather than the input rate. For instance, Query 3 produces sparse outputs, and its sink rate decreases as

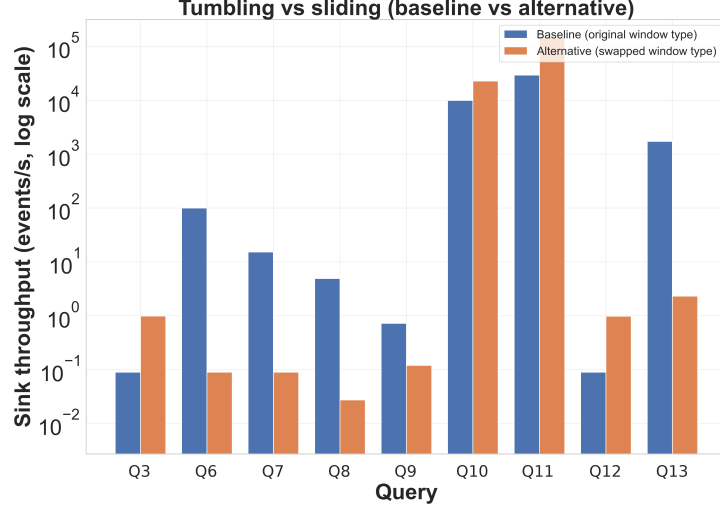


Figure 4.8: Throughput sensitivity to window size and slide interval across Queries 3 and 6–13 (sliding vs. tumbling variants)

the window size increases (from 0.21 events/s at 5 s to 0.06 events/s at 20 s). Query 8 exhibits a similar decline (0.21 to 0.06 events/s). In contrast, join tumbling queries generate numerous results per window, sustaining high sink rates (e.g., Query 10 at 10k events/s and Query 11 at 29k events/s at 10 s). Among tumbling queries, Query 10 demonstrates the greatest sensitivity to window size, as increasing the window to 20 s augments the join-state pressure, reducing input throughput to approximately 13k events/s and increasing average CPU utilization from 27.1% (10 s) to 63.3% (20 s), while memory usage rises from 224 MB to 266 MB. Converting originally sliding queries to tumbling (see Fig. 4.8) reduces sink throughput to one emission per window and decreases resource consumption accordingly. For Query 8, the tumbling variant produces very sparse outputs (0.027 events/s), necessitating longer runs to observe results as the query presents high selectivity.

Sliding windows show sensitivity to the slide interval, as increased overlap directly raises recomputation and emission rates. For Query 6, reducing the slide from 1 s to 10 ms increases sink throughput (0.98 to 100 events/s) while maintaining input throughput near 20k events/s (see Fig. 4.8). Memory usage rises due to the greater number of concurrent overlapping windows (248 to 435 MB). For Query 7 we see a similar behavior, at 10 ms slide intervals, trajectory construction dominates, reducing input throughput to 4k events/s and increasing peak memory to 2.7 GB. At 1 s slides, input throughput recovers to 20k events/s with lower memory usage (611 MB on average). Queries 8 and 9, executed at 200 events/s, follow the same trend: decreasing the slide interval increases sink rates (Query 8: from 0.06 events/s at 1 s to 4.89 events/s at 10 ms; Query 9: from 0.36 events/s at 10 s to 3.95 events/s at 1 s), while CPU and memory remain relatively stable. Converting these queries to sliding increases overlap costs most significantly for join workloads. Queries 10 and 11 experience substantial input-throughput degradation under 1 s slides, from 4k and 11k events/s, and the per-device k -NN join also degrades to 11k events/s, while increasing CPU utilization to 66.4%.

4.6.6 Network Volatility Stress Evaluations

We evaluate the nine queries (Queries 3 and 6–13) under 4 vCPU/8 GB. Each run streams the input dataset over TCP from a host-side driver into the system under test. To ensure tests complete and to isolate timing perturbations rather than pure saturation, the driver uses query-specific mean rates: Queries 3/6/10/11 at 1k events/s, Queries 7/8/9/12 at 200 events/s, and Query 13 at 20 events/s. Runs include a 60-second fixed warm-up interval, and a 600-second measurement. We compute event-time latency per window as the delay between window-end time and the timestamp at which the corresponding window result is emitted at the sink. Latency percentiles (p50/p95) are computed over completed windows only. Baseline scenario uses a steady stream with no jitter, bursts, or outages. J1/J2/J3 add per-batch jitter of $\pm 20/\pm 80/\pm 200$ ms. B1/B2/B3 create periodic burst cycles with on/off durations of 2 s/2 s, 1 s/3 s, and 0.5 s/3.5 s, respectively, keeping the same long-run mean rate. O1/O2/O3 inject periodic outages of 10/20/30 s every 60/90/120 s, respectively. Combined applies J2 + B2 + O1 simultaneously (± 80 ms jitter, 1 s on/3 s off bursts, and a 10 s outage every 60 s).

All runs in this dataset completed successfully (PASS, no errors), with throughput closely matching the intended levels. CPU usage oscillated between 1 and 2 percent, whereas memory held steady at 160-220 megabytes (see Fig. 4.10a and 4.10b). As shown in Fig. 4.9b, incoming event flow followed expected patterns throughout regular operation, timing variations, spikes, dropouts, and mixed disruptions, with no lasting slowdown occurring despite changing network behavior.

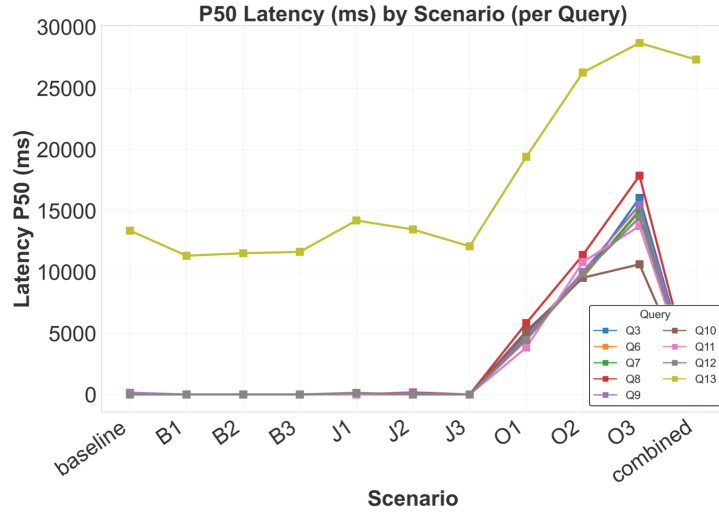
Figure 4.9a presents p50 latency. Jitter variants (J1-J3) cause only minor shifts in latency, whereas bursts (B1-B3) result in increased response times. Outages (O1-O3) are disruptive, as data pauses halt watermark progression and delay window finalization. When multiple disruptions occur, system performance remains superior to that observed during the most severe outage, indicating that total downtime has a greater impact than processing load. Query 13 exhibits the most pronounced lag among all queries. Its architecture requires substantial resources and is highly dependent on window finalization.

4.7 Chapter Summary

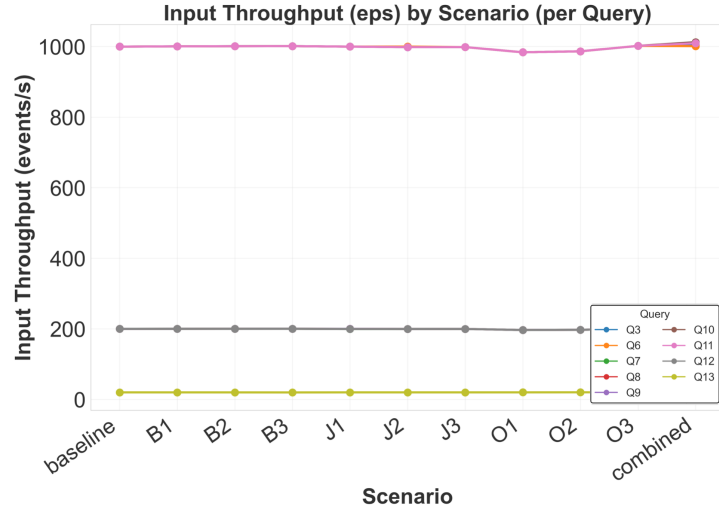
In this chapter, we presented *MobilityNebula*, the main contribution of this thesis: a mobility-aware system for continuous spatiotemporal processing that runs across the edge-to-cloud continuum. We adopted the MOD framework’s abstract model and instantiated it using *MobilityDB sequence representation*. To enable streaming of temporal objects, we extended the discrete model with event-time window awareness, resulting in the *stinstant*, *stsequence*, and *stsequenceset* types. These types couple spatiotemporal values with their window context and support trajectory construction over unbounded streams.

Instead of developing a new engine, a systems-integration approach was adopted, instantiating the proposed model on two representative systems: *MobilityFlink* (on Flink) and *MobilityNebula* (on NebulaStream). The architecture of both systems was described, detailing how MEOS operations are exposed to streaming operators.

The resulting system was evaluated using point-based and trajectory-based real-time queries in single-node and distributed settings. These experiments characterized how operator complexity, state size, and output selectivity affect throughput, latency, CPU and memory footprint, and network traffic under resource constrained edge deployments. Across the query set, *MobilityNebula* achieved the highest completion and success rates (91.9%) compared to 73.3% (TStream), 70.6% (GeoEkuiper), and 54.5% (MobyDick) for supported queries. The evaluation also identified the main pressure points for state-heavy workloads, such as windowed joins, k -NN, and trajectory



(a) p50 latency per query

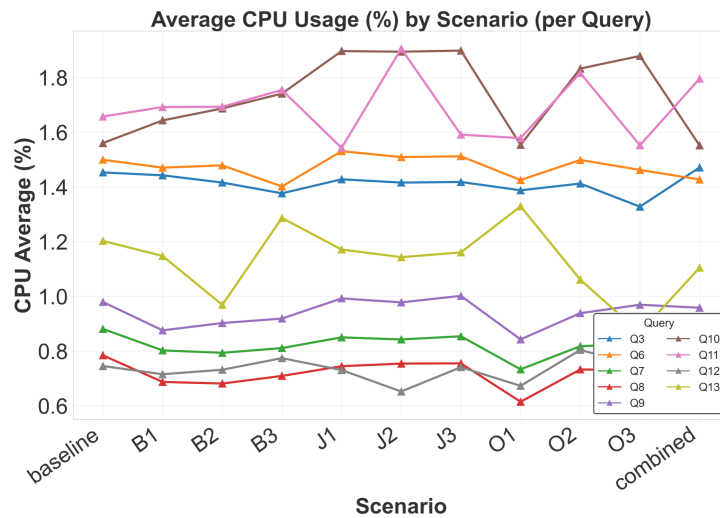


(b) Input throughput per query

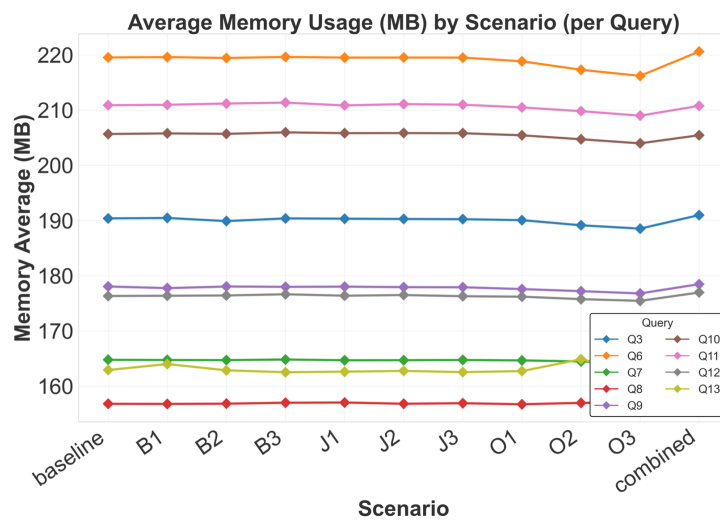
Figure 4.9: Network-volatility stress-test results under controlled jitter (J), burst loss (B), outage (O), and combined scenarios, reporting per-query p50 end-to-end latency and sustained input throughput.

construction. For instance, under 1 vCPU and 512 MB to 2 GB, the most demanding workloads sustain 9k to 14k events per second with p95 latencies of 15 to 24 seconds, while larger budgets (8 GB/2 vCPU) workloads sustain 20k events per second with p95 latencies of 1.1 seconds.

Parameter-sensitivity experiments demonstrated how window size and slide interval influence overlap, recomputation, and state pressure. Network-volatility stress tests indicated that jitter and burstiness have a limited impact on sustained throughput, whereas outages primarily affect latency by stalling watermark progress and delaying window finalization. Overall, the findings demonstrated that window-aware, trajectory semantics can be executed continuously on constrained nodes and scaled along an edge-to-cloud topology. Chapter 5 presents the deployment



(a) Avg. CPU usage per query



(b) Avg. memory usage per query

Figure 4.10: Network-volatility stress-test results under controlled jitter (J), burst loss (B), outage (O), and combined scenarios, reporting per-query average CPU utilization and average memory consumption.

of MobilityNebula in collaboration with the SNCB/NMBS, a study on trajectory outlier detection and trajectory analytics.

Chapter 5

Trajectory Processing Applications

The previous chapter presented MobilityNebula and evaluated it from a systems perspective, measuring throughput, latency, and resource consumption across a set of continuous spatiotemporal queries. Those experiments demonstrated that the system can sustain real-time mobility processing on constrained edge hardware. This chapter shifts the focus from system internals to application-level validation. We examine whether the proposed architecture and operators deliver value in operational and analytical settings that involve real mobility workloads.

The first section describes the SNCB/NMBS use case, detailing the train architecture, edge devices, and real-world integration of MobilityNebula for operational monitoring and diagnostics. The second section describes our work before trajectory stream processing. We have later used our findings to implement Kalman filter into MobilityNebula. It summarizes the methodological and analytical elements that complement the systems-oriented approach of this thesis. We start by examining trajectory outlier detection, reviewing static and streaming methods, and evaluating their suitability for mobility datasets. We introduce a ground-truth construction method, based on cross-referencing speed and bearing measurements from multiple sensors, enabling evaluation across mobility domains. Using this ground-truth, we conduct an experimental study of ten state-of-the-art outlier-detection libraries and systems across four real-world datasets. We then present the theory and implementation of an Extended Kalman Filter (EKF) outlier detector integrated into MEOS and exposed in MobilityNebula.

In addition, the chapter demonstrates how the MEOS spatiotemporal model supports expressive trajectory analytics. Using the Grab-Posisi dataset from Singapore, we perform trip segmentation, stop-based and geometry-based trip extraction, metric derivation (distance, average speed, travel time, bearing), proximity detection, and heatmap construction with MEOS and PyMEOS.¹

5.1 SNCB Use Case

This section presents the deployment of MobilityNebula in collaboration with the SNCB/NMBS. It describes the train architecture, the edge devices used, and the real-world integration of MobilityNebula for operational monitoring and diagnostics. The section detail the context, implementation, visualization, validation, and verification of the system against previously existing diagnostics.

¹<https://github.com/MobilityDB/PyMEOS>

5.1.1 Context

The Engineering Department at SNCB/NMBS aims to ensure the operational safety and reliability of over 1,200 railway vehicles across 11 distinct fleets. Figure 5.1 shows the train trajectories produced by GPS readings of six monitored trains over six months. To support this mission, SNCB/NMBS develops and maintains a remote diagnostic platform for continuous acquisition and analysis of telemetry data from onboard subsystems, including pantographs, traction converters, doors, and HVAC units. This telemetry data is transmitted to a centralized cloud infrastructure, where batch analytics support anomaly detection and the generation of maintenance work orders within the company resource planning system.

However, the scale and velocity of data generated by modern rolling stock have exposed limitations in purely cloud-centric architectures. The volume of raw sensor streams, often sampled at high frequencies, imposes significant bandwidth and storage costs, while centralized processing becomes inefficient as data volumes grow.

To resolve this, we have deployed the MobilityNebula system in the SNCB/NMBS diagnostic platform. The edge device installed on the train receives a stream of data measurements, such as Brake Pressure (BP) readings and GPS positions. MobilityNebula continuously executes mobility queries on this data before forwarding the results to downstream sinks for persistent storage and later offline analysis. Queries include detecting critical events, such as unscheduled stops, abrupt decelerations, or degradation patterns in braking systems. We employ user-defined queries expressed in SQL-like or C++ syntax. Only high-level alerts and aggregated summaries are transmitted to central systems, significantly reducing network load while preserving diagnostic fidelity.

Additionally, unlike SNCB/NMBS's previous approach, which relied on fixed Programmable Logic Controller (PLC), the system can remotely deploy queries, enabling fast reconfiguration of alert and monitoring logic and version control across fleets. This shift toward edge-enabled, data-driven maintenance enables a transition from time-based to condition-based servicing, with potential benefits including reduced downtime, extended component lifespans, and more efficient allocation of maintenance resources.

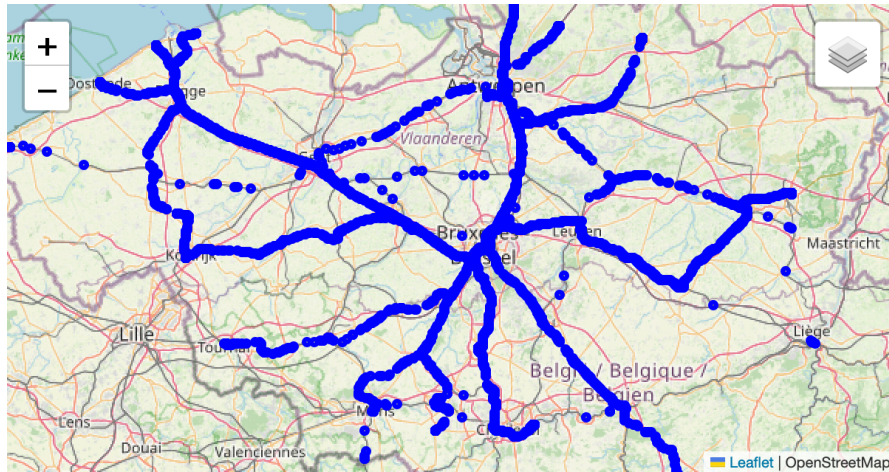


Figure 5.1: GPS trajectories of six monitored SNCB trains over six months, as observed from onboard GPS readings

The previous architecture is illustrated in Fig. 5.2. Each wagon is equipped with an nROK

device that collects GPS and sensor data. A PLC monitors the BP. A control server periodically synchronizes the timestamps from all sensors. The edge device receives the measurements via FTP and transmits them directly using MQTT. Data from all trains is subsequently processed in the cloud, with no processing occurring at the edge.

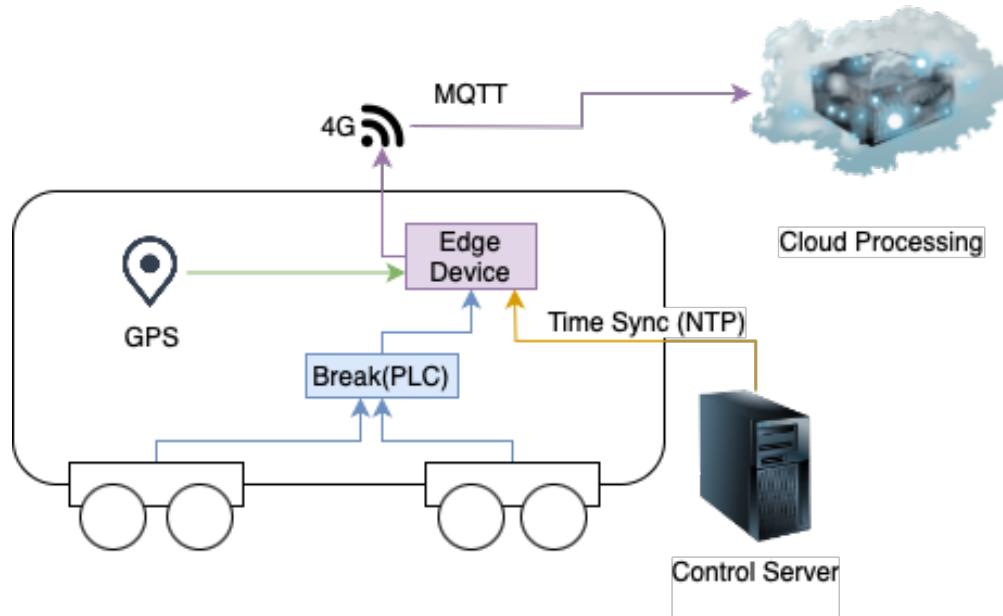


Figure 5.2: Previous SNCB diagnostic architecture, in which onboard measurements are transmitted to the cloud without edge-side processing

5.1.2 Edge Devices

The monitored trains are part of the M6 fleet of SNCB/NMBS, a series of double-deck carriages (see Fig. 5.3). These trains entered service in 2002 and remain part of SNCB/NMBS's long-distance and intercity operations. Each M6 train includes several passenger cars and a driving trailer equipped with on-board diagnostic systems that provide continuous telemetry on traction, braking, and environmental conditions. The M6 architecture features pneumatic and electrical subsystems that expose standard measurement points for BP, traction force, and door status, which we use as input sources for MobilityNebula.

The edge device used in each train is a nROK 1030 (see Fig. 5.4)². The device is a fanless rolling-stock computer based on an Intel Atom x6211E dual-core processor (6 W). The device is running Ubuntu 22.04.2 LTS.

Figure 5.5 shows the use case architecture. Each wagon is equipped with an nROK device that receives GPS and sensor readings. Processing is performed within this edge device, and alerts are transmitted via MQTT to SNCB/NMBS's headquarters and stored on a cloud server. A control server occasionally synchronizes the timestamps for all sensors and can also be used to deploy new queries remotely. It is important to note that processing should not consume

²<https://www.nexcom.com/Products/mobile-computing-solutions/railway-computer/railway-atom-compact/passenger-information-system-nrok-1030>



Figure 5.3: The SNCB M6 double-deck rolling stock used in the deployment and evaluation of MobilityNebula

the entire CPU and RAM of the device, as the department is simultaneously transmitting and running multiple diagnoses.

5.1.3 Implementation

We collaborated with SNCB/NMBS engineers to integrate MobilityNebula within their diagnostic platform. Several meetings were held to define which analysis would be meaningful for their operations and how the system would integrate with their existing infrastructure.

We defined MobilityNebula sources and sinks. The TCP source connects to train sensors that broadcast measurements, including BP and GPS data. The MQTT sink forwards aggregates and alerts to SNCB/NMBS's central monitoring system.

We then worked to determine the most relevant queries for operational monitoring. Each query addressed specific concerns raised by the engineers, such as detecting excessive brake variance or monitoring trains entering restricted areas. These included brake-system diagnostics, geofencing for maintenance zones, speed monitoring, and environmental awareness.

We conducted multiple on-site visits to SNCB/NMBS facilities to validate the design. These tests measured how the system behaved during deployments under network conditions typical of moving trains.

We have also extended MobilityNebula to SNCB/NMBS's requirements. For instance, we have implemented a variance aggregation function to detect pressure fluctuations, added a TCP source client to support their internal simulators, and tuned the MQTT sink to ensure predictable delivery and quality of service.

In addition, we have prepared practical tutorials and training sessions to ensure SNCB/NMBS engineers can operate the system independently. These covered query creation, modification, and



Figure 5.4: The nROK 1030 onboard edge device used to execute MobilityNebula queries on trains

deployment from the control server, using both the SQL-like and C++ operator syntax.

5.1.4 Query Visualizations

Figure 5.6 shows point-based Query Visualizations from Sect. 4.5.1. The visualization was built in Deck.gl and uses Kafka to input data. We show real data collected from trains, including coordinate positions, speeds, and BP. It is possible to select a query and a train to see details about the route. Each query uses colors and labels that spotlight trends and help operators detect potential issues. The queries react with alerts and flags as the stream flows, producing a visual alert when the query condition is satisfied.

We present a visualization that shows how the train's sound affects nearby buildings. Users can watch the train move along its route and see color changes that indicate noise levels. This representation helps users understand the relationship between the train's movement and its impact on the surrounding environment.

5.1.5 PLC Logic Verification

We validated the detection logic of the braking system, as shown in Query 6, in PostgreSQL environment. This step ensures that the streaming query not only meets the latency and throughput goals but also reproduces the fault-detection behavior that legacy PLC would produce. We translated the pattern into PostgreSQL using Common Table Expressions (CTE) and range-based window functions:

1. A ten-second look-back window computes the minimum and maximum FA values, flagging variances above the specified threshold in `varFA`.
2. From each flagged row, we derive `wstart` and `wend`, which delimit a secondary aggregation over FF.
3. A final filter retains only those intervals where the `varFF` variation remains below its threshold and the FA readings and event timestamps satisfy domain constraints (e.g., working hours).

