Distributed MobilityDB: A Scalable Moving Object Database Management System

MOHAMED BAKLI, Université libre de Bruxelles, Belgium and Assiut University, Egypt
MAHMOUD SAKR, Université libre de Bruxelles, Belgium and Ain Shams University, Egypt
ESTEBAN ZIMÁNYI, Université libre de Bruxelles, Belgium
NILS DIJK, Microsoft, Netherlands
MARCO SLOT, Microsoft, Netherlands

As the volume and complexity of spatiotemporal data continue to expand rapidly across various domains such as urban planning, environmental monitoring, and logistics, the demand for comprehensive data management systems becomes increasingly urgent. Handling such data entails intricate topological and analytical operations, emphasizing the necessity for robust and adaptable solutions capable of addressing diverse user queries.

This paper introduces Distributed MobilityDB[1], an open-source system engineered to manage big spatiotemporal trajectory datasets within SQL environments. Distributed MobilityDB offers capabilities for scalable spatiotemporal data management, facilitating efficient distributed query processing while seamlessly integrating with existing MobilityDB SQL operations. Key contributions highlighted in the paper encompass an adaptive spatiotemporal SQL query engine. This engine channels user SQL queries through various planning strategies for optimizing the distributed query plan, then distributing the query execution across cluster nodes transparently to the user. Various spatiotemporal query types are supported for distribution, including range selections, and joins proximity. Distributed MobilityDB is implemented as an add-on extension to PostgreSQL, which facilitates installing it on a readily running server. The paper further presents extensive experiments conducted on both cloud and on-premise environments using both real and synthetic datasets, including AIS for ship trajectories and BerlinMOD for simulated person trips.

Additional Key Words and Phrases: MobilityDB, Citus, Trajectory Processing, Moving Objects

ACM Reference Format:
Mohamed Bakli, Mahmoud Sakr, Esteban Zimányi, Nils Dijk, and Marco Slot. 2024. Distributed MobilityDB: A Scalable Moving Object Database Management System. *J. ACM* 37, 4, Article 111 (August 2024), 39 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The field of mobility data management is rapidly transforming, with the efficient processing of large-scale spatiotemporal data becoming increasingly critical. This type of data is integral to numerous businesses, including for instance global shipping [43], urban transport [32, 48], and

[1]https://github.com/mbakli/DistributedMobilityDB

Authors' addresses: Mohamed Bakli, mohamed.bakli@ulb.be, Université libre de Bruxelles, Bruxelles, Belgium and Assiut University, Asyut, Egypt; Mahmoud Sakr, Université libre de Bruxelles, Bruxelles, Belgium and Ain Shams University, Cairo, Egypt, mahmoud.sakr@ulb.be; Esteban Zimányi, Université libre de Bruxelles, Bruxelles, Belgium, ezimanyi@ulb.be; Nils Dijk, Microsoft, Amsterdam, Netherlands, nils.dijk@microsoft.com; Marco Slot, Microsoft, Amsterdam, Netherlands, marco.slot@crunchydata.com.

0004-5411/2024/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

biodiversity monitoring [34]. As the volume and complexity of spatiotemporal data escalate, they pose a significant challenge, emphasizing the need for scalable and robust management systems.

In recent years, spatial and spatiotemporal solutions have emerged within the domain of big data frameworks, presenting a plethora of methods and some system prototypes. Early system prototypes include Summit [1] and HadoopTrajectory [3], both of which extend Hadoop by incorporating trajectory types and operations. These extensions enable users to invoke trajectory-related functionalities within their map-reduce programs. In the Spark ecosystem, notable works include TrajSpark [49], TrajMessa [21], and Dragoon [13]. Basing on Spark, these systems have demonstrated performance in handling compute-intensive analytical tasks. Further some frameworks offer a SparkSQL interface, as in [11, 13, 21], enabling users to execute query operations in SQL. It is essential to note though that the Spark Catalyst processes these query operations somewhat opaquely, lacking spatiotemporal understanding. For example, when dealing with a join operation between two distributed tables, the Catalyst optimizer faces the challenge of selecting an appropriate strategy. Two primary options that can be applicable here: (1) Repartitioning both tables based on the specified partitioning key. However, this approach is not feasible when dealing with spatiotemporal keys, as they are not supported by systems for repartitioning. (2) Broadcasting one of the tables, which can be a very costly operation, particularly if the tables involved are considerably large. In practice, option (2) is often the chosen solution for queries featuring a spatiotemporal join, as the catalyst lacks understanding of the semantics behind spatiotemporal predicates to reshuffle data. To ensure optimal performance, users often find themselves compelled to script in Python, managing queries and filtering data based on their comprehension (e.g., broadcasting A, not B, and vice versa). Indexing in spatial systems such as Apache Sedona [45] could also be used to speed up one side of the join. It's worth acknowledging that the efficacy of this approach can be influenced by the order of query elements, potentially impacting overall performance.

Existing big data frameworks excel in scenarios where the advantages of distributed processing outweigh the initial overhead of loading and distributing data files. However, in scenarios where the predominant requirements involve storing data and executing frequent queries that can leverage indexes, such as the dashboard-style queries, a database platform becomes more fitting. Such applications often necessitate querying and aggregating data based on temporal, spatial, and categorical parameters, making a database platform a more suitable choice. Furthermore, applications requiring intricate spatiotemporal queries or necessitating joins between different datasets benefit from the expressive query languages, the robust data model encompassing various data types and operations, as well as the optimization strategies provided by databases.

In the realm of database systems, a couple of moving object database (MOD) systems have been developed by extending the relational database model with spatiotemporal types and operations: SECONDO [16] and MobilityDB[2] [50, 51]. Both systems adopt the same abstract data type (ADT) model but differ in their discrete representation, specifically their class hierarchies [51]. A key distinction between the two is their intended use cases: while SECONDO is designed as a research prototype, MobilityDB has been engineered as an extension to PostgreSQL with the goal of being industry-ready. Consequently, MobilityDB supports the full SQL standard [19], "part 2: SQL Data Definition and Data Manipulation Syntax and Semantics", which is a major advantage in practical applications. In MobilityDB, a moving object is abstracted as a function of time $f(t) \rightarrow geometry/geography$, where at a single time instant the moving object is represented as geometry or geography depending on the coordinate system. The *domain* of the function is the time span at which the moving object has been observed. The *range* is the spatial trajectory of the object. The object could be of any geometric type: point, line, or region, yet the common type

[2]https://github.com/MobilityDB/MobilityDB

is point, leading to the definition of temporal points. While the movement of the object in the real world is continuous, the current location tracking technologies observe it at discrete time instants depending on the sampling rate. In MobilityDB, a trajectory is represented as a sequence of geometry/geography point observations, with linear interpolation in-between.

In a previous work [4–6], we introduced a distributed moving object database system by integrating MobilityDB and Citus [3] [9]. Citus is an open source extension to PostgreSQL that enables data and query distribution across a cluster of PostgreSQL nodes in a shared nothing architecture. Citus, being the engine behind the managed database service Microsoft Azure Database for PostgreSQL [4], is a versatile tool for distributing PostgreSQL databases. It supports hash partitioning for data distribution. In this initial integration, MobilityDB tables could be distributed over a cluster of MobilityDB nodes using hash partitioning on alphanumeric columns (e.g., trip identifier). This out of the box integration already allowed several types of queries including spatial, temporal and spatiotemporal range queries to be distributed across distributed nodes. However, it did not address the complexities involved in spatiotemporal data distribution. Specifically, this approach involved several limitations: (1) Loss of spatiotemporal locality: Citus's hash partitioning method, while effective for distributing alphanumeric data, does not take spatial and temporal proximity into account. This leads to inefficient execution of spatiotemporal queries, particularly joins and kNN queries, where data locality plays a crucial role in performance; (2) Suboptimal query execution: The lack of awareness of spatiotemporal properties in Citus's query planner results in high communication costs, redundant data processing across nodes, and incorrect results, particularly for operations like proximity joins; (3) Complexity of query optimization: Relying on Citus's existing query planner without extending it to understand spatiotemporal predicates introduces significant inefficiencies.

To improve on these previous results, the work presented in this paper addresses the following main technical challenges: (1) extend data partitioning to support spatiotemporal partitioning, ensuring that moving object data is distributed in a way that preserves spatial and temporal proximity, (2) extend the SQL query planner and executor to plan and optimize spatiotemporal queries, including proximity joins and trajectory operations, across distributed nodes. Accordingly, this paper contributes a distributed MOD architecture and a corresponding open source implementation that bring the support the following capabilities:

- Distributed MOD that supports the full SQL standard
- Spatiotemporal data distribution via user-defined Functions, implementing multidimensional tiling strategy (Section 4).
- Distributed query planning, optimization, and execution for several spatiotemporal query types, including range queries, point-based and trajectory-based kNN queries, broadcast joins, intersection joins, self and distance joins, as well as joins involving trajectory tables with diverse partitioning schemes (Sections 5 and 7).
- Further, we set as a goal that this development is engineered as a PostgreSQL extension. In contrast to forking PostgreSQL, an extension allows deployment into existing PostgreSQL servers without the need for recompilation or server restart. This capability is of specific interest to industrial applications, where high availability needs to be guaranteed (Sections 3 and 6).

In the rest of the paper, the proposed Distributed MobilityDB system will be abbreviated as DistMobilityDB. The paper is structured as follows: Section 2 provides an overview of related work. Following that, in Section 3, we formally introduce the architecture of DistMobilityDB. In Sections

[3]https://github.com/citusdata/citus

[4]https://learn.microsoft.com/en-us/azure/postgresql/hyperscale/moved?tabs=direct

4–6, we delve into the details of the key system components, covering the Distribution Manager, Query Engine, and PostgreSQL Extension APIs, respectively. Section 7 is dedicated to presenting the experimental results. Finally, we conclude the paper and discuss the possible future work in Section 8.

2 RELATED WORK

In recent years, few commercial solutions for distributing relational databases have appeared, including Vitess [5] for distributing MySQL, Citus[6], Redshift[7], and TimescaleDB[8] for distributing PostgreSQL. They compare to each other in terms of their support to the different kinds of workloads: CRUD, data warehousing, analytical queries, multitenant applications, etc.

Meanwhile, the proliferation of GPS-enabled devices has generated massive amounts of mobility data. This explosion of data generation has posed many challenges in the data management community such as data partitioning for continuous trajectories and spatiotemporal joins. As a result, researchers and practitioners have proposed various prototypes for trajectory data management [38]. Most of them are based on HDFS/MapReduce or similar distributed processing frameworks such as [1, 3, 7, 11, 13, 20, 27, 28, 35, 36, 44, 49].

The only works in distributed moving object databases, up to our knowledge, as these are based on the SECONDO system, and our previous work is based on the MobilityDB system. The Distributed Arrays Algebra [18] provides a framework of distributed processing in SECONDO [17]. It is thus possible to distribute trajectory data and query processing using it. Older distributed versions of SECONDO [24, 26] provide integration with Hadoop and Cassandra to support the distribution. The main limitation in these works is the missing SQL support. The distribution is thus not hidden to the user. Similar to map-reduce, users must explicitly write a program to handle data distribution and query execution. Further, in [24, 26] the complete trajectory is duplicated across all overlapping partitions, which may replicate the size of the original dataset by one or more orders of magnitude.

As an illustration, consider a 20 GB AIS ship trajectory dataset where data replication escalates the size to 147 GB. Replication consumes substantial storage as most ship trajectories cover most of the spatiotemporal extent, resulting in replication across all overlapping tiles. However, this approach offers suboptimal performance compared to fragmentation, mainly due to that the number of spatiotemporal points is too large to be loaded and verified by any of the spatiotemporal predicates. Addressing this challenge, the approach presented in [36, 44] leverages the Map-Reduce paradigm to fragment trajectory data into multiple subtrajectories across overlapping partitions. Nevertheless, the input data comprises discrete points, and subtrajectories are stored as sets of points. In this paper, DistMobilityDB accepts a table containing complete trajectories as input, transforming it into a distributed table with fragmented trajectories, all while preserving the concept of interpolation. In this scenario, the DistMobilityDB query engine has the ability to distribute the trajectory functions and deal with them as aggregates. This would allow executing the query operations faster and in-place, rather than collecting them first which will happen but in rare cases.

In the following, we group the related work into three classes: spatiotemporal data partitioning, distributed processing algorithms for specific query types, and systems implementations. Our work belongs to the systems category, where we implemented a distributed moving object database extending on open source tools.

[5]https://vitess.io/
[6]https://www.citusdata.com/
[7]https://aws.amazon.com/redshift/
[8]https://www.timescale.com/

Spatiotemporal data partitioning. In [47] a time-hash-based trajectory data partitioning was introduced, to support range queries over discrete spatiotemporal points. The idea of temporal partitioning was also used in [39], in combination with coordinate quadtree coding for range and trajectory ID queries. In [33] a spatial trajectory clustering method extending the STR algorithm was proposed. It operates on selected points of trajectories that are grouped into subgroups with similar starting positions and similar ending positions. In [15, 23] the GeoHash algorithm was extended to support spatiotemporal range queries in MongoDB and HBase.

Spatiotemporal query distribution. Existing work targets the distribution of certain types of spatiotemporal queries, mostly by using the MapReduce framework and its open-source implementations Hadoop and Spark. P. Tampakis et al. [36] proposed a distributed subtrajectory join using MapReduce in three phases: temporal data partitioning, joining based on the plane-sweep algorithm, and refinement for grouping and sorting results. Ray et al. [29] introduced a multi-core, single-machine algorithm to join a distributed trajectory dataset with a non-distributed set of spatial objects, such as lines and polygons. The problem of joining two distributed spatiotemporal datasets was addressed in [40]. The cogroup Spark operation is used to collect, from the two datasets, the points that have proximity, and then verify the spatiotemporal relationship. In [31], a query optimizer for spatial joins was proposed to deal with three different cases for joining two spatial point datasets. Then, the optimizer applies either one-to-one join or one-to-many join to verify the intersects predicate. Sedona [45] is a spatial extension of Spark. It extends the RDD concept to support spatial data types, indexes, and geometrical operations at scale. It supports range and kNN queries. In [22], a time range count index is used on Spark for speeding up the kNN query. Other works that try to deal with the trajectory kNN queries are presented in [25, 41, 42, 46]. To sum up, all of the above approaches provide a distributed solution for specific query types supporting limited sets of query predicates. This differs from our work in DistMobilityDB, where we aim to design a distributed MOD capable of distributing a wide range of user spatiotemporal queries while leveraging the existing MOD infrastructure.

Systems. Table 1 compares the relevant systems. The distinctive feature in DistMobilityDB is the transparent query distribution which is taken over by a query engine. DistMobilityDB uses multi-dimensional tiling to split and distribute the trajectories over the worker nodes. It thus follows the general strategy of computing partial results over sub-trajectories, then aggregating the final query result. A clear advantage is that trajectories are not replicated over nodes, which avoids duplicating the data storage cost. In addition, DistMobilityDB is able to distribute a wide range of spatiotemporal selection and join queries that can be found in the literature.

Table 1. Functional comparison between the distributed trajectory data management systems. The proposed system in this paper, DistMobilityDB, is shown in the first row.

