Distributed Spatiotemporal Trajectory Query Processing in SQL

Mohamed Bakli
Université libre de Bruxelles
Brussels, Belgium
mohamed.bakli@ulb.ac.be

Mahmoud Sakr
Université libre de Bruxelles
Brussels, Belgium
Ain Shams University
Cairo, Egypt
mahmoud.sakr@ulb.ac.be

Esteban Zimányi
Université libre de Bruxelles
Brussels, Belgium
ezimanyi@ulb.ac.be

ABSTRACT

Nowadays, the collection of moving object data is significantly increasing due to the ubiquity of GPS-enabled devices. Managing and analyzing this kind of data is crucial in many application domains, including social mobility, pandemics, and transportation. In previous work, we have proposed the MobilityDB moving object database system. It is a production-ready system, that is built on top of PostgreSQL and PostGIS. It accepts SQL queries and offers most of the common spatiotemporal types and operations. In this paper, to address the scalability requirement of big data, we provide an architecture and an implementation of a distributed moving object database system based on MobilityDB. More specifically, we define: (1) an architecture for deploying a distributed MobilityDB database on a cluster using readily available tools, (2) two alternative trajectory data partitioning and index partitioning methods, and (3) a query optimizer that is capable of distributing spatiotemporal SQL queries over multiple MobilityDB instances. The overall outcome is that the cluster is managed in SQL at the run-time and that the user queries are transparently distributed and executed. This is validated with experiments using a real dataset, which also compares MobilityDB with other relevant systems.

CCS CONCEPTS

• **Information systems → Data management systems**.

KEYWORDS

MobilityDB, distributed query processing, trajectory data

ACM Reference Format:
Mohamed Bakli, Mahmoud Sakr, and Esteban Zimányi. 2020. Distributed Spatiotemporal Trajectory Query Processing in SQL. In *28th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '20), November 3–6, 2020, Seattle, WA, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3397536.3422262

1 INTRODUCTION

Due to the explosive spread of GPS-enabled devices and the popular use of cell phones, trajectory data sizes grow quickly. Spatiotemporal query serves as a basis of many services, of which processing efficiency is the key factor of the application utility. Therefore, providing scalable spatiotemporal query methods is one of the research hot spots for trajectory information processing.

MobilityDB [16] is a moving object DBMS particularly geared for managing the spatiotemporal trajectory data. It extends PostgreSQL and PostGIS with temporal and spatiotemporal database types. For instance, temporal point (`tgeompoint`) is used to represent the movement in time of a geometry point object, e.g., a vehicle. For representing its speed over time, the temporal float (`tfloat`) is used. Besides the temporal types, MobilityDB provides time types to represent the time dimension such as timestamp, timestampset, period, and periodset. The types are supported by index access methods including GiST (which is R-tree) and SP-GiST (which is an Oct-tree). MobilityDB implements more than 300 functions including distance, spatiotemporal joins, lifted operations, and temporal aggregates (more details in the Appendix). The query interface is SQL. It thus includes a query planner and optimizer. The binaries, source code, and manuals are all open source[1].

Towards addressing the challenges posed by the big trajectory data, this paper proposes a distributed version of MobilityDB.

Data Distribution Challenge. There are several challenges due to the nature of the big datasets. Some of them contain trajectories covering most of the space and time. To partition this data, the trajectory might fall into a large group of partitions, which requires either duplicating the trajectory or splitting it into pieces. The former is more straightforward in the query processing but would increase the data size, and the networking cost. The latter, on the other hand, is more efficient in terms of storage and networking, but would require to reconstruct the trajectories at the run time. Another challenge is the data skew in space, time or both. The partitioning method must adapt to the dataset characteristics[4] (e.g., its distribution) and the analysis of its dimensions[8].

Query Distribution Challenge. The goal is to keep the distribution transparent to users. So users should expect to write regular SQL queries as in the non-distributed environment. The query planner needs to be extended by understanding the semantics of spatiotemporal types and operations, and with methods to distribute them.

This paper addresses both challenges, and contributes the following:

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

- An architecture for a distributed SQL moving object database system.
- A distribution manager for transparently distributing user SQL queries.
- Supporting the common query types in the literature in an SQL database environment.
- A working implementation in the MobilityDB system.

The rest of this paper is organized as follows. Section 2 reviews the most related work from different platforms. Section 3 explains an architecture for deploying a distributed MobilityDB database on a cluster. Section 4 shows the components of the distributed storage module. For the query processing, Section 5 provides a complete picture starting from writing the query until the query execution. Section 6 evaluates the performance with a comprehensive experiments with different aspects. Extra illustrations and support material is given in the Appendix.

2 RELATED WORK

This section reviews the related work in the field of big trajectory data management.