To understand the results in Code 14 and Code 15, we mapped each SQL window and filter into sliding window operators, conditional predicates, and map transformations. We verified identical alert timings and outcome sets by replaying live SNCB/NMBS sensor readings through

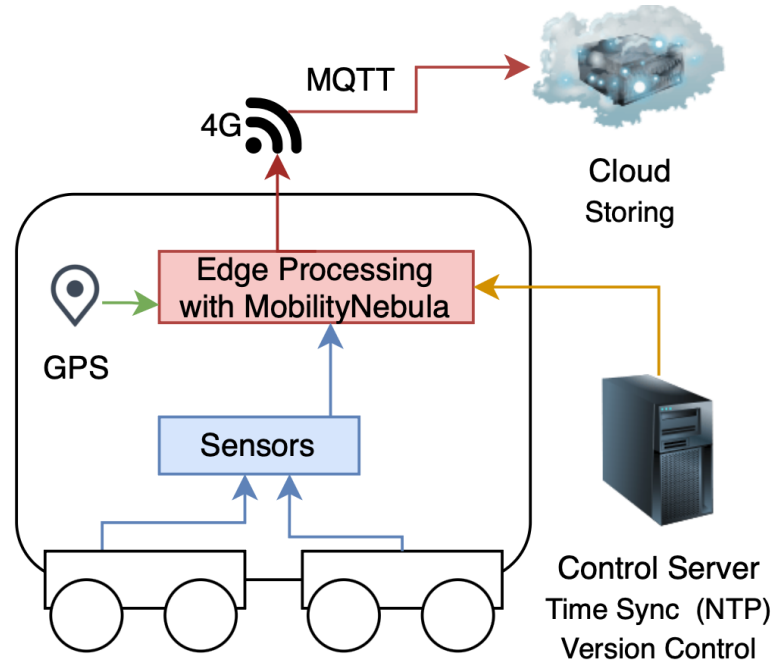


Figure 5.5: SNCB use-case architecture, showing onboard edge processing on the nROK device and MQTT-based alert delivery to the central monitoring backend

both the PostgreSQL query and our streaming query. This two-step approach, from the SQL prototype to the C++ streaming query, enables MobilityNebula to fully replace fixed-controller diagnostics while preserving the exact behavior of legacy PLC formulas.

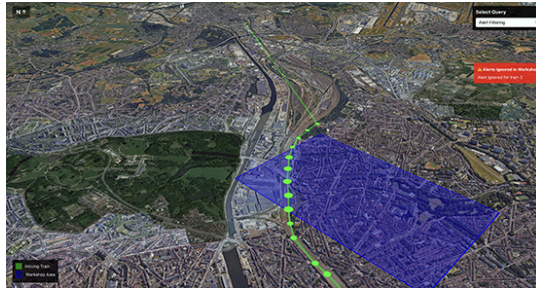
Code 14. Brake fault detection results for PostgreSQL

```
device_id, wstart, wend, varfa, varff
5,1727866234,1727866244,1.969,0.008
5,1727866235,1727866245,1.732,0.008
...
5,1727866241,1727866251,0.637,0.008
```

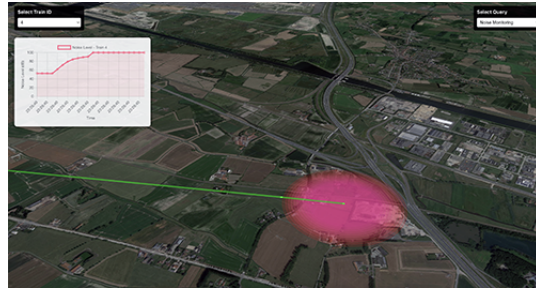
Code 15. Brake fault detection results for MobilityNebula

```
wStart,wEnd,varfa,varff
1727856300,1727866300,3.780,0.038
1727856400,1727866400,3.780,0.038
1727856500,1727866500,3.780,0.038
1727856600,1727866600,3.780,0.038
1727856700,1727866700,3.780,0.038
```

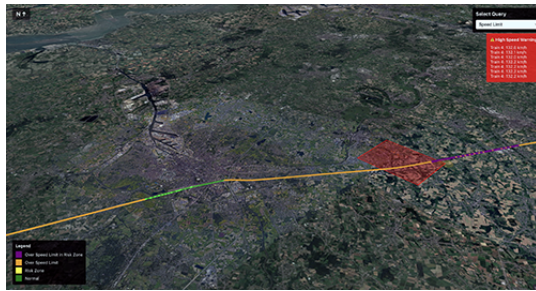
To verify that the SQL query and MobilityNebula’s query outputs match, we aligned the exact start and end timestamps of each method, taking into account how the windows are constructed and updated. MobilityNebula produces ten second windows that are updated with the event time. The SQL setup utilized the last seen timestamp, which often results in seven second windows. Cropping the original SNCB/NMBS dataset to the ten-second streaming window, from 1727856300 to 1727866300 (see Code 15), produces a maximum FA of 3.813 bar, a minimum



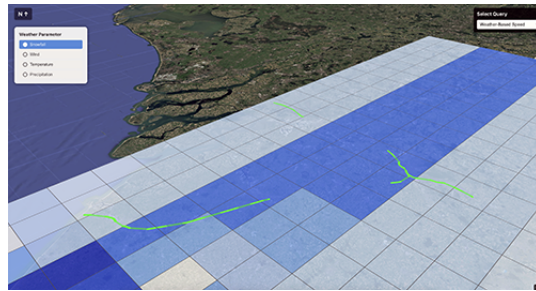
(a) Alert filtering



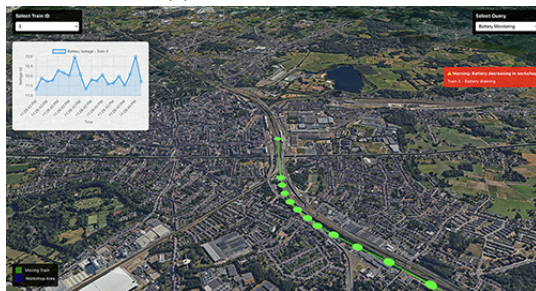
(b) Noise monitoring



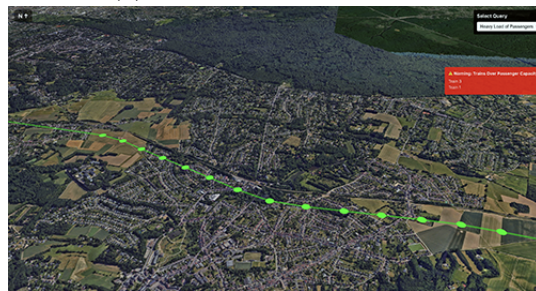
(c) Speed monitoring



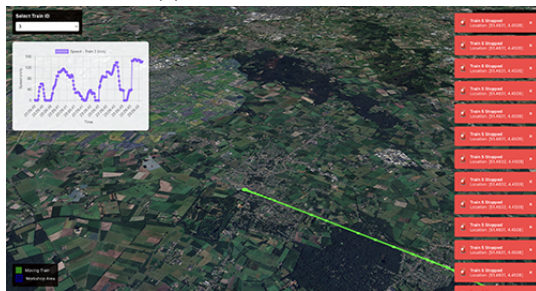
(d) Weather-based speed zones



(e) Battery monitoring



(f) Heavy load monitoring



(g) Unscheduled stops



(h) Brake monitoring

Figure 5.6: Dashboard visualizations of point-based streaming queries on SNCB data, including alerting, noise/speed monitoring, weather-based zones, and brake diagnostics

of 3.596 bar, an average of 3.7439 bar, and a variance of 0.217 bar. The corresponding FF values range from 3.967 to 4.005 bar with an average of 3.9847 bar and a variance of 0.038 bar. The SQL window has a seven-second interval and outputs identical FA variances over that subset from 1727866234 to 1727866251 (see Code 14). This one-to-one correspondence at the boundaries confirms that the declarative SQL pipeline and the MobilityNebula streaming query consume the same data points and produce equivalent diagnostics. MobilityNebula reproduces PLC brake-fault alerts while sustaining 20k events/s with near real-time latency and operating within a 1 vCPU/512 MB edge budget, making it a deployable replacement for fixed-controller diagnostics.

In summary, the SNCB/NMBS use case demonstrates the practical deployment of MobilityNebula under real operational conditions. The system operates continuously on edge devices aboard M6 trains, executes spatiotemporal queries in real-time, and delivers results equivalent to SNCB/NMBS's existing diagnostic logic. These results confirm the feasibility of distributed edge-to-cloud mobility analytics in railway environments.

5.1.6 Deployment Lessons Learned

The deployment of MobilityNebula to the SNCB/NMBS rail network involved more than just installing a research prototype on an edge device. The technical challenges that are not necessarily algorithmic are a consequence of build restrictions on embedded systems, domain restrictions imposed by the rail environment, and the disparity between domain logic for diagnosis and the semantics of continuous queries. This section describes the problems that required changes, the changes needed to solve them, and the learning experiences from a real deployment.

Cross Compilation and Reproducible Builds

Initial testing was conducted using an older version of NebulaStream that could be compiled natively on the device. For further testing and continuous development, the new version of NebulaStream was used, which introduced a Docker-based build process. The nROK environment imposes strict memory and computational constraints. However, for the updated build process, we lacked an ARM-compatible compiler capable of building the Docker image and compiling NebulaStream. Consequently, a two-step build process was implemented, with a precompiled ARM-compatible Docker image created for the nROK rather than building directly on the nROK. Compilation for the ARM environment was performed on an AWS ARM instance, ensuring architectural compatibility and sufficient resources to prevent deployment failures due to build issues.

Enterprise Network Constraints

The SNCB/NMBS network environment is highly firewalled. Some components failed in a way that was initially difficult to assess, since these errors had not occurred in an academic or home environment. In a particular case, a NebulaStream component made the implicit assumption that it could download packages from a Bitbucket repository, which was not possible within the SNCB/NMBS environment. This highlighted a crucial requirement for the edge software stack. It should support mirroring or pre-fetching all dependencies. We removed the assumption that the deployment be externally downloadable and made it work as a single pre-compiled image.

TCP Source

NebulaStream’s original TCP source was designed primarily as a client that connects to a remote server, rather than as a server that listens for inbound connections. In the SNCB/NMBS setup, the nROK was required to host the endpoint processes that would receive data pushes from the braking monitoring system. NebulaStream’s TCP source was then modified to handle this requirement and support the server role, and to accept connections, ensuring that the integration of the existing SNCB/NMBS data producer was achieved without requiring additional scripts that would increase the probability of failure and complexity of operation.

The MQTT Semantic

Early versions of the MobilityNebula deployment used MQTT settings that were sufficient in controlled environments. Still, they were too restrictive for a rail setting where intermittent connectivity and broker reconnections are routine. In our case, the primary limitation was not the MQTT protocol itself, but the expressiveness of our integration: the MQTT sink interface did not expose reliability options beyond the default configuration. We therefore extended the MQTT sink syntax to explicitly accept and configure *QoS 1* and *QoS 2*. This change allowed deployments to select delivery guarantees appropriate to the operational requirement, at least once for routine telemetry and exactly-once-style delivery semantics (at the protocol level) for safety-relevant notifications.

Performance Problems

The nROK devices host various diagnostics and operational services and cannot be dominated by a single pipeline stream. We had a bug in NebulaStream during our deployment that caused 100% CPU usage. This bug was identified and fixed by the NebulaStream team. We incorporated the patch into the MobilityNebula codebase and redeployed the system within the SNCB/NMBS limits for resource sharing. While the corrective action addressed 100% CPU usage, the experience emphasizes the need for edge feasibility that’s not only defined by latency but also by the use of shared device resources.

Query Deployment Automation

The manual multi-terminal operations process was replaced with automated operations. While preparing and testing the deployment, it was possible to start the worker in one terminal, the query in another, and engage manually when the components were restarted. However, on the train, this was not feasible. The system had to start automatically, auto-deploy the necessary queries, and recover from regular restarts without manual intervention. Therefore, we established a Docker Compose setup to automatically start the NebulaStream engine, complete the query registration steps, and initiate the auto-deployment of the query. Further, since the nROK platform was regularly restarted, we ensured the deployment could withstand daily restarts and resumed the pipeline as part of the standard start-up procedure.

Domain Diagnosis Expression into Continuous Queries

A significant portion of our deployment efforts was studying the use case. We had to learn what monitoring logic is encoded in PLC expressions, understand the brake system, its potential failures, how pressures are measured, and finally translate this knowledge into continuous queries. This translation demanded a proper treatment of these concepts. For instance, a slow brake and a fast brake may denote a normal or an emergency brake, depending on the BP, which we must

formalize in our logic. The monitoring queries required the use of statistical operators, such as variance, to effectively capture fluctuations in the pressure signals. The use of these statistical operators was essential in effectively expressing the logic concisely. The use of additional operators, such as those above, improved not only the expressiveness but also the maintainability of the overall set of queries.

5.2 Outlier Detection

Outlier detection and cleaning are essential steps in data preprocessing to ensure the integrity and validity of data analyses. In mobility applications, trajectories may contain both isolated measurement errors and systematic drifts, and the choice of detection method must match the data arrival model. Offline, or *static* methods assume the whole trajectory is available and can afford global consistency checks. Online, or *streaming* methods must operate under strict latency and memory constraints, updating their models incrementally as new points arrive. This section surveys both classes of approaches. Section 5.2.1 reviews libraries that handle static trajectory outliers. Section 5.2.2 then examines libraries designed for high-rate and low-latency streams. Table 5.1 lists each mentioned module with its name, the outlier detection technique used, the category of the outlier detection method, and the implementation language.

5.2.1 Static Outlier Detection

The reviewed works implement outlier detection algorithms that determine outliers once all trajectory records are available. We refer to these algorithms as static outlier detection algorithms. Static methods typically perform a single pass over the complete trajectory, build global models of standard movement, and then flag deviations against those models. They can leverage batch optimizations and global consistency checks to achieve high accuracy. However, they incur large memory and computation costs and cannot react to new data until the dataset is reprocessed. Therefore, static detectors are suitable for offline cleaning and validation, but they cannot support real-time monitoring or adaptive pipelines.

MovingPandas [38]³ is a Python library focused on trajectories of moving objects. Data can be represented in Pandas [72], GeoPandas,⁴ HoloViz [110], CSV, JSON, and GeoJSON. MovingPandas implements data structures in Python for the interaction and analysis of movement. The library has many trajectory manipulation functions. Focusing on outlier detection, MovingPandas implements the Kalman Filter (KF) algorithm for outlier detection, which is implemented using the Stone Soup software [100].

Scikit-learn [86]⁵ is a Python library focusing on the ease of use of machine learning algorithms. The library supports supervised and unsupervised learning algorithms, such as standard linear regression models and neural networks. It also includes tools for model fitting, data preprocessing, model selection, and evaluation. The library applies the Local Outlier Factor (LOF) algorithm and the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) for outlier detection. For LOF, the process begins by selecting features of interest, which are then isolated for further processing. It then computes the local density deviation of a data point with its neighbors. Outliers, which are samples having a considerably lower density than their neighbors, are identified by fitting the LOF model to the standardized data and determining instances where the model's prediction is inaccurate. Scikit-learn handles duplicates by using a small value instead of zero when calculating the Local Local Reachability Density (LRD), which can result in a

³<https://github.com/anitagraser/movingpandas-examples>

⁴<https://geopandas.org/en/stable/>

⁵<https://scikit-learn.org/stable/>

potentially high lrd value. Scikit-learn implements the DBSCAN algorithm for finding clusters of arbitrary shape and identifying outliers with parameters of a minimum number of points necessary to form a dense region and a maximum distance between two points to be considered neighbors.

Scikit-mobility [84]⁶ is a Python library that extends Pandas [72]. Scikit-mobility offers functions for preprocessing and cleaning trajectories, as well as analysis. The library uses heuristic filtering for outlier detection based on speed and a given threshold. This approach can be practical for identifying points in the trajectory that deviate significantly from the expected pattern, but may not be as accurate as other methods that use estimation to handle outlier points. Scikit-mobility can also employ Scikit-learn's DBSCAN algorithm for outlier detection.

PTRAIL [91]⁷ is a Python package that uses parallel computation and vectorization, making it suitable for large datasets. The package implements several preprocessing steps, such as feature extraction, filtering, interpolation, and outlier detection. PTRAIL removes outliers using a Hampel Filter (HF) [103] based on the distance and speed of the points between consecutive points. For each trajectory, the method calculates the median of a sliding window and adjacent points on each side of the trajectory. The HF also estimates the standard deviation of each point about its window median using the median absolute deviation. If a measurement differs from the median by more than the threshold multiplied by the standard deviation, the filter replaces the sample with the median.

PyMove [92, 81]⁸ is a Python library. The library implements operations for data preprocessing and pattern mining. PyMove also provides tools for data visualization, allowing users to explore and understand their data through various techniques and channels. PyMove detects outlier points based on the distance traveled, as well as the minimum and maximum speed.

MoveTk [22]⁹ is C++ library. The library implements tools for constructing, cleaning, and analyzing trajectory data. One of the key features of MoveTk is its implementation of the optimal Speed-Bounded (GSB) and optimal Acceleration-Bounded (GAB) algorithms for outlier detection. In addition to these algorithms, MoveTk implements other methods for outlier detection, including greedy and smart greedy approaches. These methods build upon the basic speed and acceleration-bounded algorithms, incorporating additional strategies and techniques to enhance their performance and accuracy. For the greedy approach, the algorithm builds a subsequence by testing if the new measurement is consistent with the last in the subsequence under the GSB model or the GAB model. In addition, the authors implement a Smart Greedy Speed-Bounded (SGSB) and Smart Greedy Acceleration-Bounded (SGAB). With SGSB and SGAB, multiple subsequences are tracked simultaneously. The next measurement is added to all subsequences that end in a consistent measurement. If there is no such subsequence, a new subsequence starting with the measurement is created. Ultimately, the longest subsequence is returned. Also, as a baseline, the library implements its interpretation of the method described in [123] as Local Greedy Speed-Bounded (LGSB). In LGSB, a graph is constructed with a vertex per measurement. Two vertices are connected if their timestamps are successive in the original trajectory and they are consistent with the speed bound. A measurement is added to the output if and only if its vertex is in a connected component of a given size. LGSB does not guarantee that the complete output is consistent according to the speed bound [123].

Argosfilter [33]¹⁰ is an R package that implements functions for managing trajectory data. The outlier detection in Argosfilter uses two different methods: one based on speed and the other based on location. The speed-based method is similar to the one provided in [123]. In contrast,

⁶<https://github.com/scikit-mobility/scikit-mobility>

⁷<https://github.com/YakshHaranwala/PTRAIL>

⁸<https://github.com/InsightLab/PyMove>

⁹<https://github.com/movetk/movetk>

¹⁰<https://cran.r-project.org/web/packages/argosfilter/argosfilter.pdf>

the location-based method is based on the algorithm described in the paper [33]. This method uses a set of spatial constraints, such as a minimum distance between two points or a maximum distance from a reference point, to identify and remove outliers from a trajectory.

StMove [94]¹¹ is an R package library. The library provides construction, filtering, and outlier detection functions. For the outlier filter, StMove applies the KF.

mvoutlier [30]¹² is an R package that is a tool for identifying outliers in multivariate data. One of the primary strategies employed by this package is distance-based outlier detection, in which the distance of each observation from others or a central directional measure is calculated. Observations falling beyond a predefined distance threshold are then tagged as outliers. Among the methods provided by the mvoutlier package are univariate outlier detection, which performs outlier detection for each variable individually, and Doutlier, which detects multivariate outliers using depth-based methods. The package also incorporates the Minimum Covariance Determinant method and the Mahalanobis distance method, which are adept at accommodating covariance between variables.

The authors in [66] implement a partition-and-detect framework for trajectory outlier detection. The method is composed of two phases, partitioning and detection. In the partitioning phase, each trajectory is split into a coarse granularity and then into a finer granularity. The detection phase computes the distance between partitions and flags those whose positions or directions stray significantly from their neighbors. This method can detect positional and angular outliers. Positional outliers are those trajectories with a different location from their neighboring trajectories. Angular outliers differ from the direction of neighboring trajectories.

Name	Outlier Detection Method	Category	Language
PTRAIL	Hampel Filter	Statistic	Python
MovingPandas	Kalman filter	Statistic	Python
Scikit-mobility	Maximum speed / DBSCAN	Heuristic / Clustering	Python
PyMove	Maximum / minimal speed	Heuristic	Python
Scikit-learn	Local Outlier Factor / DBSCAN	Density / Clustering	Python
StMove	Kalman filter	Statistic	R
Argosfilter	Maximum / minimal speed	Heuristic	R
mvoutlier	Minimum Covariance Determinant	Statistic / Distance-based	R
MoveTk	Linear speed bound	Heuristic	C++
Partition-and-Detect	Distance-based / density-based	Heuristic / Clustering	C++
MEOS heuristic	Maximum / minimal speed	Heuristic	C
MEOS EKF	Extended Kalman filter	Statistic	C

Table 5.1: Overview of static trajectory outlier-detection libraries, listing each library, its detection technique, method category, and implementation language

¹¹<https://tinyurl.com/stmove>

¹²<https://cran.r-project.org/web/packages/mvoutlier/index.html>

5.2.2 Streaming Outlier Detection

Streaming outlier detection algorithms flag anomalies as new points arrive, without waiting for a complete dataset. They keep a compact, up-to-date model, such as a sliding window of recent values, an incremental summary sketch, or a local density estimate, and test each incoming record against that model. If a point deviates sharply from the model, for example, by exceeding speed or density thresholds, the algorithm marks it as an outlier. By limiting memory use and per-record computation, streaming detectors bind their latency and scale to high-rate data sources. They trade some global accuracy for immediate response and continuous operation, making them well-suited for real-time monitoring and edge-to-cloud pipelines. Table 5.2 highlights each mentioned library with its name, the outlier detection technique used, the category of the outlier detection method (as defined in Chap. 3), and the implementation language.

River [73]¹³ is a Python library for online learning on streaming data that merges two earlier toolkits, Creme¹⁴ and scikit-multiflow.¹⁵ River represents each incoming record as a feature-value dictionary. River supports a sliding-window strategy to retain recent data and implements stateful functions, such as learners and transformers, that evolve incrementally with each new data point, maintaining internal state to enable online learning on data streams. The library implements multiple incremental learners, including decision trees, linear models, clustering methods, drift detectors, and streaming anomaly detectors. River contains methods including isolation forest, approximate k -Nearest Neighbors (k -NN), Z-score, and online k-means.

Python Streaming Anomaly Detection (PySAD) [112]¹⁶ is a library for detecting anomalies in real-time data. PySAD organizes its operational pipeline into data cleaning, feature projection, anomaly scoring, calibration, smoothing, thresholding, and online evaluation. The library supports single-pass models, including Half-Space Trees, incremental Principal Component Analysis (PCA), and k -NN-CAD. Each model updates its internal state and returns an anomaly score based on the recent values. PySAD integrates with Scikit-learn and PyOD.¹⁷

Merlion [11]¹⁸ is a Python library for time-series forecasting and anomaly detection. The library implements models for statistical time-series forecasting, such as the Autoregressive Integrated Moving Average (ARIMA) model, as well as deep-learning approaches. Merlion incorporates automated hyperparameter tuning, model ensembling, and calibrated anomaly scoring to enhance interpretability and minimize false positives.

The Massive Online Analysis (MOA) [63]¹⁹ framework is a Java toolkit for mining data streams. The framework combines stream generation, algorithm implementation, and evaluation. MOA implements classifiers, such as Hoeffding Trees, Naive Bayes, and stochastic gradient descent. For clustering, MOA offers CluStream, DenStream, and StreamKM++, which maintain fixed-size summaries of incoming data. CluStream stores compact cluster statistics that evolve without retaining all points. DenStream detects clusters by tracking high-density regions, allowing it to capture arbitrarily shaped groups. StreamKM++ constructs a small core set of representative points and applies batch clustering to that core set. The framework also supports outlier detection, concept drift handling, and recommender systems.

Apache Scalable Advanced Massive Online Analysis (SAMOA) [75] is a distributed framework for stream mining on platforms such as Storm and Apache S4. Written in Java, it reproduces MOA in its support for streaming tasks, including clustering, classification, and frequent pattern

¹³<https://riverml.xyz/dev/>

¹⁴<https://github.com/sroecker/creme>

¹⁵<https://scikit-multiflow.github.io>

¹⁶<https://pysad.readthedocs.io/>

¹⁷<https://github.com/yzhao062/pyod>

¹⁸<https://github.com/salesforce/Merlion>

¹⁹<https://moa.cms.waikato.ac.nz>

mining. SAMOA also integrates several MOA clustering algorithms for large-scale deployment.

Stream [45]²⁰ is an R package that provides a framework that integrates data stream simulation, algorithm implementation, and evaluation in a unified and extensible manner. The package establishes common abstractions for streaming sources, sliding windows, and plug-in slots for mining tasks. The package implements algorithms, including stream MOA, K-means, and DBSCAN. These algorithms process each record incrementally and generate micro-cluster summaries.

The authors in [53] integrate an EKF into Apache Flink to track vessel trajectories at scale (referred to as Flink + EKF in Table 5.2). The authors model each ship’s trajectory using a state-space model and implement both the prediction and correction steps of the EKF within a distributed dataflow. By keying state per vessel, their system maintains each filter’s a posteriori state and covariance, updates them upon each AIS report, and forecasts the next position.

Name	Outlier Detection Method	Category	Language
River	Isolation Forest / Approx. k -NN / Z-score / online k-means	Unsupervised / Density	Python
PySAD	Half-Space Trees / Incremental PCA / k -NN-CAD	Tree / Statistic / Density	Python
Merlion	ARIMA / Deep models (and ensembles)	Statistic / Deep Learning	Python
MOA	HST, LOF variants, clustering outliers	Tree / Density / Clustering	Java
Apache SAMOA	Distributed MOA detectors	Tree / Density / Clustering	Java
Stream	Micro-cluster outlier scores	Clustering	R
Flink + EKF	Extended Kalman filter	Statistic	Java
MobilityNebula	Extended Kalman filter	Statistic	C++

Table 5.2: Overview of streaming outlier-detection libraries and systems, listing each method, its category, and implementation language

5.3 Real-Time Outlier Detection

Trajectory data refers to the path a moving object follows through space over time and has extensive applications across fields such as transportation, logistics, and environmental studies. However, outliers or noise often affect this data type due to sensor errors and connectivity issues. In trajectory data, outliers can appear as sharp or unrealistic changes in direction, speed, or heading. Such irregularities may distort analytical results and produce misleading conclusions in downstream aggregation or prediction tasks.

