Mobility Trajectory Data Stream Processing Beyond the Cloud

Mariana M. Garcez Duarte
mariana.machado.garcez.duarte@ulb.be
Université libre de Bruxelles
Brussels, Belgium

Dwi P. A. Nugroho
d.nugroho@tu-berlin.de
BIFOLD, TU Berlin
Berlin, Germany

Georges Tod
georges.tod@belgiantrain.be
SNCB-NMBS Engineering
Brussels, Belgium

Evert Bevernage
evert.bevernage@belgiantrain.be
SNCB-NMBS Engineering
Brussels, Belgium

Pieter Moelans
pieter.moelans@belgiantrain.be
SNCB-NMBS Engineering
Brussels, Belgium

Elias Saerens
elias.saerens@belgiantrain.be
SNCB-NMBS Engineering
Brussels, Belgium

Esteban Zimányi
esteban.zimanyi@ulb.be
Université libre de Bruxelles
Brussels, Belgium

Mahmoud Sakr
mahmoud.sakr@ulb.be
Université libre de Bruxelles
Brussels, Belgium

Steffen Zeuch
steffen.zeuch@tu-berlin.de
BIFOLD, TU Berlin
Berlin, Germany

Abstract

The increase in Internet-of-Things (IoT) sensors mounted on moving objects has resulted in continuous spatiotemporal data streams. Nevertheless, current mobility stream processing systems remain inadequate, as they are only optimized for the cloud and cannot effectively cope with IoT workloads. General-purpose stream systems that operate across edge–to–cloud settings do not support spatiotemporal operations. In contrast, existing spatiotemporal libraries are primarily designed for historical analysis rather than real-time streaming workloads. We propose MobilityNebula, a system that supports spatiotemporal stream processing across the edge–to–cloud continuum. We evaluate the system in a real-world deployment with the Belgian railway operator (SNCB), where train-mounted edge devices ingest sensor data and execute 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.

Keywords

Spatiotemporal Data, Trajectory Data, Edge Processing, Mobility Streams

1 Introduction

The growing use of IoT sensors in moving objects, such as 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 effective in-situ spatiotemporal processing poses challenges due to high data rates, varying network conditions, and the limited CPU and memory capabilities of edge devices [30].

Stream processing frameworks, such as Kafka Streams [26] and Apache Flink [6], are well-suited for handling high-throughput event streams. However, these systems are not specifically designed to cope with IoT workloads, as they are optimized primarily for the cloud. In addition, these systems often lack built-in spatiotemporal functionality, requiring users to implement such operations manually. 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 [32].

To bridge this gap, we present MobilityNebula,[1] an integrated system for spatiotemporal stream processing on the edge–to–cloud continuum. MobilityNebula is based on the distributed NebulaStream platform [39] and the MEOS library for spatiotemporal data processing [42]. Extending spatiotemporal processing from a database context to an unbounded stream requires more than systems integration. In particular, spatiotemporal types and operations must become *stream-aware*, that is, associate spatiotemporal types with window bounds and enable trajectories to be constructed and queried at stream rate. In addition, mobility functions originating in the database domain must be mapped and translated into streaming operators that compose with windowing, grouping, and incremental execution. MobilityNebula supports point and trajectory functions and runs on constrained edge and fog devices. In earlier work, we implemented a point-based model in MobilityNebula [11, 12]. However, real-time mobility decisions depend not just on where they are, but on how objects move, making trajectory support essential. Trajectories capture more information than points, such as motion, speed, acceleration, and heading. They can help improve prediction and control, such as ETAs, incident detection, and route deviation detection. Trajectories can also inform interactions between moving objects, such as convoys and near misses. In this paper, we focus on *trajectory-based* streaming, where we define the data model and demonstrate how to build, aggregate, and query windowed trajectories in real-time on edge devices within an edge–to–cloud streaming platform.

We implement our data model in MobilityNebula and deploy it in a real-world setting. We perform queries directly on an edge device, maintaining low latency, enabling on-demand data gathering, allowing on-device operation, which is a prerequisite for deployments with intermittent connectivity. While generic engines can represent similar spatiotemporal predicates as User Defined Functions (UDFs), MobilityNebula 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

[1] https://github.com/MobilityDB/MobilityNebula

EDBT '26, Tampere (Finland)

10.48786/edbt.2026.34

rather than raw measurements. Additionally, it is designed to mitigate intermittent connectivity by executing the full pipeline on-device and transmitting only windowed trajectories and alerts when connectivity is available. The underlying query plans and operator placement across devices are managed by NebulaStream distributed runtime [11, 12], which MobilityNebula leverages without modifying its core scheduling mechanisms. In this work, we therefore concentrate on the mobility-aware type system, query expression, aggregation patterns, and operator implementation. Distributed operator placement and scale-out placement have been studied in prior work on NebulaStream and related systems [7, 8, 25] and are complementary to our contributions.

In collaboration with the Belgian railway operator (SNCB), we construct a use case and validate the platform's effectiveness by deploying MobilityNebula on edge devices onboard trains. On SNCB trains, safety checks, such as spotting brake anomalies, must be executed in a near real-time, even when the connection is unreliable. Sending raw GPS and sensor streams to the cloud without preprocessing can increase the processing pipeline's time, utilize a significant portion of the network bandwidth, and lead to increased latency.

The main technical challenge lies in the model mismatch between general-purpose stream processing systems and trajectory processing systems. Based on this mismatch, we identify two challenges for mobility stream processing engines across the edge–to–cloud continuum that are not currently addressed by existing systems:

- **Ch1: Trajectory construction at stream rate:** Build and analyze window-bounded trajectories in real-time while preserving the semantics of discrete moving objects and their window bounds.
- **Ch2: Low-latency mobility operators on constrained hardware:** Implement distance, containment, intersection, and k nearest neighbor (kNN) predicates over windowed trajectories as streaming operators that fit within the CPU and memory budgets of edge devices.

MobilityNebula targets these challenges by lifting the Moving Objects Database (MOD) framework and MEOS sequence representation into a streaming setting, and by implementing spatiotemporal operators that are lightweight enough to run on constrained devices, yet expressive to support practical mobility analytics.

Addressing Ch1–Ch2, this paper makes four contributions:

- **Ct1: Windowed discrete trajectory type system (Ch1).** We extend the MOD framework and MEOS sequence representation with window-aware spatiotemporal types (*stinstant*, *stsequence*, *stsequenceset*) that bind each temporal geometry to its window bounds. This type system encodes each trajectory and its window as a single value for streaming analytics.
- **Ct2: Aggregation as trajectory construction (Ch1, Ch2).** We formalize trajectory construction as a windowed aggregation in NebulaStream lift-combine-lower interface. When the window closes, `lower` orders the collected instants by event time and produces *stsequence*/*stsequenceset* objects.
- **Ct3: Edge-executable spatiotemporal operators (Ch2).** We integrate spatiotemporal functions into NebulaStream, such as distance, within, intersection, trajectory construction, and kNN as streaming operators. This allows the passing of MEOS objects by reference, minimizing overhead on constrained edge nodes. Temporal geometries are stored once in MEOS C format and passed by reference between operators, avoiding repeated (de)serialization and data copying. We expose MEOS functions such as `edwithin_tgeo_geo` as operators that can be combined with windowing, grouping, and aggregation while fitting within the CPU and memory budgets of edge nodes.
- **Ct4: Real-world railway deployment and validation.** We deploy MobilityNebula on SNCB trains, implementing high-risk-zone speed enforcement, trajectory creation, and proactive brake-system health monitoring, while sustaining near real-time latencies.