Systems / Features	Architecture	SQL Planning	Query Writing	Partitioning	Trajectory Representation	Range	kNN	Trajectory Functions	Broadcast Join	Distance Join	Intersection Join
DistMobilityDB	DBMS	✓	SQL	MD Tiling	Subtraj-based	S/T/ST	✓	✓	✓	✓	✓
SECONDO [18]	DBMS		Query Plan	Uniform Grid	Replication	S/T/ST	✓	✓	✓		✓
Summit [1, 2]	MapReduce		Pigeon	Two-level	point-based	S/T/ST	✓		✓		
HadoopTrajectory [3]	MapReduce		Hadoop	Rtree, Grid	Replication	ST	✓		✓		
TrajSpark [49]	RDD		Spark	Quadtree	Point-based	S/ST	✓				
TrajMesa [20]	RDD		SparkSQL	Vstore, Hstore	Replication	S/ST	✓				
UITrajMan [11]	RDD		SparkSQL	STR	Point-based	ST					
TrajStore [10]	Standalone			Quadtree	Subtraj-based	ST	✓				
Dragoon [13]	RDD		SparkSQL	STR, Grid	Point-based	ST	✓				
SharkDB [37]	Standalone			Time-Based	Subtraj-based	T	✓				

3 DISTRIBUTED MOBILITYDB ARCHITECTURE

This section presents the architecture for the proposed DistMobilityDB system. An overview of this architecture is given in Figure 1. The hardware consists of a cluster of MobilityDB nodes, i.e., PostgreSQL nodes that include the MobilityDB extension. These nodes are denoted as workers W1 ... Wn. The cluster also includes one or more coordinator nodes, denoted Ci, which run the Citus extension of PostgreSQL. The coordinator nodes break a spatiotemporal user query into tasks. These tasks are then executed by the worker nodes in parallel. The figure captures the interactions within the coordinator, responsible for query distribution and execution management. While the worker nodes also execute queries, they follow the instructions sent by the coordinator, which is responsible for orchestrating the entire distributed process. The arrows in the figure indicate the flow of information and interactions between the PostgreSQL client and the various system components. These arrows represent bidirectional communication, where the Extension APIs facilitate the handling of queries and operations, directing them to the appropriate components for processing and receiving feedback or results in return.

The essential components of DistMobilityDB consist of three main modules, described briefly below.

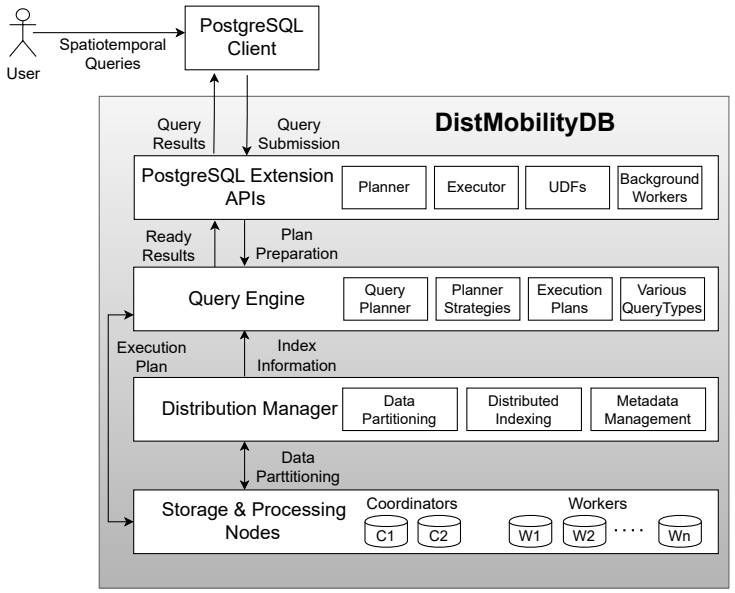


Fig. 1. The proposed Distributed MOD Architecture of DistMobilityDB: This system interacts with a user using any compatible PostgreSQL client. Core components include (1) PostgreSQL Extension APIs, which handle incoming queries and forward them to the query engine; (2) Core Query Engine, responsible for query planning and execution across nodes; (3) Distribution Manager, which manages data partitioning and sends index information to the query engine for filtering; and (4) Storage and Processing Nodes, which implement the execution plans and data partitioning through coordinators and workers.

PostgreSQL Extension APIs DistMobilityDB utilizes the available PostgreSQL hooks to modify database behavior dynamically. These hooks include the SQL query Planner, the plan Executor, User Defined Functions (UDFs), facilitating distributed data manipulation, and Background Workers,

handling asynchronous tasks for spatiotemporal data management. By leveraging these hooks, DistMobilityDB intercepts interactions between the PostgreSQL nodes that form the cluster, allowing for selective overrides of default behaviors, and achieving integration of distributed functionalities and optimizations. This avoids the inefficient query execution seen in the straightforward integration by allowing DistMobilityDB to manage spatiotemporal queries and generate the proper execution plans in the query execution process.

Query Engine The Query Engine is a cornerstone of DistMobilityDB. Upon receiving an SQL query, this engine transforms it into a distributed query plan. Its adaptability lies in its capability to seamlessly handle queries, encompassing both relational and spatial-temporal predicates, without necessitating users to modify their queries. The Query Engine addresses the shortcomings of the straightforward implementation by extending the query planner to understand the semantics behind query predicates and optimize spatiotemporal queries. This includes pushing down spatiotemporal filters to the nodes and minimizing data reshuffling for join queries. These optimizations significantly reduce the network overhead and improve the overall efficiency of distributed spatiotemporal query execution.

Distribution Manager DistMobilityDB partitions the input relation into shards that preserve spatiotemporal data locality and load balancing. Data partitioning follows the multirelational algebra MRA [8], with the addition that the partitioning is spatiotemporal. In the sequel, we denote the distributed table as multirelation following the MRA notation. It provides a two-level (global and local) distributed indexing scheme to reduce the global transmission cost and local computation cost.

Storage & Processing Nodes DistMobilityDB uses a shared nothing architecture. The cluster has a coordinator node that manages data storage and processing among worker nodes. All nodes are regular PostgreSQL instances. Only the coordinator knows about the cluster. Worker nodes do not know that they are part of a cluster. All nodes have PostgreSQL, Citus, and MobilityDB. The coordinator has in addition the DistMobilityDB extension, which handles the spatiotemporal distribution logic and delegates to Citus the communication between nodes, for instance, issuing a command and waiting for results. DistMobilityDB can also work on a single node, which then acts both as a coordinator and as a worker.

While Citus provides the underlying infrastructure for inter-node communication, DistMobilityDB introduces several layers of parallel processing that enhance query execution across multiple nodes. The system implements a custom spatiotemporal distributed query engine, complemented by various UDFs. The query engine ensures that data partitioning, task execution, and query planning are handled in a highly parallelized manner across all nodes.

To provide a clearer understanding of how the various components of the DistMobilityDB architecture relate to the proposed system's functionalities, the following sections correspond to each module:

- Distribution Manager (Section 4): This section describes the spatiotemporal-aware partitioning strategies used to efficiently distribute data across nodes, ensuring that spatial and temporal locality is preserved.
- Query Engine (Section 5): This section elaborates on the query execution process, including how the system optimizes spatiotemporal queries and leverages distributed resources for efficient execution.
- PostgreSQL Extension APIs (Section 6): This section details how we extend the PostgreSQL database system to integrate distributed functionalities and enhance spatiotemporal query processing through specific hooks.

4 DISTRIBUTION MANAGER

The Distribution Manager incorporates a spatiotemporal-aware partitioning scheme that is specifically designed to optimize the handling of moving object trajectory data. It considers both the spatial and temporal dimensions of trajectories, ensuring that data is distributed in a way that reduces the cost of queries involving moving objects. Furthermore, it maintains shorter trajectories within partitions for enhancing the performance of expensive trajectory operations (e.g., `intersects`). This is particularly important for trajectory-based queries, such as proximity joins, kNN queries, and MobilityDB trajectory functions, which require efficient access to the full moving object data.

The Distribution Manager orchestrates the transformation of a spatiotemporal relation into a distributed counterpart. It employs Multidimensional Tiling to split a given domain of any number of dimensions into a number of tiles. Intuitively, Multidimensional Tiling simply extends the traditional range partitioning provided by DBMSs such as PostgreSQL or MySQL to n-dimensional data, in our case to 2 or 3 dimensions for space and 1 dimension for time. The Multidimensional Tiling process is a dual-faceted operation involving both horizontal and vertical splitting. Horizontally, it reorganizes records within the relation, optimizing the distribution of data across distributed nodes. Vertically, it splits the trajectory objects, i.e., MobilityDB's temporal geometry type tgeompoint, into multiple fragments, enabling a granular and balanced distribution of trajectory data.

4.1 Data Model

DistMobilityDB distributes MobilityDB tables into multirelations, denoted as R^d, where d represents the dimensionality of the space in which the partitioned version of the relation R is defined. The partitioning may thus be spatial ($d = 2$) [9], or spatiotemporal ($d = 3$). These multirelations follow the multirelational algebra proposed by Ceri and Pelagatti [8]. Each multirelation R^d comprises a set of extended relations $\{R_1, R_2, ..., R_k\}$, where each R_i represents a partition of the original dataset alongside replicated indexes to maintain data integrity and query efficiency. Each extended relation R_i encapsulates an n-dimensional bounding box mbr_i that is is variable in type, accommodating temporal (T), spatial (S), or spatiotemporal (ST) dimensions, corresponding respectively to period types in MobilityDB, box types in PostGIS for 2D and 3D, and tbox and stbox types in MobilityDB for 2D, 3D, and 4D representations.

The extended relations are systematically arranged and connected through a Multidimensional Tiling Scheme (MTS), which is centralized in the coordinator node's catalog. This representation highlights the partitioning of data into smaller, manageable extended relations, each with its bounding box and descriptive data, efficiently organized and indexed for optimal query performance.

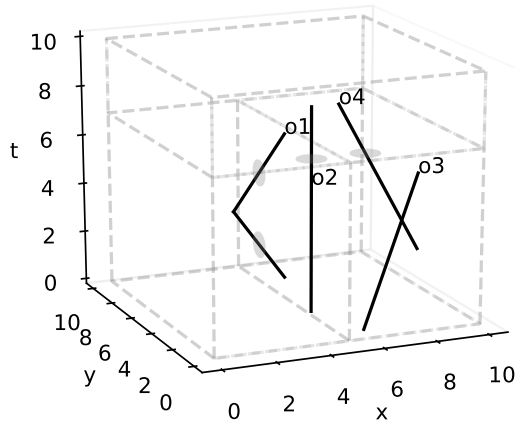
4.2 Multidimensional Tiling

We use the example in Figure 2 to illustrate Multidimensional Tiling (MD tiling). Depending on the data model, the four trajectories may be interpolated, as shown in the figure, or left as discrete points. We discuss the more general case when they are interpolated. MD tiling on point representation will then be a simplification of this discussion.

Trajectories may have other properties, which may be static (such as type, name, ...) or temporal (such as speed, gear, ...). This can be abstracted as a table with a trajectory attribute and a set of static or temporal attributes of any type. Common data partitioning methods will split such table either horizontally into groups of moving objects, or vertically into groups of columns. The two methods ignore the data proximity in the spatiotemporal space, and thus cannot help distributing topological and proximity joins. Instead, we partition the spatiotemporal space in tiles and partition the data accordingly. Figure 2 illustrates this process, where the table is partitioned both horizontally

[9]For simplicity we are not discussing 3d spatial features, although it is possible to distribute such data in this model.

(dividing records across different tiles) and vertically (breaking trajectories into shorter, multiple fragments), which enables efficient parallel processing.



id	trajectory
o_1	[(p(6, 8), t=0), (p(4, 8), t=3), (p(6, 8), t=6)]
o_2	[(p(4, 1), t=1), (p(4, 1), t=9)]
o_3	[(p(6, 1), t=0), (p(8, 1), t=6)]
o_4	[(p(8, 1), t=3), (p(5, 1), t=9)]

Fig. 2. Partitioning spatiotemporal trajectories using MD tiling.

Given a spatiotemporal space S with d dimensions , a *MD tile scheme* $MTS = \{T_1 \dots T_m\}$ is a partition of S into a set of disjoint tiles $T_i = \langle R_1, ..., R_k \rangle$, where $R_i = [lower, upper)$ is a continuous range in the domain of the i^{th} dimension. For temporal points, the number of dimensions is either 3 (2D spatial plus time) or 4 (3D spatial plus time). In the example of Figure 2, *MTS* consists of three tiles $T_1 = \langle [0, 5), [0, 10), [0, 7) \rangle$, $T_2 = \langle [5, 10), [0, 10), [0, 7) \rangle$, and $T_3 = \langle [0, 10), [0, 10), [7, 10) \rangle$.

Given a MD tile scheme $MTS = \{T_1 \dots T_m\}$ and a trajectory object o, we define the following two functions:

- Split$(o, MTS) \rightarrow \bigcup_{T_j \in MTS} \{(\bar{s}_i, j) \mid \bar{s}_i \in \text{Intersection}(o, T_j)\}$, which vertically splits an input trajectory o into multiple disjoint fragments $\bar{s}_i$, every fragment is fully included in a MD tile T_j. The result of this function is a set of pairs composed of a spatiotemporal fragment and the number of the containing tile. This partitioning improves data locality for queries involving spatial and temporal proximity, as fragments that belong to the same tile can be processed together independently.
- Merge$(\{\bar{s}_1, \dots, \bar{s}_k\})$, which joins all the fragments of an object into a single continuous trajectory. This is done by sorting the input objects in ascending order of their start time and appending them. If some of the fragments are missing, an error is thrown.

The two functions are defined such that for every object o it is always true that Merge(Split(o, MTS)) = o. Notice that splitting might result in adding interpolated points at the tile boundaries for the trajectories that span multiple tiles. During merging, the *normalization* algorithm in [51] will automatically remove these boundary points.

To minimize the cost associated with splitting and merging trajectories, DistMobilityDB employs several optimization strategies. First, the MD Tiling mechanism ensures that tiles are generated in a

manner that minimizes the number of iterations required to perform the `intersects` operation on each trajectory. The query planner initially filters data based on the tile's bounding box, allowing subsequent operations to focus solely on points within the tile's boundaries, which significantly improves efficiency. Additionally, we leverage parallel processing during the ingestion phase, where trajectory data is processed by multiple worker nodes concurrently. This parallelism allows the system to efficiently compute intersections and perform the necessary splits, which reduces the overall ingestion time. Furthermore, the GiST spatiotemporal index is created, allowing for faster lookup during the ingestion process and subsequent query operations.

Given a set of trajectory objects $O = \{o_1, ..., o_n\}$, *MD tiling* is then defined as a two-step process:

(1) Finding a MD tile scheme MTS that (a) results in balanced data partitions, and (b) preserves spatiotemporal proximity of objects within tiles and across neighbouring tiles.
(2) Partitions the set of trajectory objects O using the computed MTS. This is equivalent to compute $\forall o_i \in O, \text{Split}(o_i, MTS)$.

The MD tiling generation comprises two main phases, described next: (1) Tiling Strategy and (2) Tile Assignment & Replication.

4.2.1 ***Tiling Strategy.*** The tiling strategy aims at efficiently partition the spatial or spatiotemporal extent of a relation R with d dimensions into a series of non-overlapping tiles $\{T_1, ..., T_m\}$ where each tile $T_i = \langle T_i^1, \ldots, T_i^k \rangle$ defines a continuous range $T_i^j = [lower, upper)$ for each relation j within the domain of the i^{th} dimension.

The main goal of tiling is to ensure load balancing, in order to maintain approximately equal data size over all tiles. This enables queries to process concurrently, avoiding delays caused by uneven data distribution. In practical terms, consider the AIS ship trajectory dataset [14], where ship trajectories span a significant portion of the spatiotemporal extent. Therefore, in scenarios with notable variations, such as a trajectory containing 30,000 points compared to another with 1,000,000 points, the performance implications become particularly pronounced.

To partition the data across tiles, there exist in the literature some methods including [30, 36, 44]. While the existing methods address certain aspects of spatial and temporal data management, they do not give higher priority for ensuring load-balanced tiles in terms of both the number of partitioned shapes and the number of points within each shape. Depending on how large the imbalance is, it impact the performance of costly operations, e.g., intersection on long trajectories, reducing the query efficiency. In Section 7.7.3, we compare the query performance using multiple of these partitioning methods, and conclude that the different is indeed insignificant, because all of them still achieve some level of balanced data partitioning.