5.3.1 Theory and Implementation

We implemented an EKF within MEOS and exposed it in MobilityNebula to support online detection of such anomalies. The filter operates on temporal types, including `tfloat`, `tsequence`, and `tsequenceset`, computing Δt between consecutive observations and rebuilding the transition

²⁰<https://cran.r-project.org/web/packages/stream/>

and process noise matrices $F(\Delta t)$ and $Q(\Delta t)$ at every step. Each new record represents a measurement update in a state-space model, where the state corresponds to position and velocity, and the measurement corresponds to the observed value.

The implementation builds on the TinyEKF library,²¹ which provides a Cholesky-based update. We extended it to support temporal data with variable Δt , dynamic state and measurement dimensions, and per-sample innovation gating based on the Normalized Innovation Squared (NIS). Gating is a statistical test applied before each EKF correction to determine whether a new measurement should be trusted or ignored as an outlier, defined by its uncertainty. We compute the innovation $\nu_k = z_k - h(x_{k|k-1})$, its covariance $S_k = H_k P_{k|k-1} H_k^\top + R_k$, and the NIS value $d_k^2 = \nu_k^\top S_k^{-1} \nu_k$. In the implementation, we use the Mahalanobis distance $d_k = \sqrt{d_k^2}$ and reject a measurement when d_k exceeds a user-chosen threshold `gate_sigma`. When this occurs, the filter either emits the predicted value (*fill mode*) or skips the timestamp (*drop mode*).

Our implementation currently supports temporal numeric series and planar spatiotemporal point trajectories (for example, projected GPS tracks). In both cases, we use the TinyEKF C library as the core filtering engine, instantiated with a Constant-Velocity (CV) state model and direct position measurements. Concretely, for a two-dimensional trajectory, the state is

$$x = [x \quad v_x \quad y \quad v_y]^\top,$$

where (x, y) are positions and (v_x, v_y) are velocities, and the measurement is

$$z = [x \quad y]^\top.$$

The transition and measurement models are linear,

$$x_{k|k-1} = F(\Delta t) x_{k-1|k-1}, \quad z_k = H x_k,$$

with $F(\Delta t)$ and H defined as in the constant-velocity model above. In this linear setting, the extended Kalman filter equations implemented by TinyEKF reduce exactly to the standard Kalman filter recursion, i.e., no Jacobian linearisation is required since the dynamics $f(\cdot)$ and the measurement function $h(\cdot)$ are linear.

The CV model assumes that an object moves at approximately constant speed between updates. The model estimates both position and velocity while accounting for process noise due to unmodeled acceleration. Per spatial axis, it uses

$$F_i(\Delta t) = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}, \quad H_i = [1, 0], \quad Q_i(\Delta t) = q \begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} \\ \frac{\Delta t^3}{2} & \Delta t^2 \end{bmatrix}.$$

This form corresponds to the discretized acceleration model and is well-suited for situations where measurements are frequent, and acceleration variations remain relatively small.

In the CV model, the position is initialized from the first observation, and the initial velocity is set to zero with a comparatively large covariance. This broad prior allows the filter to adapt its velocity estimate as subsequent measurements arrive. For a D -dimensional trajectory, the EKF state dimension is $N = 2D$ (position and velocity per axis), and the measurement dimension is $M = D$ (position per axis). Internally, MEOS allocates fixed upper bounds for N and M and activates only the components corresponding to the requested spatial dimension.

After prediction, we compute the innovation $\nu = z - h(x)$ and its covariance $S = HPH^\top + R$. The NIS, also known as the squared Mahalanobis distance, is given by $d^2 = \nu^\top S^{-1} \nu$, and the

²¹Adapted from TinyEKF by Simon Levy. <https://github.com/simondlevy/TinyEKF/tree/master>

implementation compares the corresponding distance $d = \sqrt{d^2}$ to the threshold `gate_sigma`. A measurement is rejected when $d > \text{gate_sigma}$, where `gate_sigma` can be chosen in relation to the χ^2 distribution for M degrees of freedom. For $M = 2$, typical χ^2 thresholds of 5.99 (95%), 9.21 (99%), and 11.83 (99.7%) correspond to `gate_sigma` values of approximately 2.45, 3.04, and 3.44, respectively. A rejected update is one whose innovation lies beyond the expected range given the model and its noise assumptions, and it is treated as an outlier relative to the current prediction. Otherwise, the Kalman gain $K = PH^T S^{-1}$ is applied, and both the state and covariance are updated. Larger values of `gate_sigma` make the gate more permissive (fewer measurements are rejected), while smaller values impose stricter outlier rejection and reduce trust in noisy observations [55].

Matrix S is factorized using a Cholesky decomposition $S = LL^T$, and a Cholesky based inversion is used to obtain S^{-1} without relying on generic Gaussian elimination routines. If S is not symmetric positive definite, the gating step falls back to accepting the measurement (no rejection) and proceeds to the standard update.

We have implemented rejected updates in two modes. In *fill mode*, timestamps are preserved, and the filter outputs the model prediction, producing a smooth and uninterrupted trajectory suitable for visualization or long-term analysis. In *drop mode*, the timestamp is omitted, producing a shorter subsequence that contains only the accepted measurements and predictions.

The EKF recursion The algorithm follows the predict-update cycle:

$$\begin{aligned}
 \text{Predict: } \hat{x}_{k|k-1} &= f(\hat{x}_{k-1|k-1}, u_k, \Delta t), \\
 P_{k|k-1} &= F_k P_{k-1|k-1} F_k^T + Q_k(\Delta t), \\
 \text{Update: } \nu_k &= z_k - h(\hat{x}_{k|k-1}), \quad S_k = H_k P_{k|k-1} H_k^T + R_k, \\
 K_k &= P_{k|k-1} H_k^T S_k^{-1}, \\
 \hat{x}_{k|k} &= \hat{x}_{k|k-1} + K_k \nu_k, \quad P_{k|k} = P_{k|k-1} - K_k (H_k P_{k|k-1}).
 \end{aligned}$$

5.3.2 Integration and Workflow

The EKF workflow in MEOS has four stages: preprocessing, initialization, filtering, and postprocessing. In preprocessing, temporal values of types `tfloat` and `tgeompoint` are accepted for filtering, while other temporal types, as well as geodetic `tgeompoint` values, are returned unchanged.

Initialization sets the first position from the initial observation, assigns zero velocity, and uses a large prior covariance on the state. This large prior allows the filter to converge gradually as updates arrive without overcommitting to the initial velocity estimate. During filtering, the system recomputes $F(\Delta t)$, $Q(\Delta t)$, and S at each step. By construction, a larger Δt increases $Q(\Delta t)$, reflecting higher process uncertainty between widely spaced observations and broadening the predicted covariance. Gating and update are applied to each new measurement independently, in either drop or fill mode.

Postprocessing converts the filtered state sequence back into the corresponding temporal type, producing cleaned trajectories that can be processed in streaming queries, such as geofencing, event detection, or trajectory similarity analysis. For inputs that were not eligible for filtering (for example, geodetic coordinates left in degrees), MEOS forwards the original trajectory.

Although the experiments in this chapter use a linear constant-velocity model with linear position measurements, so the recursion reduces to the standard Kalman filter, we retain the EKF terminology since the MEOS implementation follows a generalized EKF interface (as in TinyEKF), supports variable state/measurement dimensions, and is designed to be extended to non-linear motion or measurement models in future work.

5.3.3 Outlier Detection in Historical Data and Real-Time Data

Outlier detection methods differ depending on whether data are processed offline or in real-time. In historical processing, the complete trajectory is available beforehand, allowing algorithms to leverage global context, such as the overall trajectory shape, long-term consistency, and relationships with neighboring objects. Methods, including DBSCAN or LOF, can recompute averages, perform interpolation, and apply density-based clustering, but they are computationally expensive and unsuitable for continuous streams.

Real-time processing, in contrast, deals with a constant flow of new observations. The goal is to detect outliers and maintain low latency and uninterrupted operation. Streaming detectors update models and thresholds incrementally. They evaluate only a short recent history or a running prediction instead of reanalyzing all past data.

The EKF in MEOS follows this real-time approach. In our implementation, the filter maintains a continuous estimate of position, velocity, and uncertainty, updating it with each new observation and comparing the result against the model prediction using an NIS-based gate. When an observation deviates more than expected from the predicted value, given the model uncertainty, we treat it as an outlier and ignore that update according to the configured drop or fill policy. These decisions occur continuously as data arrive, allowing the system to keep pace with incoming telemetry without revisiting records.

Adapting a historical method for streaming applications can be feasible if we consider a windowed trajectory as a complete trajectory. This approach offers the advantage of utilizing various methods designed for historical data in streaming mode. However, it may not accurately reflect real-time streaming and can increase computational costs, as each window requires complex calculations. In addition, it can compromise accuracy since only a sub-trajectory is analyzed, rather than the entire trajectory.

5.4 Experimental Evaluation

Our experimental evaluation focuses on data quality, specifically on detecting and removing outliers in trajectory data. In mobility datasets, outliers arise from sensor noise, GPS drift, packet loss, and other measurement inconsistencies. Because these errors can significantly impact downstream analytics, such as trajectory reconstruction, geofencing, and event detection, it is crucial to assess the effectiveness of existing tools and validate the proposed ground-truth methodology. We first describe our approach for constructing a reliable ground-truth by cross-referencing multiple sensor signals to ensure consistency in speed and bearing. We then use these ground-truths to benchmark ten open-source libraries across four datasets, comparing their accuracy and runtime performance using precision, recall, and multiple F score variants.

5.4.1 Constructing The Ground-Truth

Constructing a ground-truth for trajectory data analysis is a task that presents several challenges. The difficulties include the cost, accuracy, and scale of the required ground-truth data. Additionally, cleaning the data to remove errors and noise can be time-consuming and may only sometimes produce reliable results.

Because mobility datasets rarely include manually verified outlier labels, an operational ground-truth is constructed by applying a rule-based consistency procedure to each dataset. The resulting label set serves as a reproducible reference for a fixed configuration of the procedure, rather than representing an absolute definition of anomaly. This distinction is essential for interpreting precision and recall values, as these metrics quantify agreement with a specific

anomaly definition determined by the ground-truth configuration.

We present a method that cross-references data that originate from multiple sensors that belong to the same source. For example, the moving object has multiple sensors, such as GPS, gyrocompass, and speed sensors. Hence, the receiver receives multiple signals for the same source. Our method constructs the ground-truth by cross-referencing these signals. The intuition is that it is possible to receive faulty data for one sensor, it is much less likely that the other sensors are faulty at the same time.

Hence, our proposed method offers a different approach to traditional methods for constructing the ground-truth, such as manual annotation or data cleaning techniques including those described in [108, 111, 74]. It also allows for data integration from multiple sensors, increasing the overall accuracy of the ground-truth. We consider speed and bearing to accommodate both errors in direction and speed. Figure 5.7 illustrates the different errors one can encounter in a trajectory for speed and bearing. Algorithm 1 calculates and compares the speed and bearing with the recorded values in the data. The input consists of speed and heading thresholds (tS and tH) and the file path. For each point in the file, first we check if the IDs are the same, i.e., the points belong to the same trajectory. Second, we calculate the speed and bearing between the point and its successor. We then compare the newly calculated speed and bearing with the sensor-recorded values in Algorithm 1. If the difference in speed or heading is greater than the input thresholds, the point is considered an outlier and added to the output array. The proposed method involves iterations to define an optimal threshold for the studied parameters: heading and speed. We used visualizations of original trajectories and corrected trajectories to validate the ground-truth method. The fine-tuning can be validated by splitting the dataset and cross-validating visually. In the future, we plan to develop a self-tuning algorithm for thresholds based on machine learning. In addition, we are creating an artificial dataset where we can add errors and outliers to better validate the method.

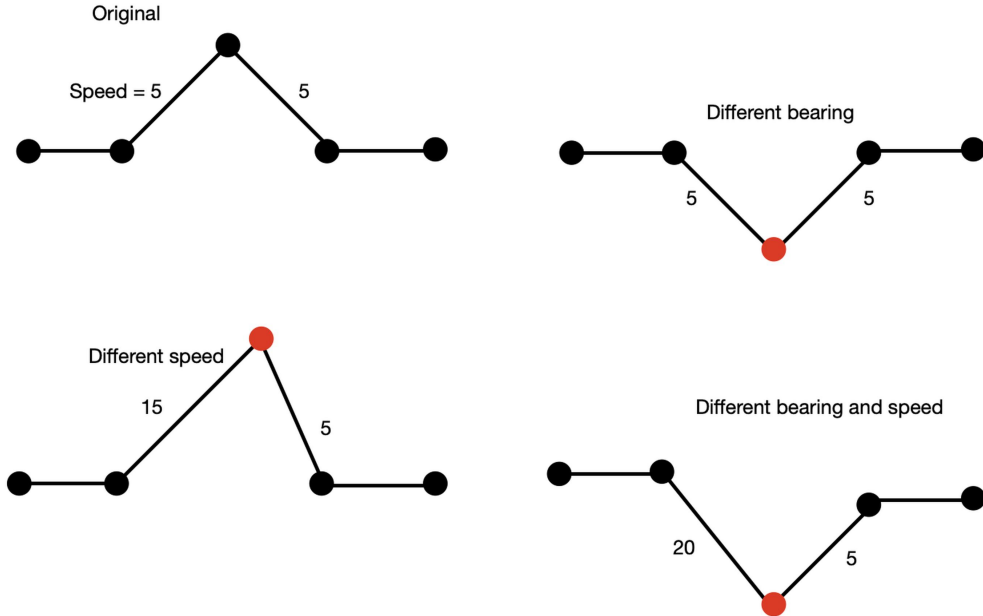


Figure 5.7: Examples of trajectory anomalies targeted by the ground-truth procedure, showing deviations in speed and/or bearing

Algorithm 1: Pseudo-code for constructing the ground-truth by comparing computed speed and bearing against sensor-reported values

Input: thresholdSpeed tS , thresholdHeading tH , FilePath $file$
Output: Outliers
 Outliers $\leftarrow \{\}$;
 file $\leftarrow$ READFILE (FilePath);
 file $\leftarrow$ ORDERBYIDANDTIMESTAMP (file);
 speed $\leftarrow 0$;
 bearing $\leftarrow 0$;
foreach point $\in file.length$ **do**
 if file[point].id == file[point+1].id **then**
 speed $\leftarrow$ CALCULATE SPEED(point, point + 1);
 bearing $\leftarrow$ CALCULATE BEARING(point, point + 1);
 if |speed - file[point].sensorRecordedSpeed| $\geq tS$ **OR**
 |bearing - file[point].sensorRecordedBearing| $\geq tH$ **then**
 Outliers.append(point);
return Outliers

Our method involves cross-referencing the data from multiple sensors to detect discrepancies and annotate them as ground-truth for outliers. We apply this method to data from modern multi-sensor tracking, such as GPS, AIS, ADS-B, Mode S, TCAS, and FLARM sensors. By combining the data from multiple sensors, we can increase the overall accuracy of the ground-truth in the presence of errors or noise in individual sensors. When comparing data from different sensors, we can analyze the data to see if there are any discrepancies or inconsistencies between all sensors. In addition, statistical tests can determine whether the data is consistent with a particular hypothesis or model. For example, we can check the calculated speed with the speed given in the data.

Our datasets for this experimental study consist of multi-sensor trajectory data. Each dataset contains data from different sensors. For example, AIS data is composed of data originating from multiple sensors, such as the speed captured by the sensor and location informed by the GPS. The first sensor is AIS data: AIS is the location tracking system for sea vessels. This data is collected from participating ships worldwide, providing diverse trajectories. AIS data is composed of data originating from multiple sensors. An AIS message is composed of 26 elements. These messages contain information from sensors such as gyrocompass, rate of turn indicator, GPS, VHF radio frequency transceivers, Information Systems (ECDIS), and radar. Each data point provides information on timestamp, type of mobile, id, latitude, longitude, heading, speed over ground, course over ground, ship type, cargo type, width, length, and destination, among other information. We utilize a total of 4.3 GB. Figures 5.8 and 5.9 illustrate the AIS dataset before and after applying the ground-truth filtering procedure.

OpenSky Network²² offers raw data stored in a historical database to study and improve air traffic control technologies and processes. Figure 5.10 shows an illustration of the raw data. Each data point in the dataset has time, id, latitude, longitude, speed, heading, Barometer altitude and altitude, timestamp, and other additional information.

Jakarta and Singapore datasets consist of GPS trajectories from Grab Tech²³ drivers' phones in Singapore and Jakarta cities in southeast Asia. Grab-Posisi [48] covers over 1 million kilometers

²²<https://opensky-network.org>

²³<https://www.grab.com/sg/>

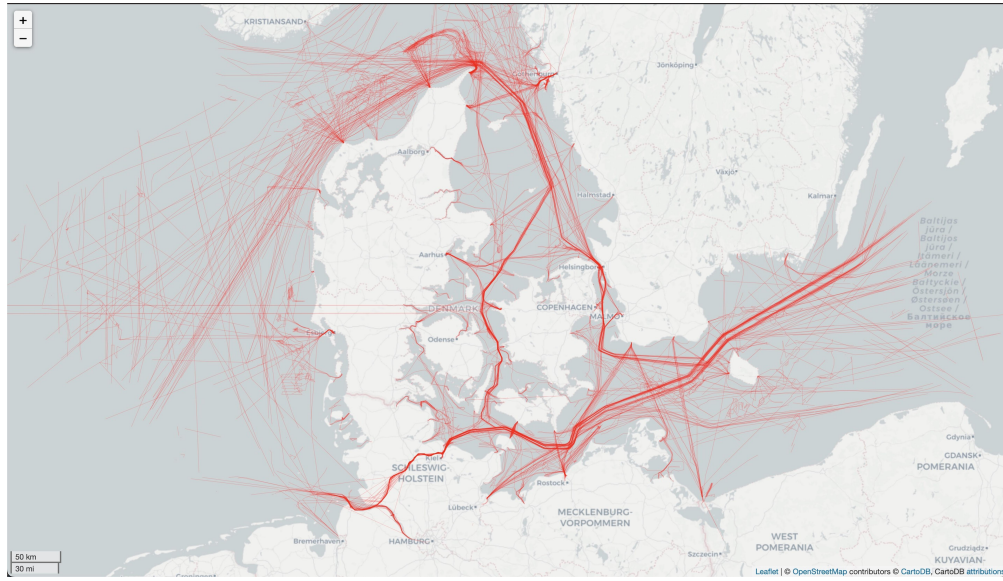


Figure 5.8: Visualization of the AIS dataset before applying the ground-truth filtering procedure

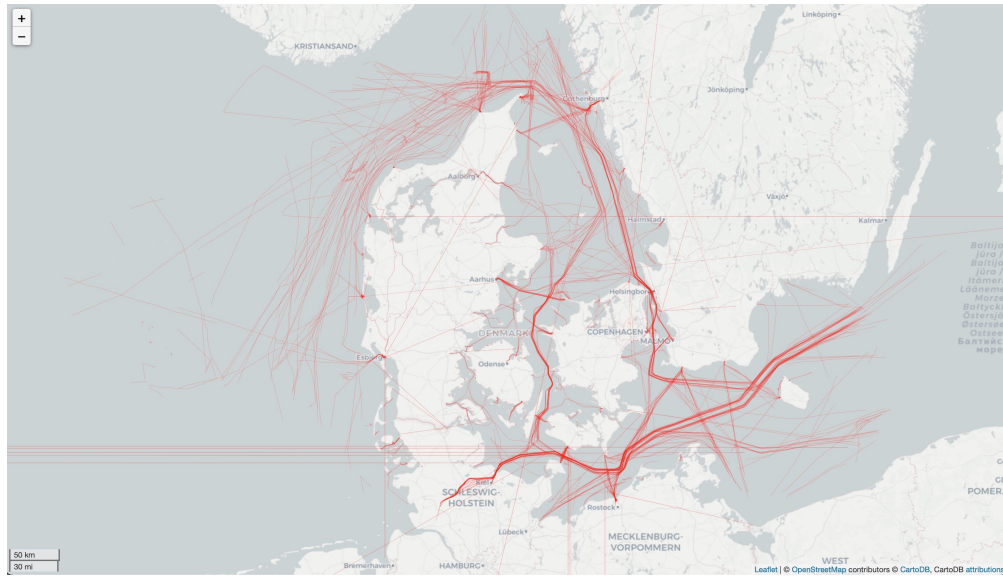


Figure 5.9: Visualization of the AIS dataset after applying the ground-truth filtering procedure

and over 88 million points. Each trajectory has information on speed, heading direction, area, and distance covered during the travel and duration. Figures 5.11 and 5.12 show the raw data.

Table 5.3 reports the total number of records, the number of outliers detected by the cross-check method, and the corresponding outlier percentage.

OpenSky data does not have a significant amount of outliers. One possible reason for the low number of outliers in the OpenSky data could be the accuracy of the sensors used to collect the

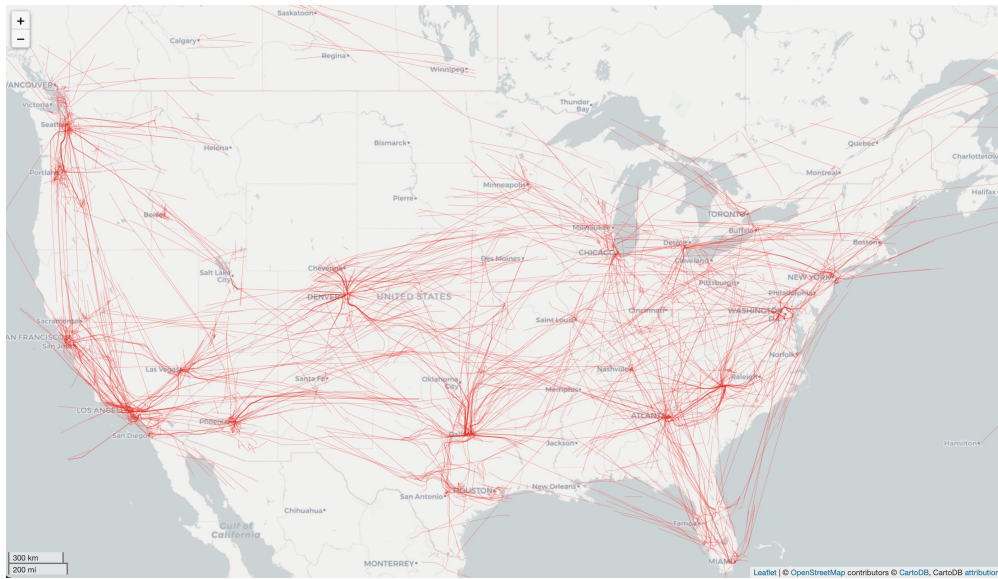


Figure 5.10: Visualization of the OpenSky dataset before applying the ground-truth filtering procedure

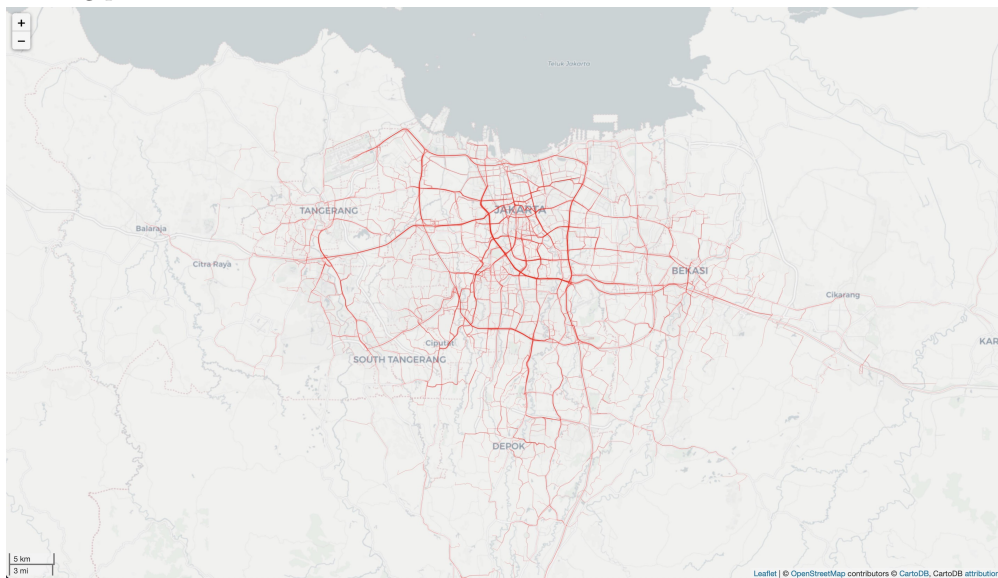


Figure 5.11: Visualization of the Jakarta Grab-Posisi GPS dataset before applying the ground-truth filtering procedure

data. If the sensors are highly accurate, there may be fewer errors or discrepancies in the data, resulting in more occasional outliers. Additionally, the data may have been adjusted or corrected in some way to account for any errors or noise, further reducing the number of outliers.