Outline. The remainder of this paper is organized as follows. Section 2 introduces concepts related to mobility stream processing. Section 3 shows the system components. Section 4 details the SNCB use case. Section 5 validates the system through experimental results. Section 6 provides an overview of prior research in continuous data processing and spatiotemporal data management. Section 7 concludes and discusses potential future work.

2 Background

This section introduces concepts of streaming spatiotemporal data. In particular, we provide background on spatiotemporal streams, mobility data, MEOS, NebulaStream, and aggregation functions.

2.1 Spatiotemporal Data Streams

Spatiotemporal streams are continuous flows that capture the movements of spatial objects over time [17]. These streams integrate data from multiple sources, including GPS receivers. Unlike generic event streams, spatiotemporal streams pair *event time* with *location*. These streams present both domain-specific challenges and opportunities [5]. This data type can have a path-dependent state in many predicates that depend on the entire trajectory history, such as average speed. Spatiotemporal streams can require expensive operators, such as calculating distance and defining k-Nearest Neighbors, 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 requires preprocessing of the sequence, as stream systems can accept late data and trajectories cannot.

2.2 Mobility Data

Mobility data is spatiotemporal data for rigid moving objects, whose geometry remains constant while their position changes over time. We assume the path between observations is interpolable and the object keeps a fixed shape and size. For example, the trajectories of cars, aircraft, ships, and trains.

2.3 MEOS

MEOS[2] (Mobility Engine, Open Source) is a C library that provides functionalities for spatiotemporal management of data [32,

[2]https://libmeos.org/

42]. MEOS implements abstractions such as spatiotemporal instants, sequences, and bounding boxes to model the temporal evolution of values. For example, a temporal float can capture a train's speed over time. Traditional approaches tend to decouple the spatial and temporal components, whereas MEOS processes them jointly, allowing queries such as tracking a train's path through specific zones at a precise time. In addition, the library decouples storage from processing, making MEOS portable to both edge-restricted platforms and cloud infrastructures.

In this paper, we use the following MEOS functions:

- `temporal_sequence(lon, lat, ts)` aggregates points (lon, lat, ts) in a temporal sequence, a time-ordered trajectory that records where the object was at each timestamp.
- `edwithin_tgeo_geo(lon, lat, ts, geometry, dist)` evaluates whether the temporal point comes within distance of a static geometry. In our queries, the predicate equals 1 when the device enters the specified area.
- `eintersects_tgeo_geo(lon, lat, ts, geom)` tests whether the temporal point ever intersects a static geometry.
- `tgeo_at_stbox(lon, lat, ts, stbox)` checks whether the temporal point lies inside a given spatiotemporal box (`stbox`), combining a spatial rectangle with a time interval.
- `nad_tgeo_stbox(lon, lat, ts, stbox)` computes the nearest approach distance, using the minimum distance between a temporal point and a spatiotemporal box.
- `nearest_approach_distance(lon, lat, ts, lon2, lat2, ts2)` computes the nearest approach distance between temporal points, using the minimum distance attained over time.
- `temporal_ext_kalman_filter(lon, lat, ts, gate, q, variance, to_drop)` returns a smoothed temporal point obtained by filtering a noisy trajectory with an Extended Kalman Filter. `gate` controls outlier detection strictness, `q` is the process noise scale, `variance` is the measurement noise variance, and `to_drop` selects whether outliers are removed or replaced by the predicted point.

Within MobilityNebula, the integration enables MEOS functions to be combined with NebulaStream operations, including windowing, grouping, and aggregation.

2.4 NebulaStream

NebulaStream [39] is an edge–to–cloud data stream processing system. The system has lower resource demands compared to large-scale engines such as Kafka Streams [26] and Flink [6], making it suitable for devices with limited CPU and memory capacities [40]. The system can accommodate varying workloads [8], such as trains that are passing through regions with volatile connectivity. By processing data at the edge, NebulaStream aims to reduce reliance on stable network connections and minimize latency, while still scaling execution across edge, fog, and cloud resources when available.

2.5 Aggregation Functions

Aggregation process all elements in a window or dataflow and return a single result. They are stateful, hence each function maintains state, updates it as new records arrive, and produces a result when required. Examples of aggregation functions include `sum`, `count`, and `average`.

NebulaStream implements aggregation using three functions: `Lift`, `Combine`, and `Lower` [18]. This design supports parallel execution and enables incremental and reusable aggregation, regardless of whether the computation occurs on sensors, fog nodes, or in the cloud. In addition, the system can update its state instead of recomputing the entire aggregation when new records arrive.

For example, the average function can be expressed as follows:

- `lift(record)` → `partial`: creates a partial state from a single record. For instance, $lift(x) = sum : x, count : 1$.
- `combine(partialA, partialB)` → `partial`: merges two partials of the same type.
  For example, $s_1, c_1 \oplus s_2, c_2 = s_1 + s_2, c_1 + c_2$.
- `lower(partial)` → `result`: produces the final result from the partial state. For example, $lower(s, c) = s/c$.

3 MobilityNebula

The goal of MobilityNebula is to support Moving Objects Database (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. 1, 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. 2 into NebulaStream's execution framework [12]. 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

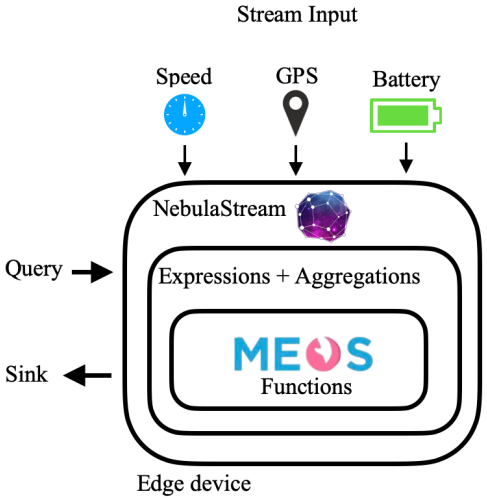


Figure 1: MobilityNebula system overview

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.

Another challenge is to handle aggregation functions over streaming mobility data. We extend MEOS by introducing temporal windows into the sequence-based representation and new abstractions called *stinstant*, *stsequence*, and *stsequenceset* (see Sect. 3.1). These data types allow the aggregation of spatiotemporal sequences within windows while preserving temporal coherence.