Here we describe a tiling algorithm that is easy to implement, while ensuring load balancing. The `ChoosePivot` function, described in Algorithm 1, illustrates it. This function seeks for each one of the d dimensions in the dataset an optimal pivot equalizing the distribution of overlapping shapes and points across partitions. It begins by defining the search space between the lower and upper bounds of the dimension of interest. The threshold ϵ plays a crucial role as a factor determining the allowed deviation percentage of the tile size. If not provided by the user, a default value of 5% is assumed.

The algorithm iteratively adjusts the pivot and recalculates the number of overlapping shapes and points on each side of the pivot. The `numInstants` function efficiently determines the count of overlapping points by leveraging a hash-partitioned table. This function achieves high performance due to two main reasons: (1) it operates in parallel across worker partitions, enabling concurrent processing, and (2) it validates overlaps using the local index within each partition, specifically the GiST index provided by MobilityDB. The pivot is considered optimal and returned if the difference between the left and right partitions falls within a tolerable range determined by the threshold

ϵ. Otherwise, the threshold ϵ is adapted based on feedback from previous iterations, with the adjustment proportional to the imbalance between leftCount and rightCount. This ensures that ϵ grows more significantly when there is a larger discrepancy between the counts. The adaptive increase calculation is governed by the conditional logic in line 21, which triggers when the count difference is less than or equal to half the optimal tile size. This approach recalibrates ϵ by a factor that increases with the relative imbalance between the partition counts. By using the minimum between the existing $\epsilon_{adaptive}$ and the new calculated value, the algorithm ensures a balanced rate of adaptation.

Algorithm 1: Choosing the Pivot point

Input: Dataset D, Tile size $tileSize$, Dimension dim, Threshold ϵ
Output: Pivot point $pivot$

1 $start, end \leftarrow$ `lowerBound`$(D,\ dim)$, `upperBound`$(D,\ dim)$
2 **if** ϵ *is not given* **then**
3 $\quad \epsilon \leftarrow 0.05$ `// Default is a 5% deviation of the tile size`
4 $pivot \leftarrow (start + end)\ /\ 2$ `// Initial pivot point`
5 $leftPivot, rightPivot \leftarrow start, end$
6 **while** ***true*** **do**
7 $\quad leftCount \leftarrow$ `numInstants`$(D, leftPivot, pivot)\ :\ \mathrm{D} \in [leftPivot,\ pivot)$
8 $\quad rightCount \leftarrow$ `numInstants`$(D, pivot, rightPivot)\ :\ \mathrm{D} \in [pivot,\ rightPivot)$
9 $\quad$ **if** $|leftCount - rightCount)| \leq tileSize * \epsilon$ **then**
10 $\quad\quad$ **return** $pivot$ `// Balanced pivot found`
11 $\quad$ **else if** $pivot > rightPivot$ **then**
12 $\quad\quad leftPivot, rightPivot \leftarrow start, end$
13 $\quad\quad pivot \leftarrow (leftPivot + rightPivot)/2$
14 $\quad\quad \epsilon \leftarrow \epsilon_{adaptive}$ `// Adaptive increase of epsilon`
15 $\quad$ **else if** $leftCount \leq rightCount$ **then**
16 $\quad\quad leftPivot \leftarrow pivot$
17 $\quad\quad pivot \leftarrow pivot + (rightPivot - pivot) * \epsilon$
18 $\quad$ **else**
19 $\quad\quad rightPivot \leftarrow pivot$
20 $\quad\quad pivot \leftarrow pivot - (pivot - leftPivot) * \epsilon$
$\quad$ `// Adaptive increase based on count difference`
21 $\quad$ **if** $|(leftCount - rightCount)| \leq\ tileSize * \epsilon * 0.5$ **then**
22 $\quad\quad \epsilon_{adaptive} \leftarrow \epsilon \times \min\left(\epsilon_{adaptive}, 1 + |leftCount - rightCount|\ /\ totalCount\right)$

After choosing the pivot point for every dimension, the partitions can be constructed by systematically dividing a relation R into a Multidimensional Tiling Scheme (*MTS*), consisting of a set of tiles, each encapsulating a portion of the dataset. The partitioning algorithm takes as input the relation R and the desired number of tiles, and determines the number of dimensions in the dataset. The algorithm iterates through each desired tile, performing the following steps:

(1) Dimension Selection, which selects the next dimension for splitting. This selection is based on the dimensions already processed and the evolving tile set, ensuring a systematic and balanced approach.
(2) Pivot Point Determination, which invokes the `ChoosePivot` function to determine a pivot point for the split along the chosen dimension. Employing a binary search approach, it

explores all possible overlapping positions to identify a pivot with approximately the required size with a tolerance ϵ of 5%. This meticulous pivot selection ensures precision and accuracy in the subsequent splitting process.

(3) Data Partitioning, which splits the input data into two subsets, R_{left} and R_{right}, based on the determined pivot point, to form $Tile_{left}$ and $Tile_{right}$, contributing to the Multidimensional Tiling Scheme.

Example: The following example demonstrates how to achieve the tiling implementation using a User-Defined Function, called `create_spatiotemporal_distributed_table`:

Distribute the trips table

```
SELECT create_spatiotemporal_distributed_table(
    'trips', 16, 'distributed_trips', 'MD_Tiling');
```

In this case, the input table trips is partitioned into 16 tiles using multidimensional tiling, with the result stored in the new distributed table distributed_trips.

4.2.2 ***Tile Assignment & Replication***. The tile assignment phase is a critical step in optimizing the distribution and replication of spatiotemporal tiles across worker nodes in DistMobilityDB. The goal is to efficiently assign tiles to nodes based on their spatiotemporal characteristics, namely, proximity, shared dimensions, and replication for fault tolerance, while promoting efficient data processing and minimizing communication latency.

Algorithm 2: Tile Assignment and Replication

Input: Spatiotemporal Tiles T, Worker Nodes N, Replicas Factor F, Weights α, β
Output: Distributed and Replicated Tiles

foreach *Node* $n_j \in N$ **do**
  foreach *Tile* $t_i \in T$ **do**
    if t_i *unassigned* **then**
      $C_i \leftarrow Centroid(t_i), C_j \leftarrow Centroid(n_j)$
      $S_{ij} \leftarrow Shared\ Dimensions(n_j)$
      $Cost(t_i, n_j) \leftarrow \alpha \cdot Dist(C_i, C_j) + \beta \cdot S_{ij}$
  Assign tile t_k *with minimum* $Cost(t_k, n_j)$ *to* n_j
  Replicate t_k *to adjacent nodes* n_k *sharing most dimensions using the replicas factor* F

The procedure described in algorithm 2 distributes tiles by iterating through the nodes until all tiles are assigned. In each iteration, it evaluates unassigned tiles t_i for their suitability to the current node n_j. A cost function $Cost(t_i, n_j)$ is computed, incorporating the Euclidean distance between the centroid of the tile, C_i, and the centroid of the node, C_j. Additionally, the number of shared dimensions, S_{ij}, is considered. Two weighting factors α and β allow fine-tuning the algorithm based on the importance of proximity and shared dimensions. The algorithm selects the tile t_k with the minimum cost for assignment to the current node n_j. Replication of the assigned tile is then performed to adjacent nodes that share the most dimensions with the assigned node. This replication strategy enhances fault tolerance and ensures redundancy within the distributed system. This proactive redundancy is leveraged by the planner, significantly mitigating network I/O during the execution of non-colocated join queries. Additionally, the weighting and replicas factors can be tuned through SQL commands. Users can adjust them dynamically by executing commands such as

`SET tile_proximity = TRUE`, `SET tile_shared_dimensions = TRUE`, and `SET tile_replicas = 1`, depending on the optimization goals.

5 QUERY ENGINE

The Query Engine is optimized for large-scale MODs by supporting complex trajectory-based queries and proximity-based joins unique to moving object databases. Unlike generic spatiotemporal systems, it handles dynamic object movements efficiently. We have extended the query planner and executor to handle and efficiently execute trajectory-specific queries, ensuring low-latency responses for moving object tracking and analysis.

We consider distributed SQL architectures in which the nodes are themselves SQL databases that can plan and execute SQL queries on local shards. In this setting, the distributed query planner is primarily concerned with breaking down a query on distributed tables into a plan that consists of: (1) sets of SQL queries that run in parallel on shards, (2) network transfer primitives such as collecting, broadcast, or reshuffling intermediate results, (3) a SQL query that combines intermediate results on the coordinator. The local SQL queries are constructed from the multirelational operator tree generated by the distributed query optimizer, based on the 1:1 mapping between relational algebra and SQL. The local queries are then planned and executed by the worker nodes using the regular SQL optimizer, which can take advantage of local (spatial) indexes.

In our approach, we partition the database twice: once using horizontal partitioning (e.g., using the standard hash partitioning provided by Citus), and once using MD tiling. This has the overhead of duplicating the distributed table twice. We think however that this overhead is comparable to building an index, where in this case it helps distributing more types of queries.

In order to decide which distributed copy to use, the query is first analyzed. If spatiotemporal selections or joins are identified, we use MD tiling copy. For all other queries, hash partitioning is used. This approach is common in databases, where one or several indices are created to speed up certain read operations. This enables distributing both spatiotemporal queries as well as trajectory queries.

When relations have multiple representations (e.g., hash partitioning and MD tiling) or the query involves complex joins, the optimizer generates all possible variants of the operator tree and selects the one that has the lowest cost. Precise cost estimation is a complex topic which is beyond the scope of this paper, but simple heuristics that consider how much work is pushed down generally perform well.

5.1 The *merge* Operator

A plan for a query on a distributed relation that uses MD tiling requires a *merge* operator in the multirelational operator tree. This operator groups the fragments by their primary key, returns the value of each attribute, and applies the Merge function defined in Section 4.2 on incoming fragments to reconstruct a trajectory (if needed). When converting the operator tree to SQL, the merge operator would be represented as a DISTINCT or GROUP BY on the tripId.

The *merge* operator is placed on top of the *collect* operator that fetches data from the nodes. Optimization rules then need to be applied taking the semantics of the *merge* operator into account. This is explained in the following sections.

5.2 Selection Optimization

In the presence of MD tiling, the *selection* operator is inserted directly above the *merge* operator prior to optimization. Some *selection* functions on trajectories can be pushed down below the *merge* and benefit from the properties of MD tiling. In particular, a *selection* operator that filters trajectories based on the intersection with a particular area, referred to as *range query*, can be

translated into an equivalent selection on tiles and pushed down. Computing the intersection for individual trajectories is then parallelized at the fragment level to minimize the overall execution time. In addition, the *collect* operator can be extended to prune away tiles that do not intersect with the query range, which reduces the overall amount of work. Figure 3 illustrates the general strategy for such a query. On the left, we see the typical execution strategy for non-distributed tables, where the selection is pushed directly to the relation. In the distributed case, shown in the middle, data is first collected and merged at the coordinator before applying the selection. However, this approach can be inefficient due to the overhead of collecting data at the coordinator. It is used only when the selection operation requires processing the entire trajectory. On the right, the figure presents a more efficient strategy where the selection is pushed down to the trajectory fragments within the distributed tiles. This approach enhances performance by minimizing data movement and applying operations on shorter trajectory fragments, leading to faster processing.

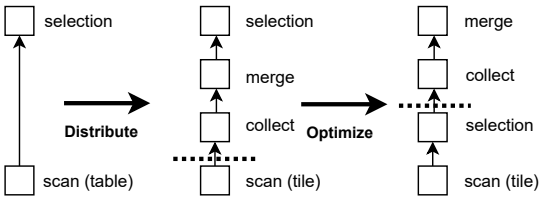


Fig. 3. Left is a plan for a simple selection query done centrally without distribution. In the middle is a naive but correct distributed execution plan, where the data is distributed in tiles at nodes, and the plan collects all data at the coordinator and performs the selection. The plan in the right utilizes the equivalance rule that selections can be pushed down to the workers and done in parallel, then the result are collected and merged at the coordinator. The dashed horizontal lines indicate the boundary between the coordinator node, and the data and processing nodes

Consider a distributed table trips(tripId, trip) where tripId is the primary key and trip is a spatiotemporal trajectory. An example of a range query is as follows:

Q1) Find trips that have passed a specific region of interest (ROI)

```
SELECT tripId FROM distributed_trips
WHERE intersects(trip, 'Polygon((...))')
```

The WHERE clause contains a range predicate that intersects the spatiotemporal trip attribute with a spatial polygon. Since the SELECT clause does not include operations that need the whole trajectory, it can be pushed down below the *merge* operator, which will only remove duplicate results (i.e., tripId) from different fragments of the same trip. Such selections are always commutative with the *collect* operator and can hence be parallelized across shards.

A *selection* operator that operates on the whole trajectory is not commutative with *merge*, since it can only be applied after the trajectory is reconstructed from the fragments. In the following example, the selection is done based on the trajectory length, a function that requires the whole trajectory.

Q2) Find trips that have traveled a distance exceeding 10 km

```
SELECT tripId FROM distributed_trips
WHERE length(trip) > 10000
```

In this case, the plan that uses a hash-partitioned table will be considered cheaper, since the *selection* can be pushed down.

Another case in which a *selection* cannot be pushed down below the *merge* is when the whole trajectory is used in other parts of the query (e.g., it is returned in the SELECT clause). The reason is that the pushed-down selection will only return a subset of the fragments, and thus, the trajectory cannot be fully reconstructed from the results. The next query illustrates an example of this case.

Q3) Find the complete trajectory of trips that have passed a specific region of interest

```
SELECT trip FROM distributed_trips
WHERE intersects(trip, 'Polygon((...))')
```

While the *selection* is of the range type, which can be pushed down, the whole trajectory in the SELECT clause cannot be reconstructed from the fragments obtained by executing this query on the tiles. That leaves the planner with two possibilities: (1) use the hash-partitioned representation, losing the benefits of fragment-level parallelism and pruning, or (2) collect only the unique identifier of the trajectories using MD tiling and then collect the full trajectories by joining with the hash-partitioned table. In the second option, the optimizer can also choose between a *reshuffle* or a *broadcast* operation based on the size of the result set (the two plans in Figure 4). Both operations allow a subsequent join to be pushed down.

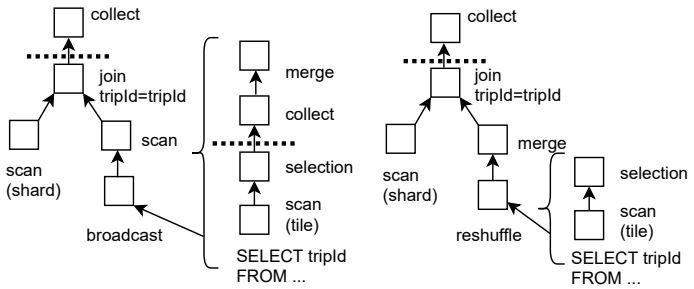


Fig. 4. The broadcast (left) v.s. reshuffle (right) approaches for collecting complete trajectory objects in the results. The broadcast operation, suited for small result sets, collects and merges intermediate results at the coordinator, then broadcasts them to all nodes to minimize further data reshuffling. In contrast, the reshuffle operation, used for large result sets, redistributes intermediate results to the workers before performing the merge operation.

Figure 4 illustrates two different plans to handling this query. The broadcast operation is used when the result set is relatively small. In this approach, the intermediate results from the shards (distributed data nodes) are first collected and merged at the coordinator. Then, these results are broadcast back to all the nodes. This ensures that all nodes have the same intermediate results to work with, minimizing the need for further data reshuffling. The query tree transformation for *broadcast* is given in the left side of the figure and involves collecting, merging, and then broadcasting the selected tripIds, as shown in the subplan enclosed in curly brackets. We denote by *scan*(*tile*) the access to the MD tiling version of the data, and we denote by *scan*(*shard*) the access to the hash partitioned version.

