MobilityDuck: Mobility Data Management with DuckDB

Nhu Ngoc Hoang[1,*,†], Ngoc Hoa Pham[1,†], Viet Phuong Hoang[1,†] and Esteban Zimányi[1,†]

[1]Université Libre de Bruxelles, Brussels, Belgium

Abstract

Efficient management and analysis of spatiotemporal data are critical for modern mobility applications, yet existing moving object database (MOD) systems oftentimes remain overly complex or heavyweight for practical analytics workflows. While MobilityDB, a notable MOD extension, can expand PostgreSQL with advanced spatiotemporal capabilities, its reliance on a traditional database architecture introduces significant overhead and complexity. In this paper, we present MobilityDuck, a DuckDB extension that integrates the MEOS library to provide support for spatiotemporal and other temporal data types in DuckDB. MobilityDuck leverages DuckDB's lightweight, columnar, in-memory executable properties to deliver efficient analytics. To the best of our knowledge, no existing in-memory or embedded analytical system offers native spatiotemporal types and continuous trajectory operators as MobilityDuck does. We evaluate MobilityDuck using the BerlinMOD-Hanoi benchmark dataset and compare its performance to MobilityDB. Our results show that MobilityDuck preserves the expressiveness of spatiotemporal queries while benefiting from DuckDB's in-memory, columnar architecture.

Keywords

Spatiotemporal, Trajectories, Mobility, DuckDB, BerlinMOD, MEOS

1. Introduction

The rapid growth of spatiotemporal data has created new opportunities for mobility analytics, where discovering patterns and trends in object trajectories plays a central role in applications such as urban planning, intelligent transportation systems, and mobility-as-a-service platforms.

Despite an extensive body of research in moving object databases (MODs), and the emergence of systems like MobilityDB, mainstream adoption is still limited by architectural complexity, setup overhead, and integration challenges in modern analytics pipelines. However, MobilityDB inherits PostgreSQL's complexity, which limits its efficiency for lightweight querying, embedded deployment, and exploratory data science workflows where ease of use and speed of integration are paramount. At the same time, DuckDB has rapidly emerged as a modern analytical database, designed to be lightweight, embeddable, and highly optimized for in-memory, columnar query execution. Nevertheless, DuckDB currently lacks first-class support for spatiotemporal data types and operators.

This paper introduces MobilityDuck, the first DuckDB extension to support spatiotemporal and temporal data types. By combining DuckDB's in-memory, vectorized execution model with MEOS's mature spatiotemporal algebra, MobilityDuck brings the expressiveness of moving object databases into a lightweight analytical engine. We also adapt the BerlinMOD benchmark to the Hanoi urban environment, producing BerlinMOD-Hanoi, a reproducible dataset and query workload for diverse mobility analytics. Our experimental evaluation shows that MobilityDuck maintains query expressiveness while delivering significant performance improvements on most benchmark tasks. Beyond engineering efforts, MobilityDuck addresses important research questions at the intersection of spatiotemporal data management and high-performance in-memory analytics, demonstrating that mature and semantically rich temporal data models can be incorporated into modern columnar, vectorized DBMSs with minimal overhead. We further present integration patterns and experimental analysis, creating a blueprint for research libraries to enter production-strength analytics platforms.

Published in the Proceedings of the Workshops of the EDBT/ICDT 2026 Joint Conference (March 24-27, 2026), Tampere, Finland

*Corresponding author.

†These authors contributed equally.

nhu.hoang@ulb.be (N. N. Hoang); ngoc.pham@ulb.be (N. H. Pham); viet.hoang@ulb.be (V. P. Hoang); esteban.zimanyi@ulb.be (E. Zimányi)

2. Background and Related Work

Research on spatiotemporal data management has a long history in both the database and GIS communities with early efforts extending spatial databases with temporal versioning or enriching temporal databases with spatial attributes [1, 2]. A comprehensive review of these early models can be found in the literature [3].

Among the many research prototypes, MobilityDB [4] has emerged as the most complete open-source implementation of a moving object database [5]. It extends PostgreSQL and PostGIS with *temporal types* and *spatiotemporal operators*, building on the MEOS (Mobility Engine Open Source) library. It supports moving points (e.g., vehicle trajectories), temporal spans, and temporal aggregates. MobilityDB has become a reference implementation for managing mobility data, but inherits PostgreSQL's overhead in query execution and storage management. However, its performance remains limited by PostgreSQL's general-purpose query engine and storage layer.

Motivated by the need for faster analytical processing and simpler deployment, several systems have explored in-memory and memory-efficient architectures for spatiotemporal data. S4STRD presents a scalable in-memory storage system for real-time trajectory data, keeping recent updates in RAM and using NoSQL backends for persistence [6]. SharkDB [7] is an in-memory, column-oriented trajectory storage system that partitions trajectories into time-based frames, allowing efficient compression, memory throughput, and parallel processing across cores. In a complementary direction, Richly et al. propose optimized spatiotemporal data structures for in-memory columnar databases, adapting memory layouts, compression, and tiering to trajectory workloads [8]. These works illustrate the feasibility and challenges of in-memory spatiotemporal storage, particularly for reducing I/O overhead, but they emphasize storage and access optimizations rather than full query semantics.