For trajectory creation, we utilize an aggregator whose partial state for each (device, window) contains (*longitude*, *latitude*, *timestamp*). The aggregator updates the trajectory incrementally in a sliding window. At window close, the aggregator either (i) emits a statistic, such as an aggregated average or minimum speed, or (ii) emits the *stsequence* for downstream operators, such as spatial filters or joins.

3.1 Data Types

The data types in this paper align with the MOD framework defined in [21], which consists of two layers: an abstract model that captures the semantics of movement and a discrete model that implements the model.

We apply the abstract model of [13, 21] and the discrete representation of MEOS [43]. The MOD framework maintains abstract movement semantics unchanged across placements, from edge to cloud, while the discrete layer adapts to streaming needs. This modeling choice is not only conceptual but also operational in MobilityNebula. To enable compatibility with MEOS data types and operations, we utilize the same MOD representation, and we extend the type system for streaming-based systems by introducing window-aware data types (*stinstant*, *stsequence*, *stsequenceset*), inspired by the discrete spatiotemporal type system of the MobyDick framework on Apache Flink [16].

3.1.1 Abstract Model. The abstract model proposed by the authors in [20] 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 geometry* represents a spatial location whose position changes continuously over time, while a *range* captures time intervals or numeric values. This conceptual foundation describes object movements over time, although it does not define methods for data storage or query execution.

Table 1: Type System - Abstract Model

Type Constructor	Signature	Type Category
integer, float, text, bool		$\rightarrow$ *BASE*
geometry, geography		$\rightarrow$ *SPATIAL*
instant, interval		$\rightarrow$ *TIME*
moving	*BASE* $\cup$ *SPATIAL*	$\rightarrow$ *TEMPORAL*
range	*BASE* $\cup$ *TIME*	$\rightarrow$ *RANGE*

Table 1 summarizes the abstract type constructors for the type system, adapted from [20]. 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.

3.1.2 Discrete Model. The discrete model implements the abstract model by defining explicit strategies for storing, updating, and querying spatiotemporal data.

The *sliced representation*, as implemented in SECONDO [19, 20], decomposes a trajectory into a series of units. Each unit stores the segment between two consecutive observations, along with the corresponding interval. Although this representation is conceptually straightforward, it inadvertently duplicates boundary events, thereby contributing to elevated storage requirements and increased I/O costs.

This paper is based on an alternative model called *sequence representation* [32]. This representation models the same trajectory as a chain of instants, storing every observation exactly once and introducing a split only when a temporal gap occurs. This method eliminates repeated bounds embedded in the *sliced representation*. Additionally, the sequence design incorporates temporal continuity within its type. As a result, operations can eliminate the continuity checks that sliced algorithms require. MEOS implements the following constructors for sequence representation: *tinstant*, *tsequence*, and *tsequenceset*.

To support continuous data ingestion, we extend this sequence representation with the concept of *temporal windows* [16]. This extension results in *windowed spatiotemporal sequence* (see Fig. 2), 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 valid time interval. Table 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.

3.1.3 Data Type Example. Consider the GPS measurements in Fig. 2a to demonstrate the discrete model in practice. Each longitude, latitude, and timestamp tuple is lifted into a single value, following the *tinstant* and *tsequence* discrete models [32]. 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
```

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 2b 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. 2b 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*. By contrast, 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. 2b 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

Table 2: Type System - Discrete Model

Type Constructor	Signature	Type Category
integer, float, text, bool		→ *BASE*
geometry, geography		→ *SPATIAL*
timestamptz, tstzset, *tstzspan, tstzspanset*		→ *TIME*
tbool, tint, tfloat, ttext, tinstant, *tsequence, tsequenceset*	*BASE* × *TIME*	→ *TEMPORAL*
tgeometry, tgeography, *tgeompoint, tgeogpoint*	*SPATIAL* × *TIME*	→ *TEMPORAL*
now, unbounded, range		→ *WINDOW*
stinstant, stsequence, stsequenceset	*TEMPORAL* × *WINDOW*	→ *STEMPORAL*
stgeometry, stgeography, *stgeompoint, stgeogpoint*	*TEMPORAL* × *WINDOW*	→ *STEMPORAL*

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.

In MobilityNebula, *stsequence* and *stsequenceset* are produced by a trajectory aggregator. 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 *tinstants* 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.

Event-time semantics and trajectory aggregation. 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 (i) extract all n points from the internal state into a vector in $O(n)$ time and $O(n)$ memory, (ii) sort the vector by event time in $O(n \log n)$ time, and (iii) 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.

3.2 MobilityNebula Query Language

We extend the NebulaStream parser to incorporate MEOS spatiotemporal functions in queries. Mobility functions are written in red in our queries. For instance, Code 1 shows a sliding-window query that selects points within three meters of a spatiotemporal box by calling the function `nad_tgeo_stbox`.

Code 1. *Distance-Based Filtering*

```
Query::from(GPS)
.filter(nad_tgeo_stbox(lon, lat, ts, POLYGON((4.3 50.6, 4.3
↪  50.7, 4.4 50.7, 4.4 50.6, 4.3 50.6))) < 3)
.window(SlidingWindow::of(EventTime(ts), Seconds(60),
↪ Seconds(3)))
```

First, we specify the input stream in Code 2. In our deployment, GPS refers to the real-time stream of points from trains, including GPS coordinates, timestamps, speed, and pressure readings.

Code 2. *Input Stream Defining*

```
Query::from(GPS)
```

Next, we apply a distance-based filter (Code 3). 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 3. *Filtering*