To evaluate the precision of various outlier detection libraries, we employ datasets such as AIS and OpenSky, which are characterized by minimal outliers. This ensures that the libraries are not overly aggressive in data point removal. In contrast, we utilize datasets from Jakarta and

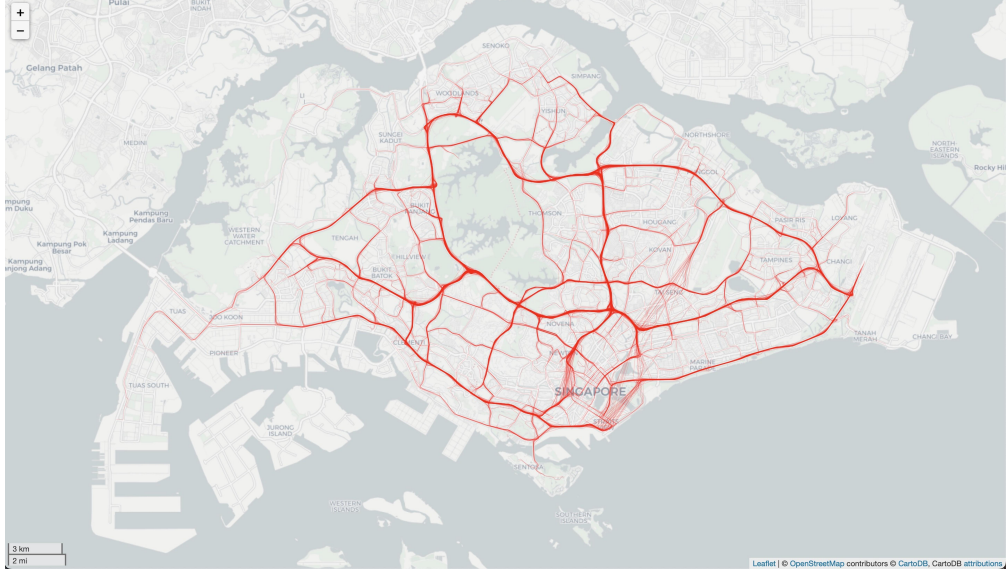


Figure 5.12: Visualization of the Singapore Grab-Posisi GPS dataset before applying the ground-truth filtering procedure

Singapore for assessing recall, which present a higher rate of outliers.

Parameter-Dependent Ground-truth Construction

The ground-truth is generated via an explicit rule-based procedure rather than manual annotation. Consequently, the labeled outlier set does not represent an inherent property of the data but is instead the output of a detector whose behavior is determined by its configuration. In the present pipeline, this configuration is governed by two thresholds: a speed-difference threshold tS and a bearing-difference threshold tH (see Alg. 1). A point is labeled as an outlier when the discrepancy between the speed or bearing computed from consecutive positions and the corresponding sensor-recorded value exceeds tS or tH . Therefore, the ground-truth should be interpreted as an operational reference produced by a configurable detector that enforces a consistent definition of anomaly for a fixed setting, but its definition is conditional on (tS, tH) and can vary substantially when these parameter values are modified.

Qualitative Effect of (tS, tH)

To assess the qualitative impact of the speed- and bearing-threshold pair (tS, tH) , we rely on the AIS and Jakarta datasets. These datasets provide complementary information on the labeling process by trajectory type. AIS trajectories follow a well-structured pattern in the maritime scenario, whereas Jakarta trajectories traverse a dense network of roads, with varying sampling rates and short local excursions. Our approach enable us to determine whether variations in the parameters affect the scale of the labeled outliers, the geographical extent, and the types of outliers identified.

The AIS data from the Helsingør to Helsingborg ferry corridor serves as a concrete example to illustrate the behavior of the ground-truth procedure under varying parameterizations. This setting is particularly informative because the traffic is dense yet strongly structured. Vessels

Dataset	Description	Source	Total Points	Outliers	%
Jakarta	Vehicle GPS trajectories	Grab-Posisi dataset [48]	16 798 804	85 275	0.5077%
Singapore	Vehicle GPS trajectories	Grab-Posisi dataset [48]	9 096 633	68 202	0.7498%
OpenSky	ADS-B, Mode S, TCAS, and FLARM	OpenSky Network ¹	5 417 090	2	0.0000369%
AIS	Ship tracks	Danish Maritime Authority ²	25 714 294	957	0.00372%

Table 5.3: Summary of the evaluated datasets, including source, total number of points, number of detected outliers, and outlier percentage.

repeatedly follow near-identical lane paths with consistent curvature near the terminal approaches.

Figure 5.13 illustrates this sensitivity in the AIS example. The original GPS measurements (shown in blue) form a dense lane structure. Under a stricter configuration ($tS = 11$, $tH = 50$), the labeled outliers (shown in pink) are relatively sparse and concentrated in a limited number of locations. This setup is expected to pick only the most noticeable deviations that are sustained enough to accumulate data according to the scoring rule. Conversely, a more permissive configuration ($tS = 5$, $tH = 20$) increases the number of labeled outliers (shown in green) and expands their spatial footprint along the corridor.

We consider the Jakarta dataset to serve as an urban mobility counterpart where the traces follow trajectories over a dense road map with high variability in sampling rate, stop-and-go patterns, and overlapping paths. This pattern constitutes an integral part of ground-truth generation since rule-based scoring cannot only depend on aggressive behavior but also on minor inconsistencies characteristic of urban motion patterns and corresponding measurement regimes (e.g., irregular sampling periods in the vicinity of intersections or sudden turning and speed oscillations). Thus, the Jakarta dataset offers a scenario where alterations of the pair (tS, tH) can influence both the magnitude of ground-truth-based outliers and their geographical distribution patterns on the map.

Figure 5.14 highlights how parameter choices reshape the labeled outlier set in a dense urban network. The original trajectories (shown in green) cover major routes as well as detailed local streets. Under the configuration $tS = 40$ and $tH = 90$, the labeled set (shown in brown) expands broadly across the network and becomes particularly dense in the urban core, where the combination of high traffic complexity and irregular sampling increases the number of local inconsistencies that can accumulate sufficient evidence. With $tS = 200$ and $tH = 180$ (shown in orange), the set of labels changes in both density and size because the temporal support and severity ratio differ, focusing not on general expansion across multiple routes but rather on deviations with high scores.

Implications for Evaluation and Interpretation

This parameter dependence has two direct implications for the interpretation of EKF agreement metrics. First, precision/recall values are conditional statements as they measure agreement with a particular instantiation of the ground-truth definition rather than agreement with an absolute notion of anomaly. Second, modifying (tS, tH) alters both the size and the spatial distribution

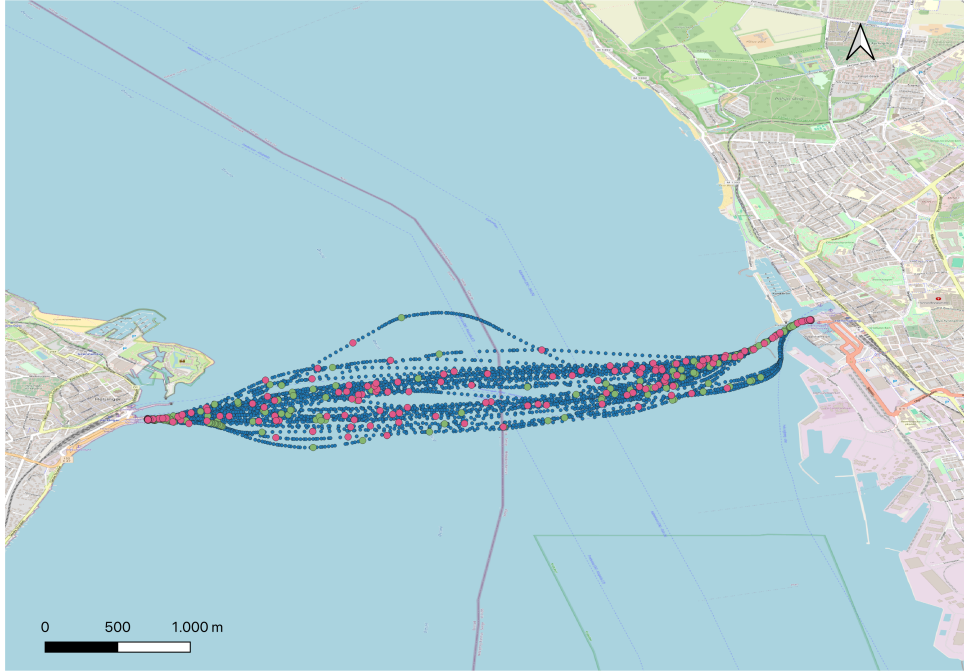


Figure 5.13: Sensitivity of AIS ground-truth to the speed and bearing thresholds (tS, tH)

of the labeled set. A stricter ground-truth tends to reduce the positive class and emphasize extreme anomalies, which may align more with innovation-based rejection. While a permissive ground-truth increases the positive class and often includes numerous semantic inconsistencies that are not identifiable from position innovation alone.

5.4.2 Experimental Study

The experimental study proposed evaluates the libraries and it consists of the compilation of various mobility datasets originating from multiple sensors, the compilation of different evaluation metrics, and the generation of ground-truth for each of these datasets. Furthermore, several libraries are assessed in this experiment. The metrics to be measured are recall, precision, F1, F2, and F0.5.

The experiments were run in ten libraries composed of MovingPandas, Scikit-mobility, Scikit-learn, PTRAIL, PyMove, MoveTk, MEOS, Argosfilter, StMove, and mvoutlier. These were described in Sect. 5.2. Figure 5.15 summarizes the runtime comparison across the four datasets. The data used in the experiment were described in Sect. 5.4.1. In addition, we include our MEOS EKF implementation in *drop mode*. MEOS EKF results should be interpreted primarily as point-wise agreement with the ground-truth labels rather than as a measure of smoothing quality. The source code is publicly available.²⁴

The European aviation industry [29]²⁵ has developed methods to reduce carbon emissions to meet climate targets. An aircraft can fly an optimal flight path and use various technologies

²⁴<https://github.com/marianaGarcez/OutlierDetectionLibraries>

²⁵<https://www.eurocontrol.int/publication/objective-skygreen-2022-2030>

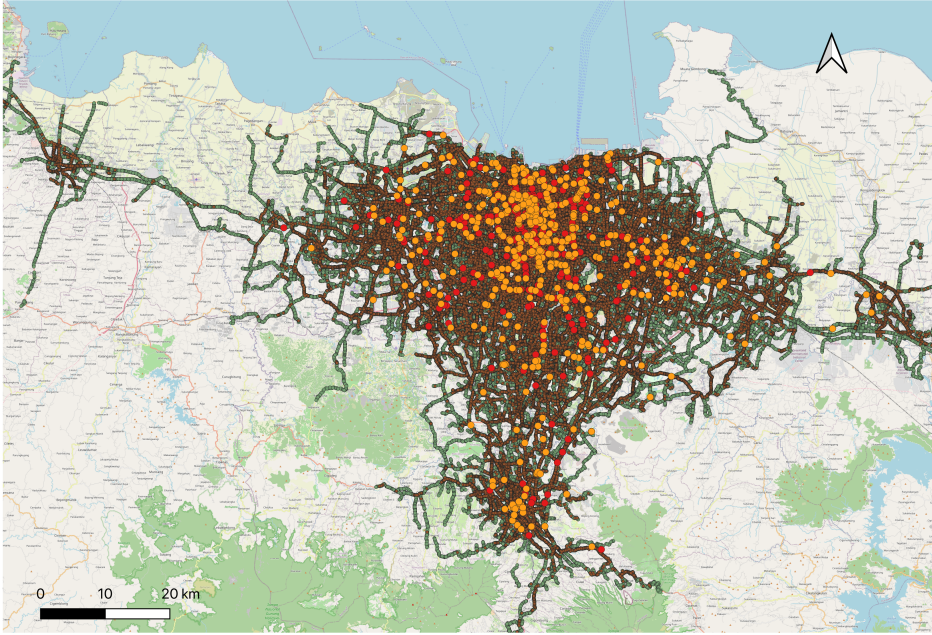


Figure 5.14: Sensitivity of Jakarta ground-truth to different speed and bearing threshold configurations (tS, tH)

and infrastructure to minimize fuel consumption and carbon emissions. This might include using modern flight planning software and meteorological data to plan for minimal fuel, using green energy at the airport to power the aircraft on the ground, and using electric taxi solutions to minimize ground-based emissions. The aircraft would also fly an optimal climb phase, follow a fuel-efficient cruising level, and use idle thrust descent to reduce fuel consumption during descent, i.e., the aircraft should change its altitude as rarely as possible. Due to this standard, we consider the OpenSky dataset to have only 2D rather than 3D data. However, it is essential to note that the assumption may not hold true in all circumstances. It may be necessary to use 3D data or incorporate altitude data in some cases to analyze flight patterns accurately [89].

In Figure 5.15, a comparative analysis of the runtimes for various libraries is presented across four distinct datasets: Jakarta, Singapore, OpenSky, and AIS. Notably, the PyMove library consistently exhibits the highest runtimes across all datasets, with an increase in computational time for larger datasets such as Jakarta and Singapore. In contrast, MEOS EKF exhibits the lowest runtime across all datasets, reflecting its constant-size per-point update (fixed-dimensional state) and C implementation.

Libraries developed in R programming language offer comparable runtimes. Similarly, libraries based on C++ (MoveTk) and C (MEOS) heuristic demonstrate runtimes similar to those of the R libraries. Furthermore, Scikit-learn exhibits competitive performance, likely attributed to its highly optimized Python implementation and algorithm. The MoveTk library displays an anomalous behavior in the execution time with the AIS dataset. This irregularity could be attributed to the underlying process employed for data reading. Given these observations, careful consideration of runtimes is advised when selecting a library for trajectory data analysis and

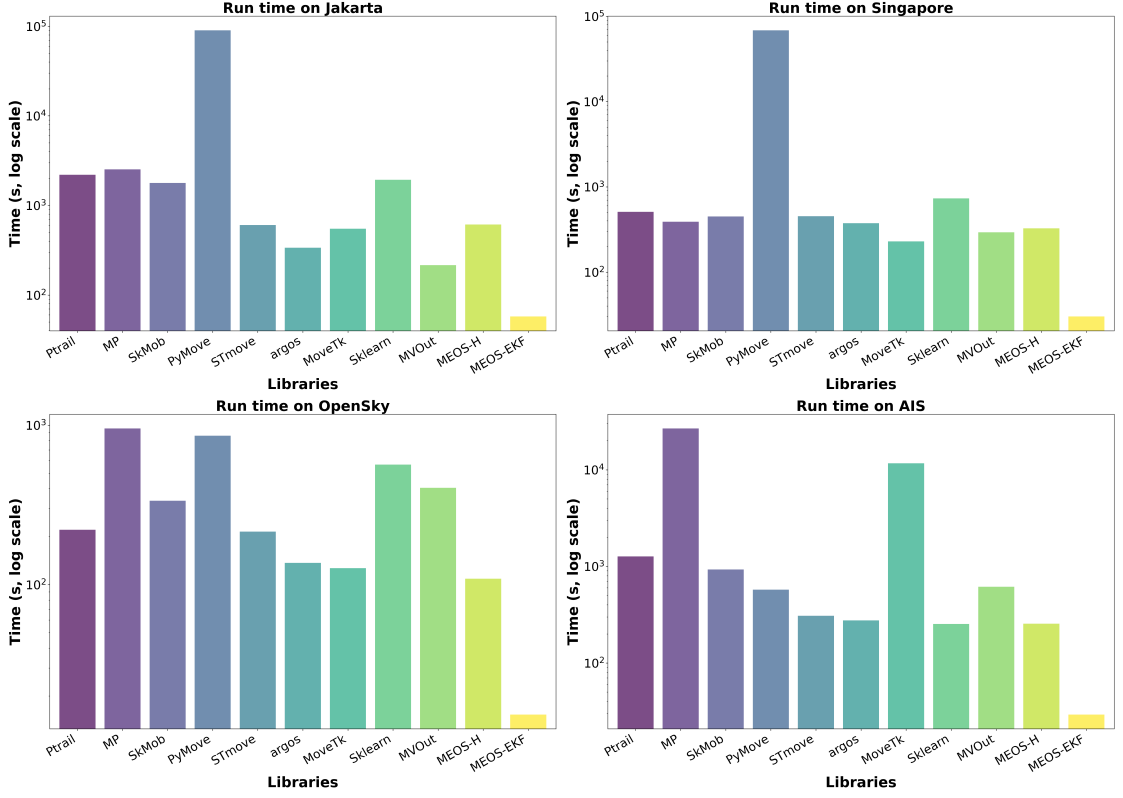


Figure 5.15: Runtime comparison of outlier-detection libraries across the Jakarta, Singapore, OpenSky, and AIS datasets

modification.

In Figures 5.16 and 5.17, we see the total runtimes across libraries developed in Python and R, respectively. The top left graph corresponds to Jakarta datasets, top right graph represents the Singapore dataset, the bottom left graph represents the OpenSky dataset runtime and the bottom right is AIS dataset runtime. What stands out is an association between the language of implementation and the efficiency of the library in question. To summarize, the choice of programming language appears to substantially impact the runtime, emphasizing the need to consider language compatibility and efficiency when evaluating libraries for specific tasks.

In order to analyze the correctness results, Figs. 5.18, 5.19, 5.20, 5.22, and 5.21 show precision, recall, and F1, F2, and F0.5 scores of each library, respectively. The scores are based on comparing the ground-truth data and library output. The accuracy, precision, recall, F0.5, F1, and F2 scores:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \quad (5.1)$$

$$\text{Precision} = TP / (TP + FP) \quad (5.2)$$

$$\text{Recall} = TP / (TP + FN) \quad (5.3)$$

$$F_{0.5} = \frac{1.25 \cdot \text{Precision} \cdot \text{Recall}}{0.25 \cdot \text{Precision} + \text{Recall}} \quad (5.4)$$

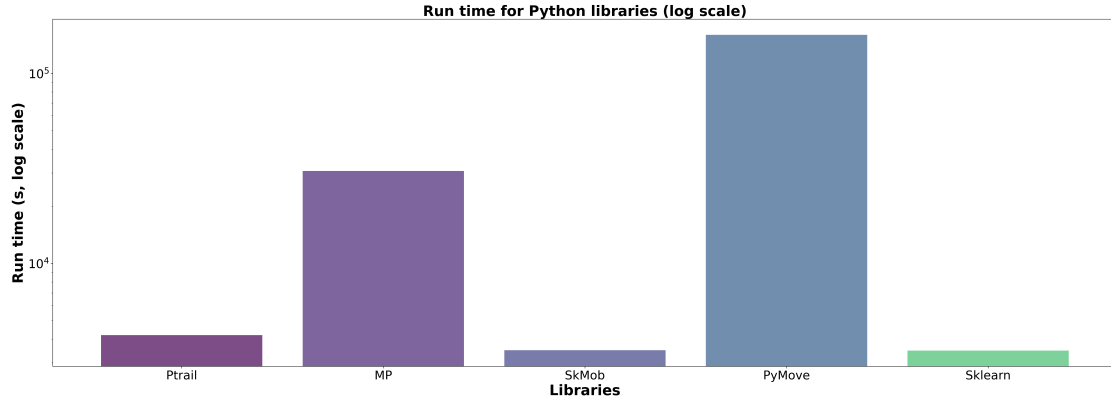


Figure 5.16: Aggregated runtime across all datasets for the evaluated Python libraries

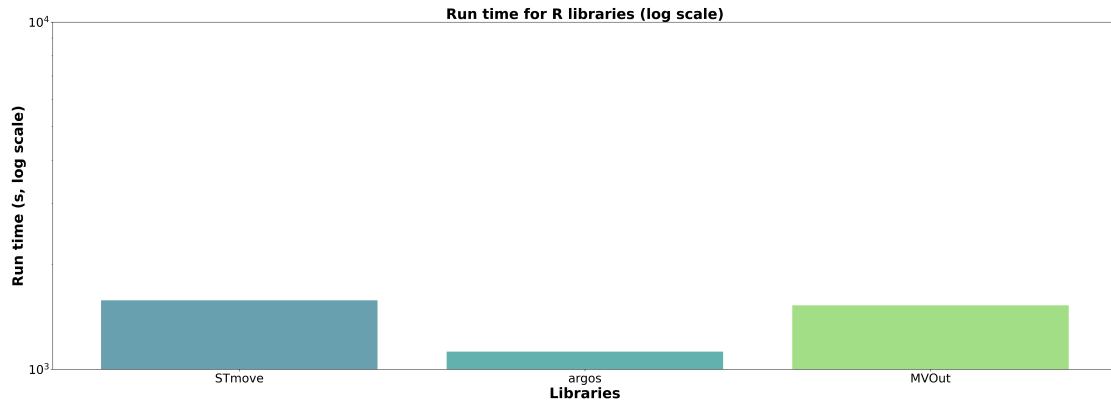


Figure 5.17: Aggregated runtime across all datasets for the evaluated R libraries

$$F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \quad (5.5)$$

$$F_2 = \frac{5 \cdot \text{Precision} \cdot \text{Recall}}{4 \cdot \text{Precision} + \text{Recall}} \quad (5.6)$$

These metrics summarize how well the outlier detection methods agree with the ground-truth on which individual measurements are outliers, under a strict point-wise evaluation. This type of evaluation is strict and requires that the outlier detection method and ground-truth align on identifying the same measurement instance, indexed by trajectory ID and timestamp, with no tolerance permitted. As a result, the evaluation answers a very specific question: agreement at the level of individual points, not agreement at the level of overall trajectory.

The ground-truth method labels provide a set of outliers. *TP* (true positive) counts measurements that the outlier detection method correctly identifies as outliers, i.e., predicted outliers and true outliers, while *FP* (false positives) counts normal measurements that are incorrectly dropped as outliers. Similarly, *FN* (false negatives) counts outliers that the outlier detection method fails to remove (missed outliers), and *TN* (true negatives) counts normal measurements that are correctly kept. Since we work at the level of individual measurements, a correct or incorrect decision concerns a specific point indexed by trajectory ID and timestamp, not the

entire trajectory being classified as anomalous.

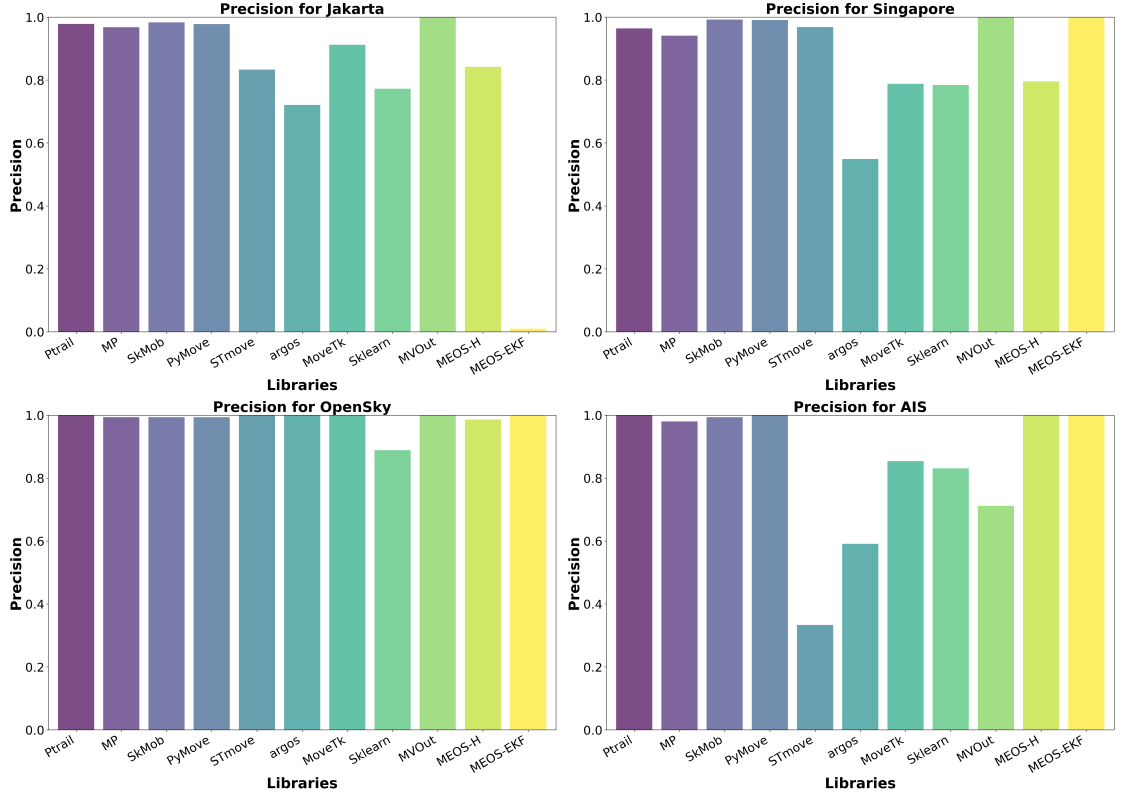


Figure 5.18: Precision of each library measured against the ground-truth across the four datasets

In Fig. 5.18, the precision is shown. The top left graph corresponds to Jakarta datasets, the top right graph represents the Singapore dataset, the bottom left graph represents the OpenSky dataset and the bottom right is the AIS dataset. The precision rate relates to the proportion of true positive predictions of outliers. This rate is homogeneous throughout libraries such as PTRAIL, MovingPandas, and Scikit-mobility. Scikit-mobility outperforms all. StMove presents the highest rate variation, especially for the AIS dataset.