In parallel, DuckDB[1] is an open-source relational database

[1]https://duckdb.org

management system developed by Mark Raasveldt and Hannes Mühleisen [9]. DuckDB is optimized for online analytical processing (OLAP) workloads, making it a suitable system for handling complex querying on large datasets [10]. The key features of DuckDB are as follows:

- **Embeddability**: Unlike traditional database systems with large servers running as stand-alone processes, DuckDB is designed to be an embedded database system that runs completely within another host process.
- **Analytical**: While other embedded systems (e.g., SQLite) focus more on transactional (OLTP) workloads, DuckDB is geared towards efficiently executing analytical SQL queries.
- **High performance**: DuckDB employs a vectorized interpreted execution engine, which optimizes CPU cache usage and allows batch processing of data.

Recent work has extended DuckDB with domain-specific extensions, e.g., for geospatial analytics via DuckDB Spatial Extension[11], or for machine learning via QuackML [12]. However, there is no native support for spatiotemporal types and operators.

Evaluating spatiotemporal DBMSs requires reproducible benchmarks. **BerlinMOD** [13] is the standard benchmark for moving object databases. It defines a synthetic mobility model, a trip generation based on an underlying road-network, and a set of queries measuring performance on indexing, joins, and aggregates. In this work, to adapt BerlinMOD to different geographic contexts, we introduced **BerlinMOD-Hanoi** (see Section 5), which applies the BerlinMOD benchmark using the Hanoi road network from OpenStreetMap data as base map. Rather than proposing new benchmark queries, BerlinMOD-Hanoi serves as a case-study dataset that enables empirical evaluation of spatiotemporal analytics in a non-European urban context.

Taken together, existing work provides mature spatiotemporal data models and efficient storage techniques, but there is limited empirical understanding of how such functionality can be embedded into an in-memory, vectorized analytical DBMS. This paper positions MobilityDuck as a systems case study that explores the engineering challenges, design tradeoffs, and performance implications of integrating MEOS-based spatiotemporal analytics into DuckDB.

3. MobilityDuck: Architecture and Implementation

3.1. Design Goals

Our primary goal with **MobilityDuck** is to enable spatiotemporal analytics within DuckDB by reusing the mature functionality of the MEOS library. The design is guided by the following principles:

- **Lightweight integration:** MobilityDuck is implemented as a DuckDB extension, preserving DuckDB's embedded deployment model.
- **Reuse of MEOS:** Instead of reimplementing temporal types and operators, we wrap MEOS natively in C++, ensuring correctness and consistency with MobilityDB.
- **DuckDB compatibility:** All types and functions are exposed as DuckDB user-defined types (UDTs) and functions, allowing seamless integration with DuckDB's SQL engine, storage manager, and vectorized execution model.

3.2. System Architecture

MobilityDuck follows a simple and modular architecture that connects DuckDB with the MEOS library through a thin C++ extension layer. At query time, DuckDB executes SQL statements as usual, while the extension intercepts calls to spatiotemporal functions and forwards them to MEOS.

Conceptually, the system has three main layers:

- **DuckDB core:** provides the SQL parser, planner, storage engine, and vectorized execution framework. MobilityDuck registers its custom types and functions within this engine at load time.
- **MobilityDuck extension layer:** acts as the bridge between DuckDB and MEOS. It defines DuckDB user-defined types and functions (e.g., `tint`, `tfloat`, `span`) based on their corresponding MEOS structures.
- **MEOS library:** provides the underlying temporal and spatial operators and data structures used by MobilityDB.

This design ensures minimal overhead while maintaining full compatibility with existing DuckDB operations.

3.3. Data Types, Functions and Operators

MobilityDuck adopts the same logical type system as MobilityDB in order to preserve the semantics of temporal and spatiotemporal data. Internally, all MEOS types are represented using the native DuckDB type BLOB, with explicit type aliases used to expose them as first-class spatiotemporal types at the SQL level.

In addition, MobilityDuck exposes functionality through three categories of functions, including **Cast functions**, **Scalar functions** and **Operators**. Cast functions implement explicit conversions between logical types in DuckDB. Scalar functions implement MobilityDuck operations following DuckDB's scalar function interface, taking one or more input values and producing a result value that can be used in SQL expressions. Finally, operators are exposed using DuckDB's function mechanism, allowing familiar spatiotemporal predicates (e.g., overlap tests) to be used directly in SQL expressions.

4. Indexing System

To accelerate spatiotemporal range queries, MobilityDuck implements an R-tree index over spatiotemporal bounding boxes (`stbox`). R-trees are specifically designed for multidimensional data and provide efficient spatial access methods for indexing geographic and spatiotemporal information by organizing data using topological containment relations, making them ideal for spatial queries [5]. In contrast, MobilityDB (built on PostgreSQL) supports both R-tree (via GiST) and quad-tree (via SP-GiST) indexes for spatiotemporal data, offering alternative indexing strategies depending on the data and query types (both types of indexes are used for benchmarking MobilityDB in Section 6).