```
.filter(nad_tgeo_stbox(lon, lat, ts, POLYGON((4.3 50.6, 4.3
↪  50.7, 4.4 50.7, 4.4 50.6, 4.3 50.6))) < 3)
```

We group the remaining points into windows (Code 4). We use event time so that each point timestamp defines its window assignment. We then specify a 60-second window duration with a three-second slide. Every three seconds, a new 60-second 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. Finally, we attach a print sink to emit the results whenever the window ends.

Code 4. *Window Defining*

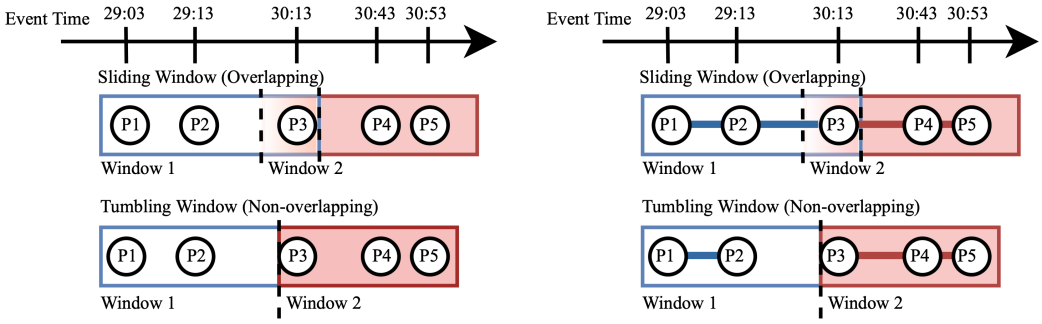
```
.window(SlidingWindow::of(EventTime(ts), Seconds(60),
↪ Seconds(3)))
```

4 Use Case

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

Longitude	Latitude	Timestamp	MobilityNebula *tinstant*
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

(a) Input (raw data values) and output (*tinstant* values)



(b) Window strategies for creating *stinstant* and *stsequence*

Figure 2: Data representation, windowed instants, and windowed sequences

or unscheduled stops, directly on edge devices located aboard trains. Historically, SNCB has utilized Programmable Logic Controllers (PLCs) to detect anomalies in sensor readings. While PLCs are fast, aligning with geographically aware rules can be challenging. Additionally, unlike SNCB's previous approach, the system can remotely deploy queries, allowing for fast reconfiguration of alert and monitoring logic.

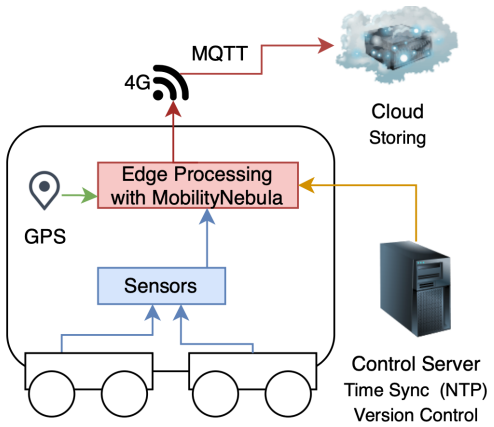


Figure 3: Use case architecture

In this use case, we utilize MobilityNebula for maintenance monitoring on trains operated by Société Nationale des Chemins de fer Belges (SNCB), with sensor data ingested directly on edge devices. The system continuously assesses brake performance to detect wear indicators before failures occur and indicates the train's location if wear is found. This data-driven approach focuses on maintaining resources by actual needs rather than adhering to fixed maintenance schedules, thereby improving reliability and reducing downtime. Figure 3 shows the use case architecture. Each wagon is equipped with an nROK device that receives GPS and sensor readings. Sensors, GPS and edge processing device are aboard the train. Processing is performed within this edge device, and alerts are transmitted via MQTT to SNCB's headquarters and stored on a cloud server in Microsoft Azure. A control server located at SNCB headquarters occasionally synchronizes the timestamps for all sensors and can also be used to deploy new queries remotely.

For this deployment, all operational decisions must be taken locally on each train with minimal latency, motivating local decision-making. Consequently, the complete safety pipeline runs on the train-mounted edge device, while the cloud-side infrastructure is used only for monitoring, data archival, and offline analysis. Therefore, the use case does not require distributing trajectory operators across multiple edge nodes. Instead, it benefits from executing the full trajectory logic as close as possible to the sensors.

4.1 Queries

We implement nine queries in MobilityNebula[3] that capture a part of the operational needs and challenges described by the SNCB maintenance team. These queries were chosen because they reflect real-world tasks that SNCB technicians perform when monitoring trains. For all the following queries, we assume a print sink is attached to print out the results.

Query 1 performs high-risk zone proximity monitoring. It identifies trains in proximity to dangerous zones by invoking `edwithin_tgeo_geo` on the SNCB stream against predefined high-risk area polygons (INPolygons) (Line 2). Any incoming temporal point within the specified distance of 20 meters in a

[3]Queries can be found at https://github.com/MobilityDB/MobilityNebula

ten-second tumbling window (Line 3) is immediately printed and generates real-time alerts for trains near hazardous locations.

Query 1. *High-Risk Zone Proximity Monitoring*

```
1 Query::from(GPS)
2 .filter(edwithin_tgeo_geo(lon, lat,ts, INPolygons, 20) ==
  ↪ 1)
3 .window(TumblingWindow::of(EventTime(ts),Seconds(10)))
4 .sink(PrintSinkDescriptor::create());
```

Query 2 performs brake-system monitoring. It first filters the SNCB stream to include only points that are not in a maintenance area (INPolygons), by applying the function `eintersects_tgeo_geo` (Line 2). It then applies a ten-second sliding window with a ten-millisecond step over the event timestamp (Line 3). Within each window, the query computes the pressure variation for the automatic brake pipe pressure (FA) and the fictitious brake cylinder pressure (FF) using the `variation` operator (Line 4), with results `varFA` and `varFF`. `varFA` and `varFF` denote the statistical variance of the respective pressure attribute within the window, computed using a custom VAR aggregation. For a pressure signal X within one window, measures how much X fluctuates around its mean. Therefore, this variance is measured in bar^2 and captures how much the pressure fluctuates around its mean. The query retains only those data points where `varFA` exceeds 0.6 bar^2, while `varFF` remains at or below 0.5 bar^2 (Line 5), indicating a potential brake issue.

Query 2. *Brake System Monitoring*

```
1 Query::from(GPS)
2 .filter(eintersects_tgeo_geo(lon, lat, ts, INPolygons) ==
  ↪ 0)
3 .window(SlidingWindow::of(EventTime(ts), Seconds(10),
  ↪ Milliseconds(10)))
4 .apply(variation(FA), variation(FF))
5 .filter(varFA > 0.6 && varFF<= 0.5);
```

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

Query 3. *Trajectory Creation*

```
1 Query::from(GPS)
2 .window(SlidingWindow::of(EventTime(ts), Seconds(10),
  ↪ Milliseconds(10)))
3 .apply(temporal_sequence(lon, lat, ts));
```

Query 4 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 ten-second sliding window with a ten-millisecond step over the event timestamp (Line 3). Within each window, the query computes the trajectory using attributes `longitude`, `latitude`, and `timestamp`, and outputs the train position as a trajectory (Line 4).

Query 4. *Trajectory Creation in a Restricted Space*

```
1 Query::from(GPS)
2 .filter(tgeo_at_stbox(lon, lat, ts, stbox xt(((4.3,50),(4.5
  ↪  ,50.6)),[2024-10-24,2024-11-26])) == 1)
3 .window(SlidingWindow::of(EventTime(ts), Seconds(10),
  ↪ Milliseconds(10)))
4 .apply(temporal_sequence(lon, lat, ts));
```

Query 5 performs trajectory creation and emits high-speed alerts in a geofenced area. It first keeps only records whose `longitude`, `latitude`, and `timestamp` are close to 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 (Line 5). It then filters where avg speed exceeds 50 m/s or min speed exceeds 20 m/s (Line 6). It then outputs the train position as a trajectory, average, and minimum speed.

Query 5. *Trajectory Creation and High-Speed Alert*