On the other hand, when the result set is large, a reshuffle operation is preferred, as shown in the right side of the figure. Reshuffling also distributes the intermediate results to workers but, in contrast to broadcasting, the data movement occurs among the workers directly, without

involving the coordinator. When reshuffling directly after the selection, we skip the *merge* normally performed after the *collection*. We will need to compensate for duplicate trip IDs being considered for joining by executing the *merge* step after the *reshuffle* operation.

5.3 Supported User Queries

In this section, we enumerate several state-of-the-art queries that users can execute on multirelations. These queries will be assessed in the experimental section using various datasets. The subsequent schema will be employed to elucidate the queries:

Database schema

```
CREATE TABLE shipsFishing(tripId INTEGER PRIMARY KEY, shipId INTEGER, trip TGEOM (POINT))
CREATE TABLE shipsCargo(tripId INTEGER PRIMARY KEY, shipId INTEGER, trip TGEOM (POINT))
CREATE TABLE ports(portId INTEGER PRIMARY KEY, geom GEOMETRY (POLYGON))
```

The tables *shipsFishing* and *shipsCargo* are multirelations partitioned using three dimensions (x, y, t). Both tables include the *trip* attribute of temporal point type. The *ports* table, a PostGIS relation, contains a geom attribute of geometry type and serves as a smaller reference table replicated across all worker nodes.

Range Query - Range(q,R): Given a query range q and a multirelation $R = \{r_1, r_2, ..., r_i\}$, range(q,R) finds all objects $o \in R$ that overlap the range defined by q, such that

$$\text{Range}(q, R) := \bigcup_{r_i \in R} \{\forall o \in r_i \mid o \in q, o \in \text{Overlapping}(q, r_i)\}$$

The query range can be temporal-only, spatial-only, or spatiotemporal defined by any of the PostGIS (e.g., geometry(polygon)) or MobilityDB (e.g., period, tgeom(point)) types. An example of how the user writes a range query is as follows:

Find ships that have passed a specific region of interest (ROI)

```
SELECT shipId, tripId FROM shipsFishing
WHERE intersects(trip, 'Polygon((...))')
```

The WHERE clause contains a range predicate that intersects the spatiotemporal trip attribute with a spatial polygon.

Intersect-Join Query- $R \bowtie_{Intersects} S$: Let $R = \{r_1, r_2, ..., r_i\}$ and $S = \{s_1, s_2, ..., s_j\}$ be two multirelations of spatiotemporal objects o_{r_i}, o_{s_j} and the predicate is any of the `intersect` functions of PostGIS and MobilityDB. The query returns all pairs of objects that have intersections. The formal representation is as follows:

$$R \bowtie_{\text{Intersects}} S := \{(r_i, s_j) \in (R \times S) \mid \text{pair}(o_{r_i}, o_{s_j}) \in \text{Intersect}(r_i, s_i)\}$$

As an example, consider the following query.

Find ships involved in accidents through their trips

```
SELECT R.tripId, S.tripId FROM shipsCargo R, shipsFishing S
WHERE intersects(R.trip, S.trip)
```

The `intersects` predicate verifies whether the pair of objects are spatiotemporal intersect or not. This query will be executed as a colocated join query if the two multirelations share the same

tiling scheme MTS however it will be treated as a non-colocated join query if the tiling scheme is not the same.

Distance-Join Query- $R \bowtie_{Distance} S$: Let $R = r_1, r_2, ..., r_i$ and $S = s_1, s_2, ..., s_i$ be two multirelations of spatiotemporal objects o and a distance threshold d. The user can use any distance function in MobilityDB. The result of the d distance join query is the set of objects containing all the possible different pairs from that have a distance of each other smaller than, or equal to d:

$$R \bowtie_{\text{Distance}} S := \{(r_i, s_j) \in (R \times S) \,|\, \text{pair}(o_{r_i}, o_{s_j}) \in \text{Dist}(r_i, s_i) \leq d\}$$

As an example, consider the following query.

Find ships that are closer than or equal to 500 meters

```
SELECT R.tripId, S.tripId FROM shipsCargo R, shipsFishing S
WHERE edwithin(R.trip, S.trip, 500)
```

The dwithin predicate returns true when the first two arguments are closer than or equal to the distance threshold in the third parameter, here 10 meters. This query thus reports the pairs of trips that have ever been at a distance of 10 meters or less to each other.

K Nearest Neighbor Search Query- $KNNS(R, q, k)$: Given a multirelation $R = \{r_1, r_2, ..., r_i\}$, a query object q (e.g., point, trajectory), and a number k. The result of the kNN query with respect to the query object is an ordered set of objects $o \in R$ with the k smallest distances from q. The formal definition is as follows:

$$KNNS(R, q, k) := \{\forall_o \in R \,|\, r_i \in R : \\ \text{Dist}(r_i, q) \leq \text{Dist}(r_{i+1}, q) \leq \ldots \leq \text{Dist}(r_n, q), \,|\, \text{Dist}(R, q) \,|\, \leq k\}$$

As an example, consider the following query.

Find the closest K trips to a given point

```
SELECT tripId FROM shipsCargo
ORDER BY distance(trip, 'Point(...)') asc
LIMIT k
```

The `distance` function computes the distance between each ship trajectory and a given `point` object. Executing this query employs a two-phase strategy: initially collecting distances, followed by the subsequent selection of the top k in the second phase. The query is planned using the Filer and Refine strategy, described in Section 5.4.

5.4 Planner Strategies

A pivotal aspect of DistMobilityDB is the query planner that utilizes MD tiling, offering significant advantages in the distribution of spatial, temporal, and spatiotemporal joins. Through the meticulous partitioning of data based on spatial and temporal dimensions, MD tiling transforms spatiotemporal join queries into colocated joins, eliminating the need for extensive data reshuffling. Notably, the planner leverages stored catalog information detailing the spatiotemporal extent of the tiles, enabling a strategic approach to minimize data reshuffling for other spatiotemporal join operations. This proactive strategy enhances query performance and optimizes the distributed processing of complex spatial and temporal relationships within MobilityDB.

This section elucidates five key strategies integral to the query planner of DistMobilityDB query planner, showcasing the system's prowess in handling spatial, temporal, and spatiotemporal joins seamlessly.

5.4.1 ***Early-Stage Filter***. We propose an effective filter operation that early prunes a large amount of data. It is triggered for all queries with predicates that require a topological comparison between two objects. It adds the necessary filtering predicates (e.g., overlaps) to trigger the local index before expensive operations are performed. This strategy aims to avoid unnecessary memory, CPU, connections, and network costs.

Consider a query Q that involves a topological comparison between objects A and B, the strategy identifies relevant predicates P for early application in the query process. This is denoted as:

$$P = \{p_1, p_2, ..., p_n\} \quad \text{where each } p_i \text{ is a topological predicate applicable to } A \text{ and } B$$

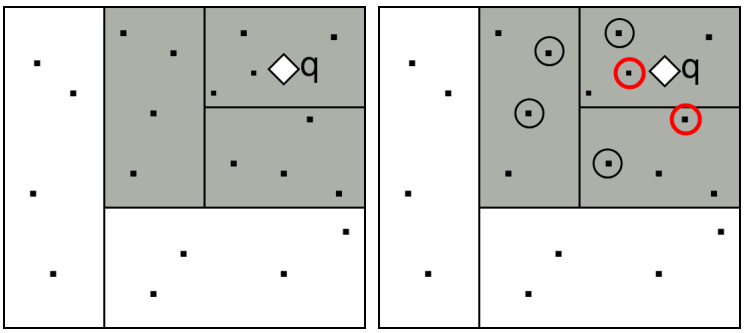
Using the identified predicates P, the strategy utilizes local indexes to effectively prune the dataset. Let D represent the dataset, and D' denote the pruned dataset after applying predicate pushdown. The pruning process can be expressed as:

$$D' = \{d \mid d \in D, \exists p \in P : p(d)\}$$

Here, $p(d)$ represents the application of predicate p to data element d, filtering out irrelevant entries.

5.4.2 ***Filter and Refine***. Consider a query Q that requires a two-phase approach for execution, necessitating the division of the query into two distinct operations. The initial query acts as a filtering operation employing the Early-Stage Filter strategy. Additionally, it involves modifications to the query clauses, such as transferring the distance operation from the Orderby Clause to the Selection Clause, allowing the utilization of the results in another query. The second query functions as a post-processing operation.

A concrete example is the kNN query, which entails computing distances between each spatiotemporal object in a multirelation and a specific spatial object. In this case, the query is processed in two steps: (1) Filter the Multidimensional Tiling Scheme (*MTS*) to eliminate unnecessary tiles, pushing the query down to local tiles after incorporating the distance function into the query selection; (2) Select k objects with minimum distances. The local index is leveraged in Step 1 to expedite the KNN selection process. In Step 2, the query is finalized by aggregating all results and selecting the top k. Figure 5 provides a visual representation of how this query is planned.



(a) Step1: Filter and push down **(b)** Step 2: Refinement

Fig. 5. An example kNN search query, k=2. (a) the query node q is highlighted as a diamond. The relevant shards are identified and highlighted in gray. The query is only pushed to these shards. (b) per shard, the top two results are computed in parallel (highlighted in black circles). The coordinator will then collect and merge these results to select the global top k among them (highlighted in red circles)

5.4.3 ***Broadcast Join.*** In scenarios involving a join query Q between a large multirelation R and a smaller-sized relation S, the Broadcast Join strategy proves effective. This method involves replicating the smaller table across all worker nodes, allowing for parallel computation of the join operation. The coordinator subsequently aggregates and collects the results. The Broadcast Join operation is expressed as: $R \bowtie_{\text{intersects}} S$. Here, $\bowtie$ signifies the join operation, and intersects denotes the spatiotemporal intersection condition. It is crucial to highlight that workers have the ability to optimize their local queries by leveraging available spatiotemporal indexes. This approach often leads to faster results, primarily due to the usage of the local MobilityDB indexes. Additionally, the use of bounding boxes for fragments, compared to full trajectories, enables more effective pruning.

As an illustrative example, consider the query:

Find ships that have traveled through a specific set of ports

```
SELECT t1.shipId, t.tripId, p.portId FROM shipsFishing t, ports p
WHERE intersects(t.trip, p.geom)
```

If the ports relation is replicated in all worker nodes, this query will compute the intersection join locally between the fragments in every tile, and then report all intersecting pairs to the coordinator. The merging in the coordinator will remove duplicate results.

5.4.4 ***Colocated Join.*** Consider a query Q requiring a join operation between multirelations $R1 \bowtie_{jp_1} R2 \bowtie_{jp_2} \ldots \bowtie_{jp_n} RN$, each characterized by a consistent Multidimensional Tiling Scheme (MTS) and an equal number of tiles T. A join operation is considered colocated if and only if it can be executed without necessitating the redistribution or broadcasting of the data slices across nodes. Formally, the colocated join condition can be expressed as:

$$\text{Colocated}(\bowtie_{i=1}^{n} R_i) = \begin{cases} 1, & \text{if } \forall i, \text{tiles in } R_i \text{ reside on a common node with matching boundaries} \\ 0, & \text{otherwise} \end{cases}$$

The colocated join operation consists of two primary steps: filtering and refinement. In the filtering step, a filter operation F is applied to each multirelation to identify candidate tiles based on a specific predicate. The filtering process is formally defined as:

$$F(R_i) = \{t \mid t \in R_i \wedge \text{predicate}(t)\}$$

where t represents a tile in multirelation R_i and predicate(t) is the condition that identifies relevant tiles (e.g., spatial overlap). This step efficiently narrows down the search space by leveraging indexes, thereby speeding up the join operation. Following the filtering, the refinement step involves applying additional predicates to the filtered tile pairs to ensure accurate results. This step is essential for resolving the details of the join and is represented as:

$$\text{Refine}(F(R_i), F(R_j)) = \{(t_i, t_j) \mid t_i \in F(R_i), t_j \in F(R_j), \text{ joinPredicate}(t_i, t_j)\}$$

where joinPredicate(t_i, t_j) represents a join condition such as `intersects` applied to the pairs of tiles (t_i, t_j) from the filtered sets of R_i and R_j respectively. These steps collectively ensure that the colocated join operation is executed efficiently, leveraging the locality of data and minimizing the need for extensive data movement across the network.

An illustrative example of a colocated join query, along with its succinct corresponding plan, is depicted in Figure 6. This figure helps to understand the execution strategy of colocated joins.

Find trips that have spatiotemporal intersection

```
SELECT a.tripId , b. tripId
FROM shipsFishing a, shipsFishing b
WHERE intersects(a. trip , b. trip )
```

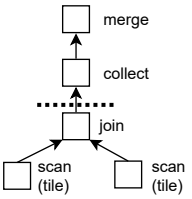


Fig. 6. Left is a self-join spatiotemporal intersection query. Right represents the corresponding query plan, where the spatiotemporal intersection is computed between fragments within a single tile, and the join operation is pushed down to the workers for execution.

5.4.5 ***Non-colocated Join.*** This strategy is essential for executing complex queries Q across multiple multirelations $R1 \bowtie_{jp_1} R2 \bowtie_{jp_2} \ldots \bowtie_{jp_n} RN$, especially when each relation is partitioned using its own Multidimensional Tiling Scheme (MTS) with a distinct set of tiles T. Such joins are characteristic for their requirement to operate across tiles distributed over various worker nodes, often necessitated by topological predicates like distance joins. The non-colocated join involves three steps: (1) reshuffling, (2) filtering, and (3) refinement. The reshuffling step involves reorganizing data across nodes to ensure related tiles from different multirelations are brought to a common computational context. The reshuffling function $\mathcal{R}$ can be expressed as:

$$\mathcal{R}(R_i, R_j) = \bigcup_{t_i \in R_i, t_j \in R_j} \text{Reshuffle}(t_i, t_j)$$

where $\text{Reshuffle}(t_i, t_j)$ aligns tiles t_i and t_j across different nodes. The reshuffling process is engineered to minimize data movement across the cluster, obviating the necessity for a comprehensive reshuffling process. In this approach, only the intersecting fragment of relation R_i with the Multidimensional Tiling Scheme (MTS_{R_j}) of relation R_j undergoes relocation. The reshuffling can be expressed as:

$$I(R_i, MTS_{R_j}) = \{r_i \mid r_i \in R_i, \exists r_j \in R_j : \text{Intersects}(r_i, MTS_{R_j}(r_j))\}$$

where $I(R_i, MTS_{R_j})$ represents the set of fragments in R_i that intersect with any part of MTS_{R_j}. This method is optimal in scenarios with significant overlap between datasets, such as in large-scale geospatial analyses involving intersecting regions. It reduces computational load and expedites the query by focusing on areas of overlap.

In heterogeneous data environments, characterized by significant variations in tiling methods between R_i and R_j, or in scenarios where the absence of statistical information, can result in inefficiencies, a different approach is required. This involves a complete reshuffling for one of the relations using the Multidimensional Tiling Scheme (MTS) of the other relation. For instance, consider a scenario where relation R_i covers a small extent while relation R_j covers a larger extent. In such cases, a complete data reshuffling for R_j is done only for the data that overlaps with the MTS of relation R_i. This approach ensures better parallelization of the processing. By aligning the tiling schemes, this method alleviates challenges associated with diverse data distributions, thereby fostering a more streamlined query execution process. The Filtering and Refinement steps are similar to what was explained in the colocated join.

For illustration, consider Figure 7, where two relations R_1 and R_2 are depicted with different tiling schemes. Consequently, tiles such as $R_1.tile_1$ and $R_2.tile_1$ may be situated on different workers.