The indexing system integrates with DuckDB's query optimizer to enable efficient spatial query processing. Index scans are registered as specialized operators on `stbox` data. The indexing system seamlessly integrates with MEOS spatial functions, ensuring that:

- Spatiotemporal bounding boxes are correctly extracted from temporal geometries,
- R-tree insertion and search operations use MEOS spatial predicates,
- Index maintenance operations preserve spatial integrity, and
- Query results are consistent with MobilityDB semantics.

4.1. Index Construction

Our implementation supports two distinct scenarios for index construction, each optimized for different use cases in database operations.

4.1.1. Incremental Construction: Index-First Approach

In the first scenario, an index already exists on a table, and new data is being inserted. When new data is inserted into a table that already has an RTree index, the `Append` method handles incremental updates. This method evaluates index expressions on the new data and constructs index entries using the MEOS RTree insertion functionality. The `Construct` method processes data chunks and inserts them into the RTree structure. Once the `stbox` is prepared, the method applies the MEOS library's insert function `rtree_insert` to handle the actual insertion into the RTree data structure.

4.1.2. Bulk Construction: Data-First Approach

The second scenario occurs when creating an index on a table that already contains data, typically through a `CREATE INDEX` statement. This situation requires a different strategy optimized for processing large volumes of existing data. Our implementation follows a three-phase pipeline that leverages parallel processing for efficiency:

- **Phase 1: Data Collection** As DuckDB's execution framework scans the table in parallel across multiple threads, each thread processes its assigned data partition through the `Sink()` method. This method receives chunks of data containing `stbox` values and row identifiers, appending them to thread-local storage.
- **Phase 2: Data Combination** The `Combine()` method consolidates thread-local collections into a single global dataset through thread-safe merging operations. This consolidation is protected by a mutex to ensure data consistency.
- **Phase 3: Index Construction** In this phase, the system constructs the actual index entries from the collected data. For each chunk, the task deserializes `stbox` data, performs SRID normalization, and collects valid entries into arrays. It then calls the `BulkConstruct` method with these arrays, which inserts entries through `rtree_insert`.

4.2. Query Optimization and Index Scan Injection

DuckDB's query optimizer automatically replaces sequential scans with index scans when applicable predicates are detected. To enable this optimization, the index registers a scan matcher for operators between two `stbox` operands. When a spatial filter predicate matches the indexable pattern, the optimizer substitutes the original table scan with an index scan operator. MobilityDuck currently supports pattern matching for the spatial overlap operator (`&&`) between two `stbox` operands. During query optimization, when the optimizer encounters a filter expression containing this operator, the index attempts to bind the operands. If one operand is a constant `stbox` value, the index can perform an efficient bounding-box search using the R-tree structure.

During index scan execution, the scan normalizes the query's spatial reference system (SRID) to ensure geometric consistency. Then, the normalized bounding box queries the R-tree structure using the underlying MEOS R-tree implementation, which returns identifiers of all entries whose bounding boxes overlap with the query region. Finally, the scan operator iterates through these candidate row IDs, retrieving and returning qualifying tuples to the query pipeline.

5. BerlinMOD-Hanoi

BerlinMOD is the de facto benchmark for evaluating spatiotemporal database systems. However, its MobilityDB implementation tailors the road network and mobility model to Brussels, Belgium. To extend its applicability, we created **BerlinMOD-Hanoi**, an adaptation of the benchmark to the urban setting of Hanoi, Vietnam. This allows us to evaluate MobilityDuck on realistic mobility data from a non-European city with different traffic patterns, densities, and cultural mobility habits. BerlinMOD-Hanoi datasets, SQL scripts, and visualization functions are publicly available.[2]

5.1. Dataset Preparation

We followed the BerlinMOD methodology but replaced Brussels's road network with the one from OpenStreetMap (OSM) for Hanoi:

- Extracted the Hanoi road network using `osm2pgsql` and `osm2pgrouting`, configured with BerlinMOD's `mapconfig.xml` to select road types.
- Constructed a routable network topology with pgRouting.
- Applied the BerlinMOD trip generation logic, adjusted with population statistics of Hanoi's administrative regions using a customized SQL script `hanoi_preparedata.sql`.

The generated trips simulate commuting activities, sampled according to home–work distributions derived from administrative region statistics. Each trip is represented as a temporal sequence of positions (`tgeompoint`) with associated time instants, fully compatible with MEOS/MobilityDB types.

[2] https://github.com/MobilityDB/MobilityDB-BerlinMOD-Hanoi

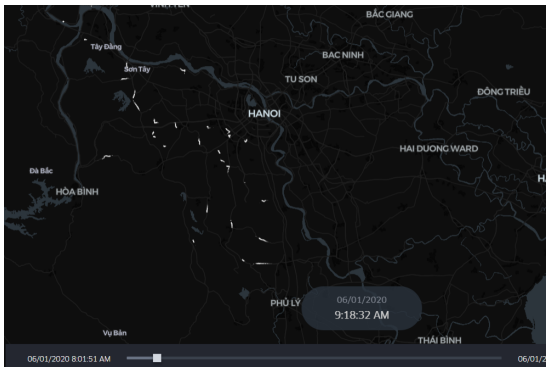


Figure 1: Visualization of animated synthetic trips in Hanoi generated by BerlinMOD-Hanoi.