In Fig. 5.19, the metric of interest is the recall, which measures the ability of a library to identify true positives correctly, in this context, outliers within the dataset. The top left graph corresponds to Jakarta datasets, the top right graph represents the Singapore dataset, the bottom left graph represents the OpenSky dataset and the bottom right is the AIS dataset. A high recall score indicates that the library detects a large percentage of the true outliers, reducing the likelihood of missing significant outliers. Conversely, a low recall score suggests the library may be susceptible to false negatives, resulting in omitting non-outlier data points. Of particular note is the performance of MoveTk, which displays commendable consistency across various datasets and emerges with the highest recall rate. Argosfilter and Scikit-learn closely follow this. On the other hand, mvoutlier and PTRAIL exhibit significant variability in their recall rates, particularly underperforming in larger datasets. It suggests that both libraries offer less reliable outlier detection capabilities, a factor worth considering when selecting a library for specific data analysis requirements.

The F1 score is a metric used for comparing the overall performance of the different libraries

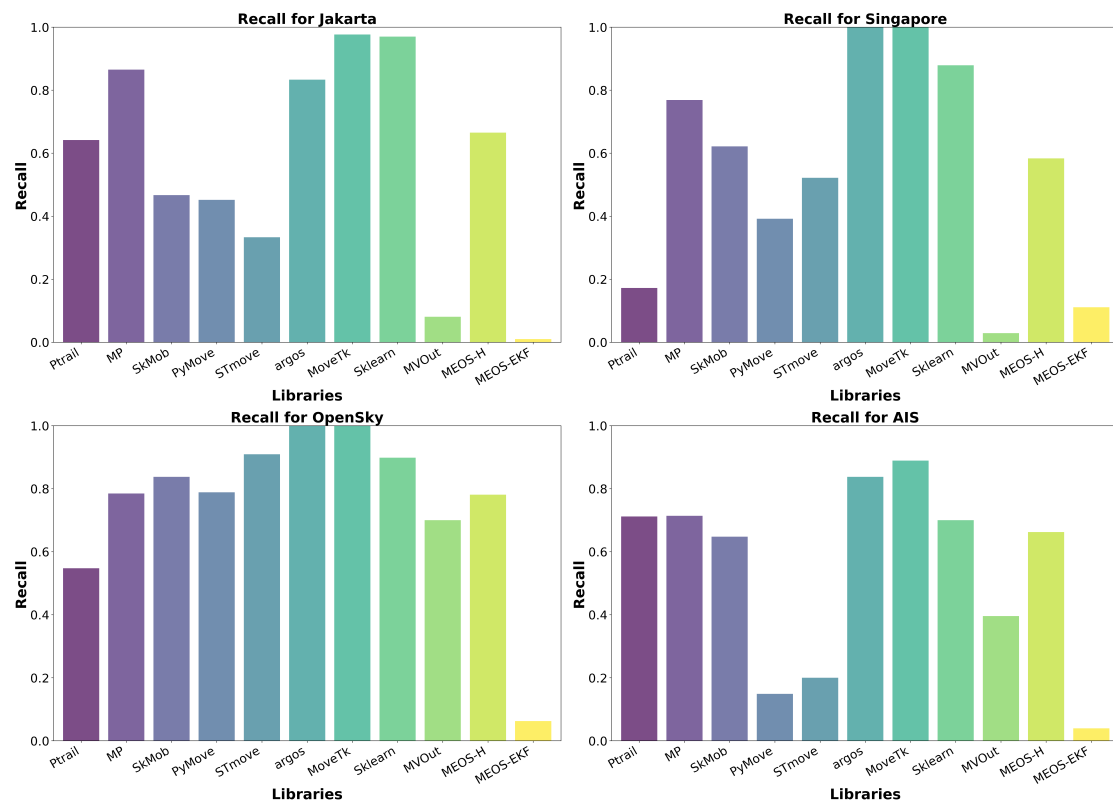


Figure 5.19: Recall of each library measured against the ground-truth across the four datasets

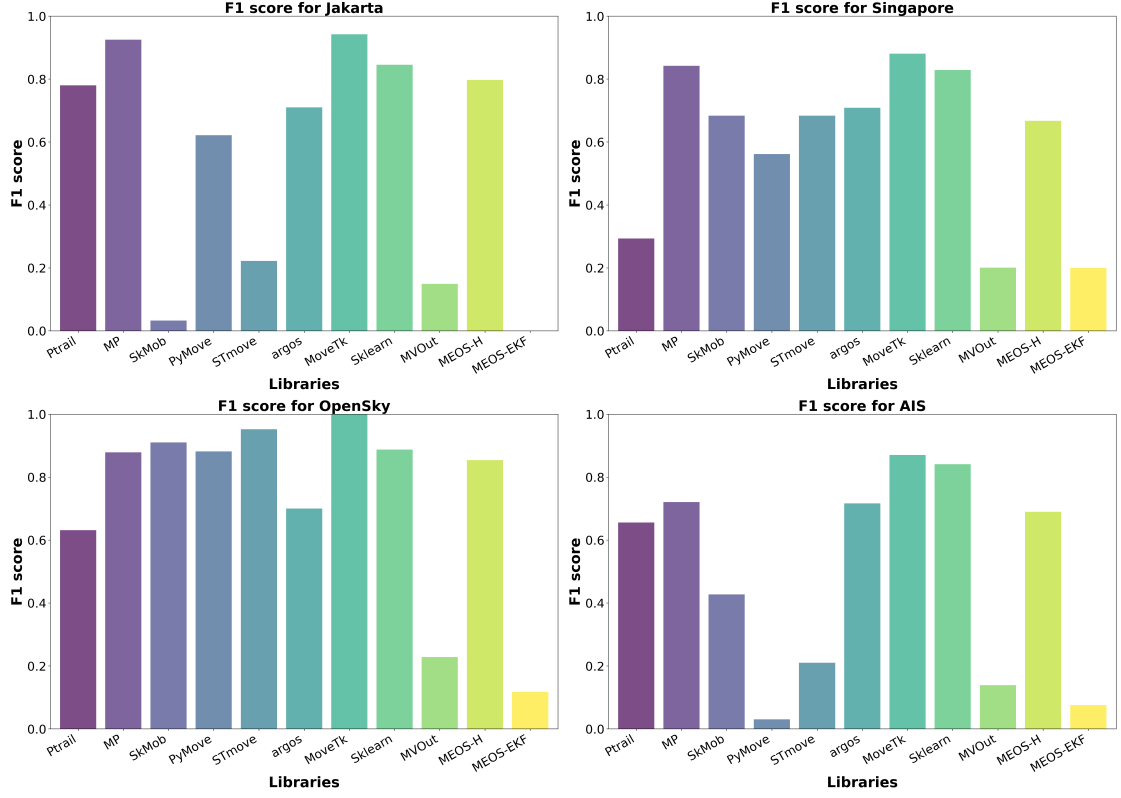


Figure 5.20: F1 score of each library measured against the ground-truth across the four datasets

(see Fig. 5.20). The top left graph corresponds to Jakarta datasets, the top right graph represents the Singapore dataset, the bottom left graph represents OpenSky dataset and the bottom right is the AIS dataset. The MoveTk library has the biggest rate for the F1 score in all datasets. Followed by the Argos filter library. In the Jakarta dataset, MovingPandas has the second-highest F1 score. Followed by Argos filter and PyMove. In contrast, StMove and Scikit Mobility have a low rate for the F1 score. MovingPandas has the third highest rate for the Singapore dataset, followed by StMove. PTRAIL has the lowest rate in this dataset. All libraries have a similar performance in the OpenSky dataset, except for PTRAIL, which has the lowest score in the first file. In the AIS dataset, MoveTk, PTRAIL, MovingPandas, and Argos filter have the highest score. In contrast, PyMove and StMove have the lowest rate.

To emphasize precision in our analysis, we have integrated the F0.5 score (see Fig. 5.21), which holds greater weight in precision over recall. The left graph corresponds to Jakarta datasets, the right graph represents the Singapore dataset. Conversely, the F2 score has been included to prioritize recall, shown in Fig. 5.22. The left graph represents the OpenSky dataset and the right is the AIS dataset. The analysis of F0.5 and F2 scores across various libraries and datasets has yielded valuable insights into the performance of each library with a focus on precision and recall, respectively. When examining both F0.5 and F2 scores, we observe significant variability in library performance across different datasets, emphasizing the data-dependent nature of their effectiveness. Regarding F0.5 scores, MovingPandas and Scikit Mobility are the top-performing libraries, excelling in precision, while MoveTk also demonstrated strong performance. However, when shifting the focus to F2 scores and prioritizing recall, Argos Filter and MoveTk are the top

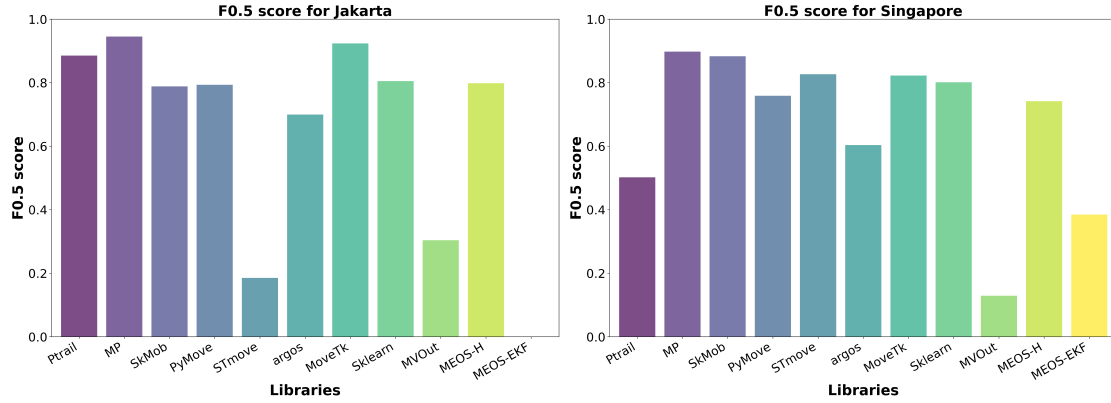


Figure 5.21: F0.5 score of each library measured against the ground-truth (precision-weighted)

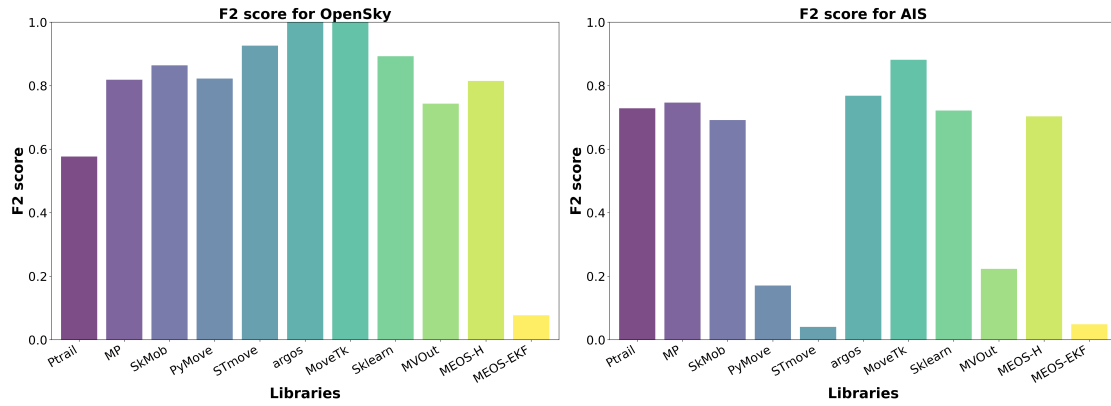


Figure 5.22: F2 score of each library measured against the ground-truth (recall-weighted)

performers, indicating their effectiveness in capturing true outliers. These findings suggest that the library choice should be context-specific, considering the specific requirements of the task.

Across datasets, MEOS EKF exhibits a highly conservative approach to measurement rejection. This approach results in very high precision on datasets with infrequent outliers, such as OpenSky and AIS, but leads to consistently low recall and, consequently, low F_1 and F_2 scores. For the Jakarta dataset, both precision and recall are nearly zero, reflecting minimal point-wise agreement with the ground-truth labels. This outcome is anticipated, as MEOS EKF rejects points through innovation-gating under a local motion model, while the ground-truth labels are based on a broader, multi-signal consistency criterion. The implications of this mismatch are examined in detail in Sect. 5.4.3.

These results suggest that while MovingPandas shows strong performance in precision, MoveTk excels in recall. MovingPandas stands out with its high-precision (0.970) and strong performance in F0.5 (0.920) and F1 (0.842) scores, indicating its effectiveness in balancing recall and precision with a slight bias towards precision. MoveTk shows better recall (0.966) and F1 score (0.923), suggesting its ability to identify true outliers. MEOS heuristic is a good choice, exhibiting balanced performance with F0.5 (0.842), F1 (0.752), F2 (0.707), Precision (0.906), and Recall (0.673). Libraries such as Scikit-learn and Argosfilter also demonstrate balanced capabilities, with Scikit-learn performing well in F1 (0.851) and recall (0.862), and Argosfilter in F2 (0.948).

and recall (0.918). On the other hand, mvoutlier has lower scores in F1 (0.179) and recall (0.301), implying limitations in its predictive accuracy. Despite their high precision, PTRAIL and PyMove show moderate recall performance, indicating a potential trade-off between precision and recall in these libraries. In contrast, MEOS EKF functions as a conservative online filter, delivering competitive runtime performance and high precision in scenarios with minimal outliers. However, it exhibits limited recall when evaluated using a strict point-wise agreement protocol with rule-based ground-truth (Sect. 5.4.3). This outcome suggests a discrepancy between the EKF’s model-based rejection criterion and the semantics of the ground-truth labels, thereby motivating an evaluation approach that also considers trajectory-level quality and motion-model consistency.

5.4.3 Interpreting EKF Results

In addition to the cross-library benchmark, we validated the MEOS EKF rejection rate and behavior against the ground-truth outlier labels. The purpose of this verification is to determine whether there is a point-wise correspondence between measurements removed by the EKF and those identified as outliers by the ground-truth algorithm.

Observed Behavior

Over a wide range of gates and noise values (process noise Q and covariance matrix R), the overlap of labeled outliers and the predicted outliers remains small. In datasets where ground-truth outliers are extremely sparse, precision can nevertheless appear high because the EKF rejects only a small number of measurements and incurs few false positives. However, recall remains low because many ground-truth outliers do not trigger large single-step innovations, so the overall F scores remain low despite the high precision. Consequently, recall is low and the F -scores ($F_{0.5}$, F_1 , F_2) remain close to zero: many measurements rejected by the EKF are not labeled as outliers by the ground-truth procedure (high FP), while many ground-truth outliers are not rejected by the EKF (high FN). At the same time, accuracy is high because the vast majority of measurements are non-outliers, are retained by the EKF, and are also labeled as non-outliers by the ground-truth (high TN). Thus, accuracy is dominated by class imbalance and is not informative about the quality of outlier detection. Taken together, the low F Scores and high accuracy are consistent with the EKF’s role as a probabilistic state estimator optimized for smoothing under a linear-Gaussian motion prior, rather than as a dedicated outlier detector.

AIS Ferry Example

Figure 5.23 illustrates the practical significance of low pointwise agreement between two outlier detection method MEOS EKF and MovingPandas KF. The original AIS points represent frequent trips of a ferry. In this setting, the removed point patterns differ broadly. Figure 5.23(a) shows MovingPandas removal patterns, which tend to be sparse and be centered around a few points in space, rather than being scattered throughout the corridor. The exclusion patterns in the MEOS EKF, in Figure 5.23(b), are also sparse but exclusions are found in areas around which individual measurements are difficult to fit with a smooth motion pattern.

When comparing Figure 5.23(a) and (b), it is shown that the two methods differ in aggressiveness and tend to select different points. MovingPandas removals are consistent with a rule-driven notion of anomaly, using plausibility checks based on reported speed, bearing, or local consistency to flag points that violate movement constraints. MEOS EKF outlier detection follows the statistical gate implied by the innovation test: points are rejected when the normalized residual

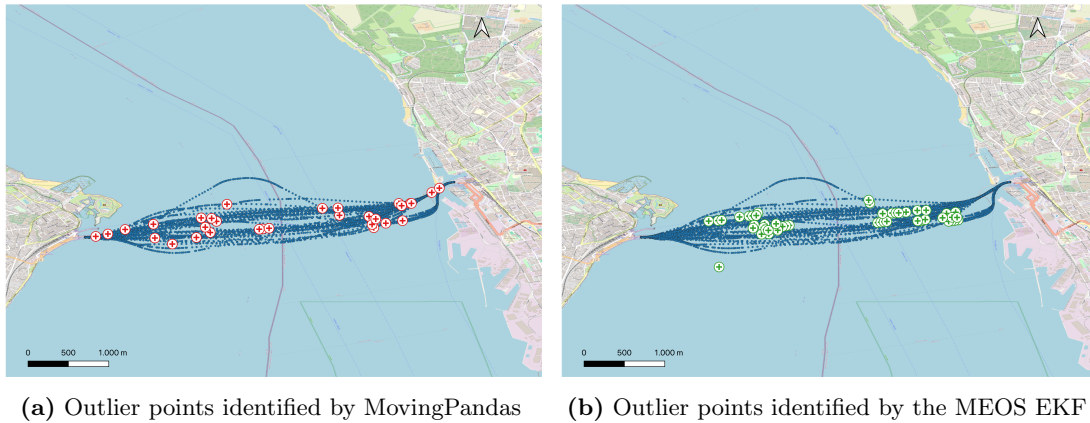


Figure 5.23: Comparison of removed point patterns on AIS trajectories for MovingPandas and the MEOS EKF

exceeds the filter’s predicted variance, targeting isolated spikes, fragmented reports, and rare deviations that cannot be absorbed by state adaptation.

In addition, an important detail is that MovingPandas’ examples recommend an initial `OutlierCleaner` pass, i.e., a local, rolling-window correction step using the Mean Absolute Deviation (MenAD), before running the KF smoother. This matters for the interpretation of the experimental study, since the pre-cleaning step removes a significant amount of points, which tends to improve precision-oriented metrics and is consistent with the strong results reported for MovingPandas in Sect. 5.4.2.

Jakarta Example

Figure 5.24 shows Jakarta city with the original trajectories (seen in green). The ground-truth method flags an extensive set of outliers (seen in red). This pattern aligns with a ground-truth procedure that encodes broad, rule-based anomalies, such as plausibility of speed/acceleration or multi-signal inconsistencies, which can frequently occur in urban mobility data. In comparison, MEOS EKF outliers (seen in yellow) are relatively sparse. The EKF rejects measurements that are statistically inconsistent with a local motion model and an assumed noise threshold. The method can also tolerate deviations through state adaptation. Therefore, even aggressive gate/ Q/R settings cannot force the EKF to approximate an outlier to the ground-truth when the innovation magnitude exceeds the innovation threshold.

STIB Example

In addition, we use data collected by STIB, the main urban mobility operator in the Brussels region. The gathered data contains irregularities, making the data cleaning process difficult. We extracted 1250 trajectories containing start and end timestamps and 66795 position. The coordinate system used is WGS84. The temporal window is about 1 hour of operation. As it is a public transportation dataset, it shows high variability in duration and distance, with a median of 15,8 min and 3,24 km, respectively. Figure 5.25 shows the total data collected.

We compare the spatial distribution of removed points across all evaluated methods on the STIB dataset to highlight how different outlier definitions (rule-based, heuristic, and model-based) lead to distinct removal patterns. This qualitative comparison complements the point-wise metrics

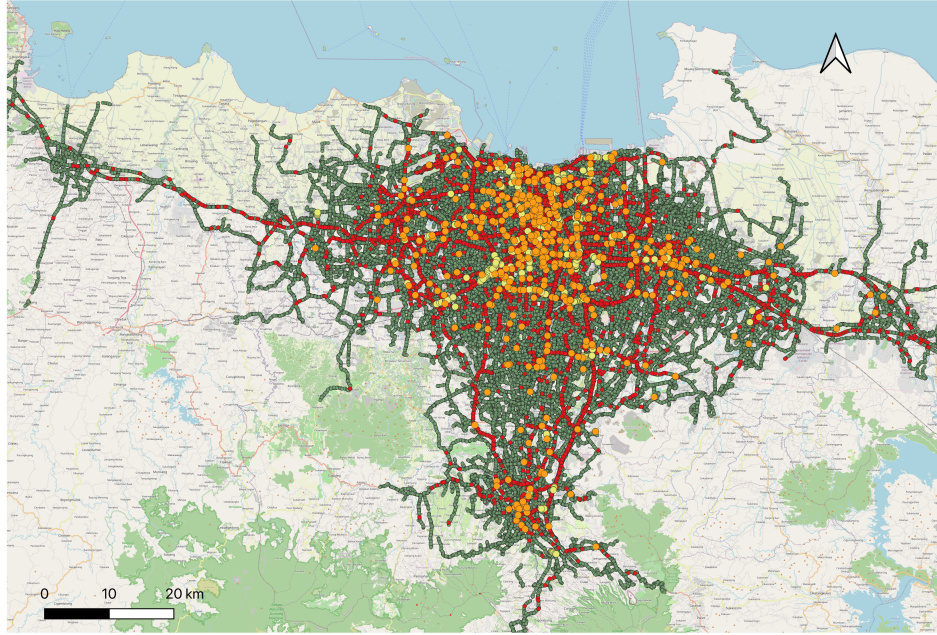


Figure 5.24: Jakarta example comparing points rejected by the MEOS EKF against the ground-truth outlier labels

reported earlier by exposing whether a method removes isolated spikes, short noisy segments, or broader trajectory regions. In particular, we focus on MEOS EKF and MovingPandas KF.

In Fig. 5.26, MovingPandas `KalmanSmootherCV` removals are shown in red, MovingPandas `OutlierCleaner` removals are shown in blue, and MEOS EKF removals are shown in pink. The figure provides a consolidated view by overlaying the STIB trajectories with the points removed by each evaluated method, highlighting that the methods remove different spatial patterns even when removal counts are similar.

We apply MovingPandas cleaning algorithms and quantify the data into two categories: `OutlierCleaner` (pre-cleaning) and `KalmanSmootherCV`. We then quantify how each category is present within the STIB dataset: With 66,795 recorded positions, `OutlierCleaner` removes 885 points (1,32%), while `KalmanSmootherCV` removes 184 points (0,27%). We measure 45 points of overlap with either method, for a total of 1024 unique removals (1.53%). Among the unique removals, 82.1% are attributed to the pre-filter rules used in the `OutlierCleaner` method, whereas 13.5% were removed by the `KalmanSmootherCV` method. As we can see in Figure 5.27, both methods implement very different definitions of what an outlier point means, with `OutlierCleaner` being a rule-driven pre-cleaning method. In contrast, `KalmanSmootherCV` primarily operates as a state estimator, removing points based on prior speed after the pre-cleaning stage. `KalmanSmootherCV` (seen in red) and rule-driven pre-cleaning `OutlierCleaner` (seen in blue). Figure 5.28 summarizes the number of points removed by each MovingPandas stage, contrasting `KalmanSmootherCV` (red) with the rule-driven pre-cleaning `OutlierCleaner` (blue). `OutlierCleaner` removes many more points (885) than `KalmanSmootherCV` (184). We can therefore conclude that MovingPandas outlier detection reported in Sect. 5.4.2 occurs mainly



Figure 5.25: Visualization of the raw STIB trajectories used in the qualitative removal-pattern comparison

during the pre-cleaning stage.

In Fig. 5.29, we observe that, despite MEOS EKF and MovingPandas KalmanSmootherCV removing the same number of points (184 points), the two methods disagree on which points are outlier, reflecting different operational definitions of outliers as there are discrepancies in rule-driven pre-cleaning versus innovation-gated state estimation. The figure quantifies the points of each processing stage within the STIB dataset.

Method Validation Implications

These results separate two questions that are often conflated in trajectory cleaning. First, the EKF can be validated as a correct gated state estimator by internal consistency with innovation statistics, gate triggers, and by the observation that rejected points frequently correspond to obvious spikes or glitches. Second, reproducing a semantic ground-truth outlier set is a different goal: it requires a detector whose decision boundary is aligned with the anomaly definition used by the evaluation protocol. Our experiments indicate that tuning $\text{gate}/Q/R$ alone cannot force a KF smoother to approximate the semantic label set.