Assume a query that joins the two relations on the intersection of their trajectories. Therefore, the query targets the spatiotemporal intersection between the tiles, rather than the entire tiles.

(a) Intersection Join **(b)** Distance Join

Fig. 7. Intersection and distance join on different MTS: The boxes illustrate tiles from the multirelations R_1 and R_2, highlighted in white, with their spatiotemporal bounding boxes defined using the MBR function. (a) Shows the intersection areas between tiles, shown in gray, which contain the data used for query processing. (b) Illustrates a distance join, where tile boundaries are expanded by the query distance, highlighted in gray with dashed borders.

As shown in left part of Figure 7, only the common part of the extents of the two tiles need to be copied.

Handling Intersection and Distance Joins: For intersection joins, the system targets the spatiotemporal intersection between the tiles. Assuming a join operation between $R_1.\text{tile}_1$ and R_2, the operation can be mathematically represented as:

$$R_1.\text{tile}_1 \bowtie R_2 = R_1.\text{tile}_1 \bowtie \bigcup_{k=1}^{T} \text{IntersectionMBRs}(R_1.\text{tile}_1, R_2.\text{tile}_k)$$

Where $\text{IntersectionMBRs}(R_1.\text{tile}_1, R_2.\text{tile}_k)$ calculates the intersecting extents between the tiles from R_1 and R_2.

In cases involving distance joins, the tiles are expanded by a given distance before assessing overlaps. This ensures that all relevant spatial interactions within the specified distance are considered. The expansion and subsequent join can be expressed as:

$$R_1.\text{tile}_1 \bowtie R_2 = R_1.\text{tile}_1 \bowtie \bigcup_{k=1}^{T} \text{ExpandedIntersectionMBRs}(R_1.\text{tile}_1, R_2.\text{tile}_k, \text{distance})$$

Here, $\text{ExpandedIntersectionMBRs}(R_1.\text{tile}_1, R_2.\text{tile}_k, \text{distance})$ denotes the operation to calculate the intersecting extents between the expanded $R_1.\text{tile}_1$ and tiles from R_2.

These strategies collectively enable the system to perform complex non-colocated joins, involving a variety of spatial, temporal, and spatiotemporal operations, across distributed nodes efficiently and accurately. For instance, consider the following query:

Find fishing ships that were within 500m of cargo ships

```
SELECT DISTINCT t1.tripId AS tripId1, t2.tripId AS tripId2
FROM shipsFishing t1, shipsCargo t2
WHERE edwithin(t1.trip, t2.trip, 500)
```

The edwithin predicate returns true when the first two arguments are closer than or equal to the distance threshold in the third parameter, here 10 meters. This query thus reports the pairs of trips that have ever been at a distance of 10 meters or less to each other. According to the non-colocated join strategy, the query planner distributes this query using the plan in Figure 8. The subplan on the right selects the subfragments in the neighbouring tiles that fall within in the given distance range. The result is then reshuffled to the corresponding workers, and the joins are computed locally.

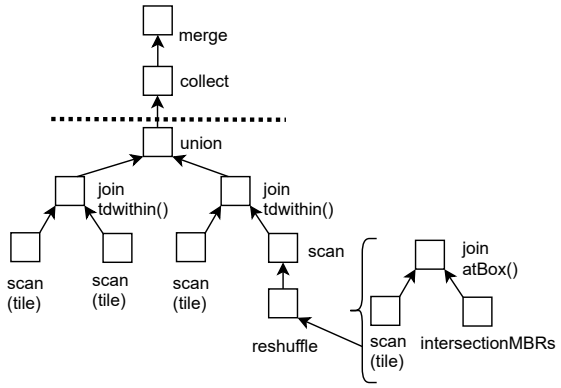


Fig. 8. Non-colocated distance joins that joins tiles with themselves and with reshuffled subfragments from the IntersectionMBRs

If the join predicate is commutative, as in this example, it suffices to copy the overlapping part from one tile to the second, but not the other way around. To achieve this optimization, the query optimizer needs to know about the commutativity of join predicates. This is decided based on the cost of the copying direction and the used predicates and operations.

5.5 Parallel Query Execution

DistMobilityDB achieves parallel query execution through a combination of local execution on shards and distributed query optimization. When a query is submitted, the system generates a distributed query plan that divides the query into sub-queries, which are then executed in parallel across different nodes. Each node executes its portion of the query independently using local spatiotemporal indexes, and intermediate results are later combined. For queries involving large joins, such as proximity joins or distance-based joins, the system parallelizes execution by ensuring that the required data is pre-distributed based on spatiotemporal locality to reduce the need for costly data reshuffling.

Moreover, the planner decides whether to perform colocated or non-colocated joins based on the query type and data distribution. Colocated joins enable queries to be executed in parallel directly on local fragments, while non-colocated joins require a reshuffling of data across nodes but still execute fragments independently in parallel. The use of background workers for asynchronous tasks such as data replication, synchronization, and maintenance further enhances the system's ability to parallelize operations.

5.6 Abstract Syntax Tree Transformation

The first step in our implementation takes the SQL query and produces its abstract syntax tree (AST) using the PostgreSQL query parser. In this step, the query is syntactically checked, and optimized with respect to the global schema in the coordinator node. Therefore, this tree does not include the notion of distribution. We enrich this AST with annotations about the distributed tables, partitioning scheme (e.g., MD tiling), replicated tables, spatiotemporal types, and operations, etc. These annotations give the following steps an understanding of the semantics of the query elements and enable the matching of the patterns that are candidate to optimization.

The annotated AST is then analyzed to match the selection and join patterns for MD tiling described in Section 5. Each query pattern produces an execution strategy composed of multiple

SQL queries to Citus. These include queries to copy the data across shards, in the case of non-colocated joins, and queries to do selection and colocated joins. The planner then delegates to Citus the distribution of each of these queries. When the planner is presented with a query that does not match any of the query patterns, it directly forwards the query to Citus to use the standard hash partitioning.

During planning, data localization logic is used to decide which tiles, hence workers, may contribute to the query result. For certain types of queries, it is possible to localize the search into a spatial, temporal, or spatiotemporal range in the multidimensional data space. This step will use the MD tiling catalogue to identify the partitions that overlap this range and the nodes that store these partitions. Only these partitions will need to be involved in the query execution. The AST tree is further annotated with this information.

Consider the non-colocated join query in Section 5.4. After the AST transformation, the planner will detect that it is a non-colocated join query and will produce Citus queries corresponding to the plan in Figure 8. The data reshuffle subplan (the rightmost one in Figure 8) will be translated into this query:

Reshuffling query in DistMobilityDB

```
WITH intersectionMBRs AS (
  SELECT t1.tile AS tile1, t2.tile AS tile2,
    intersection(t2.bbox, expand(t1.bbox, 10)) AS intersectionBox
  FROM catalogue t1, catalogue t2
  WHERE t1.tile < t2.tile AND t1.bbox && expandSpatial(t2.bbox, 500)
  ORDER BY tile1,tile2 )
INSERT INTO trips_reshuffled
SELECT t.carId, t.tripId, t.tile1, atSTBox(t.Trip, c.intersectionBox)
FROM trips t, intersectionMBRs c
WHERE t.tile = c.tile2 AND t.trip && c.intersectionBox;
```

This query will be forwarded to Citus, which will distribute it over the network nodes. The CTE expands every tile with the given distance in the user query, and produces the boxes that overlap with the neighbours. The `INSERT ... SELECT` will populate the (temporary) distributed table `trips_reshuffled` with the subfragments (`atSTBox(.,.)`) that fall in these overlap boxes. Citus will reshuffle the results using hash partitioning according to the `t.tile1` key in the SELECT clause. In this way, every tile will receive all the subfragments from its neighbouring tiles inside its partition of the `trips_reshuffled` table.

Compiled query in DistMobilityDB

```
SELECT DISTINCT T1.tripId AS tripId1, T2.tripId AS tripId2
FROM trips t1, trips t2
WHERE t1.tile = t2.tile AND t1.trip && expandSpatial(t2.trip, 500)
  AND edwithin(t1.trip, t2.trip, 500)
UNION
SELECT DISTINCT T1.tripId AS tripId1, T2.tripId AS tripId2
FROM trips t1, trips_reshuffled t2
WHERE t1.tile = t2.tile AND t1.trip && expandSpatial(T2.trip, 500)
  AND edwithin(T1.trip, T2.trip, 500);
```

The two queries in the UNION are similar to the original user query except for the additional join condition `t1.tile=t2.tile`. With this condition, Citus understands that the join is colocated and is able to distribute the query.

A shortened version of the generated query plan is as follows:

Query plan in DistMobilityDB

```
1  Distributed Query Plan (Distance/Near Join Query):
2   -> Scanning the catalog: (MD tiling)
3    -> Total number of partitions: 128
4   -> Repartitioning:
5    -> Predicate: edwithin, Type: distance, Distance: 500
6    -> Number of reshuffled tuples: 17031 over 128 shards
7   -> Merge:
8   -> Collect:
9     -> Number of Parallel Tasks: 256
10    -> Self Join:
11     -> Task 1 (WorkerNode1 - Local Plan):
12      -> Nested Loop
13       -> Seq Scan on trips_shard_1 t2
14       -> Index Scan using trips_shard_1_spgist_idx on trips_shard_1
15        -> Index Cond: (trip && expandspatial((t2.trip), 500))
16        -> Filter: (edwithin(trip, t2.trip, 500))
17    -> Neighbour Join:
18     -> Task 1 (WorkerNode1 - Local Plan):
19      -> Nested Loop
20       -> Seq Scan on neighbors_trips_shard_1 t2
21       -> Index Scan using trips_shard_1_spgist_idx on trips_shard_1
22        -> Index Cond: (trip && expandspatial((t2.trip), 500))
23        -> Filter: (edwithin(trip, t2.trip, 500))
```

The planner first scans the catalog (line 2) to get information about the distributed table scheme. Then, based on the detected distance predicate, it recognizes the query type and develops a repartitioning plan(Lines 4-6) for it. In line 9, the planner reveals that there are 256 colocated parallel queries that are conducted across all worker nodes, half of them are self joins and the others are a join between the neighbors of each shard. As shown, the plan was designed to trigger the index (Lines 14,21) by adding the necessary index predicates (Lines 15,22) to prune data before performing the expensive operations (Lines 16,23). The `expandSpatial` and `overlapping(&&)` predicates are added by the planner to verify the overlapping of the spatiotemporal boxes of t1 and t2 after expanding the trip of t2 by 500 meters. Finally, the collect operator (line 8) gathers all partial results and passes them to the Merge operator (line 7) to generate the final results.

5.7 Query Executor

The Query Executor is a pivotal component of DistMobilityDB, responsible for executing the query plans generated by the Query Planner. It operates in a distributed environment, where each worker node executes its local queries concurrently while coordinating with the coordinator node for overall query execution. The architecture of the Query Executor is designed to optimize performance through parallel processing and efficient resource management.

The execution process involves several key steps:

- Decomposition: The Query Executor receives the distributed query plan and transforms it into sub-queries that can be executed independently on each worker node.
- Local Execution: Each worker node utilizes its local resources to execute the assigned sub-queries. The executor leverages the available spatiotemporal indexes to enhance performance, particularly for spatiotemporal queries.
- Aggregation of Results: Once the sub-queries are executed, the results are collected and sent back to the coordinator node for aggregation. The coordinator node then merges the results to form the final output, ensuring that any duplicates are removed and that the results are properly formatted for the user.

In DistMobilityDB, we introduce an adaptive execution framework designed to dynamically adjust execution plans in response to fluctuating node performance. This framework enhances efficiency by reassigning queries based on node availability. For example, consider a scenario where a query is initially directed to node X. If node X is currently overwhelmed or busy, the framework proactively mitigates delays by rerouting the query. Instead of the query languishing in the execution queue of node X, the query executor smartly delegates it to an equivalent, replicated tile (i.e., secondary version) on node Y. This strategy ensures more fluid and responsive query handling, effectively optimizing resource utilization and reducing wait times in the system.

The execution plans of DistMobilityDB are designed to be adaptive, enabling real-time adjustments based on the current workload and network conditions. This adaptability is crucial for handling spatiotemporal data, which often exhibits variable density and distribution patterns. The adaptive mechanism is modeled as: $P_{adaptive} = h(P_{concurrent}, W, N)$, where h represents the adaptive function, W denotes the current workload, and N symbolizes network conditions.

Each execution plan in DistMobilityDB is thus a combination of well-coordinated global and local plans, underpinned by concurrent and adaptive mechanisms. This holistic approach ensures efficient processing of spatiotemporal queries across distributed environments, significantly improving the system's scalability and responsiveness to complex query demands.

Global Plans: On the coordinator node, global plans (P_{global}) are executed and the node is responsible for overarching tasks such as data reshuffling (R) and aggregation (A), and indexing (I). It is represented as: $P_{global} = f(R, A, I)$, where f is the function that combines reshuffling, aggregation, and indexing strategies to optimize the overall query execution.

Local Plans: On the worker nodes, local plans (P_{local}) are executed. These plans are tailored to handle specific data fragments (D_s) within each node through a specific set of tiles (D_t). The local plan for a node is expressed as: $P_{local} = g(D_s)$, where g denotes the function that processes the assigned data fragment.

Concurrent Plans: For complex queries, especially those involving spatiotemporal joins such as proximity joins, DistMobilityDB adopts a concurrent execution strategy. This involves executing local joins (J_{local}) on worker nodes simultaneously with global reshuffling (R_{global}) and aggregation processes on the coordinator node. This concurrent execution is represented as: $P_{concurrent} = J_{local} \parallel R_{global}$, where $\parallel$ signifies the concurrent execution of local and global operations.

5.8 Technical Challenges in Query Distribution

The query distribution engine in DistMobilityDB faced multiple technical challenges, particularly in distributing and optimizing spatiotemporal SQL queries across nodes. One of the most complex aspects was ensuring that query execution plans could be dynamically adjusted based on the nature of the query whether spatial, temporal, or a combination. Each query comes with two major challenges: (1) heterogeneous predicates that interact in complex ways and (2) a dynamic data context that presents data reshuffling for processing the query. Optimizing queries involving

spatiotemporal joins presented a unique challenge, as it required the system to balance between using local indexes and minimizing network communication.

While the importance of data reshuffling is clear for some queries, the process itself is fraught with challenges: (1) high dimensionality requires careful consideration of the relationships between dimensions to preserve data integrity and efficient load balancing; (2) dynamic data distribution changes as new data is introduced and queries are processed, necessitating adaptive reshuffling strategies; and (3) cross-node communication overhead arises from the need to transfer substantial amounts of data between nodes.

6 POSTGRESQL EXTENSION APIS

The PostgreSQL Extension APIs are mechanisms that allow for extending the capabilities of the PostgreSQL database system. This section explores the components that constitute our proposed PostgreSQL extension. A PostgreSQL extension comprises two essential components: a collection of SQL objects, including metadata tables, functions, and data types, and a shared library which can be dynamically loaded into a PostgreSQL server during runtime, i.e., calling `CREATE EXTENSION DistMobilityDB`. Except for the parser, all database modules within PostgreSQL are designed to be extensible. The parser, being code-generated during the build process, remains non-extensible, forcing syntactic interoperability between different extensions. Upon loading a PostgreSQL extension, it gains the ability to modify PostgreSQL's behavior by leveraging specific hooks. In the context of DistMobilityDB, the extension employs the following hooks:

Planner. This hook serves as a intercepts the default query planner, injecting custom query planning logic into the PostgreSQL database system. Through the utilization of this hook, DistMobilityDB gains the capability to dynamically shape the query planning process, presenting tailored optimization strategies for managing various spatiotemporal query scenarios within the database. Following the parsing of a query by PostgreSQL, DistMobilityDB conducts an initial check to determine if the query involves a DistMobilityDB table. Then, DistMobilityDB proceeds to generate a plan tree, representing a distributed query plan.