Hadoop-based Systems. Summit [1] provides a trajectory data management on top of Hadoop. It supports hierarchical partitioning: temporal followed by spatial. The temporal partitioning is based on the time granularity and the user has to decide. The spatiotemporal points inside each temporal partition are indexed, either spatial-based or segmentation-based. Summit does not support the notion of interpolation. A trajectory is thus a set of spatiotemporal points. It supports three main query types: spatiotemporal range, join query, and kNN query. It also provides operations such as overlaps and trajectory similarity. Following the same approach of segmenting the trajectory, the work in [12] proposes an algorithm, called DTJb, where it manages the subtrajectory join query processing in two MapReduce phases. The trajectory data is partitioned into temporal partitions. A quadtree is used to partition the data spatially inside each temporal partition. Three query types are supported: trajectory, range, and join queries.

In contrast, HadoopTrajectory [3] provides a distributed query processing on continuous trajectories. It extends Hadoop with moving objects types and operations[2]. The types are supported by global and local indexes. The grid and R-tree are used to partition the trajectory data. The full trajectory is stored in one partition and is supported by spatiotemporal filter predicates such as overlaps. HadoopTrajectory provides three query types: trajectory, range and join queries. These queries are supported by trajectory processing operators in the MapReduce layer such as length, passes, and speed.

Spark-based Systems. TrajSpark [15] is a spark-based system for in-memory trajectory processing. It proposes two RDD extensions for managing trajectory segments. That is, the trajectory data points are partitioned into separate partitions according to their spatiotemporal information. These RDDs are supported by global and local index methods for improving search performance. Three main queries are proposed: single-object query, spatiotemporal-based range query, and kNN query. These queries are supported by operations such as overlaps and intersects. For the trajectory query (i.e., single-object query, the trajectory data for a specific object is collected from partitions using the object identifier and specific time range. UITrajMan [5] is also a spark-based extension for managing big trajectory data. It partitions the trajectory data using the STRPartitioner method. The system employees global and local indexing. In addition, it supports querying by trajectory identifier, range-query, and kNN query. UITrajMan supports trajectory operations such as distance and intersects. Dita [11] proposes a pivoting strategy for partitioning the trajectory data. An index, called trieIndex, is used as a global and local index. The index is built using the first, last, and pivot points of each trajectory. Four query types are supported: trajectory-ID query, range query, join query, and kNN query.

NoSQL-based Systems. TrajMesa [6] proposes a horizontal storage schema in H-store, where each trajectory is stored in one row. Additional columns are added to store the trajectory identifier and its spatiotemporal box. The data is indexed using two methods: the XZT method which is a temporal index and the XZ2 method which is a spatial index. It provides four query types: ID-temporal query, spatial range query, kNN query, and similarity query. THBase [9] provides a subtrajectory data management. It uses a hybrid grid structure for indexing the spatiotemporal information of all trajectories. The index consists of two levels: the first level is a time period index and the second level is a spatial index using the quadtree.The querying part supports: query by identifier, spatiotemporal range query, and kNN query.

Moving Object Database. SECONDO Distributed2Algebra [10] provides a set of functions for distributing data and queries across multiple SECONDO instances. One of the instances is used as a coordinator for partitioning and querying. The data distribution can be done using one of two partitioning types: object identifier (e.g., hash and range), and spatiotemporal partitioning (e.g., grid). For the grid partitioning, the trajectory is replicated in all overlapping grid cells. It provides querying such as spatiotemporal range query, join query, and trajectory query.

There are some other systems such as SharkDB [13] which provides a column-oriented trajectory data storage and in-memory processing. It proposes a time frame-based structure for storing the trajectory points into a number of frames. Each frame can be for one minute. Furthermore, the frames are compressed using the I/P frame encoding. SharkDB supports two main queries: window query and kNN query. The window query must include the time interval for better utilization of the frame-based structure. Y. Zheng in [7] proposes a cloud-based trajectory query processing framework based on Microsoft Azure. The trajectory data is stored in two locations. Azure table is used for storing the full trajectory, where each trajectory is distinguished by its ID. Azure Redis, which is a key-value store, is used to store the index. The index is built using a table-based suffix tree method that is based on the results of the map-matching algorithm. Two main queries are provided: trajectory-ID query and range query.

All the previous systems require an effort from the user to understand how to manage the queries. Moreover, most of these systems support limited trajectory operations. For more operations such as the trajectory length and speed, the user needs to write a complete MapReduce program to implement them. In addition, the distribution in SECONDO is not declarative. Therefore, this paper tries to fill these gaps between the user and the distributed systems by providing a distributed trajectory query processing in SQL.

3 SYSTEM OVERVIEW

The distributed MobilityDB database architecture is illustrated in Figure 1. It consists of a cluster of MobilityDB instances (also called nodes in the sequel), where one of them acts as the *coordinator*. A MobilityDB instance is a PostgreSQL database, that has both PostGIS and MobilityDB extensions installed. One physical machine might thus host multiple MobilityDB instances[2]. The coordinator has an additional extension called the *distribution manager*, which is the core proposal of this paper. The other nodes, called *workers*, are regular MobilityDB instances. They do not actually know that they are part of a distributed cluster. This means that besides their role as workers in the distributed cluster, they may be independently serving other DB clients. All the distribution logic is thus performed by the distribution manager in the coordinator node.