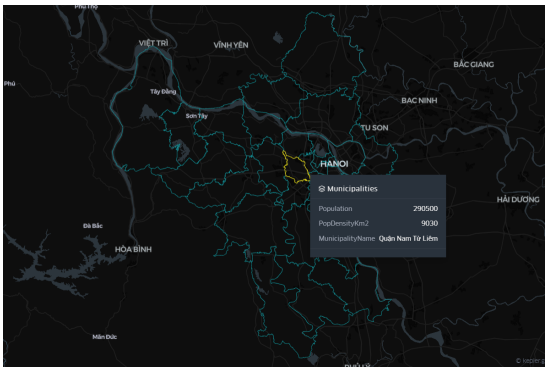


Figure 2: Hanoi administrative regions used in BerlinMOD-Hanoi.

5.2. Dataset Characteristics

BerlinMOD-Hanoi produces scalable datasets through the scale factor (SF) parameter. We also provide **GeoJSON exports** of trips and administrative regions, enabling visualization in **Kepler.gl**.[3] Figure 1 shows an animation of the synthetic trips generated by BerlinMOD-Hanoi, while Figure 2 shows the administrative boundaries used to sample realistic home and work locations.

6. Experimental Evaluation

This section demonstrates the utilization of MobilityDuck in trajectory manipulation using the BerlinMOD-Hanoi dataset. Initial data exploration is conducted by integrating MobilityDuck with DuckDB's Python API and traditional Python libraries for visualization. Additionally, MobilityDuck (DuckDB) is evaluated against MobilityDB (PostgreSQL) using 17 range queries provided by the BerlinMOD benchmark.

All experiments in this section were conducted on an Oracle virtual machine running Ubuntu 20.04, 4 CPUs, 24GB RAM, 20GB swap memory. The BerlinMOD-Hanoi datasets used were generated at 4 different scale factors (see Section 6.2.1). MobilityDuck was built with DuckDB version 1.3.2. The exploratory steps using DuckDB's Python API were run with Python 3.9, DuckDB Python client 1.3.2. MobilityDB benchmarking was conducted on PostgreSQL 15.13.

6.1. Use Case Demonstration

This section presents a preliminary demonstration to showcase MobilityDuck's capacity in manipulating spatiotemporal data as well as its potential in integrating with external APIs and libraries.

The demonstration utilizes the existing BerlinMOD-Hanoi dataset containing trips, where each row shows the coordinates (longitude and latitude) of a specific vehicle made during a given trip at a specific timestamp. The coordinates and timestamp of a row are used to create a `tgeompoint` value, which is ideal for representing a temporal geometry (which, in this case, is a `POINT`). Then, the `tgeompoint` values are aggregated by vehicle IDs and trip IDs to create `tgeompointSeq` values, which are temporal geometries with the additional sequence subtype to represent the evolution of the geometries over a sequence of time instants. Finally, to facilitate visualization in Python, the `tgeompointSeq` values are turned into trajectories in `GEOMETRY` type using the `trajectory()` function.

The `GEOMETRY` data type, which is prevalent in other spatial database systems such as PostgreSQL (extended with PostGIS), is supported in DuckDB by the Spatial extension.[4] As such, handling the `GEOMETRY` type is beyond the scope of MobilityDuck. The latest version of MobilityDuck includes a preliminary interface with Spatial's `GEOMETRY` and `WKB_BLOB` types to ensure the usability of MEOS functions originally involving geometries. When integrating with Python, the geometries can be loaded using the Shapely library[5] to return a GeoPandas[6] dataframe for further processing and visualizing.

Having loaded the data and conducted the aforementioned data preparation steps, a number of operations are run and their results are captured and visualized:

1. Show the trajectories of all trips (Figure 3)
2. Show the trip(s) that cross the highest number of districts (Figure 4)
3. Show the trips that cross Hai Ba Trung district (Figure 5)
4. Show the total distance traveled per district (Figure 6)
5. Show 6 districts with the highest number of trips crossing them, and show parts of the trips that cross the districts (Figure 7)

The SQL queries and Python script for recording and visualizing results are available as a Jupyter Notebook in the example section of MobilityDuck repository.

6.2. BerlinMOD-Hanoi Benchmarking

6.2.1. Introduction

Performance of MobilityDuck (in DuckDB) and MobilityDB (in PostgreSQL) are compared using 17 range-style queries on the BerlinMOD-Hanoi dataset of 4 scale factors: SF-0.05, SF-0.1, SF-0.15, and SF-0.2. Table 1 summarizes the scale factors.

While MobilityDuck includes an initial R-tree index implementation, at the time of our experiments, the index was in its early stages of development and did not support all spatiotemporal data types required by the benchmark workload.