```
1 Query::from(GPS)
2 .filter(edwithin_tgeo_geo(lon, lat, ts,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(device_id)
4 .window(SlidingWindow::of(EventTime(ts), Seconds(45),
  ↪ Seconds(5)))
5 .apply(temporal_sequence(lon, lat, ts), avg(gps_speed), min
  ↪ (gps_speed))
6 .filter((avg_speed> 50) || (min_speed> 20));
```

Query 6 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 (Line 3). Here, `GPS2` denotes a second logical view of the GPS stream. For each pair, it computes the nearest approach distance between the two points (Line 4), retains only pairs where the left-side latitude is positive (`lat` > 0.0, Line 5), and emits the resulting `mindist` values.

Query 6. *Positional Divergence for a Device*

```
1 Query::from(GPS)
2   .joinWith(GPS2, device_id == device_id2)
3   .window(TumblingWindow::of(EventTime(ts), Seconds(10)))
4   .apply(nearest_approach_distance(lon, lat, ts, lon2, lat2,
  ↪  ts2))
5   .filter(lat > 0.0)
```

Query 7 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_dist` (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 7. *Global Closest Device Pairs (Top-k Closest Pairs)*

```
1 Query::from(GPS)
2 .joinWith(GPS2, device_id < device_id2)
3 .window(TumblingWindow::of(EventTime(ts), Seconds(10)))
4 .apply(nearest_approach_distance(lon, lat, ts, lon2, lat2,
  ↪ ts2))
5 .apply(topK(mindist, 10));
```

Query 8 performs trajectory smoothing with an extended Kalman filter. In each ten-second 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 outlier dropping (Line 3). The cleaned trajectory is then emitted.

Query 8. *Trajectory Denoising with Extended Kalman Filter*

```
1 Query::from(GPS)
2 .window(TumblingWindow::of(EventTime(ts), Seconds(10)))
3 .apply(temporal_ext_kalman_filter(temporal_sequence(lon,
  ↪ lat, ts), 3.0, 0.01, 1.0, false));
```

Query 9 performs a windowed *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 nearest neighbors (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* nearest neighbors according to a pre-computed distance attribute.

Query 9. *Windowed Per-Device kNN Join*