5.5 Trajectory Functions

In addition to studying data-quality methods, we examine how the MEOS spatiotemporal model can be used for offline trajectory analytics. This section illustrates MEOS and PyMEOS using a large real-world dataset from Singapore, demonstrating how the same temporal and

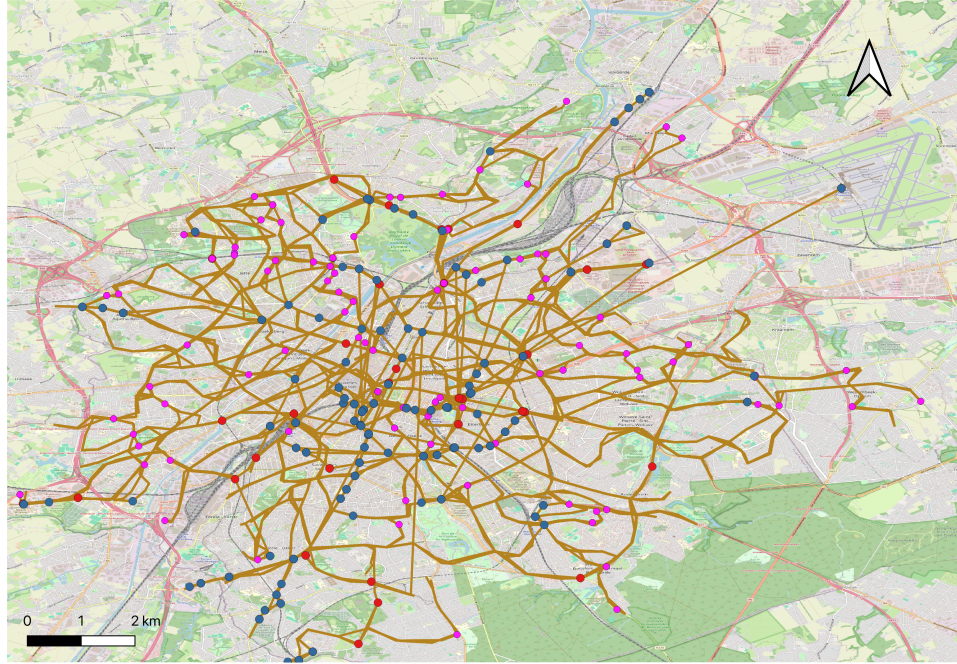


Figure 5.26: Overlay of STIB trajectories with points removed by MovingPandas (OutlierCleaner and KalmanSmootherCV) and the MEOS EKF

spatiotemporal types employed in MobilityNebula support expressive analytical operations, such as trip construction, segmentation, metric extraction, and proximity detection. We highlight the flexibility of the underlying model and demonstrate how MEOS enables mobility studies that complement the real-time capabilities evaluated earlier in the chapter.

The analysis is based on the Grab-Posisi dataset [48], which contains GPS trajectories collected from vehicles moving in the city of Singapore. The dataset spans more than one million kilometers of travel over a one-month period and comprises over 88 million points stored in CSV files.²⁶ Using this dataset, we (1) ingest GPS records and construct trajectories, (2) create spatiotemporal bounding boxes around two frequently visited locations (“Singapore Airport” and “Gardens by the Bay”) and filter trajectories within a chosen time interval, (3) segment the filtered trajectories into single-leg journeys between these locations, (4) compute derived metrics such as total distance, speed profiles, and average bearing, and (5) identify instances where vehicles come within a fixed proximity threshold of one another. Figures 5.30-5.38 illustrate the resulting trajectories, the spatial filters, and examples of detected close encounters.

5.5.1 Implementation in MEOS

We construct trajectories by reading GPS data files containing coordinates and object identifiers (`carId`). We collect these coordinates and link points based on their `carId`. Using these joint coordinates, we create a trajectory. We utilize a `tsequence`, a data structure that encapsulates a sequence of temporal values to structure these trajectories. As the data from the file is processed,

²⁶<https://engineering.grab.com/sg/blog/grabposisi/>

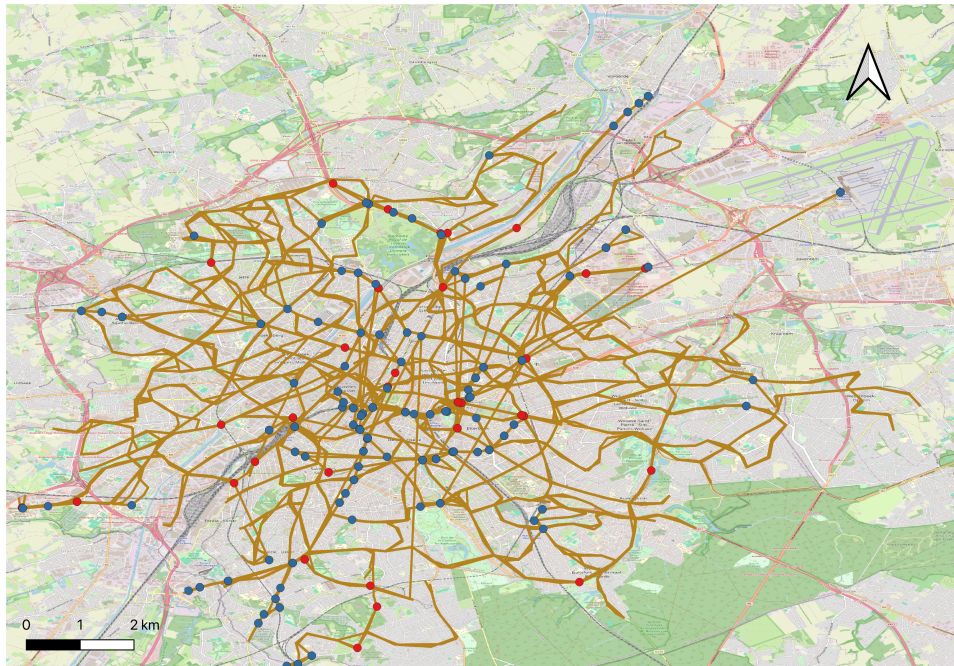


Figure 5.27: Spatial distribution of points removed by the two MovingPandas stages (Outlier-Cleaner and KalmanSmootherCV) on STIB

points are systematically appended to their respective `tsequence`.

We filter the GPS trajectories that intersect the areas: ‘Singapore Airport’ and ‘Gardens by the Bay.’ We create an `stbox`, a spatiotemporal bounding box by joining these two areas’ maximum and minimum coordinates and setting a period of time: ‘08-04-2019’ to ‘22-04-2019’, namely **Area of Interest** (see Fig. 5.31).

We then segment these collected trajectories into single-leg journeys, such as a direct trip from ‘Singapore Airport’ to ‘Gardens by the Bay.’ First, we identify trips intersecting both areas (see Code 17). Considering only the resulting trips, we segment them based on the timestamp indicating when the car was first detected at the origin and last detected at the destination with `tpoint_at_geom`.

Code 16. Extracting trips that intersect Singapore Airport or Gardens by the Bay

```
for (int i = 0; i < no_cars; i++){
  for (int j = 0; j < no_places; j++){
    Temporal *atgeom = tpoint_at_geom(
      (const Temporal *) cars[i].trip, places[j].geom);
    if (atgeom){
      carsTrips[no_trips].carID = cars[i].carID;
      carsTrips[no_trips++].trip = cars[i].trip;
    }
  }
}
```

We extract trips that intersect either Singapore Airport or Gardens by the Bay using `tpoint_at`

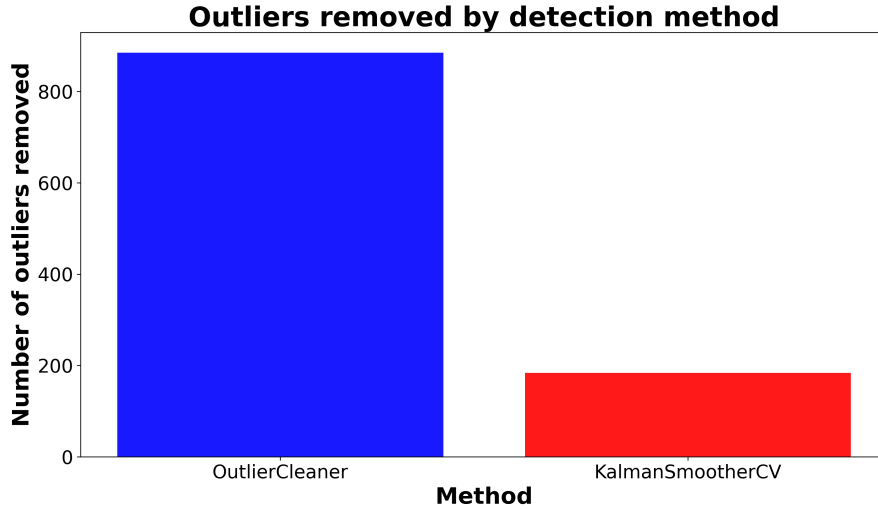


Figure 5.28: Counts of removed points per MovingPandas stage on the STIB dataset

`_geom` (see Code 16).

We apply analytical functions to determine key metrics of each car’s journey. To store the results of these metrics, we utilize a `tfloat` data type. A `tfloat` is a data structure that represents a float value over time. The total distance traveled is achieved by accumulating the distance between each GPS point in the trip. Additionally, we compute the average speed for each trip. The average bearing of travel is also determined based on the angular data associated with the vehicle’s path. Furthermore, we generated a list of instances where cars were within 10 m from each other by comparing the position of each car and timestamp. In Fig. 5.38, we show an example with two cars.

Code 17. Filtering trajectories that intersect both areas of interest

```
for (size_t i = 0; i < numberCarsinAreaofInterest; i++){
    if (eintersects_tgeo_geo(
        cars[i].trip, places[0].geom)
    && (eintersects_tgeo_geo(
        cars[i].trip, places[1].geom))){
        cars[no_cars].carID = cars[i].carID;
        cars[number_cars++].trip = cars[i].trip; }}

```

5.5.2 Implementation in PyMEOS

The procedures outlined above can be equivalently implemented using PyMEOS.²⁷ In addition, PyMEOS provides visualization functionalities using matplotlib.

First, we visualize the trajectories using a heatmap, which allows us to identify the more transited parts. To do so, we first compute the bounding box containing all the trajectories and tile it into boxes of 500 m × 500 m. Then, we compute, for each tile, how many trajectories intersect it and plot it using Plotly (see Fig. 5.32).²⁸ The heatmap (see Fig. 5.37) reveals the

²⁷<https://github.com/marianaGarcez/OutlierDetectionLibraries/blob/main/PyMEOS/PyMEOS-ACM.ipynb>

²⁸<https://plotly.com/python/>

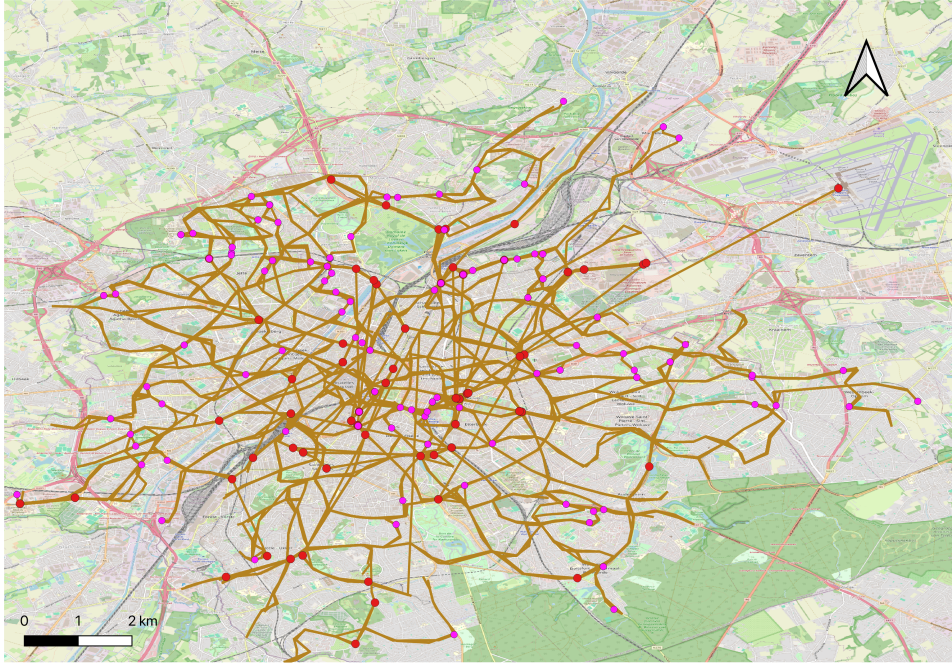


Figure 5.29: Comparison of removed point patterns on STIB for MovingPandas KalmanSmootherCV and the MEOS EKF

main corridors connecting the regions to central Singapore and highlights several secondary routes serving intermediate neighborhoods.

Initially, we employ a general bounding box encompassing both the ‘Singapore Airport’ and ‘Gardens by the Bay’ to exclude any cars that do not operate within these areas. Afterward, we introduce bounding boxes for each location to identify the cars whose trajectories intersect these areas. We can see the result of this process in Fig. 5.33. Figure 5.33a depicts trips that do not cross into the area of interest. On the other hand, Fig. 5.33b shows trips that traverse the area of interest. In Fig. 5.33c, the focus is on trips that intersect only one of the two designated areas, while Fig. 5.33d presents trips that pass through both specified areas. Figures 5.33e and Fig. 5.33f show a zoomed-in version of the previous two Figures, showing in detail the trajectories that cross the areas.

We split the trajectories into single-leg trajectories using two strategies using the areas of interest geometries (see Sec. 5.5.2) and using the stops method (see Sec. 5.5.2).

Geometry-Based Segmentation

For the geometry method, for every car, it first computes whether the car is in ‘Singapore Airport’ or in ‘Gardens by the Bay’ at every moment, storing this information as `tbool`, a temporal boolean (Lines 1–5). Then, we compute the periods in which the car is outside, removing the first and last segments (Lines 6–9) and uses them to clip the original trajectories into parts that go from one area to the other (Lines 10–11). In addition, we compute the metrics for each single-leg trajectory, including speed and bearing (see Figs. 5.35 and 5.34).

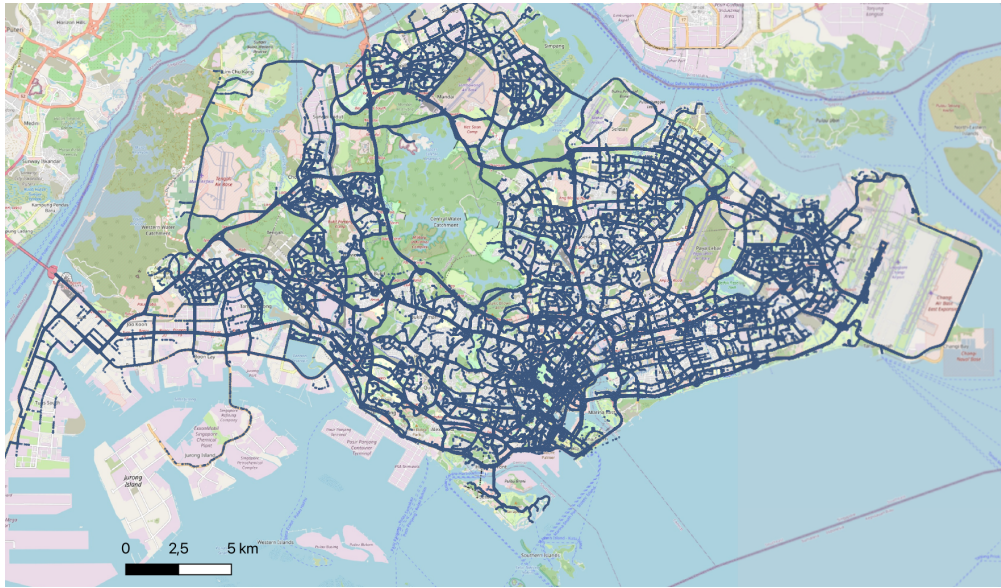


Figure 5.30: Example trajectories from the Singapore Grab-Posisi dataset used for offline analytics

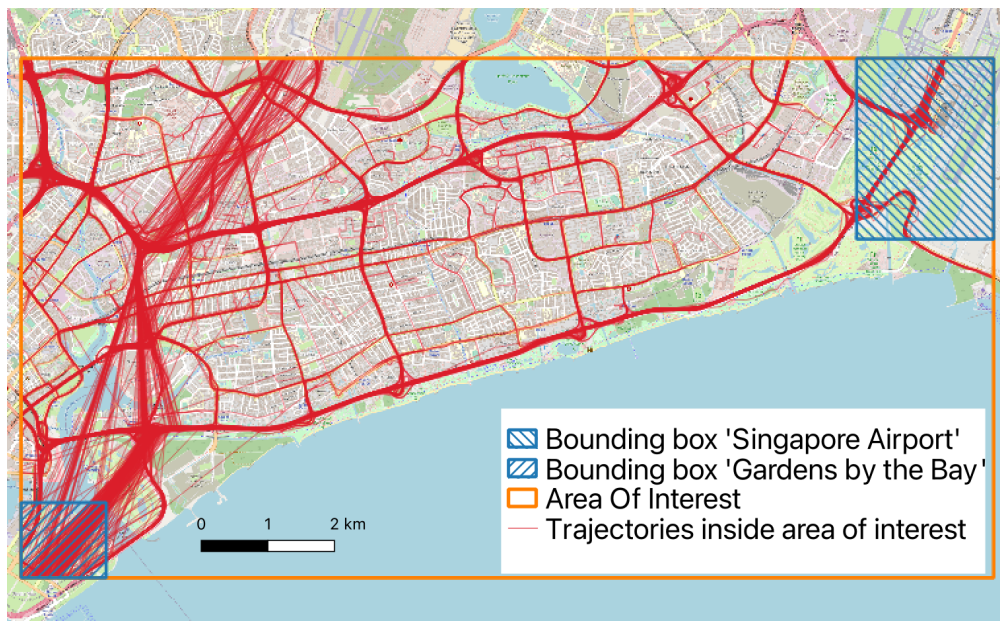


Figure 5.31: Trajectories overlaid with spatiotemporal bounding boxes for Singapore Airport and Gardens by the Bay

Code 18. Trajectory split using bounding boxes

```
st['Is Inside'] = st['Trajectory'].map(lambda t:
    trip.is_spatially_contained_in(bay)
    .temporal_or(trip.is_spatially_contained_in(airport)))
```

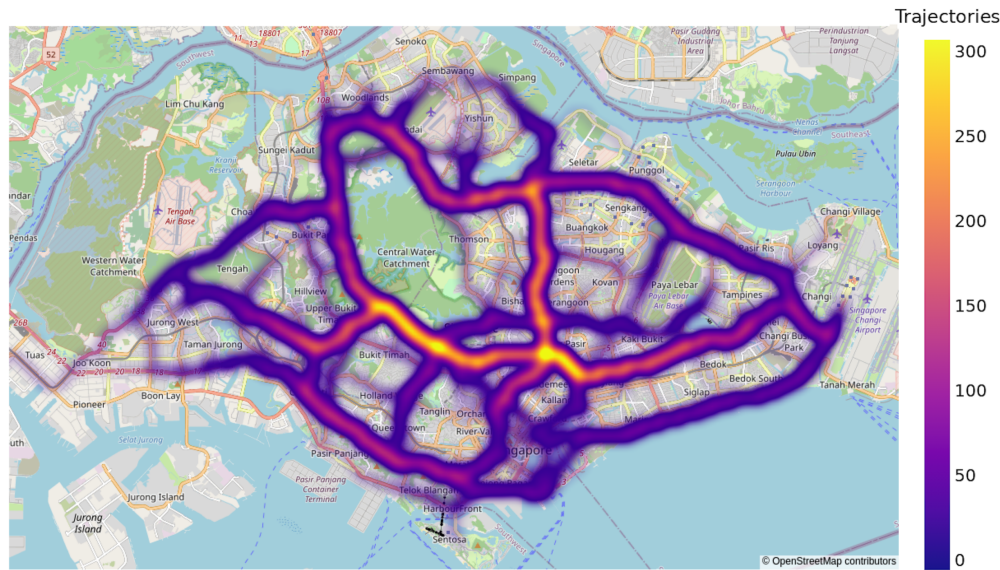



Figure 5.32: Spatial density heatmap of trips between Singapore Airport and Gardens by the Bay

```
st['Time Inside'] = st['Is Inside'].map(lambda t:
    t.when_true())
st['Period'] = st['Time Inside'].map(lambda t:
    t.to_period())
st['Time Outside'] = st.apply(lambda row:
    row['Is Inside'].at(t['Period']).when_false(), axis=1)
st['Single Trips'] = st.apply(lambda row:
    row['Trajectory'].at(['Time Outside']), axis=1)
```

Stop-Based Segmentation

Figure 5.36 shows the speed and bearing signals restricted to the trip segments produced by the stop-based method. The stop-based method relies on behavioral information instead of explicit spatial boundaries. A *stop* is defined as a period of remaining within 5 m for at least 5 minutes. Utilizing PyMEOS, we apply the `stops` operator to each trajectory using these parameters, which allows us to identify the time intervals during which the vehicle is stationary or moves only within a limited radius.

For trajectories with no detected stops, the method returns a single trip equal to the full movement interval. For trajectories with one or more stops, the result is a set of movement segments separated by the stop periods. As in the geometry-based case, we propagate the corresponding speed and bearing sequences to each segment by restricting them to the time span of the trip.

Segmentation Strategies Comparison

Table 5.4 presents a summary of the results for both segmentation methods applied to the bay-airport trajectories. The geometry-based method identified a total of 1,518 single-leg trips, resulting in an average of 1.05 segments per original trajectory. While most trajectories corre-

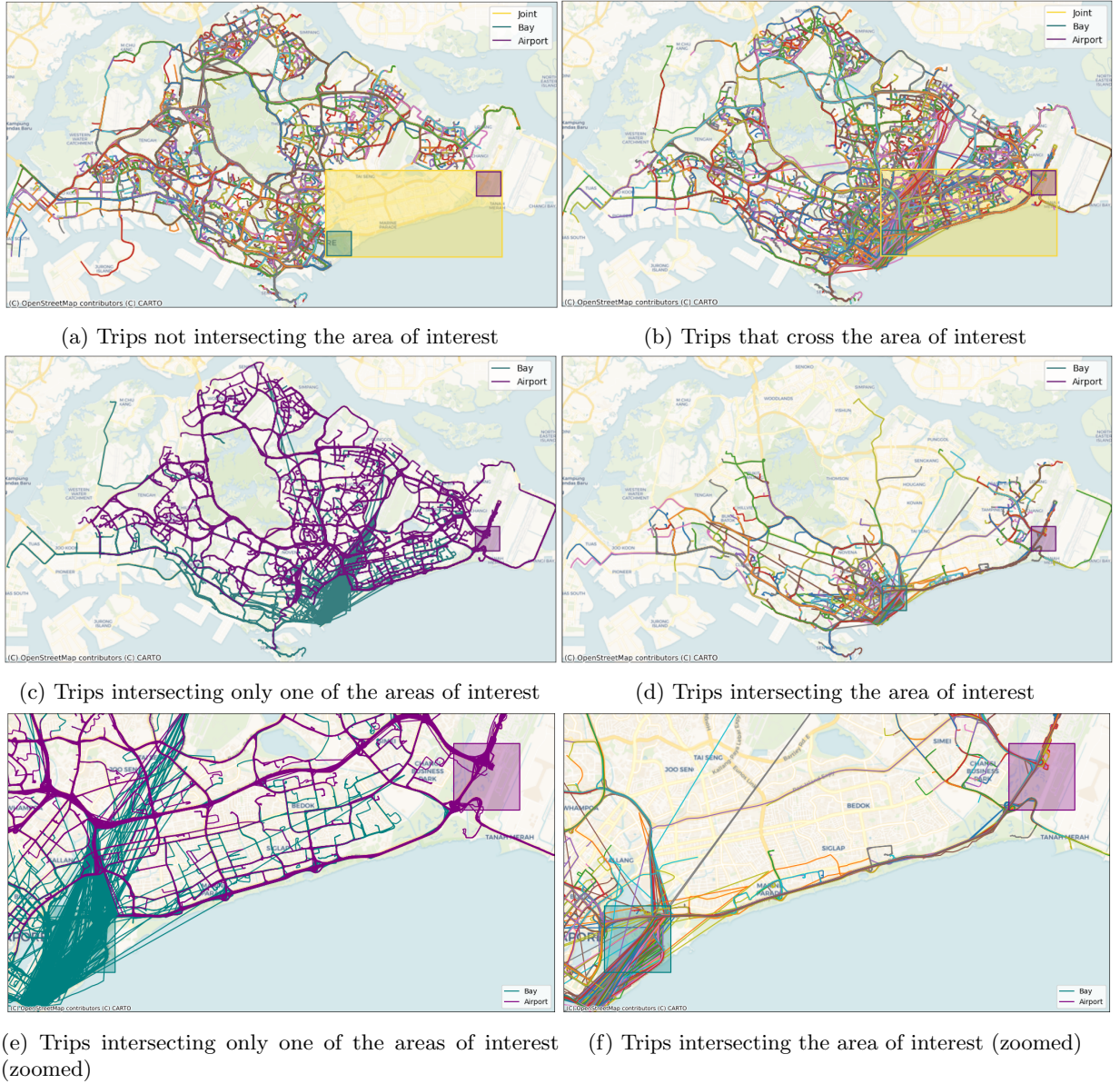


Figure 5.33: Filtering outcomes for trip selection, showing trajectories that intersect none, one, or both areas of interest (with zoomed views)

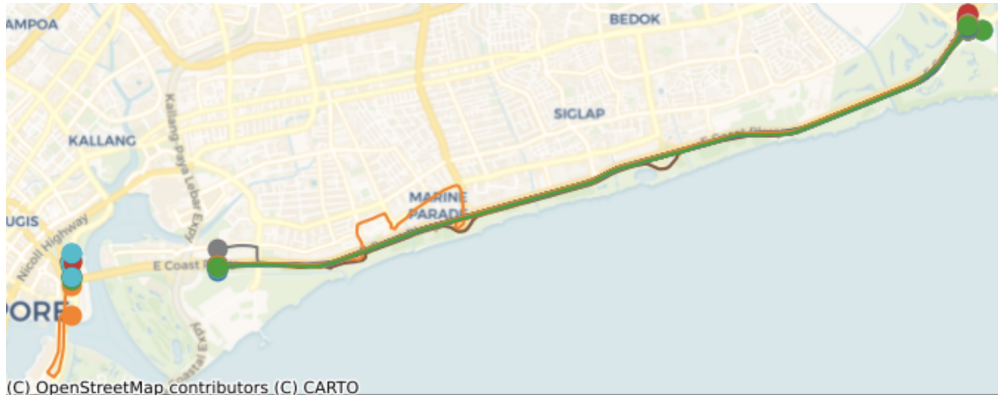


Figure 5.34: Single-leg trip geometries segmented between Singapore Airport and Gardens by the Bay using the geometry-based method

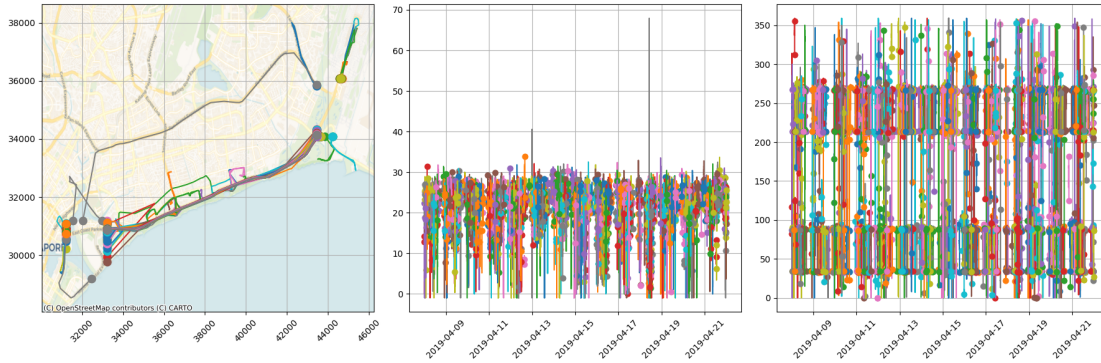


Figure 5.35: Speed and bearing profiles of geometry-based single-leg trip segments

sponded to a single trip, a small subset produced two or three legs due to multiple crossings of the areas of interest. In comparison, the stop-based method resulted in 1,448 single-leg trips, with an average of exactly 1.00 segment per trajectory. In this approach, nearly all trajectories are represented by a single segment, as very few included stops that met the 5 m/5 min criterion.