[3] https://kepler.gl/

[4] https://duckdb.org/docs/stable/core_extensions/spatial/overview.html
[5] https://shapely.readthedocs.io/en/stable/
[6] https://geopandas.org/en/stable/

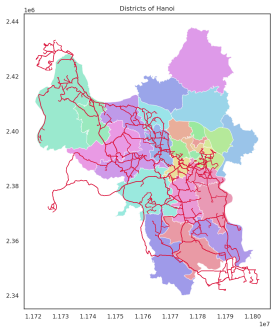


Figure 3: Trajectories of all trips

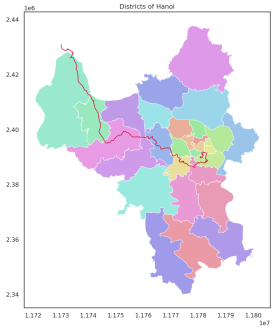


Figure 4: Trajectory of the trip crossing the highest number of districts

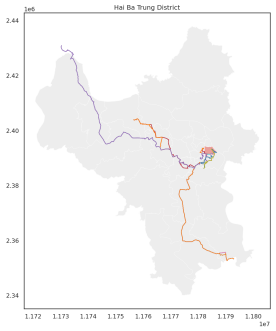


Figure 5: Trips crossing the Hai Ba Trung District

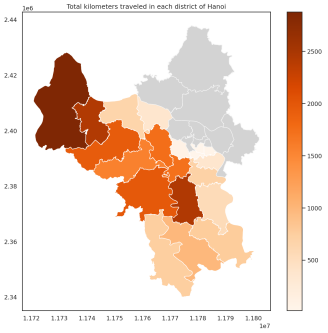


Figure 6: Districts by total distance traveled

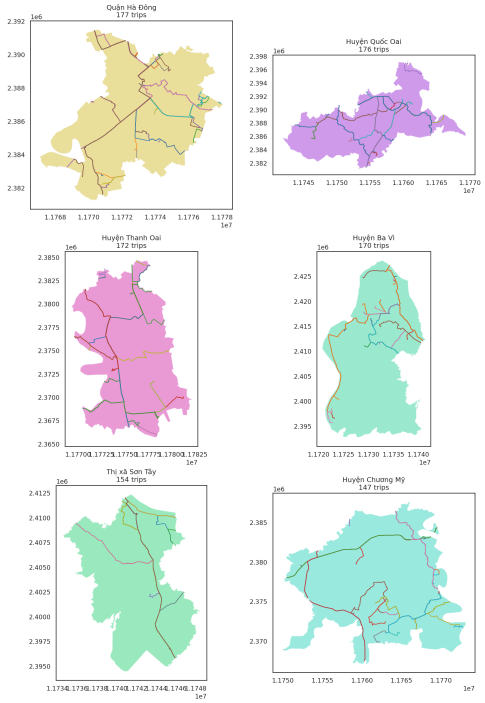


Figure 7: Top 6 districts with highest numbers of trips crossing; trips are clipped to districts

Table 1
BerlinMOD-Hanoi datasets at 4 different scale factors (SF) used for the benchmark

Scale factor	# vehicles	# trips	# raw GPS points
SF-0.05	447	9,491	35,670,635
SF-0.1	632	18,910	72,888,909
SF-0.15	775	26,919	101,557,323
SF-0.2	894	35,319	131,250,325

As a result, we ran all MobilityDuck benchmarks without index supports. In contrast, MobilityDB was evaluated both with R-tree (GiST) and quad-tree (SP-GiST) indexes enabled on relevant columns. The loading phase is excluded from the evaluation, and only the elapsed times of running the queries are used for the subsequent comparisons. We introduce next a selected number of queries. All queries are available in the benchmark section of the accompanying GitHub repository.

Query 5: *What is the minimum distance between places, where a vehicle with a license from* `Licenses1` *and a vehicle with a license from* `Licenses2` *have been?*

```
WITH Temp1(License1, Trajs) AS (
  SELECT l1.License, ST_Collect(list(
      trajectory(t1.Trip)::GEOMETRY))
  FROM Trips t1, Licenses1 l1
  WHERE t1.VehicleId = l1.VehicleId
  GROUP BY l1.License ),
Temp2(License2, Trajs) AS (
  SELECT l2.License, ST_Collect(list(
      trajectory(t2.Trip)::GEOMETRY))
  FROM Trips t2, Licenses2 l2
  WHERE t2.VehicleId = l2.VehicleId
  GROUP BY l2.License )
SELECT License1, License2, ST_Distance(
    t1.Trajs, t2.Trajs) AS MinDist
FROM Temp1 t1, Temp2 t2
ORDER BY License1, License2;
```

This query processes high volumes of trajectory values, which involves casting between `WKB_BLOB` and `GEOMETRY` type, heavily increasing the runtimes. To optimize such bulky operations, we implemented MobilityDuck-native equivalents of DuckDB's spatial functions that take `GEOMETRY` as input, such as `ST_Collect()` and `ST_Distance()`. The modified version of the query is shown below:

```
WITH Temp1(License1, Trajs) AS (
  SELECT l1.License,
    collect_gs(list(trajectory_gs(t1.
      Trip)))
  FROM Trips t1, Licenses1 l1
  WHERE t1.VehicleId = l1.VehicleId
  GROUP BY l1.License ),
Temp2(License2, Trajs) AS (
  SELECT l2.License,
    collect_gs(list(trajectory_gs(t2.
      Trip)))
  FROM Trips t2, Licenses2 l2
  WHERE t2.VehicleId = l2.VehicleId
  GROUP BY l2.License )
SELECT License1, License2,
   distance_gs(t1.Trajs, t2.Trajs) AS
     MinDist
FROM Temp1 t1, Temp2 t2
ORDER BY License1, License2;
```