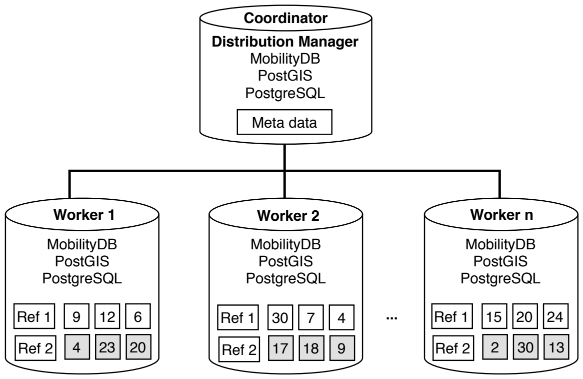


Figure 1: System Architecture

The coordinator maintains a catalogue of *metadata* about the cluster structure, and the logical and physical data partitions. The physical storage of data occurs in the worker nodes. A worker node would thus store, as database tables, partitions of the big tables, replicas of partitions that are stored in other worker nodes, and replicas of smaller *reference* tables.

Figure 2 gives a high level overview of the components of the distribution manager, which are briefly described as follows:

Distributed Storage. (Section 4) Storage is organized according to one of the proposed trajectory data partitioning methods. To partition a table, the distribution manager creates the same table schema in the coordinator and in all workers. The coordinator remains empty, and the data is distributed over the tables in the workers. The metadata about each partition is stored in the *metadata* catalog. In addition, it is possible to build a local index (R-tree, Oct-tree, or B-tree), where every worker nodes indexes only its objects. This is all realized by SQL functions that are defined in the distribution manager.

Distributed Query Processing. (Section 5) It is a pipeline of four main components: (1) A query parser for identifying query elements such as distributed table, replicated tables, spatiotemporal types (e.g.,`tpoint`), predicates (e.g.,`overlaps`, `contains`), and functions (e.g.,`length`, `atGeometry`); (2) A query designator that analyzes the query and the catalogue, and selects the workers that will be invoked during the query execution; (3) A query planner for generating a distributed query plan, according to the query predicates; and (4) Multiple query executors for tracking the execution of the distributed plan in workers, deal with node failures, and collects the results.

[2]Hosting multiple MobilityDB instances on a single machine might not be optimal w.r.t performance. The performance depends on factors like how the disk storage is organized, and the number of available cores.

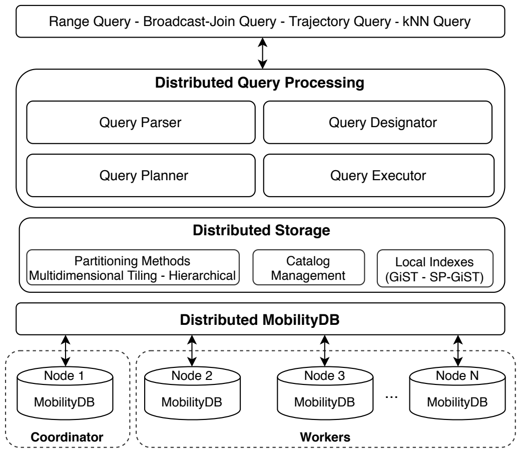


Figure 2: Distribution Manager

4 DISTRIBUTED STORAGE

Given a big trajectory table, data partitioning is critical for efficiently processing the query. The goal of the partitioning is threefold. Firstly it is desired that the partitions have similar sizes, in order to balance the load over the worker nodes. Secondly, it is desired to maintain the spatiotemporal proximity between the objects in a partition. This is specially critical to the performance of spatiotemporal joins. Finally, it is required to minimize the amount of data copying in case the partitioning needs to duplicate the data in multiple partitions. In other words, if the total data size is d, and the number of the workers is w, we want to produce w partitions, each of which has a size close to $\frac{d}{w}$, where the spatiotemporal extent of every partition is minimal.

Using own analysis, and the survey of literature, we could identify two main approaches of partitioning: hierarchical and multidimensional tiling. In hierarchical partitioning, first the spatial extent is partitioned then the temporal extent or the other way around. The latter proved more effective in the literature, e.g., [1, 12]. Multidimensional tiling partitions the n-dimensional space into grid tiles, and assigns the data objects to them, e.g., [14].