```
1 Query::from(GPS)
2 .joinWith(GPS2, device_id != device_id2)
3 .window(TumblingWindow::of(EventTime(ts), Seconds(10)))
4 .apply(nearest_approach_distance(lon, lat, ts, lon2, lat2,
  ↪ ts2))
5 .groupBy(device_id)
6 .apply(knn_agg(mindist, device_id2, 3));
```

5 Experiments

We run nine queries from Sect. 4.1. We evaluate edge deployability by testing our implementation under strict resource caps and comparing it with related work (see Sect. 5.1). We analyze performance through a mobility computation breakdown (see Sect. 5.2), parameter sensitivity experiments (see Sect. 5.3) and network volatility stress evaluations (see Sect. 5.4). In addition, we verify the correctness of MobilityNebula query by comparing the result with the existing fixed PLC logic used by SNCB (see Sect. 5.5).

5.1 Edge-Constrained Evaluation

To validate the edge deployability of MobilityNebula, we test our implementation under strict container resource caps in Docker.

We implement the nine queries (see Sect. 4.1 and Table 3 for supported queries per system) in TStream [34],[4] GeoEKuiper [22], [5] and MobyDick [16][6] with the same TCP stream input workload of 20k events/s. During these tests, we measure CPU usage, memory usage, throughput, and latency. Each query runs for 30 seconds. We tested with multiple configurations, in a single-node core under three memory configurations: 512 MB, 2 GB and 8 GB of RAM. We utilized 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 evaluated the conditions under which the queries were either terminated or timed out without producing any sink output.

Table 3 maps each query (Q1–Q9) to the systems capable of expressing it without altering semantics or requiring altering its core source code. For GeoEKuiper, Queries 6, 7, and 9 required code adaptations to construct a trajectory-like structure. We chose GeoEKuiper for these adaptations since it is a baseline, 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.

[4]https://github.com/marianaGarcez/SpatialFlink
[5]https://github.com/marianaGarcez/GeoEkuiper
[6]https://github.com/marianaGarcez/mobydick

Table 3: Implementation of Queries Q1–Q9 Across Systems.

Query	MobilityNebula	MobyDick	TStream	GeoEKuiper
Q1	✓	✓	✓	✓
Q2	✓	✓	✓	✓
Q3	✓	✓	✓	✓
Q4	✓	✓	✓	✓
Q5	✓	✓	✓	✓
Q6	✓	–	–	✓
Q7	✓	–	✓	✓
Q8	✓	–	–	–
Q9	✓	–	✓	✓

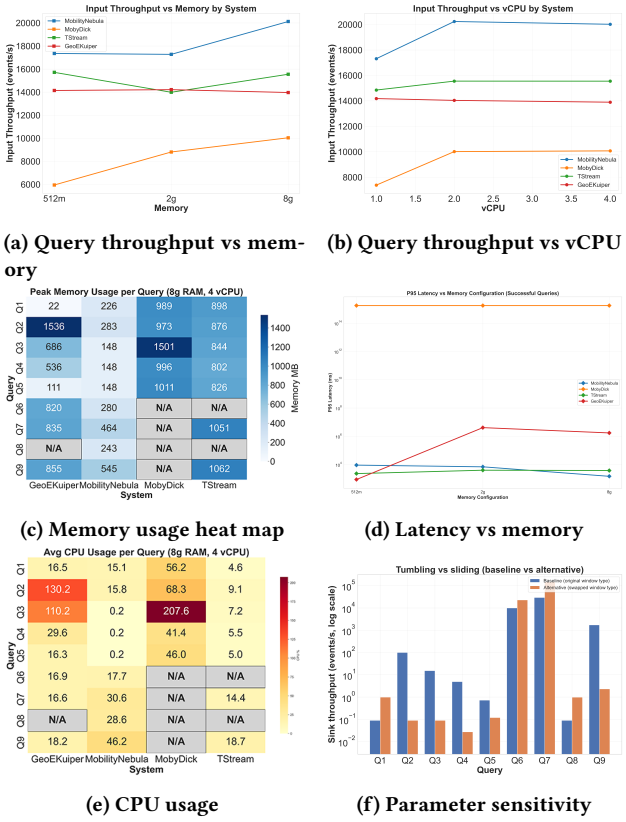
The runner registers `sncb_stream` and the query rule via the CLI, connects to a local MQTT broker for input and output, and otherwise uses the engine defaults.

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 [24]. 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. Figure 4 shows the results as the mean of five runs.[7]

Figure 4a 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-closure operations under load. TStream sustains near-target ingestion on its supported queries once memory is at least 2 GB, maintaining 20k events/s across Q1–Q5 and the heavier Q7/Q9 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 Q1–Q4 and Q7/Q9), indicating that its typical working set is closer to 0.7–1.1 GB on these workloads (see Fig. 4c).

Figures 4c and 4e 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, Q9). CPU utilization similarly stays within limits, with the most compute-intensive operators occurring in heavy workloads, such as EKF smoothing Q8 and *k*NN Q7–Q9). In contrast, the baselines show increased or more volatile resource footprints on trajectory-based workloads. GeoEKuiper peaks at 1,536 MB in Q2 alongside elevated CPU (130.2%), and MobyDick peaks at 1,501 MB with very high CPU

[7]Full graphs at https://github.com/MobilityDB/MobilityNebula/wiki/Edge-Constrained-Evaluation-Results



(a) Query throughput vs memory **(b) Query throughput vs vCPU**

(c) Memory usage heat map **(d) Latency vs memory**

(e) CPU usage **(f) Parameter sensitivity**

Figure 4: Edge-constrained evaluation results

in Q3 (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 4d 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 Q3/Q5 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 closure 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-warmup ingestion in extreme cases (see Fig. 4a). 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. MobilityNebula achieves the highest success rate, 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%.

5.2 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 Q3 and Q5: 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 wall 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 kernel rather than in the surrounding windowing logic.[8]

5.3 Parameter Sensitivity

We measure parameter sensitivity across the nine queries by varying the window size and slide interval around the baselines in Sect. 4.1. 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 (Q1, Q6–Q9), we added a sliding variant at the baseline size with a 1 s slide, and for sliding queries (Q2–Q5), we added a tumbling variant at the baseline size. For tumbling queries (Q1, Q6–Q9), we test sizes 5/10/20 s and for sliding queries (Q2–Q5), we sweep window sizes (Q2: 5/7/10/20 s; Q3/Q4: 5/10/20 s; Q5: 15/45/90 s) and slide intervals (Q2–Q4: 10 ms/100 ms/1 s; Q5: 1/5/10 s).

Sink throughput is primarily determined by emission frequency and selectivity, rather than the input rate. For instance,

[8] Full per-query percentage breakdowns are provided in the artifacts

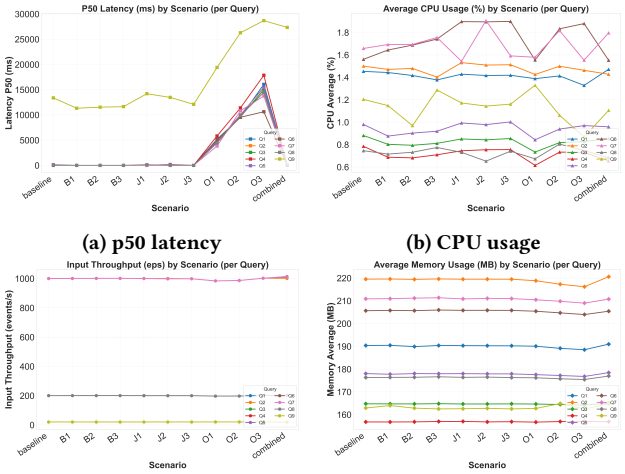
Q1 produces sparse outputs, and its sink rate decreases as the window size increases (from 0.21 events/s at 5 s to 0.06 events/s at 20 s). Q8 exhibits a similar decline (0.21 to 0.06 events/s). In contrast, join tumbling queries generate numerous results per window, sustaining high sink rates (e.g., Q6 at 10k events/s and Q7 at 29k events/s at 10 s). Among tumbling queries, Q6 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. 4f) reduces sink throughput to one emission per window and decreases resource consumption accordingly. For Q4, 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 Q2, reducing the slide from 1s to 10ms increases sink throughput (0.98 to 100 events/s) while maintaining input throughput near 20k events/s (see Fig. 4f). Memory usage rises due to the greater number of concurrent overlapping windows (248 to 435 MB). For Q3 we see a similar behavior, at 10ms slide intervals, trajectory construction dominates, reducing input throughput to 4k events/s and increasing peak memory to 2.7 GB. At 1s slides, input throughput recovers to 20k events/s with lower memory usage (611 MB on average). Q4 and Q5, executed at 200 events/s, follow the same trend: decreasing the slide interval increases sink rates (Q4: from 0.06 events/s at 1 s to 4.89 events/s at 10 ms; Q5: 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. Q6 and Q7 experience substantial input-throughput degradation under 1s slides, from 4k and 11k events/s, and the per-device kNN join also degrades to 11k events/s, while increasing CPU utilization to 66.4%.

5.4 Network Volatility Stress Evaluations

We evaluate the nine queries 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: Q1/Q2/Q6/Q7 at 1000 events/s, Q3/Q4/Q5/Q8 at 200 events/s, and Q9 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 ±20/±80/±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. 5b and 5d). As shown in



(a) p50 latency (b) CPU usage

(c) Throughput (d) Average memory usage

Figure 5: Network volatility stress evaluation results

Fig. 5c, 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 5a 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 closure. 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 9 exhibits the most pronounced lag among all queries. Its architecture requires substantial resources and is highly dependent on window closure.

5.5 PLC Logic Verification

We validated the detection logic of the braking system, as shown in Query 2, in a traditional SQL 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 PLCs 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 filter retains 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 5 and Code 6, 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 sensor readings through both the PostgreSQL query and our streaming query. This two-step approach, from the SQL prototype to the

streaming query, enables MobilityNebula to fully replace fixed-controller diagnostics while preserving the exact behavior of legacy PLC formulas.

Code 5. *Brake Fault Detection Results for PostgreSQL*

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

Code 6. *Brake Fault Detection Results for MobilityNebula*

```
wStart_ms,wEnd_ms,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 stream query outputs match, we aligned the start and end timestamps of each method, taking into account how the windows are constructed and updated. MobilityNebula produces 10-second windows that are updated with the event time. The SQL setup utilized a 10-second window based on the last seen timestamp, which often results in 7-second windows. Cropping the SNCB dataset to the MobilityNebula streaming window, from `1727856300` to `1727866300` ms (see Code 6), produces a maximum FA of 3.813 bar, a minimum of 3.596 bar, an average of 3.7439 bar, and a variance of 0.217 bar^2. 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^2. The SQL window outputs identical FA variances over that subset from `1727866234` to `1727866251` (see Code 5). This correspondence confirms that, for the overlapping interval, the declarative SQL pipeline and the MobilityNebula streaming query compute identical variances, so the seven-second look-back used in SQL does not introduce bias in the fault-detection logic.

6 Related Work

In this section, we survey prior work on continuous data processing and spatiotemporal data management, focusing on three domains: stream processing systems, spatiotemporal database, and spatiotemporal stream processing systems. MobilityNebula fills the intersection of low-latency stream processing and spatiotemporal mobility analytics across the edge–to–cloud continuum.