This version uses `trajectory_gs()`, a version of `trajectory()` that returns a GSERIALIZED object as a BLOB in DuckDB instead of the well-known binary WKB_BLOB format. `collect_gs()`, a modified version of `ST_Collect()`, then takes an array of these geometries and aggregates them into a collection. Finally, `distance_gs()` takes two geometries, still in GSERIALIZED format, and returns the distance between them. This optimized query overcomes the drawback of the current interface with Spatial's GEOMETRY data type by utilizing MEOS' PostGIS-based functions.

Query 7: *What are the license plate numbers of the passenger cars that have reached the points from `Points` first of all passenger cars during the complete observation period?*

```
WITH Timestamps AS (
  SELECT DISTINCT v.License, p.PointId,
      p.Geom,
    MIN(startTimestamp(atValues(t.Trip,
      p.Geom::WKB_BLOB))) AS Instant
  FROM Trips t, Vehicles v, Points1 p
  WHERE t.VehicleId = v.VehicleId AND
    v.VehicleType = 'passenger' AND
    t.Trip && stbox(p.Geom::WKB_BLOB)
      AND
    ST_Intersects(trajectory(t.Trip)::
      GEOMETRY, p.Geom)
  GROUP BY v.License, p.PointId, p.Geom
    )
SELECT t1.License, t1.PointId, t1.Geom,
    t1.Instant
FROM Timestamps t1
WHERE t1.Instant <= ALL (
  SELECT t2.Instant
  FROM Timestamps t2
  WHERE t1.PointId = t2.PointId )
ORDER BY t1.PointId, t1.License;
```

This query first creates the common table expression (CTE) `Timestamps` containing the vehicle information and the timestamps at which a passenger car reaches the points. In the SELECT statement, the query utilizes the `startTimestamp()` and `atValues()` functions. The `atValues()` function takes in a temporal value (in this case, temporal point geometry `tgeompoint` from `Trips`) and a base value (in this case, point geometry from `Points1`) to return the temporal value restricted to the second argument. Essentially, this function takes the full trip and returns only the temporal geometry values at the points from `Points1`. This value is then passed to `startTimestamp()` which, as the name suggests, returns the start timestamp of the temporal value, equivalent to the earliest timestamp at which a trip reaches any point from `Points1`. The third join condition utilizes the *overlaps* (&&) predicate to filter trips that overlap with the point geometry by first creating a spatiotemporal bounding box (`stbox`) around the point.

Query 10: *When and where did the vehicles with license plate numbers from `Licenses1` meet other vehicles (distance < 3 meters) and what are the latter licenses?*

```
WITH Temp AS (
  SELECT l1.License AS License1, t2.
      VehicleId AS Car2Id,
    whenTrue(tDwithin(t1.Trip, t2.Trip,
        3.0)) AS Periods
  FROM Trips t1, Licenses1 l1, Trips t2
      , Vehicles v
  WHERE t1.VehicleId = l1.VehicleId AND
    t2.VehicleId = v.VehicleId AND
    t1.VehicleId <> t2.VehicleId AND
    t2.Trip && expandSpace(t1.trip::
        STBOX, 3.0) )
SELECT License1, Car2Id, Periods
FROM Temp
WHERE Periods IS NOT NULL;
```

This query first creates the CTE `Temp` to store the `Periods` when the vehicles met other vehicles within the spatial constraint. The `tDwithin()` function used in the projection is one of the spatial relationships generalized for temporal geometries. This function first computes, at each instant, whether the distance between the temporal points (`Trip` values from the `Trips` table) is less than or equal to 3. The function yields a `tbool` (temporal boolean) value representing the condition at all time instants of the trips. The resulting `tbool` value is passed to `whenTrue()`, which returns the time when the temporal boolean takes the value `true` as a `tstzspanset` value. This mobility type represents sets of ranges of `timestamptz` values. To filter out trips that are very far from each other, the `expandSpace()` function is used in the fourth join condition. The trip (from `t2`) is first cast into the `stbox` type, representing a spatiotemporal bounding box around the whole trip. `expandSpace()` expands the spatial dimension of this bounding box by 3 units. Only trips that overlap with this expanded box, filtered using the *overlaps* (&&) predicate, are kept.

6.2.2. Results and Discussions

Figure 8 visualizes the runtimes of all 17 queries, across 4 scale factors, and for 3 scenarios: using MobilityDuck on DuckDB (yellow bars), MobilityDB on PostgreSQL with R-tree indexes (GiST, dark blue bars), and MobilityDB with quad-tree indexes (SP-GiST, light blue bars).