In the following, we elaborate the two methods of partitioning. The following schema will be used for illustration in the rest of the paper. The *trips* table is the one to be distributed, and it has the *trip* attribute of type temporal point. The other two are smaller reference tables that will be replicated on all worker nodes.

```
trips<tripId:int primary key, trip:tpoint>
streets<id:int, geom:geometry>
periods<id:int, timePeriod: period>
```

4.1 Hierarchical Partitioning

We use two levels of partitioning: a temporal one followed by a spatial one. In the temporal partitioning, the time extent of the dataset is split into equi-sized intervals. Trajectories are copied into all temporal partitions that overlap their time extent. Inside every temporal partition, a quadtree is employed to partition the data spatially. It is desired that the partition resulting from the two steps is fine enough, that the parts are smaller than the desired partition size, i.e., $\leq \frac{d}{w}$. Beyond this requirement, the number of temporal partitions and the parameters of the quadtree are not of big importance. For example, it would be reasonable that the overall number of splits is around 10x the number of workers. After that, the parts in every temporal partition are z-ordered, and distributed over the final partitions in a balanced way. This is iteratively done for every temporal partition. The result is that one data partition is compiled per worker node, where all partitions have similar sizes. This partitioning algorithm is implemented in the distribution manager as an SQL user function, as follows:

```
SELECT create_hierarchical_partitions(trips, trip);
```

For the reference tables, typically small sized, we want to replicate them on all workers:

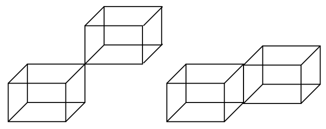
```
SELECT create_reference_table(streets);
SELECT create_reference_table(periods);
```

4.2 Multidimensional Tiling

Hierarchical partitioning works well when the data has some temporal density patterns. For example, car movement will be more dense during day, and less during night, so the temporal partitioning would be at the day granularity. For the more general case, where such a pattern can not be assumed, multidimensional tiling can adapt more to the spatiotemporal distribution of the data. Given a three or four dimensional trajectory dataset, respectively a three- or four-dimensional grid tiles are constructed to cover the whole data extent. Again it is desired that the size of every tile is smaller than the partition size.

One trajectory may overlap multiple grid tiles. Unlike hierarchical partitioning which would copy the trajectory in every overlapped partition, we split the trajectories at the tile boundaries. So if a trajectory overlaps three tiles, it will be split into three segments, and each tile will only store its respective segment.

Algorithm 1 shows the construction of the grid tiles. The first step (Lines 1-3) initializes some variables. Here, G is level 0 of the grid which contains the spatiotemporal extent of the trajectory dataset; d is the number of dimensions; m is the total number of executors in the workers; τ is the maximum number of points for all trajectories inside each grid tile. Lines 6-13 examine the number of trajectory points inside each grid tile. If it exceeds the threshold, we further split the tile into equi-sized 2^d tiles. Then the trajectory segments are re-partitioned into sub-segments according to the spatiotemporal boundaries of the new grid tiles. Each sub-segment is assigned to exactly one tile. We recursively do this step until all tiles will contain less trajectory points than the threshold. The grid tiles are assigned Z-order codes, to speed up the merging process. Then, we apply an adaptive grouping method. The goal is to generate similar sized partitions from the grid tiles while preserving spatial proximity of objects in one partition. The algorithm outputs a number of tiles that is greater than the number of workers.



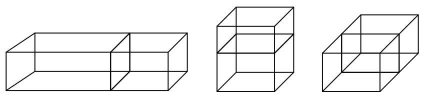


Figure 3: Merge grid tiles

Algorithm 1: Multidimensional Tiles

```
Input  : Trajectory Dataset $D$
Output : A list of disjoint tiles $G = \{G_1, G_2, \ldots, G_n\}$ of $D$ , where
         $D = Data(G_1) \cup Data(G_2) \cup \ldots \cup Data(G_n)$
1  $G \leftarrow$ Extent(D)
2  $d \leftarrow$ Dim($G$) /* either 3 (x,y,t) or 4 (x,y,z,t) */
3  $m \leftarrow$ numExecutors()
4  $\tau \leftarrow$ numPoints(D)/$m$
5  /* Segmentation Phase */
6  do
7      foreach unchecked grid tile $g \in G$ do
8          if numPoints(g) $> \tau$ then
9              Divide the grid tile into $2^d$ tiles and partition
                 trajectories $T$ into segments: $T_{s1}, T_{s2}, \ldots, T_{sk}$
                 according to the spatiotemporal bounds of the new
                 grid tiles $G_i$
10             if $T_{sk}$ is within the grid tile $G_i$ then
11                 /* Assign the trajectory segment to the tile */
12                 $G_i \leftarrow T_{sk}$
13         $G_i \leftarrow$ Mark the grid tile as a checked tile
14     $T_{max} \leftarrow$ Get the max number of trajectory points in the tiles
15 while $T_{max} > \tau$
16 /* Encoding Phase */
17 Encode grid tiles in $G$ with SFC(Z-curve)
18 Output $\leftarrow G$
```