The trip number distribution further illustrates the difference in behavior. For the geometry-based method, 1,448 trajectories appear as a single segment, 64 trajectories are split into two segments, and six trajectories are split into three segments. For the stops-based method, all 1,448 segments correspond to the first leg (Trip #1), no higher trip numbers are observed. In other words, geometry-based splitting introduces additional legs when vehicles re-enter or oscillate around the areas of interest. In contrast, stop-based splitting remains conservative and rarely introduces extra segments at the chosen thresholds.

From a mobility perspective, this behavior is appropriate for the car domain. Vehicles tend to perform relatively direct movements between the two regions, often without long stationary periods, so the stop-based method yields mostly single-leg journeys. The geometry-based method reveals finer structure in routes that partially enter or graze the areas of interest, which may be useful when studying access patterns to specific facilities. For applications involving ships or slower modes, one would expect the stops-based method to produce a higher number of segments due to longer and more frequent stops.

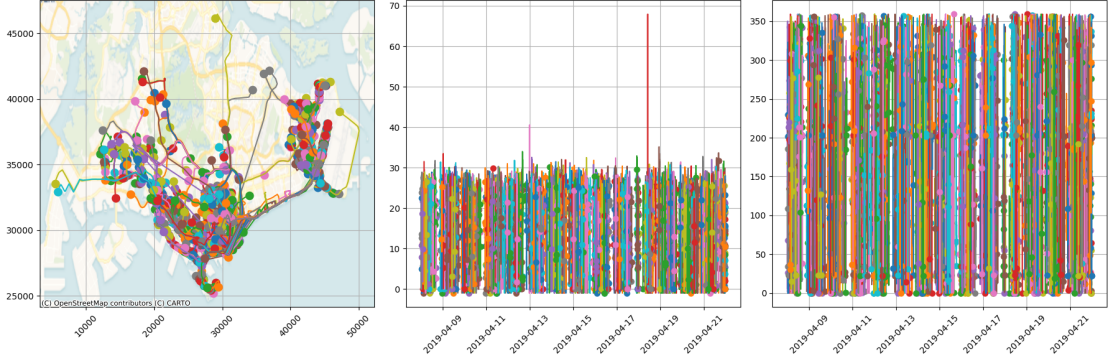


Figure 5.36: Speed and bearing profiles of stop-based single-leg trip segments

Method	Description	Single-leg Trips	Avg. Trips / Trajectory
Geometry-based	Split based on spatial containment using bounding boxes around Singapore Airport and Gardens by the Bay	1 518	1.05
Stop-based	Split based on detected stops (within 5 m for at least 5 min)	1 448	1.00

Table 5.4: Single-leg trip counts produced by geometry-based and stop-based segmentation

5.5.3 Metric Extraction

Beyond segmentation, MEOS and PyMEOS support the derivation of aggregate metrics for each trajectory or single-leg trip. For every trip, we compute the total traveled distance, the time-weighted average speed, the average bearing, and the travel time. Distance is obtained by summing segment lengths in projected coordinates (EPSG:3414), while speed and bearing are aggregated using time-weighted averages over the corresponding `tfloat` sequences.

The distribution of distances is concentrated between 10 km and 25 km, with only a few long cross-island journeys exceeding 100 km. The average-speed distribution peaks between 15 and 18 m/s, matching expected speeds on Singapore expressways. Travel-time durations exhibit a right-skewed distribution, with most trips lasting 15-35 minutes and a long tail representing congestion or stop-and-go periods. The distance-time scatter plot shows a near-linear relationship between the two, with dispersion around the line reflecting temporal variations in traffic conditions.

5.5.4 Area of Interest Heatmap

To obtain an overview of mobility intensity, we construct a trajectory density heatmap from the bay-airport trips. Using the spatial extent of all trips, we define a regular grid of 500 m $\times$ 500 m tiles in the projected coordinate system and count how many trajectories intersect each tile. We then project the tile centroids back to WGS84 and render the counts as a density map using Plotly, overlaid on an OpenStreetMap base layer.

The heatmap (see Fig. 5.37) reveals the main corridor connecting the airport region to the Gardens by the Bay area.

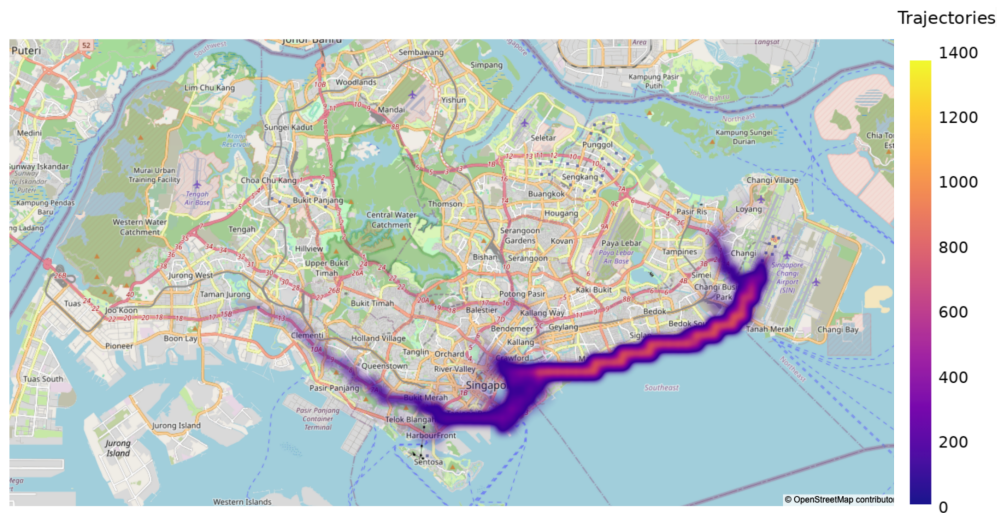


Figure 5.37: Trajectory-density heatmap of trips between Singapore Airport and the Gardens by the Bay area

5.5.5 Close Cars Identification

We detect proximity events between vehicles. Given two trajectories, MEOS can evaluate whether they were ever within a given distance threshold of each other using the `is_ever_within_distance` operator. We apply this check pairwise across trajectories and record all pairs of vehicles that come within 10 m at any time during the observation period. The resulting set of candidate encounters is stored in a separate table, which includes the identifiers of both vehicles and the corresponding trajectory segments.

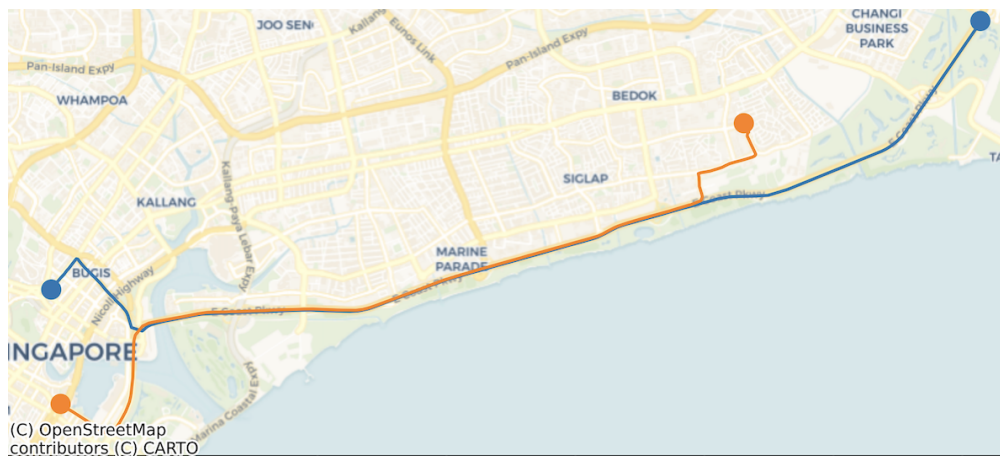


Figure 5.38: Example proximity event where two vehicles are detected within 10 m of each other

These proximity events have several potential applications. In transportation safety, they may highlight situations where vehicles share the same lane in dense traffic or approach each other at low headway. In mobility analysis, they can reveal convoys, shared routes, or localized congestion. The example in Figure 5.38 shows two vehicles traveling in close proximity along a shared path,

illustrating the types of patterns that can be identified with this approach.

5.6 Chapter Summary

This chapter demonstrated the capabilities of MobilityNebula in supporting real-world deployments and highlighted the role of MEOS in enabling mobility analytics. The SNCB/NMBS use case was introduced, illustrating the integration of *MobilityNebula* into an operational remote-diagnostics pipeline for railway vehicles. The onboard architecture and edge constraints were described, followed by an account of the integration process, including sources, sinks, and query deployment. The chapter also showed how continuous mobility queries, such as geofencing, speed and stop event detection, and brake-pressure variance monitoring, can be executed on trains to generate alerts and aggregated summaries.

Trajectory outlier detection was examined. Both static and streaming detection approaches were reviewed, and a ground-truth construction methodology was introduced, which relies on cross-checking speed and bearing consistency across sensors. This methodology facilitated the comparison of ten state-of-the-art open-source outlier-detection libraries across four real-world datasets. Insights from this study led to the implementation of an EKF outlier detector within MEOS, which was made available as a streaming operator in MobilityNebula, enabling online trajectory cleaning and enhancing the reliability of downstream mobility analytics.

The expressiveness of MEOS (and PyMEOS) spatiotemporal model and operations for trajectory analytics was demonstrated using the Grab-Posisi dataset. Trip segmentation was performed using geometry-based and stop-based strategies, and their behaviors were compared. Metric derivation, including distance, time-weighted average speed, bearing, and travel time, was demonstrated. Additionally, a trajectory-density heatmap was constructed, and proximity events were detected by identifying vehicle pairs that came within 10 m of each other. Collectively, these applications validate that the system and type model support operational edge monitoring and trajectory analytics, adding to our contributions and future directions discussed in Chapter 6.

Chapter 6

Conclusions and Future Work

Moving objects, such as trains, ships, and cars, are increasingly equipped with IoT sensors that generate continuous streams of spatiotemporal data. Processing these streams in real-time is valuable for applications such as maintenance monitoring, emergency event detection, and routing. Meanwhile, no existing system satisfies the combined requisites of this context. General stream processing engines, such as Flink and NebulaStream, are optimized for scalar and relational workloads and lack native support for mobility semantics, such as trajectory creation and spatial predicates. Spatiotemporal database systems, such as MobilityDB, implement expressive mobility operators, but are built for batch and historical workloads and cannot sustain continuous ingestion with low latency. The edge devices on board trains and ships impose strict budgets for CPU and memory, and intermittent network connectivity, making data transfer to the cloud impractical for tasks sensitive to latency.

In this thesis, we present a framework for distributed spatiotemporal stream processing for mobility data, targeting deployments across the edge-to-cloud continuum. We develop two systems: *MobilityFlink* and *MobilityNebula*. *MobilityFlink* serves as a proof of concept, while *MobilityNebula* has been tested in a real-world setting for the SNCB/NMBS use case.

MobilityFlink extends Flink to support spatiotemporal processing in large-scale cloud deployments. The system integrates MEOS types and functions through the JMEOS interface, enabling Flink to process mobility data within its streaming runtime. We demonstrate that mobility queries, such as spatial joins, range filters, and trajectory construction, can be expressed directly within Flink’s operator model.

MobilityNebula, in contrast, focuses on distributed and resource-constrained environments, enabling real-time spatiotemporal analytics along the edge-to-cloud continuum. We integrate MEOS operator set into NebulaStream and demonstrate that mobility queries, such as geofencing, geospatial complex event detection, and trajectory creation, can be executed on low-power edge devices.

The Engineering Department at SNCB/NMBS aims to ensure the operational safety and reliability of railway vehicles across distinct fleets. To support this mission, SNCB/NMBS develops and maintains a remote diagnostic platform for continuous acquisition and analysis of telemetry data from onboard subsystems, including pantographs, traction converters, doors, and HVAC units. This telemetry data is transmitted to a centralized cloud infrastructure, where batch analytics support anomaly detection and the generation of maintenance work orders within the company resource planning system. However, the scale and velocity of data generated by modern rolling stock have exposed limitations in purely cloud-centric architectures. The volume of raw sensor streams, often sampled at high frequencies, imposes significant bandwidth and storage costs,

while centralized processing becomes inefficient as data volumes grow. To resolve this, we have deploy the MobilityNebula system in the SNCB/NMBS diagnostic platform. The SNCB/NMBS deployment validated MobilityNebula in a real-world industrial setting. The system was executed directly on trains to process telemetry data from onboard sensors, such as brake pressure (BP) and GPS signals, in real-time. We have implemented queries that capture a portion of the operational needs and challenges described by the SNCB/NMBS maintenance team, ranging from detecting trains entering maintenance zones to monitoring BP and noise levels.

In addition to these systems, we contribute a comparative study on outlier removal in trajectory data. We evaluate ten open-source libraries across four large-scale datasets, representing urban, maritime, and air-traffic domains. We propose a ground-truth construction methodology that cross-references multi-sensor data streams using speed and bearing consistency. We provide a structured approach to verify data quality and evaluate algorithm performance in realistic settings.

Finally, we implement an Extended Kalman Filter (EKF) within MEOS for online outlier detection. We combine model-based prediction with statistical gating to identify and remove inconsistent measurements in real-time. We then implement this operation into MobilityNebula.

In summary, we contribute to the theory and practice of distributed spatiotemporal stream processing. From a theoretical perspective, we define a window-aware, sequence-based spatiotemporal model that extends the Moving Objects Database ((MOD) framework into a streaming context. From a practical standpoint, we deliver two operational systems: MobilityFlink for cloud-based analytics and MobilityNebula for edge-to-cloud deployments, which realize this model through real implementations and field-tested use cases.

6.1 Summary of Contributions

The contributions in this thesis are relevant to three groups: (1) mobility and transport operators, such as SNCB/NMBS, that need to process sensor measurement streams directly in edge devices even when the connectivity is unstable, (2) IoT system engineers that build pipelines of distributed analysis of fleets, vessels, smart infrastructures and that need expressive mobility queries under rigorous resource restrictions, and (3) data management researchers who work in spatiotemporal systems, stream processing, edge processing, and need open systems for processing mobility flows. We revisit the four research questions from Chap. 1 and summarize what was achieved for each.

RQ 1 - Data Model: *How can we define abstract and discrete window-aware spatiotemporal types that enable mobility operations in streaming environments?*

We have introduced three window-aware discrete types *stinstant*, *stsequence*, and *stsequenceset* that extend the sequence representation in MEOS with explicit window boundaries. These types associate each spatiotemporal value with a window interval, allowing the incremental construction of trajectories within the pipeline flow without loss of temporal semantics.

The hypothesis that such types can be constructed and used in real-time under edge-device constraints was confirmed. MobilityNebula creates and uses the *stsequence* in operators, maintaining the memory cost for events proportional to the number of instances inside the window.

RQ 2 - Systems Integration: *How can general streaming engines be extended with mobility awareness so that spatiotemporal queries run end-to-end with low overhead?*

We instantiated the proposed data model in two streaming engines: MobilityFlink (Flink), a cloud-oriented system, and MobilityNebula (NebulaStream), an edge-to-cloud system. Both systems integrate MEOS via thin adaptation layers without significantly modifying the stream

engine’s core code, confirming the hypothesis that the systems integration approach is viable.

The evaluation shows that geometric predicates such as `edwithin_tgeo_geo` and `eintersects_tgeo_geo` account for 10–16% of query execution time, and the remaining cost is attributable to NebulaStream window management, state, and watermark-oriented finalization.

We have chosen to prioritize MobilityNebula development over MobilityFlink, since its execution environment is not based on the JVM and MobilityFlink’s exclusive cloud deployment model is incompatible with the low-latency objectives and edge constraints of the SNCB/NMBS use case.

RQ 3 - Expressiveness for Real Use Cases: *Can real-world mobility operations, such as geofencing, Geospatial Complex Event Processing (GCEP), trajectory construction, stop detection, speed/bearing analytics, and railway diagnostics, be expressed as continuous queries and executed on constrained edge devices?*

We have implemented 13 continuous queries in MobilityNebula, covering geofencing, geospatial complex event processing, trajectory construction, trajectory smoothing, global closest pairs computation, and k -Nearest Neighbors (k -NN).

The SNCB/NMBS implementation validated these queries in an operational real-world context on board M6 trains, demonstrating functional equivalence between the streaming query results and the Programmable Logic Controller (PLC) reference diagnostic logic used by SNCB/NMBS maintenance engineers. The hypothesis that declarative queries can represent industrial operational needs was confirmed for the implemented set of requirements of SNCB/NMBS.

RQ4 - Data Quality and Outliers: *Which trajectory outlier detection methods are most effective and practical for mobility datasets, and how can such methods be incorporated into real-time pipelines?*

We carried out a comparative evaluation of 10 open-source libraries for outlier detection in trajectories across four large-scale data sets covering urban, maritime, and air traffic domains, and proposed a methodology for building reference data based on multi-sensor cross-checking of speed and course consistency.

The results confirmed the hypothesis that the library methods vary substantially between domains, and there is no method that dominates in all data sets. We also implemented a EKF within MEOS, and we integrated it as a flow operator in MobilityNebula, confirming that model-based incremental filtering is compatible with the flow pipeline in data rates and deployment window sizes in SNCB/NMBS.

6.2 Future Work

Although the four research questions have been addressed, each answer raises new open questions. The following future directions are organized around the identified gaps in the research questions: the integration of MEOS functions (RQ 2), the expansion of operational query coverage for SNCB/NMBS (RQ 3), and the improvement of data quality in the pipeline (RQ 4). Additionally, improvements in MobilityNebula at the infrastructure level, such as spatiotemporal indexing and hardware acceleration, are necessary to sustain performance as data rates and window sizes increase beyond the current implementation at SNCB/NMBS (RQ 1, RQ 2).

Complete Integration of MEOS Functionalities in MobilityNebula

The current system exposes a significant subset of MEOS operations in MobilityNebula, including spatiotemporal predicates such as `eintersects`, `edwithin`, and `nad_tgeo_stbox`, in addition

to aggregation operators such as `tsequence` and `ext_kalman_filter`, but MEOS offers a much broader set of spatiotemporal operations and data types that are not yet available as streaming operators.

Expanding this support demands a systematic integration process as each MEOS function should be mapped to the most adequate streaming operation (map, filter, window function, etc.), and the integration layer, composed of logical and physical layers, plugin wrapper, and parser, should be extended to each new operation. A challenge faced during development was that NebulaStream functions were originally designed to be unary or binary, whereas MEOS integration required three or more arguments, for example, to construct a point using longitude, latitude, and timestamp. This required explicit extensions of the logical and physical layer functions, types, and query parser.

A complete integration should automate these extensions via a code-generation script that preserves MEOS expressiveness, including lifting semantics and temporal synchronization, while respecting streaming constraints such as window-based processing, time semantics, and the arrival of out-of-order data. The creation of this pipeline would significantly reduce the engineering effort needed to expose MEOS function as operators.

In-Memory Spatial Index in MobilityNebula

In the evaluated queries in MobilityNebula, operations such as k -NN require pairwise distance computation among all points within each window. Without the index structure, pairwise evaluation scales quadratically with window size, which, while sufficient at current window sizes with tens of tuples at 1 Hz GPS, could become a bottleneck at higher data rates or larger windows. The introduction of in-memory spatial indexes, such as h3,¹ s2² indexes, and R-trees could potentially reduce the complexity.

However, maintaining these indexes within strict memory budgets of embedded devices, such as an Intel Atom 6 W processor with shared CPU and RAM across multiple diagnostic processes poses a non-trivial challenge. Future work could investigate index update strategies, such as spatial partitions with delta updates, spatial approximations that trade precision for memory efficiency, and indexing techniques based on learning, such as learned search [83] and adaptation of spatial stream distribution.

Hardware Acceleration of Spatiotemporal Operators in MobilityNebula

Spatial operators such as k -NN are computationally intensive: pairwise evaluation results in $O(N^2)$ operator invocations per window, a load that could dominate end-to-end latency at higher ingestion rates.

Hardware platforms such as GPUs and FPGAs offer potential acceleration for parallel and regular operations, but they introduce complexity in scheduling and a trade-off in data movement between the CPU and GPU, which need to be managed in resource-constrained environments. In the context of NebulaStream, the path to offloading to the GPU passes through the Nautilus engine, a hybrid execution engine that combines interpreted and tracing-based Just-in-Time (JIT) compilation, lowering hot paths to an intermediate representation compiled to native code via different backends, including LLVM.

Although this approach allows for the automatic generation of code, it limits developers' flexibility, as it requires them to write kernel-specific code for GPUs or CUDA, adapted for spatiotemporal operations whose irregularities and data dependencies make it difficult for a

¹<https://h3geo.org/docs/highlights/indexing/>

²<https://s2geometry.io>

general compiler to generate optimal code. Overcoming this challenge, whether by extending the CUDA language with spatiotemporal primitives so that the compiler can recognize them and optimize them by introducing support for kernel injection defined by the user in the Nautilus pipeline, still remains an open challenge.

Semantic-Aware Spatiotemporal Workflows in MobilityNebula

MobilityNebula processes raw spatiotemporal data, such as GPS coordinates, timestamps, and brake pressure sensor readings, but the SNCB deployment revealed that real operational data carry a rich context that the system currently ignores: the historical maintenance records can establish a specific baseline for equipment that could inform alert thresholds.

Making the processing semantic-aware can allow operators to incorporate uncertainty measures, indicate sensor health, annotate provenance, and add additional operational context directly into the filtering and querying steps. For example, a geofencing operator can weigh their decision based on the GPS signal trustworthiness, reducing false positives during periods of degraded signal, or a brake pressure anomaly detection operator could adjust its limits based on historical maintenance records.

The main technical challenge is defining a semantic schema and propagation rules sufficiently expressive to capture semantic diversity without significantly augmenting the operator state or increasing processing latency. Future work could explore propagation models for annotation integration into the streaming pipeline, and mechanisms for trust-based processing that join semantic reasoning with adaptive models to improve the robustness of feature filtering and extraction for downstream machine learning models.

Real-time Point Cloud Data Processing

A possible extension of the work presented in this thesis is supporting real-time point cloud processing in MobilityNebula. A point cloud is a digital 3D representation of an object or a place. It is constructed from a dense set of points with coordinates (x, y, z) and may include additional attributes, such as RGB values or semantic classifications. These data are collected by LiDAR sensors, which are sometimes mounted on moving devices such as trains, vehicles, and drones. Point cloud data is now central to infrastructure inspection, obstacle detection, and structural monitoring.

Current solutions for processing point clouds, such as the `pgPointCloud` extension combined with PostGIS,³ offer expressive operators for point clouds. However, these solutions are designed for historical data as they assume the entire dataset is available when processing. This approach is incompatible with real-time processing scenarios. More recently, the authors in [118] proposed a classifier that combines per-frame feature extraction with multi-frame Bayesian decision optimization to recognize moving vehicles, pedestrians, and cyclists. Meanwhile, the authors in [109] demonstrated a lightweight convolutional network for moving-object segmentation that can run at sensor rate on a Field-Programmable Gate Array (FPGA) and on a desktop GPU. These approaches show that near-sensor, per-frame LiDAR processing is feasible on constrained hardware. Incremental spatial indexes have been proposed for ingestion-time queries, like M3NO. However, to the best of our knowledge, no existing system provides declarative, edge-deployable stream processing with point-cloud operators.