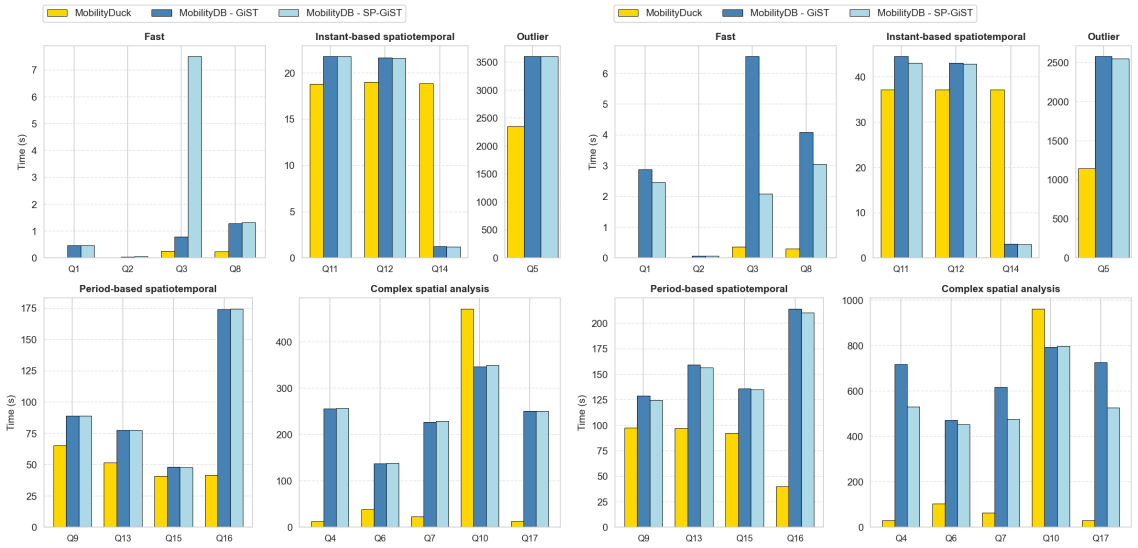
MobilityDuck outperforms MobilityDB both with and without indexes across all scale factors in 13 out of 17 queries (1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 16, 17). For Query 5, MobilityDuck still manages to achieve the best runtimes in all scale factors except SF-0.15. Similarly, for Query 15, MobilityDuck achieves the best runtimes in all scales except SF-0.2 (where it runs approximately 3.7 seconds slower than MobilityDB).

MobilityDuck shows lower performance across all scale factors for Query 10 and Query 14. This can be explained due to the fact that both of these queries extensively call upon table columns with indexes when running with MobilityDB in PostgreSQL.

Overall, in the majority of cases, MobilityDuck without indexes outperforms MobilityDB with indexes. These results demonstrate the effectiveness of integrating spatiotemporal querying directly within DuckDB's analytical engine, rather than relying on external index-based acceleration. The strong performance of MobilityDuck, even in the absence of specialized indexes, suggests an effective utilization of DuckDB's architecture to handle mobility workloads.

6.2.3. Scalability Limits on Commodity Hardware

While the system demonstrated robust performance up to SF-0.2 (corresponding to approximately 20GB of on-disk



(a) Query runtimes at SF-0.05

(b) Query runtimes at SF-0.1

(c) Query runtimes at SF-0.15

(d) Query runtimes at SF-0.2

Figure 8: Runtimes (in seconds) for the BerlinMOD-Hanoi benchmark queries at SF-0.05, SF-0.1, SF-0.15, and SF-0.2

data), experiments at SF 0.3 and 0.5 (corresponding to about 303 million raw GPS points) encountered resource exhaustion on the test configuration. Despite DuckDB's out-of-core capabilities (spilling to disk), the system experienced critical memory saturation (near 100% utilization of RAM and swap), resulting in process termination. We attribute this limitation to the specific characteristics of spatiotemporal workloads on commodity hardware. Unlike standard relational data, spatiotemporal query processing involves complex intermediate structures such as high-cardinality geometries or variable-length spatial objects, incurring significant memory overhead that might not always be immediately spillable. We therefore interpret this as a practical upper bound for interactive spatiotemporal analytics using an in-memory analytical engine on commodity hardware under the evaluated workload. Importantly, this limitation reflects the constraints of the execution environment rather than an inherent scalability limit of DuckDB itself. Larger scale factors are expected to be feasible on machines with higher memory capacity or through distributed and cloud-based deployments.

7. Limitations and Future Work

The latest implementation of MobilityDuck integrates MEOS and binds spatiotemporal types and functions adapted from the implementation of MobilityDB. As MEOS and MobilityDB are both constantly evolving with frequent additions of types and functionalities, the volume of such adaptation can grow very rapidly. Future development of MobilityDuck can benefit from an automated tool for generating bindings of all types and functions to ensure the most complete and up-to-date implementation on a par with MEOS.