Algorithm 2 merges the grid tiles of the previous algorithm. The output is a list of spatiotemporal partitions that are less than or equal the number of parallel workers on the cluster. Line 1 adjusts to tile capacity after each iteration. For each pair of tiles (Lines 4-10) try to merge them under the conditions: (1) the two tiles share one face. For example, if the data has three dimensions, each face can be represented using two dimensions (i.e StBox(x1,y1,x2,y2), StBox(x1,t1,x2,t2), or StBox(y1,t1,y1,t2)), (2) the total number of segment points does not exceed the threshold, and (3) none of the two tiles are used in the previous iterations. As shown in Figure 3, we do not merge tiles that share a point (a) or an edge (b) because the object moves in at least two dimensions. Therefore, we do merging if both tiles share a face as in (c). Checking if two tiles share common

Algorithm 2: Merge Grid Tiles

```
Input  : Grid tiles $G = \{G_1, G_2, \ldots, G_n\}$, Total number of points
         per tile $\tau$, Maximum number of partitions $m$
Output : A list of disjoint spatiotemporal partitions
         $S = \{S_1, S_2, \ldots, S_m\}$ of $D$, where $D = S_1 \cup S_2 \cup \ldots \cup S_m$
1  $\tau_{inc} \leftarrow$ avgNumOfPointsPerTile(G)
2  while numTiles(G) > m do
3      mergedTiles = an empty list
4      foreach grid tile $g1 \in G$, $g2 \in G$ do
5          if Pair(g1,g2) share one face & numPoints(Pair(g1,g2)) <
             $\tau$ & One of the two tiles $\notin$ mergedTiles then
6              Expand the bbox of g1 to fill the data of both tiles
7              $g1 \leftarrow$ Merge(Trajs(g1), Trajs(g2))
8              mergedTiles_i $\leftarrow g1, g2$
9      /* Increase the threshold and do another round */
10     $\tau \leftarrow \tau + \tau_{inc}$
11 Output $\leftarrow$ DataPartitions(G)
```

face is done using their Z-order codes, which is more efficient than computing their topological relationships. The segments are collected from both tiles and merged for each trajectory, then stored in the resulting tile. Line 3-9 are repeated as long as the number of tiles is greater than the number of workers.

This partitioning is defined as an SQL user function, as follows:

```
SELECT create_MDTile_partitions(trips, trip);
```

4.3 Local Index

Because the partitions are regular database tables in the worker nodes, every worker can build own indexes on its partition. MobilityDB supports three index types for spatiotemporal attributes: generalized search tree (GiST), space partitioning GiST (SP-GiST), and an R-tree for equality comparisons. The distribution manager, transparently to the user, distributes the `CREATE INDEX` SQL statement on distributed table to the worker nodes, so that each of them will create a local index.

4.4 Catalog Management

The distribution manager uses the catalogue to store metadata about the cluster nodes, distributed/replicated tables, partitions, and filtering predicates. This information helps the planner to distribute and optimize user queries. In the cluster nodes, we store the capability information for each node. For the distributed/replicated tables, we store the partitioning method information and statistics about each table such as the extent and number of partitions. Regarding the partitions information, we store data such as the node number and spatiotemporal box. Finally, we store the filtering predicates to be used by the spatiotemporal filter that are described in detail in Section 5.2.

5 DISTRIBUTED QUERY PROCESSING

The distribution manager is implemented as a wrapper around the standard PostgreSQL planner. It receives the user query, passes it to the standard parser, enriches the parse tree with tags from the distribution catalogue, designates the worker nodes that need to be invoked, produces the distributed plan, monitors the execution, and produces the final result. Currently the following classes of queries can be distributed, with the intention to support more types in the future.

- **Range Query.** It retrieves the trajectories that overlap a given query range. The range can be spatial, temporal, or spatiotemporal, for example:

```
SELECT tripId FROM trips
WHERE intersects(trip, 'Polygon((...))') AND
  trip && period '[2012-01-01, 2012-01-05)'
```

  The where clause contains two range predicates: the intersects predicate checks whether the spatiotemporal trip attribute ever intersects a given polygon region, i.e., a spatial range. The second predicate checks the temporal overlapping between the bounding box of the *trip* attribute and a given time period. In Section 5.5, the distributed execution plans for this query and the following ones will be discussed.

- **Broadcast Join Query.** It joins a distributed table with one or more reference tables, for example:

```
SELECT S.id, T.tripId
FROM trips T, streets S, periods P
WHERE intersects(T.trip, S.geom) AND
  T.trip && P.timePeriod
```

  This query joins the distributed *trips* table with the two reference (i.e., replicated) tables *streets* and *periods*, based on their spatial and temporal intersection respectively.