Executor. The hook executor in PostgreSQL is a crucial component that allows DistMobilityDB to influence the query execution process dynamically. DistMobilityDB utilizes the hook executor to optimize the execution of distributed queries related to spatiotemporal data, ensuring efficient execution of the distributed query plan and retrieval of intermediate and final results in a distributed environment.

User Defined Functions (UDFs). DistMobilityDB leverages PostgreSQL User-Defined Functions (UDFs) to enable the distribution of spatiotemporal data across a distributed environment, manipulate metadata, and facilitate remote function calls. PostgreSQL UDFs are callable from SQL queries within transactions. The distribution of data is accomplished by invoking UDFs in SQL queries, wherein the UDFs orchestrate parallel execution of desired spatiotemporal computations across the distributed architecture.

Background Workers. PostgreSQL background workers are auxiliary processes designed to execute tasks independently of the main server process, contributing to system efficiency and parallelism. In the context of DistMobilityDB, background workers play a vital role in handling asynchronous and distributed tasks related to spatiotemporal data management. These workers are leveraged by DistMobilityDB to perform background operations such as data synchronization, node availability checks, maintenance tasks, and distributed query processing. By utilizing background workers, DistMobilityDB enhances scalability and responsiveness, ensuring optimal performance

in scenarios involving extensive spatiotemporal datasets and complex distributed computing tasks within the PostgreSQL environment.

By leveraging these hooks, DistMobilityDB possesses the ability to intercept interactions between the client and the PostgreSQL engine, specifically focusing on DistMobilityDB tables. This interception empowers DistMobilityDB to selectively replace or enhance the default behavior of PostgreSQL, aligning seamlessly with the distributed functionalities and optimizations uniquely offered by DistMobilityDB.

7 EXPERIMENTAL EVALUATION

This section presents a comprehensive experimental study evaluating the performance of DistMobilityDB in comparison to other systems, utilizing both real-world and synthetic datasets. For the real dataset, we experiment using the most used query types in the domain such as range, kNN, and spatiotemporal joins either colocated or non-colocated using the `distance` and `intersects` predicates. For the synthetic dataset, we run all the BerlinMOD [12] benchmark queries. We conduct a comparison between DistMobilityDB and other major trajectory data frameworks for which the source code is available. The comparison is done using the query types supported in each system. Notice that the spatiotemporal algebra provided by MobilityDB is only partially supported in other big data frameworks. Therefore, the BerlinMOD benchmark queries will only be used for the comparison between the MD tiling and the hash partitioning, and not for the other systems as the queries contain many algebraic operations that are not supported by the other systems.

7.1 Experimental Setting

Two sets of experiments were conducted to evaluate DistMobilityDB. The first experiment was conducted in Microsoft Azure using up to eight E16s_v3 instances as worker nodes, and one instance as coordinator. Each instance has 16-vcore processors, 128 GB of RAM, and 8 TB disk. The instances run CentOS 7 with PostgreSQL 16.1, PostGIS 3.1.4, MobilityDB 1.1, and Citus 12.1. The Summit framework is installed with Hadoop 2.10.2. The Apache Sedona framework 1.5 is installed on the nodes with Spark 3.5. The comparison with Apache Sedona is limited to spatial queries as it supports only spatial data processing.

The second experiment was conducted on an on-premise cluster comprising four nodes. These nodes were used to compare DistMobilityDB with the recent distributed version of SECONDO [18]. We tried as much as we could to write the queries in Secondo as the implementation of the queries is very complicated and it needs to call many operations and to use many indexes. We provide a GitHub link[10] for those queries. Each node within the on-premise cluster featured an Intel(R) Xeon(R) CPU E5520@2.27GHz, 24GB RAM, and 500GB HDD.

7.1.1 ***Datasets***. For the synthetic dataset, we used the MobilityDB implementation[11] of the BerlinMOD [12] data generator. This generator produces realistic trajectories simulating persons' trips going from home to work in the morning, back to home in the afternoon, and leisure trips in the evenings and weekends. The data generator is configured with a scale factor parameter that generates trips for a number of simulated vehicles (e.g., cars, trucks) during a number of days. It is also a benchmark for moving object databases. It provides two sets of queries: BerlinMOD/R and BerlinMOD/N. In our experiments, we used the BerlinMOD/R queries, which contain a set of range, broadcast, colocated, and non-colocated join queries that are the main focus of this paper.

[10] https://github.com/mbakli/MobilityDb-Secondo-Queries.git
[11] https://github.com/MobilityDB/MobilityDB-BerlinMOD

Table 2. The statistics of the experiments datasets

	AIS- Real ship tracks	BerlinMOD - Synthetic
Input size	1TB	ScaleFactor:20 (100GB)
#Trajectories,#instants,#days	486.2K, 5.3Bn, 455	3M, 3.4Bn, 127
Avg instants per trajectory	10.6K	2K

For the real dataset, we used the AIS (Automatic Identification System) ship trajectory dataset obtained from the Danish Maritime Authority.[12] The AIS data includes static data such as ship name and MMSI (Maritime Mobile Service Identity), and dynamic data such as longitude, latitude, and time. Each record represents a single spatiotemporal observation. The data is stored in CSV files, where each file represents the movements of one day, and its size is about 1.7GB. The data is loaded into a distributed hash-partitioned table to speed up the preprocessing steps. We filtered out points that are outside the projection and points that have the same time for the same ship identifier. Then, the table is exported into a single CSV file to be then loaded into Summit and Apache Sedona. For MobilityDB, we construct multiple trajectories for every ship identifier based on the time gaps. These trajectories are connected by a sequence attribute for defining their order. This is done because there is no attribute in the dataset to define the end of each ship trajectory. Static data is stored once for each trajectory. Trajectories are built passing the spatiotemporal points ordered by time to the constructor function of the MobilityDB `tgeompoint` type, which then returns a spatiotemporal trajectory after doing the interpolation. The statistics about the two datasets are shown in Table 2.

7.1.2 ***Data Partitioning***. The trajectories for the two datasets in MobilityDB are stored in a big table called trips, which is partitioned twice in two separate tables using MD tiling and hash partitioning. Both partitioning methods are configured to generate the same number of partitions (shards and tiles, respectively). To speed up the spatiotemporal queries, a GiST index is built on the trip column for each local partition. The same partitioning is used four times while varying the cluster size: 32 tiles over two worker nodes, 64 tiles over 4 worker nodes, 96 tiles over 6 worker nodes, and 128 tiles over 8 worker nodes. The number of tiles is taken based on the available cores. The other lookup relations that are used in the BerlinMOD queries are replicated in all worker nodes. The ANALYSE command is used for tables to collect statistics about each column so that the planner can generate an efficient plan.

The trajectories in Summit are partitioned using two-level indexing: first temporal and then spatial. A day granularity is chosen since the trips in both datasets occurred in a small number of days, which leverages the fact that Summit filters first by time. For spatial indexing, an RTree is built for each temporal partition separately. For Apache Sedona, the trajectories are partitioned spatially using an Rtree index as Sedona only supports spatial data processing. For Secondo, the trajectories are partitioned using a spatiotemporal grid and the partitions are indexed using the Rtree index.

7.2 Range Queries

In this section, we assess the performance of range queries using three experiments, where the range type can be spatiotemporal (3D), spatial (2D), or temporal (1D). This shows us how the system behaves when we have one partitioning method for all range types. We built a function that generates nine random range sizes for every range type, 0.5%, 1%, 2%, 4%, 8%, 10%, 12%, 14%, and 16% of the data extent. Then, we randomly generate 10 ranges for each range size. The comparison is done using the following: (1) DistMobilityDB using MD-tiling, (2) Hash Partitioning in Citus, (3)

[12]https://www.dma.dk/SikkerhedTilSoes/Sejladsinformation/AIS/Sider/default.aspx

the Summit framework using its two-level indexing, and (4) Apache Sedona using the RTree index. The main query that is used in the experiment is as follows:

Find cargo ships that have passed a specific spatiotemporal range

```
SELECT tripId FROM shipsCargo
WHERE intersects(trip, givenRange)
```

where the given range can be a spatial, temporal, or spatiotemporal range according to the experiment.

In DistMobilityDB, the distributed query planner triggers the local index to prune trips that do not overlap with the range. Then the spatial or spatiotemporal range intersection will be verified with each candidate trip using the predicate *Intersects*.

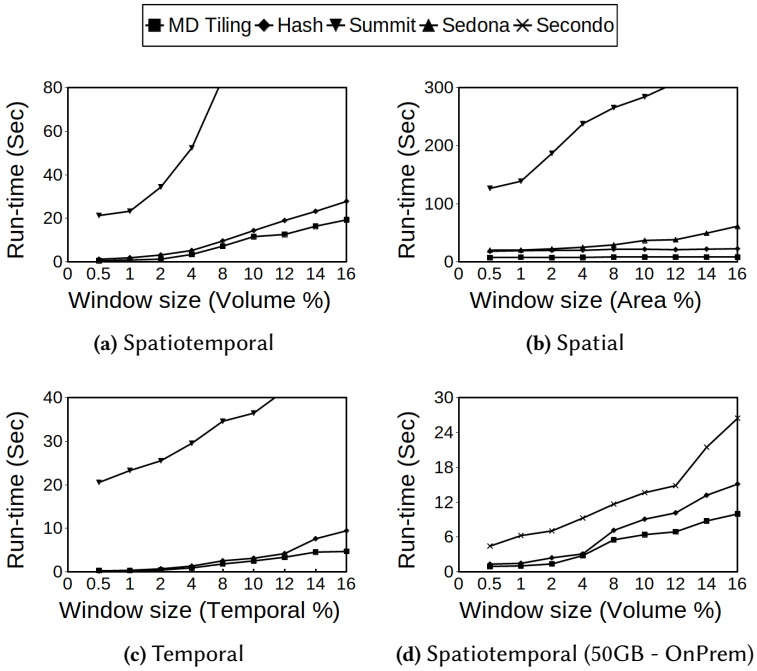


Fig. 9. Range Queries on the AIS Dataset: The x-axis represents the sizes of randomly selected query ranges, and the y-axis indicates the query runtime in seconds.

Figure 9 shows the performance of the experiments. DistMobilityDB and Citus hash partitioning perform fastest in all the runs, with DistMobilityDB slightly faster. This is owing to the fact that hash partitioning creates balanced shards based on the number of trips inside each partition rather than the number of instants. Whereas, in the AIS dataset, the number of instants per trajectory varies between a few hundred to tens of thousands, causing the shards to be unbalanced. This affects the performance when we have a costly operation that needs to be applied for each instant of each trip.

In Summit, the partitioning is done in two levels: temporal then spatial. Therefore, as expected, it performs better in temporal and spatiotemporal range queries Figures 9a, 9c. It is much slower in spatial range queries Figure 9b, because a lot of data is scanned even though most of it is not needed

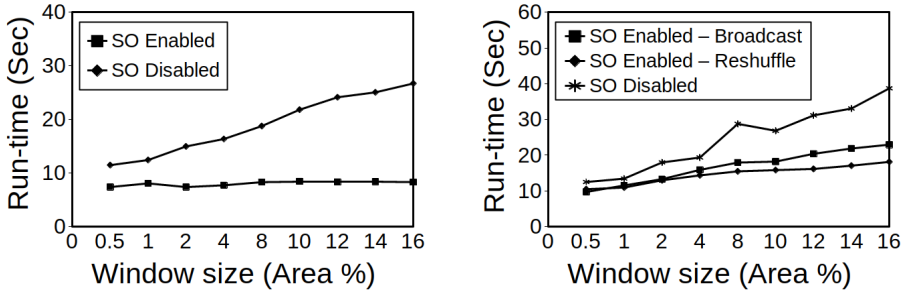
for the query. On the spark side, the run time of spatial range queries in Sedona is comparable to DistMobilityDB and Citus (slightly higher). The spatial RDD is indexed and then the index is used to perform in-memory range filtering. Figure 9d shows the performance with the distributed SECONDO. We used a smaller dataset as Secondo replicate trajectories so it consumes more storage. As shown in the figure, DistMobilityDB performs better since the spatiotemporal intersects function validates short trajectories and a small number of candidates. This is not the case in Secondo, where the intersects function is applied to a larger number of candidates and for the entire trajectory which has thousands of points.

To summarize, the DistMobilityDB gives the best performance due to the following reasons: (1) the pruning step of the query optimizer that is done on small shards extents before proceeding with the query predicates, (2) the power of the database index on local shards after the selection step, and (3) the load balancing between shards w.r.t the small number of instants.

7.3 Selection Optimization Performance

The goal of this experiment is to measure the effectiveness of the selection optimization described in Section 5.2 of the paper. We compare the performance of DistMobilityDB with and without this optimization enabled. The experimental queries are the same as Query 1 and Query 3 as described in Section 7.2 on range queries.

For Query 1, two configurations were tested: (1) Selection Optimization Enabled, where the selection operator is pushed down below the merge operator, allowing for local execution at the worker nodes; and (2) Selection Optimization Disabled, where the selection operator is processed centrally at the coordinator after data collection from the worker nodes. For Query 3, three configurations were tested: (1) Selection Optimization Enabled with Broadcast, where the planner uses the broadcast operator to share intermediate results across the hash-partitioned table; (2) Selection Optimization Enabled with Reshuffle, where the planner employs the reshuffle operator to distribute and colocate intermediate results with the hash-partitioned table; and (3) Selection Optimization Disabled, where the query is executed directly on the hash-partitioned table without leveraging fragment-level parallelism or pruning.



(a) Performance of Q1 in Section 5.2 **(b)** Performance of Q3 in Section 5.2

Fig. 10. Impact of Selection Optimization (SO) on Query Execution Time: The x-axis represents the sizes of randomly selected query ranges, and the y-axis indicates the query runtime in seconds.

As shown in Figure 10, we evaluated the impact of selection optimization on query performance using the query execution time, defined as the duration from query submission to the return of results. In Figure 10a, with selection optimization enabled, query execution times were significantly reduced compared to the non-optimized configuration. This improvement stems from partially

executing the selection operation at the worker nodes, which minimizes the volume of data transferred to the coordinator. In contrast, without optimization, all data must be collected by the coordinator before applying the selection, resulting in higher data transfer and making performance more vulnerable to factors such as network bandwidth.

Figure 10b demonstrates that Selection Optimization (SO) Enabled with Reshuffle consistently achieves the lowest runtime across most window sizes. The reshuffle operator effectively minimizes data transfer overhead and enhances parallelism, leading to faster query execution, particularly as the window size increases. The SO Enabled with Broadcast configuration performs slightly slower than the reshuffle option but still outperforms the disabled configuration. The broadcast operator distributes intermediate results across the hash-partitioned table, replicating data transferred volumes, though less efficiently than the reshuffle approach. As window size increases, requiring more data to be broadcast, the runtime remains relatively stable but slightly higher than the reshuffle configuration. In contrast, SO Disabled incurs a significantly higher runtime due to the lack of fragment-level parallelism, which increases the overhead of evaluating the `intersects` operation. This kind of topological operation is particularly costly when the input trajectory contains thousands of points, as is the case in the input dataset. In the optimized configurations, this overhead is minimized by distributing data fragments across worker nodes.

7.4 kNN Queries

There are numerous forms of kNN (k-Nearest Neighbor) query, we focus on finding the top-k trajectories that are closest to a given query geometry. This type of query is common in trajectory pattern analysis and is represented with a triple (geometry, time period, distance in meter). The geometry can be a point or trajectory. The time period and the distance are thresholds limiting the search space.