In working with GEOMETRY type, we do not work with this type directly due to its specialized implementation by

the Spatial extension. In the latest MobilityDuck implementation, we use an additional proxy layer where the functions that are supposed to return GEOMETRY type will, instead, return either WKB_BLOB or VARCHAR types, which are standardized. The casting to and from these types and GEOMETRY is left for the Spatial extension to handle, which is enforced by adding ::GEOMETRY, ::WKB_BLOB, etc. to the relevant values. This interface, while simple in terms of implementation, results in unnecessary overheads when dealing with data of type GEOMETRY, as previously discussed in Section 6.2. Future development of MobilityDuck will focus on a more refined integration with the Spatial extension in order to support the GEOMETRY-related functions more natively and efficiently.

MobilityDuck currently does not support the `geography` type. Future work will examine the native support for this data type, such as with the use of the Geography extension,[7] in order to develop MobilityDuck's interface for handling this type.

8. Conclusion

This paper introduces MobilityDuck, a DuckDB extension which integrates the mobility data management capacity of MEOS into a lightweight, in-memory analytical database system. By embedding spatiotemporal types and trajectory operators directly into an in-memory analytical engine, MobilityDuck is among the first systems to bridge the gap between traditional moving object databases and modern embedded analytics systems. It enables efficient analysis of large spatiotemporal datasets while preserving DuckDB's strengths in performance, simplicity, and seamless integration with data science environments. By using the BerlinMOD-Hanoi dataset, we demonstrated the performance of MobilityDuck. First, we presented a use case scenario that integrates MobilityDuck with DuckDB's Python API and the Jupyter Notebook environment, enabling fast visualizations of query results and showing the quick integration of MobilityDuck into the existing DuckDB ecosystem. Secondly, we compared the runtime of MobilityDuck against MobilityDB using a set of benchmark queries of BerlinMOD. In the majority of cases, MobilityDuck achieved better results than MobilityDB with two types of indices, showing its potential for developing a unified, high-performance analytical framework for spatiotemporal data.

For future work, we aim to expand the spatiotemporal analytics capabilities of MobilityDuck by adding support for the remaining types and functions of MEOS, and potentially develop an automated tool for keeping MobilityDuck up-to-date with both MEOS and MobilityDB. Additionally, we plan to further develop the indexing capabilities of MobilityDuck to support indexing more spatiotemporal data types, as well as to re-evaluate its performance once indexing has been more thoroughly supported.

9. Artifacts

All relevant code and instructions for building MobilityDuck as well as the use case demonstration and benchmark are available on the official GitHub repository.[8] In addition, precompiled binary extension packages are also provided in GitHub repository for Linux, macOS, and DuckDB-Wasm. At present, Windows is not supported due to compatibility limitations in the underlying MEOS library. Support for Windows will be made available once MEOS provides the necessary compatibility. All code and data related to BerlinMOD-Hanoi are also available.[9]

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT in order to paraphrase and reword. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] R. Newell, D. Theriault, M. Easterfield, Temporal gis: Modeling the evolution of spatial data in time, Computers & Geosciences 18 (1992) 427–433.

[2] S. Gebbert, E. Pebesma, The grass gis temporal framework, International Journal of Geographical Information Science 31 (2017) 1273–1292.

[3] N. Pelekis, B. Theodoulidis, I. Kopanakis, Y. Theodoridis, Literature review of spatio-temporal database models, The Knowledge Engineering Review 19 (2004) 235–274.

[4] E. Zimányi, M. Sakr, A. Lesuisse, Mobilitydb: A mobility database based on postgresql and postgis, ACM Transactions on Database Systems (2020) 19:1–19:42.

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

[6] T. Pham, D. Nguyen, K. Doan, S4strd: A scalable in memory storage system for spatio-temporal real-time data, in: Proc. of the 2015 IEEE Int. Conf. on Smart City/SocialCom/SustainCom, IEEE, 2015, pp. 896–901.

[7] H. Wang, K. Zheng, J. Xu, B. Zheng, X. Zhou, S. Sadiq, SharkDB: An in-memory column-oriented trajectory storage, in: Proc. of the 23rd ACM Int. Conf. on Information and Knowledge Management, 2014, pp. 1409–1418.

[8] K. Richly, Memory-efficient storing of timestamps for spatio-temporal data management in columnar in-memory databases, in: Proc. of the Int. Conf. on Database Systems for Advanced Applications, Springer, 2021, pp. 542–557.

[9] M. Raasveldt, H. Mühleisen, Duckdb: An embeddable analytical database, in: Proc. of the 2019 Int. Conf. on Management of Data, ACM, 2019.

[10] M. Raasveldt, H. Mühleisen, Data management for data science-towards embedded analytics, in: Proc. of the 10th Conference on Innovative Data Systems Research, 2020.

[11] M. Gabrielsson, PostGEESE? Introducing The DuckDB Spatial Extension, 2023. https://duckdb.org/2023/04/28/spatial.html.

[12] P. Gabel, quackML: A duckdb extension implementing a full service ai/ml engine, 2025. https://github.com/parkerdgabel/quackML.

[13] C. Düntgen, T. Behr, R. Güting, Berlinmod: A bench-

[7] https://duckdb.org/community_extensions/extensions/geography.html

[8] https://github.com/MobilityDB/MobilityDuck

[9] https://github.com/MobilityDB/MobilityDB-BerlinMOD-Hanoi

mark for moving object databases, The VLDB Journal 18 (2009) 1335–1368.