- **Trajectory Query.** It is a query that retrieves a trajectory object by its identifier or by another attribute filter, for example:

```
SELECT tripId, trip
FROM trips
WHERE tripId in (1, 2, 10, ...) AND
  speed(trip) ?> 25
```

  This query returns the trips in a given set of identifiers, if their speed has *ever equal been greater than* 25 m/s ($? > 25$). The challenge in this query is that in the select clause it is required to retrieve the trip attribute. If the trip attribute is segmented (i.e., multidimensional tiling partitioning is used), then the segments need to be returned from the partitions and merged in the coordinator node to construct the full trajectory.

- **kNN-Query.** It retrieves the k trajectories that have had a smallest average distance with a given trajectory, for example:

```
SELECT tripId, twAvg(trip <-> 'tpoint(...)') Dist
FROM trips
ORDER BY Dist asc
LIMIT 2
```

  The distance operator $< - >$ computes the distance between the trip attribute and a given `tpoint` object, as a temporal float. The *twAvg* function computes a time-weighted average for this distance. The combination of ORDER BY and LIMIT tells PostgreSQL that this is a kNN query, and triggers index filter.

5.1 Query Parser

The standard query parser of PostgreSQL parses and transforms the user query into a parse tree structure. However it does not know about the catalogue information that we store for the distributed tables. Therefore, we enrich the parse tree with tags that will help the planner to generate the distributed plan. For instance, the following query: "which trips reached a certain bus station at specific instant?"

```
SELECT tripId, atGeometry(trip, 'Polygon((...))')
FROM trips
WHERE intersects(
  atTimestamp(trip, timestamp '...'), 'Polygon((...))'))
```

The parse tree of this query is shown in Figure 4. The blue tags are enriched by our query parser. The tags mark the distributed tables, predicates, and arguments. They also mark the functions that operate over the spatiotemporal attributes in the distributed tables.

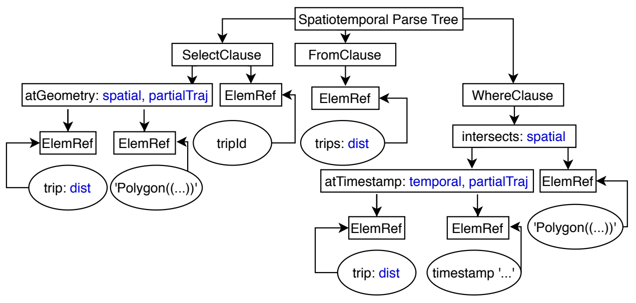


Figure 4: Query parsing example

5.2 Query Designator

It checks which workers (i.e., data partitions) may contribute to the result. This is done by inspecting the query predicates, e.g., *overlaps*(&&), *contains* (@>), *contained* (<@), and the extent of the data partitions which is stored in the catalogue. We also support spatial and temporal filtering separately based on the function arguments as most of these functions have different arguments. In other words, both of the MobilityDB geometric operators and topological functions can be used in the user query. For example, the operator *&&* is used to filter trajectories that overlaps a spatial, a temporal, or spatiotemporal object such as `tpoint`, `period`, `geometry`, `stbox`, `box2d`, etc.

5.3 Query Planner

The query planner starts from the user SQL and its enriched parse tree. It then creates an SQL statement per worker. Workers execute in parallel. The coordinator collects the worker results, and constructs the overall query results. For the currently supported types of queries (Section 5), the planner needs to analyze and distribute both the SELECT and the WHERE clauses of the user query. For the WHERE clause, the handling amounts to rewriting the table name into the name of the partition at the designated worker.

The distribution of the SELECT clause depends on the partitioning method, either or not individual trajectories are split over partitions. In the case of hierarchical partitioning, trajectories are not split. They are copied in all overlapping partitions. The same SELECT clause is then sent to workers, and the role of the coordinator is to remove the duplicate results. For the temporal aggregates, a combiner function is added to the post-processing tree, to be executed in the coordinator node on the results collected from workers.

For example, the function `tcount`, which is used to calculate at each point in time the number of defined trips, is executed as a `tcount` on the worker nodes, then as a temporal sum `tsum` combiner function in the coordinator. This mapping between the aggregate function into worker and combiner functions is stored in the catalogue.

When the distributed table is partitioned using a method that splits the trajectories (i.e., multidimensional tiling), the planner needs to take into account that worker queries process trajectory segments, instead of the whole trajectory. Thus, the query functions computed on trajectory segments distribute the functions accordingly. For example, the `cumulativeLength` function returns a `tfloat` value that represents the travelled distance, as a function in time. When the trajectory is split over multiple partitions, the cumulative length will independently be computed by workers for every segment, starting from zero. The workers will thus compute partial results, that the coordinator will need to combine and generate the final result. To achieve this, we propose a mechanism that a distributed function is defined in terms of three functions: worker, combiner, and final. The worker function executes in parallel in worker nodes, and process the trajectory segments. Both combiner and final functions execute in the coordinator, where the former combines pairs of workers results, and the latter produces the final result. As such, it is delegated to developers to define the way in which query functions are distributed. This information is stored in the distribution catalouge, and used here by the planner. A detailed example is illustrated in Appendix A.2.