6.1 Stream Processing Systems

The authors of the survey [14] review several stream processing systems that support continuous data handling. Early stream engines such as STREAM [17] and Aurora [1] introduced continuous SQL queries and sliding windows within database systems, enabling real-time filtering and aggregation. Building on the MapReduce paradigm [9], systems such as Apache Flink [6], Spark Streaming [38], Google Dataflow [27], and Storm[9] scaled-out stream processing to large clusters. Apache Kafka Streams [26] decouples producers and consumers, creating durable, replayable queues that form the backbone of many modern pipelines, as exemplified by Uber's real-time architecture [15]. In addition, other approaches include distributed SQL streaming database systems such as RisingWave [36]. More recently, with the increasing deployment of IoT sensors and the emergence of heterogeneous IoT network architectures, systems such as NebulaStream [8, 18, 39] have introduced mechanisms to cope with execution across the edge–to–cloud continuum, including edge-node execution across different node architectures, heterogeneous hardware support, and incremental stream query placement. Current stream processing systems, however, lack spatiotemporal processing capabilities at the edge to meet the growing amount of data gathered by sensors mounted on moving objects.

[9] http://hortonworks.com/hadoop/storm/

6.2 Spatiotemporal Systems

Database extensions such as SECONDO [19] and Parallel SECONDO [28] introduced moving object support into a generic Database Management System, offloading parallel execution to Berkeley DB and Hadoop. Hermes [31] extended Oracle with trajectory models inside the object-relational model. HadoopTrajectory [3] and ST-Hadoop [2] augment SpatialHadoop with grid and 3-D R-tree indexes that prune data before distribution. TrajSpark [41] mitigates repartitioning costs through dynamic partitions and time-decay scoring. MobilityDB [43] extends PostGIS with spatiotemporal types (*tinstant*, *tsequence*), GiST/SP-GiST indexes, and functions for trajectory management. The work in [4] presents a distributed version of MobilityDB. The above systems target historical analytics and lack mechanisms for low-latency stream processing.

6.3 Spatiotemporal Streaming Systems

Uber's infrastructure [15] integrates Kafka [26], Flink [6], Pinot [23], and Presto [33] to manage data ingestion and provide geo-aware dashboards, but it does not provide trajectory management. Sedona [37] is an extension to Spark SQL with spatial RDDs and spatial indexes designed for batch analytics scenarios. GeoFlink [35] adds spatial types and grid index to Flink, enabling continuous range queries, k-nearest-neighbor searches, and spatial joins over static geometries. TStream [34] builds on GeoFlink by introducing window-bounded grid indexes and trajectory operators allowing TStream to handle real-time path queries over moving objects. However, TStream is optimized for JVM-based Flink clusters rather than in resource-constrained edge devices or operator placement across the edge–to–cloud continuum. While architectures like MobiSpaces [10] have proposed designs for mobility stream processing across edge and cloud, such systems have not yet been fully integrated or realized in practice—to the best of our knowledge. MobyDick framework [16] provides an abstract model and implements discrete spatiotemporal data types as a Flink library. Tornado [29] augments Storm with grid-based structures and FAST keyword indexing to support spatial–keyword stream processing with adaptive load balancing. GeoEKuiper [22] provides spatial SQL operators, as range, join, kNN, and predicate queries at the edge over a cloud-cooperated IoT stream processor. However, it remains point-based as it does not provide a trajectory type system, trajectory construction, or mobility-specific operators that reason about motion over windows. Consequently, it cannot express window-bounded trajectory queries on constrained hardware.

7 Conclusion

MobilityNebula is a stream processing framework for trajectory-based mobility analytics on edge devices. We integrate MEOS data types and operations into NebulaStream to enable low-latency

spatiotemporal analytics on resource constrained edge devices. We issue nine queries, from identifying trains entering high-risk zones to smoothing trajectories. The analysis indicates that real-time spatiotemporal processing can be achieved in resource limited IoT scenarios. We define and implement a window-aware trajectory type system, a lift–combine–lower aggregation pattern for trajectory construction, and a lightweight operator integration that enables MOD semantics to run on constrained devices within a distributed stream processor.

In earlier work, we have demonstrated point-based queries in distributed edge-cloud and edge-fog deployments [11, 12]. This paper complements these works by focusing on trajectory processing in resource constrained devices. In our use case, a single edge node per train is sufficient. This design is intentional as safety-critical decisions, such as brakes monitoring, require in situ processing with minimal latency and must remain operational during intermittent connectivity, motivating our edge-first design. At the same time, MobilityNebula is implemented as an extension of the NebulaStream distributed runtime, and the presented operators and data types are compatible with NebulaStream's query planning and placement mechanisms. Exploring distributed placement optimizations specific to trajectory operators and broadening the set of application domains are promising directions for future work. Quantifying transport-level disconnection and late-reconnection behavior, such as alarm continuity, buffer overflow rate, and recovery time is part of ongoing work. We plan to continue extending the system with features, such as in-memory spatial indexes, as well as support for speeding up streaming workloads and integrating GPU acceleration.

Acknowledgments

This work was partially funded by the EU's Horizon Europe research program grant No. 101070279 MobiSpaces.

Artifacts

Source code and configurations are available at: https://github.com/MobilityDB/MobilityNebula,
https://github.com/marianaGarcez/SpatialFlink,
https://github.com/marianaGarcez/GeoEkuiper,
https://github.com/marianaGarcez/mobydick,
Full figures that are presented in this paper and extra content are available at: https://github.com/MobilityDB/MobilityNebula/wiki.

References

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. 2003. Aurora: a new model and architecture for data stream management. *The VLDB Journal* (2003), 120–139.

[2] L. Alarabi, M. Mokbel, and M. Musleh. 2018. ST-Hadoop: a MapReduce framework for spatio-temporal data. *GeoInformatica* (2018), 785–813.

[3] M. Bakli, M. Sakr, and T. Soliman. 2018. A spatiotemporal algebra in Hadoop for moving objects. *Geo-spatial information science* 21, 2 (2018), 102–114.

[4] M. Bakli, M. Sakr, and E. Zimányi. 2020. Distributed spatiotemporal trajectory query processing in SQL. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*. 87–98.

[5] T. Brandt and M. Grawunder. 2018. GeoStreams: A survey. *Comput. Surveys* 51, 3 (2018), 1–37.

[6] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* (2015), 28–33.

[7] X. Chatziliadis, E. Zacharatou, A. Eracar, S. Zeuch, and V. Markl. 2024. Efficient Placement of Decomposable Aggregation Functions for Stream Processing over Large Geo-Distributed Topologies. *VLDB Endowment* 17, 6 (2024), 1501–1514.

[8] A. Chaudhary, K. Beedkar, J. Karimov, F. Lang, S. Zeuch, and V. Markl. 2025. Incremental Stream Query Placement in Massively Distributed and Volatile Infrastructures. In *Proceedings of the 41st IEEE International Conference on Data Engineering*. IEEE, 376–390.

[9] J. Dean and S. Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* (2008), 107–113.

[10] C. Doulkeridis, G. Santipantakis, N. Koutroumanis, G. Makridis, V. Koukos, G. Theodoropoulos, Yannis Theodoridis, Dimosthenis Kyriazis, Pavlos Kranas, D. Burgos, R. Jimenez-Peris, M. 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. 2023. MobiSpaces: An Architecture for Energy-Efficient Data Spaces for Mobility Data. In *IEEE International Conference on Big Data*. 1487–1494.