We propose integrating a dedicated input formatter for the `sensor_msgs/PointCloud2` message type in ROS2, which is the streaming middleware for point clouds, as well as for raw UDP packets produced by LiDAR sensors such as Velodyne and Ouster. The current `DataType` catalog of

³<https://pgpointcloud.github.io/pointcloud/functions/postgis.html>

NebulaStream only supports scalar types and a VARSIZED binary type. It must be extended with a composite point type (x, y, z, i, t). Next, a set of physical operators must be registered through the PhysicalFunctionRegistry. These operators could mirror the pgPointcloud API while focusing on real-time processing. In addition, it could be possible to extend the Point Data Abstraction Library (PDAL)⁴ to work in real-time.

Expanded Monitoring and Alerts for SNCB/NMBS

MobilityNebula’s current implementation in SNCB/NMBS validates the system in real operational conditions, processing directly on edge devices. Still, it covers a subset of operational needs identified by the engineers from SNCB/NMBS. The same infrastructures of sensors present onboard the M6 trains, including brake pressure reading, GPS, door system, HVAC units, and pantographs, already offer the necessary data for a broader operational analysis, such as low battery detection, monitoring of door opening and closing, schedule deviation, emergency brake usage detection, and predictive maintenance of auxiliary systems.

Extending MobilityNebula to support these queries requires an integration of heterogeneous sources, such as GPS positions, climate sensors, and weather conditions, in unified workflows for streaming. The implementation of this process was demonstrated to be non-trivial, as the translation of the knowledge of diagnosis coded in PLC logic to continuous queries expressed by streaming operators, such as the brake detection query by the aggregation of the variance, demanded multiple interactions with SNCB/NMBS engineers, face-to-face visits, and study of the domain of context.

⁴<https://pdal.org/en/2.10.1/>

Bibliography

- [1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.
- [2] T. Akidau, E. Begoli, S. Chernyak, F. Hueske, K. Knight, K. Knowles, D. Mills, and D. Sotolongo. Watermarks in stream processing systems: semantics and comparative analysis of Apache Flink and Google Cloud Dataflow. *Proceedings of the VLDB Endowment*, 14(12):3135–3147, 2021.
- [3] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.
- [4] T. Akidau, S. Chernyak, and R. Lax. *Streaming systems: The what, where, when, and how of large-scale data processing*. O’Reilly Media, Inc., 06 2018.
- [5] L. Alarabi, M. Mokbel, and M. Musleh. ST-Hadoop: A MapReduce framework for spatiotemporal data. *GeoInformatica*, 22(4):785–813, 2018.
- [6] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
- [7] M. Bakli, M. Sakr, and T. Soliman. HadoopTrajectory: A Hadoop spatiotemporal data processing extension. *Journal of Geographical Systems*, 21(2):211–235, 2019.
- [8] M. Bakli, M. Sakr, and E. Zimányi. Distributed moving object data management in MobilityDB. In *Proceedings of the ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial ’19)*, pages 1–10. ACM, 2019.
- [9] M. Bakli, M. Sakr, and E. Zimányi. Distributed mobility data management in MobilityDB. In *Proceedings of the IEEE International Conference on Mobile Data Management (MDM ’20)*, pages 238–239. IEEE, 2020.
- [10] L. Benson, P. Grulich, S. Zeuch, V. Markl, and T. Rabl. Disco: Efficient distributed window aggregation. In *Proceedings of the International Conference on Extending Database Technology (EDBT ’20)*, pages 423–426. OpenProceedings.org, 2020.
- [11] A. Bhatnagar, P. Kassarjian, C. Liu, T. Lan, W. Yang, R. Cassius, D. Sahoo, D. Arpit, S. Subramanian, G. Woo, A. Saha, A. Jagota, G. Gopalakrishnan, M. Singh, K. Krithika, S. Maddineni, D. Cho, B. Zong, Y. Zhou, C. Xiong, S. Savarese, S. Hoi, and H. Wang. Merlion: End-to-end machine learning for time series. *The Journal of Machine Learning Research*, 24(1):10682–10687, 2023.

- [12] T. Brandt and M. Grawunder. GeoStreams: A survey. *ACM Computing Surveys*, 51(3):1–37, 2018.
- [13] M. Breunig, H. Kriegel, R. Ng, and J. Sander. LOF: Identifying density-based local outliers. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '00)*, pages 93–104. ACM, 2000.
- [14] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.
- [15] K. Cao, Y. Liu, G. Meng, H. Liu, A. Miao, and J. Xu. Trajectory outlier detection on trajectory data streams. *IEEE Access*, 8:34187–34196, 2020.
- [16] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4):28–38, 2015.
- [17] A. Chaudhary, K. Beedkar, J. Karimov, F. Lang, S. Zeuch, and V. Markl. Incremental stream query placement in massively distributed and volatile infrastructures. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE '25)*, pages 376–390. IEEE, 2025.
- [18] A. Chaudhary, J. Karimov, S. Zeuch, and V. Markl. Incremental stream query merging. In *Proceedings of the International Conference on Extending Database Technology (EDBT '23)*, pages 604–617. OpenProceedings.org, 2023.
- [19] A. Chaudhary, S. Zeuch, and V. Markl. Governor: Operator placement for a unified fog-cloud environment. In *Proceedings of the International Conference on Extending Database Technology (EDBT '20)*, pages 631–634. OpenProceedings.org, 2020.
- [20] N. Chaudhry, K. Shaw, and M. Abdelguerfi. *Stream data management*. Springer, 01 2005.
- [21] GEOS contributors. *GEOS computational geometry library*. Open Source Geospatial Foundation, 2025.
- [22] B. Custers, M. Kerkhof, W. Meulemans, B. Speckmann, and F. Staals. Maximum physically consistent trajectories. *ACM Transactions on Spatial Algorithms and Systems*, 7(4):1–33, 2021.
- [23] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [24] M. Duarte, D. Nugroho, G. Tod, E. Bevernage, P. Moelans, E. Tas, E. Zimányi, M. Sakr, S. Zeuch, and V. Markl. Mobility stream processing on NebulaStream and MEOS. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '25)*, pages 79–82. ACM, 2025.
- [25] M. Duarte and M. Sakr. An experimental study of existing tools for outlier detection and cleaning in trajectories. *GeoInformatica*, 29(1):31–51, 2024.
- [26] A. Eldawy. SpatialHadoop: Towards flexible and scalable spatial processing using MapReduce. In *Proceedings of the ACM SIGMOD PhD Symposium (SIGMOD'14)*, pages 46–50. ACM, 2014.

- [27] E. Eldawy and H. Mokhtar. Clustering-based trajectory outlier detection. *International Journal of Advanced Computer Science and Applications*, 11(5):133–139, 2020.
- [28] M. Ester, H. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD '96)*, pages 226–231. AAAI Press, 1996.
- [29] EUROCONTROL. Objective Skygreen 2022–2030: The economics of aviation decarbonisation towards the 2030 Green Deal milestone. Technical report, EUROCONTROL, 2022.
- [30] P. Filzmoser and M. Gschwandtner. mvoutlier: Multivariate outlier detection based on robust methods. R package, 2017. Software.
- [31] L. Forlizzi, R. Güting, E. Nardelli, and M. Schneider. A data model and data structures for moving objects databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '00)*, pages 319–330. ACM, 2000.
- [32] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos. A survey on the evolution of stream processing systems. *The VLDB Journal*, 33(2):507–541, 2023.
- [33] C. Freitas, C. Lydersen, M. Fedak, and K. Kovacs. A simple new algorithm to filter marine mammal Argos locations. *Marine Mammal Science*, 24(2):315–325, 2008.
- [34] Y. Fu and C. Soman. Real-time data infrastructure at Uber. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '21)*, pages 2503–2516. ACM, 2021.
- [35] Z. Galić. *Spatiotemporal data streams*. Springer, 09 2016.
- [36] Z. Galić, E. Meskovic, and D. Osmanovic. Distributed processing of big mobility data as spatiotemporal data streams. *GeoInformatica*, 21(2):263–291, 2017.
- [37] M. Garofalakis, J. Gehrke, and R. Rastogi. *Data stream management: processing high-speed data streams*. Springer, 01 2016.
- [38] A. Graser. MovingPandas: Efficient structures for movement data in Python. *GI Forum*, 1:54–68, 2019.
- [39] PostgreSQL Global Development Group. Just-in-time compilation. <https://www.postgresql.org/docs/current/jit-reason.html>, 2025. Accessed: October 2025.
- [40] P. Grulich, A. Lepping, D. Nugroho, V. Pandey, B. Del Monte, S. Zeuch, and V. Markl. Query compilation without regrets. *Proceedings of the ACM on Management of Data (PACMMOD)*, 2(3):1–28, 2024.
- [41] R. Güting, V. Almeida, D. Ansorge, T. Behr, Z. Ding, T. Hose, F. Hoffmann, M. Spiekermann, and U. Telle. SECONDO: An extensible DBMS platform for research prototyping and teaching. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE '05)*, pages 1115–1116. IEEE, 2005.
- [42] R. Güting, M. Böhlen, M. Erwig, C. Jensen, N. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems*, 25(1):1–42, 2000.

- [43] R. Güting and M. Schneider. *Moving objects databases*. Elsevier Science, 09 2005.
- [44] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.
- [45] M. Hahsler, M. Bolaños, and J. Forrest. Introduction to stream: An extensible framework for data stream clustering research with R. *Journal of Statistical Software*, 76(14):1–50, 2017.
- [46] R. Hu, L. Chen, and Y. Wang. An efficient outlier detection algorithm for data streaming, 2025.
- [47] W. Huang and X. Deng. GeoEkuiper: A cloud-cooperated geospatial edge stream processing engine for resource-constrained IoT devices with higher throughput. *IEEE Internet of Things Journal*, 11(18):30094–30113, 2024.
- [48] X. Huang, Y. Yin, S. Lim, G. Wang, B. Hu, J. Varadarajan, S. Zheng, A. Bulusu, and R. Zimmermann. Grab-Posisi: An extensive real-life GPS trajectory dataset in Southeast Asia. In *Proceedings of the ACM SIGSPATIAL International Workshop on Prediction of Human Mobility (IWCTS '19)*, pages 1–10. ACM, 2019.
- [49] I. Ilyas and X. Chu. *Data cleaning*. ACM, 07 2019.
- [50] J. Im, K. Gopalakrishna, S. Subramaniam, M. Shrivastava, A. Tumbde, X. Jiang, J. Dai, S. Lee, N. Pawar, J. Li, and R. Aringunram. Pinot: Real-time OLAP for 530 million users. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '18)*, pages 583–594. ACM, 2018.
- [51] A. Jain. Data clustering: 50 years beyond K-means. *Pattern Recognition Letters*, 31(8):651–666, 2010.
- [52] R. Jimenez-Peris, D. Burgos-Sancho, F. Ballesteros, M. Patiño Martinez, and P. Valduriez. Elastic scalable transaction processing in LeanXcale. *Information Systems*, 108(C):1–18, 2022.
- [53] K. Juraszek, N. Saini, M. Charfuelan, H. Hemsén, and V. Markl. Extended Kalman Filter for large scale vessels trajectory tracking in distributed stream processing systems. In *Proceedings of the ECML PKDD Workshop on Advanced Analytics and Learning on Temporal Data (AALTD '19)*, pages 151–166. Springer, 2019.
- [54] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE '18)*, pages 1507–1518. IEEE, 2018.
- [55] A. Kelly. A 3D state space formulation of a navigation Kalman Filter for autonomous vehicles. Technical Report CMU-RI-TR-94-19, Carnegie Mellon University, Robotics Institute, 05 1994.
- [56] D. Khalyeyev, T. Bureš, and P. Hnětynka. Towards characterization of edge-cloud continuum. In *Proceedings of Software Architecture: ECSA 2022 Tracks and Workshops (ECSA '22)*, pages 215–230. Springer, 2023.
- [57] B. Khazael, M. Asl, and H. Malazi. Geospatial complex event processing in smart city applications. *Simulation Modelling Practice and Theory*, 122:102675, 2023.

- [58] M. Khodarahmi and V. Maihami. A review on Kalman Filter models. *Archives of Computational Methods in Engineering*, 30(1):727–747, 2022.
- [59] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. Baveja. Lambda architecture for cost-effective batch and speed big data processing. In *Proceedings of the IEEE International Conference on Big Data (Big Data '15)*, pages 2785–2792. IEEE, 2015.
- [60] E. Knorr, R. Ng, and V. Tucakov. Distance-based outliers: Algorithms and applications. *The VLDB Journal*, 8(3):237–253, 2000.
- [61] J. Kotecha and P. Djuric. Gaussian particle filtering. *IEEE Transactions on Signal Processing*, 51(10):2592–2601, 2003.
- [62] J. Krämer and B. Seeger. Semantics and implementation of continuous sliding window queries over data streams. *ACM Transactions on Database Systems*, 34(1):1–49, 2009.
- [63] P. Kranen, H. Kremer, T. Jansen, T. Seidl, A. Bifet, G. Holmes, B. Pfahringer, and J. Read. Stream data mining using the MOA framework. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA '12)*, pages 309–313. Springer, 2012.
- [64] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of the International Workshop on Networking Meets Databases (NetDB '11)*, pages 1–7. Proceedings of the NetDB, 2011.
- [65] S. Krishnan and J. Gonzalez. *Building your next big thing with Google Cloud Platform: A guide for developers and enterprise architects*. Apress, 05 2015.
- [66] J. Lee, J. Han, and X. Li. Trajectory outlier detection: A partition-and-detect framework. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE '08)*, pages 140–149. IEEE, 2008.
- [67] S. Lee and M. West. Performance comparison of the distributed extended Kalman Filter and Markov Chain distributed Particle Filter. *IFAC Proceedings*, 43(19):151–156, 2010.
- [68] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: A new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment*, 1(1):274–288, 2008.
- [69] J. Lin. The Lambda and the Kappa. *IEEE Internet Computing*, 21(05):60–66, 2017.
- [70] J. Lu and R. Güting. Parallel SECONDO: Practical and efficient mobility data processing in the cloud. In *Proceedings of the IEEE International Conference on Big Data (Big Data '13)*, pages 107–125. IEEE, 2013.
- [71] A. Mahmood, A. Daghistani, A. Aly, M. Tang, S. Basalamah, S. Prabhakar, and W. Aref. Adaptive processing of spatial-keyword data over a distributed streaming cluster. In *Proceedings of the ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '18)*, pages 219–228. ACM, 2018.
- [72] W. McKinney. Data structures for statistical computing in Python. In *Proceedings of the Python in Science Conference (SciPy '10)*, pages 56–61. scipy.org, 2010.

- [73] J. Montiel, M. Halford, S. Mastelini, G. Bolmier, R. Sourty, R. Vaysse, A. Zouitine, H. Gomes, J. Read, T. Abdessalem, and A. Bifet. River: Machine learning for streaming data in Python. *Journal of Machine Learning Research*, 22(1):1–8, 2021.
- [74] S. Moosavi, B. Omidvar-Tehrani, and R. Ramnath. Trajectory annotation by discovering driving patterns. In *Proceedings of the ACM SIGSPATIAL Workshop on Smart Cities and Urban Analytics (UrbanGIS '17)*, pages 1–4. ACM, 11 2017.
- [75] G. Morales and A. Bifet. SAMOA: Scalable advanced massive online analysis. *Journal of Machine Learning Research*, 16(5):149–153, 2015.
- [76] A. Ng, M. I Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *Proceedings of the International Conference on Neural Information Processing Systems: Natural and Synthetic (NIPS '01)*, pages 849–856. MIT Press, 2001.
- [77] R. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In *Proceedings of the International Conference on Very Large Data Bases (VLDB '94)*, pages 144–155. Morgan Kaufmann Publishers Inc., 1994.
- [78] S. Nittel. Real-time sensor data streams. *SIGSPATIAL Special*, 7(2):22–28, 2015.
- [79] S. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. Campbell. Samza: Stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.
- [80] F. Oesterreicher and I. Vajda. A new class of metric divergences on probability spaces and its applicability in statistics. *Annals of the Institute of Statistical Mathematics*, 55(3):639–653, 2003.
- [81] A. Oliveira. Uma arquitetura e implementação do módulo de visualização para biblioteca PyMove. Bachelor's thesis, Universidade Federal do Ceará, 2019.
- [82] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference (ATEC '99)*, pages 183–191. USENIX Association, 1999.
- [83] V. Pandey, A. V. Renen, E. T. Zacharatou, A. Kipf, I. Sabek, J. Ding, V. Markl, and A. Kemper. Enhancing in-memory spatial indexing with learned search. In *Datenbanksysteme für Business, Technologie und Web (BTW '25)*, pages 303–330. Gesellschaft für Informatik, Bonn, 2025.
- [84] L. Pappalardo, F. Simini, G. Barlacchi, and R. Pellungrini. Scikit-mobility: A Python library for the analysis, generation, and risk assessment of mobility data. *Journal of Statistical Software*, 103(4):1–38, 2022.
- [85] R. Pearson, Y. Neuvo, J. Astola, and M. Gabbouj. Generalized Hampel Filters. *EURASIP Journal on Advances in Signal Processing*, 2016(1):87, 2016.
- [86] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [87] N. Pelekis, Y. Theodoridis, S. Vosinakis, and T. Panayiotopoulos. Hermes: A framework for location-based data management. In *Proceedings of the International Conference on Extending Database Technology (EDBT '06)*, pages 1130–1134. Springer, 2006.

- [88] D. Pokrajac, A. Lazarevic, and L. Latecki. Incremental local outlier detection for data streams. In *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining (CIDM '07)*, pages 504–515. IEEE, 2007.
- [89] M. Sakr and R. Güting. Spatiotemporal pattern queries in SECONDO. In *Proceedings of the International Symposium on Spatial and Temporal Databases (SSTD '09)*, pages 422–426. Springer, 2009.
- [90] M. Sakr, A. Vaisman, and E. Zimányi. *Mobility data science: From data to insights*. Data-centric systems and applications. Springer, 2025.
- [91] H. Salman, Y. Haranwala, V. Bogorny, C. Renso, V. Fonseca, and A. Soares. PTRAIL - a Python package for parallel trajectory data preprocessing. *SoftwareX*, 19:101176, 2022.
- [92] A. Sanches. Uma arquitetura e implementação do módulo de pré-processamento para biblioteca PyMove. Bachelor’s thesis, Universidade Federal do Ceará, 2019.
- [93] M. Schoemans, W. Aref, E. Zimányi, and M. Sakr. Multi-entry generalized search trees for indexing trajectories. In *Proceedings of the ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '24)*, pages 421–431. ACM, 2024.
- [94] D. Seidel, E. Dougherty, and W. Getz. Exploratory movement analysis and report building with R package stmove. *bioRxiv*, 2019.
- [95] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. Presto: SQL on everything. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE '19)*, pages 1802–1813. IEEE, 2019.
- [96] A. Shaikh, H. Kitagawa, A. Matono, and K.-S. Kim. A distributed and scalable framework for low-latency continuous trajectory stream processing. *IEEE Access*, 12:159426–159444, 2024.
- [97] S. Shaikh, H. Kitagawa, A. Matono, K. Mariam, and K. Kim. GeoFlink: An efficient and scalable spatial data stream management system. *IEEE Access*, 10:24909–24935, 2022.
- [98] J. Shi, Z. Pan, J. Fang, and P. Chao. RUTOD: Real-time urban traffic outlier detection on streaming trajectory. *Neural Computing and Applications*, 35(5):3625–3637, 2021.
- [99] The MathWorks Inc. *Optimization toolbox version: 9.4 (R2022b)*. The MathWorks Inc., Natick, Massachusetts, United States, 2022.
- [100] P. Thomas, J. Barr, B. Balaji, and K. White. An open-source framework for tracking and state estimation. In *Proceedings of SPIE, Vol. 10200: Signal Processing, Sensor/Information Fusion, and Target Recognition XXVI (SPIE '17)*, page 1020008, 2017.
- [101] J. Traub, P. Grulich, A. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Scotty: General and efficient open-source window aggregation for stream processing systems. *ACM Transactions on Database Systems*, 46(1):1–46, 2021.
- [102] J. Traub, P. Grulich, A. Rodriguez Cuellar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Efficient window aggregation with general stream slicing. In *Proceedings of the International Conference on Extending Database Technology (EDBT '19)*, pages 97–108. OpenProceedings.org, 2019.

- [103] M. Trofficus. Hampel filter in Python. Software, 2021. Python implementation.
- [104] C. Urrea and R. Agramonte. Kalman Filter: Historical overview and review of its use in robotics 60 years after its creation. *Journal of Sensors*, 2021(1):1–21, 2021.
- [105] J. Verwiebe, P. Grulich, J. Traub, and V. Markl. Survey of window types for aggregation in stream processing systems. *The VLDB Journal*, 32(5):985–1011, 2023.
- [106] Y. Wang and Z. Liu. A sneak peek at RisingWave: A cloud-native streaming database. In *Proceedings of the ACM International Conference on Distributed and Event-Based Systems (DEBS '22)*, pages 190–193. ACM, 2022.
- [107] T. White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 04 2015.
- [108] S. Wu, E. Zimányi, M. Sakr, and K. Torp. Semantic segmentation of AIS trajectories for detecting complete fishing activities. In *Proceedings of the IEEE International Conference on Mobile Data Management (MDM '22)*, pages 419–424. IEEE, 2022.
- [109] Xing Xie, Haowen Wei, and Yongjie Yang. Real-time LiDAR point-cloud moving object segmentation for autonomous driving. *Sensors*, 23(1):547, 2023.
- [110] S. Yang, M. Madsen, and J. Bednar. HoloViz: Visualization and interactive dashboards in Python. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22)*, pages 4846–4847. ACM, 2022.
- [111] X. Yang, L. Tang, X. Zhang, and Q. Li. A data cleaning method for big trace data using movement consistency. *Sensors*, 18(3):824, 2018.
- [112] S. Yilmaz and S. Kozat. PySAD: A streaming anomaly detection framework in Python. *arXiv preprint arXiv:2009.02572*, 2020.
- [113] J. Yu, Z. Zhang, and M. Sarwat. Spatial data management in Apache Spark: the GeoSpark perspective and beyond. *GeoInformatica*, 23(1):37–78, 2019.
- [114] Y. Yu, L. Cao, E. Rundensteiner, and Q. Wang. Outlier detection over massive-scale trajectory streams. *ACM Transactions on Database Systems*, 42(2):1–33, 2017.
- [115] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the USENIX Conference on Hot Topics in Cloud Computing (HotCloud '12)*, pages 1–6. USENIX Association, 2012.
- [116] S. Zeuch, A. Chaudhary, B. Monte, H. Gavriilidis, D. Giouroukis, P. Grulich, S. Bress, J. Traub, and V. Markl. The NebulaStream platform: Data and application management for the internet-of-things. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR '20)*, pages 1–11. cidrdb.org, 2020.
- [117] S. Zeuch, E. Zacharatou, S. Zhang, X. Chatziliadis, A. Chaudhary, B. Monte, D. Giouroukis, P. Grulich, A. Ziehn, and V. Markl. NebulaStream: Complex analytics beyond the cloud. *Open Journal of Internet Things*, 6(1):66–81, 2020.
- [118] MingFang Zhang, Rui Fu, YingShi Guo, and Li Wang. Moving object classification using 3D point cloud in urban traffic environment. *Journal of Advanced Transportation*, 2020:1583129, 2020.

- [119] S. Zhang, J. Soto, and V. Markl. A survey on transactional stream processing. *The VLDB Journal*, 33(2):451–479, 2024.
- [120] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '96)*, pages 103–114. ACM, 1996.
- [121] Z. Zhang, C. Jin, J. Mao, X. Yang, and A. Zhou. TrajSpark: A scalable and efficient in-memory management system for big trajectory data. In *Proceedings of the International Joint Conference on Web and Big Data (APWeb-WAIM '17)*, pages 11–26. Springer, 2017.
- [122] X. Zheng, D. Yu, C. Xie, and Z. Wang. Outlier detection of crowdsourcing trajectory data based on spatial and temporal characterization. *Mathematics*, 11(3):620, 2023.
- [123] Y. Zheng. Trajectory data mining: An overview. *ACM Transactions on Intelligent Systems and Technology*, 6(3):1–41, 2015.
- [124] A. Ziehn. Complex event processing for the internet-of-things. In *Proceedings of the VLDB PhD Workshop (VLDB '20)*, pages 1–4. CEUR-WS.org, 2020.
- [125] E. Zimányi, M. Duarte, and V. Diví. MEOS: An open-source library for mobility data management. In *Proceedings of the International Conference on Extending Database Technology (EDBT '24)*, pages 810–813. OpenProceedings.org, 2024.
- [126] E. Zimányi, M. Sakr, M. Bakli, M. Schoemans, D. Tsesselis, and R. Choquet. MobilityDB: Hands on tutorial on managing and visualizing geospatial trajectories in SQL. In *Proceedings of the ACM SIGSPATIAL International Workshop on APIs and Libraries for Geospatial Data Science (SpatialAPI '21)*. ACM, 2021.
- [127] E. Zimányi, M. Sakr, A. Lesuisse, and M. Bakli. MobilityDB: A mainstream moving object database system. In *Proceedings of the International Symposium on Spatial and Temporal Databases (SSTD '19)*, pages 206–209. ACM, 2019.