If it is the case that the trajectories are segmented and need to be reconstructed, the distributed query is executed in two phases. In the first phase, the worker queries will execute the filter and the join predicates and return the object identifiers. In the second phase, the worker queries will retrieve all segments of the result trajectories. The coordinator will then merge these segments, reconstruct the trajectories, and execute the select clause of the user query. To illustrate, consider the following query which retrieves the trip trajectories that temporally overlap any of the time periods in table *periods*:

```
SELECT DISTINCT T.trip
FROM trips T, periods P
WHERE T.trip && P.timePeriod
```

Assuming that the *trips* table is partitioned by the *trip* attribute using a multidimensional tiling, the planner will generate a two phase plan as follows:

```
--Phase 1
SELECT DISTINCT T.tripId
FROM trips T, periods P
WHERE T.trip && P.timePeriod
--INTO intermediateResult

--Phase 2
SELECT tripId, merge(trip)
FROM
  (SELECT trip FROM worker1.trip
    WHERE tripId in intermediateResult)
  UNION
  ...
  UNION
  (SELECT trip FROM trip_part_n
    WHERE tripId in workern.trip)
GROUP BY tripId
```

Phase 1 query will be distributed over the workers to only evaluate the predicate and return the primary key of the *trips* table[3]. The result is stored in an intermediate structure. In the second phase, all workers are queried with this list of identifiers to return the respective trajectory segments. The coordinator will then call the *merge* function, which we implement, to concatenate the trajectory segments. The result of the second phase will finally be processed in the coordinator to produce the final query result.

5.4 Query Executor

The executor is the module monitors the running of the query on worker nodes, and the production of the final query results on the coordinator. As discussed in the previous section, the query might need to be executed in one or two phases. We choose as a design option to implement an executor per query type. That is, currently there are four executors corresponding to the four supported query types: range, broadcast join, trajectory, and kNN. This choice allows for more query-specific optimizations. Nevertheless, they all share common functions such as sending queries to individual workers, collecting back the worker results, and generating the final result in the generator. It is also the task of the query executor to manage nodes failures, but this is not implemented in the moment.

5.5 Execution Plans

This section illustrates the distributed query plan for the queries in Section 5. These plans have been generated by the PostgreSQL EXPLAIN command, on a cluster with 36 workers, hence 36 partitions $S = \{S_1, S_2, ..., S_{36}\}$. We also illustrate the effect of the different data partitioning methods on the generated plans. The number of trip segments in each partition is less than or equal to 10,000.

Range-Query. The generated plan is as follows:

```
1  Distributed MobilityDB Plan (Range Query):
2   -> Scanning the global index:(Multidimensional Tiling)
3    -> Total number of partitions:36
4     -> Filter: (shard_bbox && StBOX('Polygon((...))',
          period '[2012-01-01, 2012-01-05)'))
5      -> Partitions Removed by Filter: 27
6   -> Remove Duplicates
7    -> Number of Parallel Tasks: 9
8     -> Task 1 (WorkerNode1):
9      -> Local Query:
10      -> SELECT tripId FROM trips_shard_3
11        WHERE intersects(trip, 'Polygon((...))') AND
12              trip && period '[2012-01-01, 2012-01-05)'
13     -> Local Plan:
14       -> Index Scan using trips_shard_3_spgist_idx on
             trips_shard_3
15       -> Index Cond: (trip && StBOX)
```

The plan shows that the query is of range type, and that the data is partitioned using a multidimensional tiling. The query designator removed 27 partitions (Line 5) that do not contribute to the results. Therefore, the query is distributed to 9 partitions, each of them will receive the same query, only replacing the partition name. The worker queries will be managed independently by workers, and will run in parallel. The query of WorkerNode1 is shown in lines 8-15, the query will access the trips_shard_3 table. It replaces the table name in the user query with the corresponding partition in the worker. This query, being valid SQL, gets independently optimized by the PostgreSQL instance in the worker. The local plan that is generated at the worker is given in lines 13-15. It performs an index scan on the SP-GiST index (Line 14) to optimize the local query execution. Note that this index has been created at the coordinator, and was transparently created at the workers by the distribution manager.