[11] M. Duarte, D. Nugroho, G. Tod, E. Bevernage, P. Moelans, E. Saerens, E. Zimányi, M. Sakr, and S. Zeuch. 2025. Mobility Data Stream Processing Beyond the Cloud. In *Proceedings of the 33rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 682–685.

[12] M. Duarte, D. Nugroho, G. Tod, E. Bevernage, P. Moelans, E. Tas, E. Zimányi, M. Sakr, S. Zeuch, and V. Markl. 2025. Mobility Stream Processing on NebulaStream and MEOS. In *Proceedings of the 2025 ACM International Conference on Management of Data*. ACM, 79–82.

[13] L. Forlizzi, R. Güting, E. Nardelli, and M. Schneider. 2000. A Data Model and Data Structures for Moving Objects Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 319–330.

[14] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos. 2023. A survey on the evolution of stream processing systems. *The VLDB Journal* 33, 2 (2023), 507–541.

[15] Y. Fu and C. Soman. 2021. Real-time Data Infrastructure at Uber. In *Proceedings of the ACM International Conference on Management of Data*. ACM, 2503–2516.

[16] Z. Galić, E. Meskovic, and D. Osmanovic. 2017. Distributed processing of big mobility data as spatio-temporal data streams. *GeoInformatica* (2017), 263–291.

[17] M. Garofalakis, J. Gehrke, and R. Rastogi. 2016. *Data Stream Management: Processing High-Speed Data Streams*. Springer.

[18] P. Grulich, A. Lepping, D. Nugroho, V. Pandey, B. Del Monte, S. Zeuch, and V. Markl. 2024. Query Compilation Without Regrets. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.

[19] R. Güting, V. Almeida, D. Ansorge, T. Behr, Z. Ding, T. Hose, F. Hoffmann, M. Spiekermann, and U. Telle. 2005. SECONDO: an extensible DBMS platform for research prototyping and teaching. In *Proceedings of the 21st IEEE International Conference on Data Engineering*. IEEE, 1115–1116.

[20] R. Güting, M. Böhlen, M. Erwig, C. Jensen, N. Lorentzos, M. Schneider, and M. Vazirgiannis. 2000. A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems* (2000), 1–42.

[21] R. Güting and M. Schneider. 2005. *Moving Objects Databases*. Elsevier Science.

[22] W. Huang and X. Deng. 2024. GeoEkuiper: A Cloud-Cooperated Geospatial Edge Stream Processing Engine for Resource-Constrained IoT Devices With Higher Throughput. *IEEE Internet of Things Journal* 11, 18 (2024), 30094–30113.

[23] J. Im, K. Gopalakrishna, S. Subramaniam, M. Shrivastava, A. Tumbde, X. Jiang, J. Dai, S. Lee, N. Pawar, J. Li, and R. Aringunram. 2018. Pinot: Realtime OLAP for 530 Million Users. In *Proceedings of the 2018 ACM International Conference on Management of Data*. ACM, 583–594.

[24] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE 18)*. IEEE, 1507–1518.

[25] A. Kozar, B. Monte, S. Zeuch, and V. Markl. 2024. Fault Tolerance Placement in the Internet of Things. *Proceedings of the ACM on Management of Data* 2, 3, Article 138 (May 2024), 29 pages.

[26] J. Kreps, N .Narkhede, and J. Rao. 2011. Kafka: A Distributed Messaging System for Log Processing. In *Proceedings of the 6th International Workshop on Networking Meets Databases*. 1–7.

[27] S. Krishnan and J. Gonzalez. 2015. In *Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects*. Apress, Chapter Google Cloud Dataflow.

[28] J. Lu and R. Güting. 2013. Parallel SECONDO: Practical and efficient mobility data processing in the cloud. In *Proceedings of the 2013 IEEE International Conference on Big Data*. IEEE, 107–125.

[29] A. Mahmood, A. Daghistani, A. Aly, M. Tang, S. Basalamah, S. Prabhakar, and W. Aref. 2018. Adaptive processing of spatial-keyword data over a distributed streaming cluster. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*.

[30] S. Nittel. 2015. Real-time sensor data streams. *SIGSPATIAL Special* (2015), 22–28.

[31] N. Pelekis, Y. Theodoridis, S. Vosinakis, and T. Panayiotopoulos. 2006. Hermes: A Framework for Location-Based Data Management. In *Proceedings of the 22nd International Conference on Extending Database Technology*. Springer, 1130–1134.

[32] M. Sakr, A. Vaisman, and E. Zimányi. 2025. *Mobility Data Science: From Data to Insights*. Springer.

[33] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. 2019. Presto: SQL on Everything. In *Proceedings of the 35th IEEE International Conference on Data Engineering*. IEEE, 1802–1813.

[34] A. Shaikh, H. Kitagawa, A. Matono, and Kyoung-S. Kim. 2024. A Distributed and Scalable Framework for Low-Latency Continuous Trajectory Stream Processing. *IEEE Access* (2024), 159426–159444.

[35] S. Shaikh, H. Kitagawa, A. Matono, K. Mariam, and K. Kim. 2022. GeoFlink: An Efficient and Scalable Spatial Data Stream Management System. *IEEE Access* (2022), 24909–24935.

[36] Y. Wang and Z. Liu. 2022. A sneak peek at RisingWave: a cloud-native streaming database. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*. ACM, 190–193.

[37] J. Yu, Z. Zhang, and M. Sarwat. 2019. Spatial data management in Apache Spark: the GeoSpark perspective and beyond. *GeoInformatica* (2019), 37–78.

[38] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. 2012. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 2012 USENIX Conference on Hot Topics in Cloud Computing*. USENIX Association, 10 pages.

[39] S. Zeuch, A. Chaudhary, B. Monte, H. Gavriilidis, D. Giouroukis, P. Grulich, S. Bress, J. Traub, and V. Markl. 2020. The NebulaStream Platform: Data and Application Management for the Internet of Things. In *Proceedings of the 10th Conference on Innovative Data Systems Research*. www.cidrdb.org.

[40] S. Zeuch, E. Zacharatou, S. Zhang, X. Chatziliadis, A. Chaudhary, B. Monte, D. Giouroukis, P. Grulich, A. Ziehn, and V. Mark. 2020. NebulaStream: Complex Analytics Beyond the Cloud. *Open Journal Internet Things* (2020), 66–81.

[41] Z. Zhang, C. Jin, J. Mao, X. Yang, and A. Zhou. 2017. TrajSpark: A Scalable and Efficient In-Memory Management System for Big Trajectory Data. In *Web and Big Data*. Springer, 11–26.

[42] E. Zimányi, M. Duarte, and V. Diví. 2024. MEOS: An Open Source Library for Mobility Data Management. In *Proceedings of the 27th International Conference on Extending Database Technology*. OpenProceedings.org, 810–813.

[43] E. Zimányi, M. Sakr, and A. Lesuisse. 2020. MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS. *ACM Transactions on Database Systems* 45, 4 (2020), 1–42.