Four experiments are carried out to show the query performance using different test cases. The first experiment is a point-based kNN, where the input is a given point, time period, and k. A trajectory-based kNN is the second experiment, in which the input is a given trajectory, time period, and k. The third and the fourth experiments are similar to the first and the second experiments but spatial only. To run the experiments, we generate six random points from the extent, each with a different time period. For the trajectory-based kNN, we retrieve six random trajectories from the table. We vary the k values as follows: 5, 10, 20, 30, 40, 50, 60, 70, 80, and 90. For each k, we run the experiment using the six random values five times and take the average of the 30 runs. The query is as follows:

Find the closest K trips to a given geometry whether it is a trajectory or a point

```
SELECT tripId FROM shipsCargo
WHERE trip && givenPeriod
ORDER BY trip |=| givenGeom LIMIT K
```

The query planner of DistMobilityDB prunes trips that do not overlap with the given time period. Then, it sends the query to the candidate workers to calculate the distances in parallel. The distance for the top-k candidate trips is then reported from the workers and merged at the coordinator node. The operator | = | is used to calculate the smallest distance between each candidate trip and the given geometry.

Figures. 11a and 11b show the performance of the kNN on the point basis with and without time filter. The local index in MobilityDB plays an important role as the combination of the ORDER BY and LIMIT clauses trigger the index which helps the planner to retrieve only trajectories that can

contribute to the query results, resulting in a faster query. Moreover, since the index is used in both cases and the query reference is just a point, the time filtering predicate has little effect on performance.

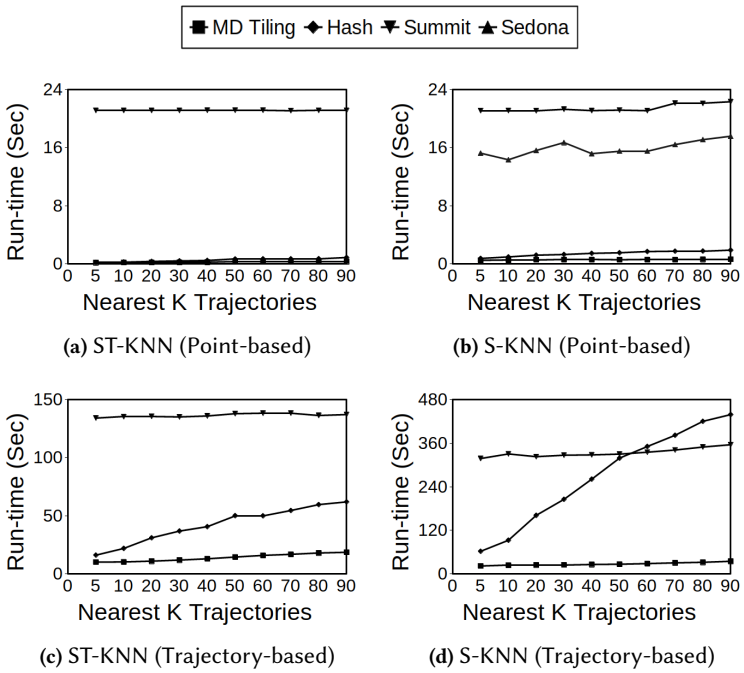


Fig. 11. kNN queries with the AIS dataset: (a) Top k-trajectories to random spatiotemporal points. (b) Same as (a) but only spatial. (c) Top k-trajectories to random spatiotemporal trajectories. (d) Same as (c) but only spatial.

In Summit, two factors cause the slower performance: (1) the overhead in starting the YARN job which executes the kNN algorithm on the candidate partitions, and (2) the spatial index inside each temporal partition retrieves more data to be scanned. The performance remains mostly unchanged regardless the value of k, which is expected as the kNN algorithm does not take time if the data is already partitioned across the mappers. Sedona outperforms Summit in this experiment due to the in-memory processing. Still, it is slightly slower than the DistMobilityDB and Citus Hash due to the YARN overhead and the garbage collection.

Figures 11c and 11d illustrate the performance of kNN queries on a trajectory basis, both with and without a time filter. The observations and explanations echo those discussed earlier for point-based kNN queries. However, it is crucial to note that calculating the distance between two trajectories is a computationally expensive operation, incurring additional costs across all systems. Notably, for large values of k, Summit outperforms Citus Hash in spatial kNN. Despite this, DistMobilityDB maintains a superior performance due to its MD-tiling strategy, which divides long trajectories into more manageable sub-trajectories with a reduced number of points. This fragmentation contributes to the efficiency of spatiotemporal query processing, making DistMobilityDB faster even in scenarios with large values of k.

7.5 Self- and Distance Join Queries

The distance-join query identifies the pairs of trips that move close with respect to a distance threshold. In this query, the user can join the distributed trips tables and verify the distance between them using the `edwithin` predicate. MD-tiling mainly targets to solve this kind of queries, because it splits the trajectories in a way that preserves the spatiotemporal proximity. The three systems: Citus Hash partitioning, Summit, and Sedona lack support for spatiotemporal distance joins.

In this experiment, we run a self-join query on the distributed trips table. We vary the distance from 500 to 4000 meters and vary the cluster size (i.e., the number of worker nodes (n) and the number of shards(s)). The goal is to show the impact of these changes on the query run-time. In most cases, the query will require data reshuffling as detailed in Section 5.4. The query is as follows:

Find fishing ships that are closer than or equal to 500 meters

```
SELECT T1.tripId Trip1ID, T2.tripId Trip2ID FROM shipsFishing T1, shipsFishing T2
WHERE edwithin(T1.trip, T2.trip, 500)
```

The experiments conducted demonstrate the impact of the parallel execution of DistMobilityDB on query performance. As shown in Figure 12a, query response times improve significantly as the number of worker nodes increases, confirming that the system efficiently distributes query workloads across nodes. The query run-time increases linearly when the search distance expands because that results in more reshuffled data, which is normally located near the neighbor boundary. This extra data does not much affect the query performance as there is a local GIST index for each shard which only retrieves a few extra candidates to be verified then by the `edwithin` predicate. The run-time shown represents the sum of the data reshuffling time and the join query processing time. Figure 12b indicates that reshuffling data takes roughly 12% of the total time, while the join query consumes the rest of the time because each shard must be joined to itself and to the data of its neighbors.

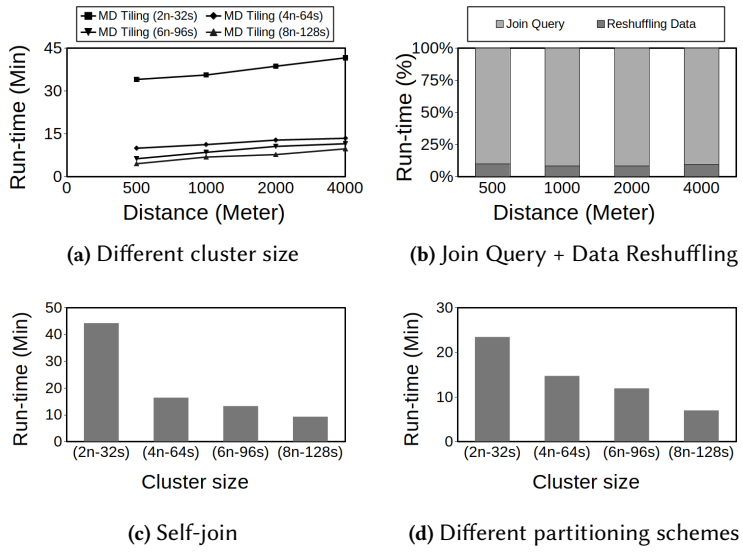
7.6 Intersection Join Queries

The intersection join query verifies the spatiotemporal intersection between two trajectories in space and time. We run two experiments using DistMobilityDB. The first is a self-join, finding the pairs of trips in the same table that intersects, e.g., accident. Thanks to MD-tiling, this is a colocated join. The planner simply broadcasts the query to all workers. In the second experiment, a join between two different distributed tables is conducted, where the two tables have different partitioning schemes. As a result, data reshuffling is always required. The query is given below:

Find ships involved in accidents through their trips

```
SELECT T1.tripId id1, T2.tripId id2 FROM shipsPassenger T1, shipsCargo T2
WHERE intersects(T1.trip, T2.trip)
```

Figure 12c illustrates the runtimes in the two experiments, varying the cluster parameters. It shows linear scalability when the cluster size increases, which is expected. Figure 12c shows the performance for the self-join query that was done on all table shards. The performance shown in Figure 12d represents the intersection between two different tables, i.e., different partitioning schemes. The query runtime represents the sum of the data reshuffling time and the join query processing time. The runtime in this second experiment is less than in the self-join experiment because the two tables are subsets of the AIS dataset: the passenger ship trips joined with the cargo ship trips. The query processing did not take much time even with the data reshuffling time as the



(a) Different cluster size **(b)** Join Query + Data Reshuffling

(c) Self-join **(d)** Different partitioning schemes

Fig. 12. Run-time of distance & intersection joins using MD tiling in the AIS dataset: (a) Varying distance and cluster size. (b) Percentage between data reshuffling and distance-join query processing. (c) Self intersection-join. (d) Intersection join on different partitioning schemas.

number of candidate trips in both tables is not too many. Therefore, the intersects predicate verifies the intersections faster. Although the predicate may take longer in case each trajectory has more instants.

7.7 BerlinMOD Benchmark

In this section, we evaluate the performance of DistMobilityDB using the BerlinMOD benchmark, a specialized benchmark designed to test MOD systems with a variety of query types. The evaluation is conducted across three dimensions. First, we assess the system's performance and scalability in different cluster configurations, showcasing its ability to handle increasing workloads with varying node counts. Second, we perform an evaluation against other state-of-the-art MOD systems, such as SECONDO, focusing on query performance. Finally, we analyze the impact of using alternative partitioning strategies on query execution efficiency.

7.7.1 Query Performance Evaluation in Different Cluster Sizes. We split the 17 BerlinMOD queries into 3 classes: broadcast joins (Q1–Q4, Q7–Q9, Q11–Q15, Q17), spatiotemporal intersection joins (Q5, Q16), and spatiotemporal distance joins (Q6, Q10). Broadcast joins are the queries that join the distributed trips relation with one or more replicated lookup relations. They can thus be answered using both the hash-partitioned and MD-tiled copies of the data. We are interested in evaluating the difference in query response time using the two partitioning methods while varying the cluster size and the number of data shards.

Figure 13a shows the runtime of the first batch of the broadcast join queries. We observe that there is no significant difference due to the partitioning method except that the spatiotemporal join operations (e.g., `contains`, `intersects`) are done faster in the MD tiling as the trajectory fragments are shorter. Figure 13b shows the runtime of the second batch that is mainly for spatial joins. We noticed that the MD tiling gives better performance when the number of tiles becomes

larger. This is because the short trajectories improve the selectivity of local indexes, despite the fact that the hash partitioned table has the same local indexes, which do not work as expected due to the long trajectories. In general, hash partitioning with long trajectories does not perform well in case the query contains expensive operations. Although the number of trajectories per partition is load-balanced, the partition size varies from one to the other since each trajectory has thousands of points, which increases the size and hence influences the loading and processing time.

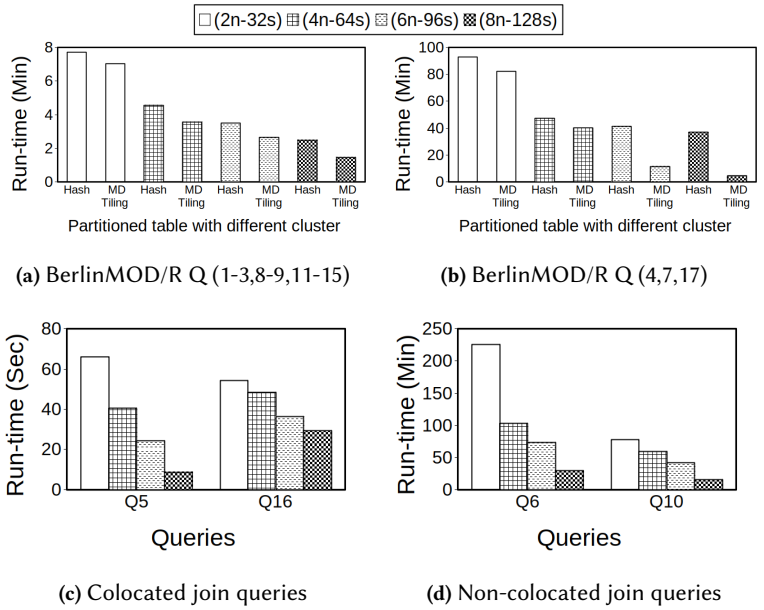


Fig. 13. BerlinMOD/R join queries when varying the cluster size and the number of tiles: (a) Run-time of all spatiotemporal, temporal, and non spatiotemporal broadcast-join queries. (b) Run-time of all spatial broadcast-join queries. (c) Spatiotemporal colocated joins using MD tiling. (d) Spatiotemporal non-colocated joins using MD tiling.

In contrast, intersection and distance joins (i.e., spatiotemporal joins) can only be distributed on the MD tiled version. We modify Q5 and Q16 in BerlinMOD and add a condition that the vehicles meet, in order to evaluate this capability in our planner. In their SQL, we add a join predicate `intersects(t1.trip, t2.trip)`, to check whether the two trips have *ever intersected*. Because intersecting pairs will always be in the same tiles, these two queries are considered colocated joins. Figure 13c shows their response time in a varying number of machines and tiles. For the other experiment, Q6 and Q10 involve a distance join, e.g, using the edwithin function. Their execution plans thus include two parts: data reshuffling between neighboring tiles, followed by colocated joins. Figure 13d illustrates their response times, varying the number of machines and tiles. In conclusion, these two experiments prove that query processing is highly scalable and can reduce the execution time by increasing the cluster size and the number of tiles.

7.7.2 Query Performance Evaluation Across MOD Systems. Figure 14 shows the run-time performance between Secondo and DistMobilityDB for some of the Berlinmod queries that we managed to build. The significant difference is in queries 4 and 6. Query 4 runs the intersects predicate and it takes more time as the trip is fully stored in one shard and it has to be checked many times as each trip is replicated into all overlapping tiles. The same case for query 6 except that it includes

a distance predicate, which will require full data reshuffling for all neighbors. As an advantage, DistMobilityDB reshuffles only the intersection part with neighbors. In conclusion, Secondo users must independently devise a query management plan for each query. This differs from DistMobilityDB, where users write SQL queries as usual, and the system automatically generates accurate and optimized distributed execution plans transparently.

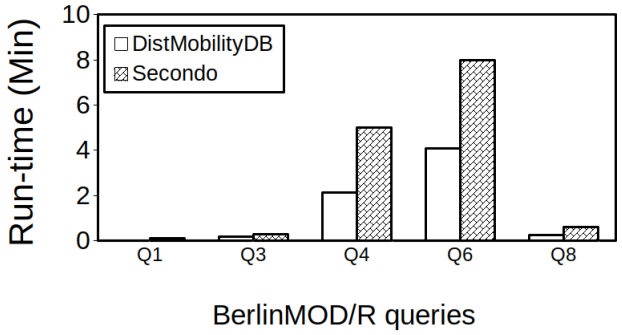


Fig. 14. Performance Comparison of BerlinMOD/R Queries with DistMobilityDB and Secondo: The x-axis represents a selected set of BerlinMOD queries, while the y-axis depicts the query runtime in seconds.

7.7.3 Query Performance Evaluation with Alternative Partitioning Strategies. This experiment evaluates the impact of different partitioning methods on query processing. We compare the MD tiling approach described in Section 4.2 with three state-of-the-art trajectory data partitioning strategies: DTJb [36], which employs an equi-depth histogram for temporal partitioning; Quadtree [30], which divides the data into spatial partitions; and Rtree [44], which creates spatiotemporal partitions. Each partition is treated as part of a multirelation, which is distributed across the cluster nodes. To ensure consistency across all partitioning strategies, each partition is indexed using the GiST index provided by MobilityDB. We used the BerlinMOD benchmark with a scale factor of 10, generating a dataset of 50 GB of trajectories. We selected a set of queries from the benchmark, including two temporal queries, two spatial queries, and two spatiotemporal queries.

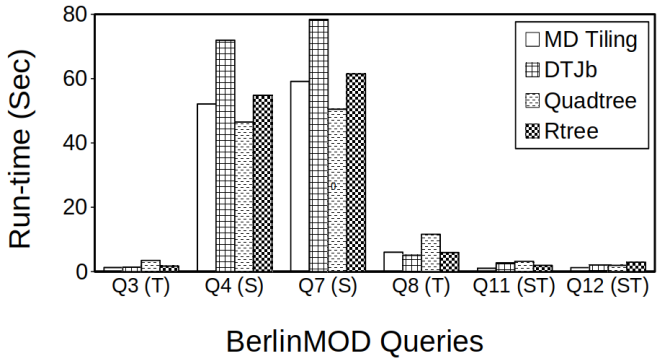


Fig. 15. Performance Comparison of BerlinMOD/R queries using Various Partitioning Strategies: The x-axis represents a set of BerlinMOD queries categorized as Temporal (T), Spatial (S), and Spatiotemporal (ST) queries, while the y-axis depicts the query runtime for each partitioning strategy.

Figure 15 presents the performance evaluation for the execution of six queries. For temporal queries (Q3 and Q8), DTJb outperforms the other methods due to its temporal partitioning, which aligns well with the nature of these queries. However, MD Tiling and Rtree also give reasonable performance, while Quadtree incurs higher execution times because its spatial partitioning results in more data being scanned. While the local index helps reduce unnecessary data processing, overlaps between the index and partitioned data can still result in processing irrelevant data.

For spatial queries (Q4 and Q7), Quadtree achieves the best performance. This is attributed to the query planner, which effectively filters data both globally and locally using the local index, narrowing down the subset of data that needs to be processed. This is particularly beneficial for the computationally expensive predicates such as the `intersects` predicate which is used in these queries. In contrast, other methods are less efficient because their local indexing retrieves a larger volume of data, increasing the overhead for executing the `intersects` predicate. For spatiotemporal queries (Q11 and Q12), the runtimes are very close across all partitioning methods, as the spatial and temporal query ranges are relatively small. Overall, in summary, this experiment shows that the data partitioning algorithm does not significantly affect the query performance. The MD Tiling algorithm presented in this paper has the benefit of being easy to implement, while similar query performance close to sophisticated Quadtree and Rtree partitioning methods.

8 CONCLUSIONS AND FUTURE WORK

This paper introduced a design for distributing spatiotemporal data, and query processing in a moving object database. Multidimensional tiling partitions the spatiotemporal space into disjoint tiles, and fragments the trajectories accordingly. It thus arranges the data in a way that preserves the spatiotemporal proximity. The paper then elaborated a concept for SQL query planning and optimization. The proposed merge operator helped split the problems of query distribution and query optimization. Considerable effort is still needed to prove the conditions upon which the merge operator is commutative and/or associative with the other operators of the multi relational algebra, considering both trajectories and other spatial and spatiotemporal data structures.

REFERENCES

[1] Louai Alarabi. 2019. Summit: A Scalable System for Massive Trajectory Data Management. *SIGSPATIAL Special* 10, 3 (2019), 2–3.

[2] Louai Alarabi, Mohamed F. Mokbel, and Mashaal Musleh. 2017. ST-Hadoop: A MapReduce Framework for Spatio-Temporal Data. In *Proceedings of the 15th International Symposium on Advances in Spatial and Temporal Databases, SSTD 2017*. Springer, Arlington, VA, USA, 84–104.

[3] Mohamed Bakli, Mahmoud Sakr, and Taysir Hassan A. Soliman. 2019. HadoopTrajectory: a Hadoop spatiotemporal data processing extension. *Journal of Geographical Systems* 21, 2 (2019), 211–235.

[4] Mohamed Bakli, Mahmoud Sakr, and Esteban Zimányi. 2019. Distributed Moving Object Data Management in MobilityDB. In *Proceedings of the 8th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial '19)*. Article 1, 10 pages.

[5] Mohamed Bakli, Mahmoud Sakr, and Esteban Zimányi. 2020. Distributed Mobility Data Management in MobilityDB. In *MDM 2020*. 238–239.

[6] Mohamed Bakli, Mahmoud Sakr, and Esteban Zimányi. 2020. Distributed Spatiotemporal Trajectory Query Processing in SQL. In *Proceedings of the 28th International Conference on Advances in Geographic Information Systems* (Seattle, WA, USA) *(SIGSPATIAL '20)*. Association for Computing Machinery, New York, NY, USA, 87–98.

[7] Mohamed S. Bakli, Mahmoud A. Sakr, and Taysir Hassan A. Soliman. 2018. A spatiotemporal algebra in Hadoop for moving objects. *Geo-spatial Information Science* 21, 2 (2018), 102–114.

[8] S. Ceri and G. Pelagatti. 1983. Correctness of Query Execution Strategies in Distributed Databases. *ACM Transactions on Database Systems* 8, 4 (1983), pp. 577–607.

[9] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. 2021. Citus: Distributed PostgreSQL for Data-Intensive Applications. In *Proc. of the 2021 International Conference on Management of Data (SIGMOD '21)*. 2490–2502.

[10] Philippe Cudre-Mauroux, Eugene Wu, and Samuel Madden. 2010. TrajStore: An adaptive storage system for very large trajectory data sets. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 109–120.

[11] X. Ding, L. Chen, Y. Gao, C.S. Jensen, and H. Bao. 2018. UlTraMan: A Unified Platform for Big Trajectory Data Management and Analytics. *Proc. of the VLDB Endowment* 11, 7 (2018), 787–799.

[12] Christian Düntgen, Thomas Behr, and Ralf Hartmut Güting. 2009. BerlinMOD: a benchmark for moving object databases. *The VLDB Journal* 18, 6 (2009), 1335–1368.

[13] Ziquan Fang, Lu Chen, Yunjun Gao, Lu Pan, and Christian S. Jensen. 2021. Dragoon: a hybrid and efficient big trajectory management system for offline and online analytics. *VLDB J.* 30, 2 (2021), 287–310.

[14] Chengxu Feng, Bing Fu, Yasong Luo, and Houpu Li. 2022. The Design and Development of a Ship Trajectory Data Management and Analysis System Based on AIS. *Sensors* 22, 1 (2022).

[15] Xuefeng Guan, Cheng Bo, Zhenqiang Li, and Yaojin Yu. 2017. ST-hash: An efficient spatiotemporal index for massive trajectory data in a NoSQL database. In *2017 25th International Conference on Geoinformatics*. 1–7.

[16] Ralf Güting, Thomas Behr, and Christian Düntgen. 2010. SECONDO: A Platform for Moving Objects Database Research and for Publishing and Integrating Research Implementations. *IEEE Data Eng. Bull.* 33 (06 2010), 56–63.

[17] Ralf Hartmut Güting, Victor Almeida, Dirk Ansorge, Thomas Behr, Zhiming Ding, Thomas Höse, Frank Hoffmann, Markus Spiekermann, and Ulrich Telle. 2005. SECONDO: An Extensible DBMS Platform for Research Prototyping and Teaching. In *Proc. of the 21st International Conference on Data Engineering, ICDE'05*. 1115–1116.

[18] Ralf Hartmut Güting, Thomas Behr, and Jan Kristof Nidzwetzki. 2021. Distributed arrays: an algebra for generic distributed query processing. *Distributed and Parallel Databases* 39 (2021), 1009–1064.

[19] International Organization for Standardization. 2023. ISO/IEC 9075:2023 – Information Technology – Database Language SQL. Available from ISO at https://www.iso.org/standard/76584.html.

[20] R. Li, H. He, R. Wang, S. Ruan, T. He, J. Bao, J. Zhang, L. Hong, and Y. Zheng. 2021. TrajMesa: A Distributed NoSQL-Based Trajectory Data Management System. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[21] R. Li, H. He, R. Wang, S. Ruan, Y. Sui, J. Bao, and Y. Zheng. 2020. TrajMesa: A Distributed NoSQL Storage Engine for Big Trajectory Data. In *Proc. of the 36th International Conference on Data Engineering*. IEEE, 2002–2005.

[22] Ruiyuan Li, Rubin Wang, Junwen Liu, Zisheng Yu, Huajun He, Tianfu He, Sijie Ruan, Jie Bao, Chao Chen, Fuqiang Gu, Liang Hong, and Yu Zheng. 2021. Distributed Spatio-Temporal k Nearest Neighbors Join. In *Proceedings of the 29th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '21)*. 435–445.

[23] ZhengYu Li and ZhuoFeng Zhao. 2021. MGeohash:Trajectory data index method based on historical data pre-partitioning. In *2021 7th International Conference on Big Data Computing and Communications (BigCom)*. 241–246.

[24] Jiamin Lu and Ralf Hartmut 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 Computer Society, Santa Clara, CA, USA, 107–25.

[25] Nehal Magdy, Mahmoud A. Sakr, and Khaled El-Bahnasy. 2017. A generic trajectory similarity operator in moving object databases. *Egyptian Informatics Journal* 18, 1 (2017), 29–37.

[26] Jan Kristof Nidzwetzki and Ralf Hartmut Güting. 2017. Distributed SECONDO: an extensible and scalable database management system. *Distributed and Parallel Databases* 35, 3–4 (2017), 197–248.

[27] Chunyao Qian, Chao Yi, Chengqi Cheng, Guoliang Pu, Xiaofeng Wei, and Huangchuang Zhang. 2019. GeoSOT-Based Spatiotemporal Index of Massive Trajectory Data. *ISPRS Int. J. Geo Inf.* 8 (2019), 284.

[28] Jiwei Qin, Liangli Ma, and Jinghua Niu. 2019. THBase: A Coprocessor-Based Scheme for Big Trajectory Data Management. *Future Internet* 11, 1 (2019), 10.

[29] S. Ray, A. Demke B., N. Koudas, R. Blanco, and A. K. Goel. 2015. Parallel in-memory trajectory-based spatiotemporal topological join. In *2015 IEEE International Conference on Big Data (Big Data)*. 361–370.

[30] Suprio Ray, Bogdan Simion, Angela Demke Brown, and Ryan Johnson. 2014. Skew-resistant parallel in-memory spatial join. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management (SSDBM '14)*. Article 6, 12 pages.

[31] Ibrahim Sabek and Mohamed F. Mokbel. 2017. On Spatial Joins in MapReduce. In *Proc. of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '17)*. Article 21, 10 pages.

[32] Mahmoud Sakr, Esteban Zimányi, Alejandro Vaisman, and Mohamed Bakli. 2023. User-centered road network traffic analysis with MobilityDB. *Transactions in GIS* 27, 2 (2023), 323–346.

[33] Zeyuan Shang, Guoliang Li, and Zhifeng Bao. 2018. DITA: A Distributed In-Memory Trajectory Analytics System. In *Proc. of the 2018 International Conference on Management of Data, SIGMOD '18* (Houston, TX, USA). ACM, 1681–1684.

[34] Dina Sharafeldeen, Mohamed Bakli, Alsayed Algergawy, and Birgitta König-Ries. 2021. ISTMINER: Interactive Spatiotemporal Co-occurrence Pattern Extraction: A Biodiversity case study. INFORMATIK 2021. , 565–579 pages.

[35] Renchu Song, Weiwei Sun, Baihua Zheng, and Yu Zheng. 2014. PRESS: A Novel Framework of Trajectory Compression in Road Networks. *Proc. VLDB Endow.* 7, 9 (may 2014), 661–672.

[36] Panogiostis Tampakis, Chirstos Doulkeridis, Nikos Pelekis, and Yannis Theodoridis. 2020. Distributed Subtrajectory Join on Massive Datasets. *ACM Transactions on Spatial Algorithms and Systems* 6, 2 (2020), 8:1–8:29.

[37] Haozhou Wang, Kai Zheng, Xiaofang Zhou, and Shazia Sadiq. 2015. SharkDB: An In-Memory Storage System for Massive Trajectory Data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15*. 1099–1104.

[38] Sheng Wang, Zhifeng Bao, J. Shane Culpepper, and Gao Cong. 2021. A Survey on Trajectory Data Management, Analytics, and Learning. *ACM Comput. Surv.* 54, 2 (2021), 36 pages.

[39] Shuang Wang and Hakan Ferhatosmanoglu. 2020. PPQ-Trajectory: Spatio-Temporal Quantization for Querying in Large Trajectory Repositories. *Proc. VLDB Endow.* 14, 2 (oct 2020), 215–227.

[40] Randall T. Whitman, Bryan G. Marsh, Michael B. Park, and Erik G. Hoel. 2019. Distributed Spatial and Spatio-Temporal Join on Apache Spark. *ACM Trans. Spatial Algorithms Syst.* 5, 1 (2019), 28 pages.

[41] Dong Xie, Feifei Li, and Jeff M. Phillips. 2017. Distributed Trajectory Similarity Search. *Proc. VLDB Endow.* 10, 11 (2017), 1478–1489.

[42] Munkh-Erdene Yadamjav, Farhana M. Choudhury, Zhifeng Bao, and Baihua Zheng. 2021. Time Period-Based Top-k Semantic Trajectory Pattern Query. In *Proc. of the 26th International Conference on Database Systems for Advanced Applications (DASFAA '21)*. pp. 439–456.

[43] Zhaojin Yan, Yijia Xiao, Liang Cheng, Rong He, Xiaoguang Ruan, Xiao Zhou, Manchun Li, and Ran Bin. 2020. Exploring AIS data for intelligent maritime routes extraction. *Applied Ocean Research* 101 (2020), 102271.

[44] Bin Yang, Qiang Ma, Weining Qian, and Aoying Zhou. 2009. TRUSTER: TRajectory Data Processing on ClUSTERs *(DASFAA '09)*. 768–771.

[45] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. 2019. Spatial Data Management in Apache Spark: The GeoSpark Perspective and Beyond. *Geoinformatica* 23, 1 (2019), pp. 37–78.

[46] Haitao Yuan and Guoliang Li. 2019. Distributed In-memory Trajectory Similarity Search and Join on Road Network. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1262–1273.

[47] Zhongwei Yue, Jingwei Zhang, Huibing Zhang, and Qing Yang. 2018. Time-Based Trajectory Data Partitioning for Efficient Range Query. In *Database Systems for Advanced Applications*. 24–35.

[48] Jinrui Zang, Pengpeng Jiao, Sining Liu, Xi Zhang, Guohua Song, and Lei Yu. 2023. Identifying Traffic Congestion Patterns of Urban Road Network Based on Traffic Performance Index. *Sustainability* 15, 2 (2023).

[49] Zhigang Zhang, Cheqing Jin, Jiali Mao, Xiaolin Yang, and Aoying Zhou. 2017. TrajSpark: A Scalable and Efficient In-Memory Management System for Big Trajectory Data. In *Proc. of the APWeb-WAIM Joint Conference on Web and Big Data* (Beijing, China). Springer, 11–26.

[50] Esteban Zimányi, Mahmoud Sakr, Arthur Lesuisse, and Mohamed Bakli. 2019. MobilityDB: A Mainstream Moving Object Database System. *Proc. of the 16th International Symposium on Spatial and Temporal Databases, SSTD* (2019).

[51] Esteban Zimányi, Mahmoud Attia Sakr, and Arthur Lesuisse. 2020. MobilityDB: A Mobility Database Based on PostgreSQL and PostGIS. *ACM Transactions on Database Systems* 45, 4 (2020), pp. 19:1–19:42.