[3]The planner knows about the primary key from the catalogue

Broadcast Join Query. The query plan is as follows:

```
1  Distributed MobilityDB Plan (Broadcast-Join Query):
2   -> Scanning the global index: (Hierarchical):
3    -> Total number of partitions: 36
4   -> Remove Duplicates
5    -> Number of Parallel Tasks: 36
6     -> Task 1 (WorkerNode1):
7      -> Local Query:
8       -> SELECT S.id, T.tripId FROM trips T, streets S,
            periods P WHERE intersects(T.trip,S.geom) AND T
            .trip && P.timePeriod
9      -> Local Plan:
10      -> Nested Loop Join Filter: (t.trip && (p.timePeriod
            )::stbox)
11       -> Seq Scan on periods p
12        -> Nested Loop
13         -> Seq Scan on streets S
14         ->  Index Scan using trips_shard_1_spgist_idx on
               trips_shard_1
15          -> Index Cond: (trip && S.geom)
16          -> Filter: intersects(trip, S.geom)
```

The plan detects that the query is broadcast join, and that the data is partitioned using the hierarchical partitioning. Since the tables *streets* and *periods* are reference/replicated tables, the query is simply broadcasted to all the 36 workers. The coordinator will collect the results and remove duplicates (line 4). The duplicates are caused by the hierarchical partitioning, which copies the trajectory in all overlapping partitions.

Trajectory Query. The query plan is as follows:

```
1  Distributed MobilityDB Plan (Trajectory Query):
2   -> Scanning the global index:(Multidimensional Tiling)
3    -> Total number of partitions: 36
4   -> Merge: (trip)
5     -> Merge Key: tripId
6    -> Number of Parallel Tasks: 36
7     -> Task 1 (WorkerNode1):
8      -> Local Query:
9       -> SELECT tripId, trip FROM trips_shard_4 WHERE
            tripId in (1,2,10) AND speed(trip) ?> 25
10     -> Local Plan:
11      -> Bitmap Heap Scan on trips_shard_4
12       -> Filter: (speed(trip) ?> 25)
13       -> Bitmap Index Scan on
             trips_shard_4_tripId_btree_idx on
             trips_shard_4
14        -> Index Cond: (tripId = ANY ({1,2,10}))
```

This is an example of the two phase execution plan. It is invoked here because the select clause requires to return the trip trajectory attribute, and because the data is partitioned using the multidimensional tiling method. As shown in lines 6-9, the query is sent to all partitions for obtaining all segments of the trajectory. Then in line 4, phase 2, the merge operations is invoked to reconstruct the complete trajectory. The B-tree index on tripID (Line 13) is invoked as part of the second phase of execution to speed up the search by identifier.

kNN-Query. The query plan is as follows:

```
1  Distributed MobilityDB Plan (kNN Query):
2   -> Scanning the global index: (Hierarchical):
3    -> Total number of partitions:36
4   -> Limit
5   -> Sort
6   -> Number of Parallel Tasks: 5
7    -> Task 1 (WorkerNode2):
8     -> Local Query:
9      -> SELECT tripID, twAvg(Trip <-> 'tpoint(...)')
10        FROM trips_shard_18 ORDER BY Dist asc LIMIT 2
11     -> Local Plan:
12       -> Index Scan using trips_shard_18_gist_idx on
              trips_shard_18
13       -> Order By: (trip <-> 'tpoint(...)')
```

The query is internally mapped into a kNN query, because of the planner has a special rule to understand the combination of distance, orderby, and limit as a kNN query. Since GiST and SP-GiST indexes in PostgreSQL support kNN queries, MobilityDB leverage this to support trajectory kNN. As shown in lines 12-13, the local GiST indexes will be used at workers. The the coordinator then performs the combination of sort and limit (lines 4-5), to obtain the final results.

6 EXPERIMENTS

This section presents an experimental evaluation of the proposed Distributed MobilityDB. We assess two aspects: query performance, and scalability (different cluster parameters). In addition, we provide comparisons with the state-of-the-art systems that are based on different platforms: (1) Summit [1], which is a Hadoop-based extension for trajectory data management; (2) HadoopTrajectory [3] which is also a Hadoop-based extension; (3) Dita [11] which is a Spark-based extension for spatial trajectory data analytics; and (4) SECONDO Distributed2Algebra [10], which is a distributed database algebra in the SECONDO moving object database system.

6.1 Experiment Setting

Cluster Setting. All experiments have been done on a cluster consisting of four physical machines. Every machine has: Intel(R) Xeon(R) CPU E5520@2.27GHz, 24GB RAM, 500GB HDD, 2 sockets with 4 cores and 2 threads per core, where the total number of workers is up to 16. We use 12 workers on each machine and 4 workers left for the operating system. One of the cluster machines is chosen as a coordinator. Therefore, the total number of workers on the cluster is 36 (i.e., 12 workers * 3 machines). In MobilityDB, the nodes have the same stack that consists of PostgreSQL, PostGIS, and MobilityDB. The Summit and HadoopTrajectory systems are installed on the nodes with Hadoop-2.7.3. For installing the Dita system, Apache Spark-2.2.0 is installed on all nodes. For the Distributed2Algebra, SECONDO 4.2.0 is installed on all the four machines. In all experiments with the three systems, one machine acts as the coordinator, and the remaining three machines act as workers.

Dataset. The experiments are conducted on a real publicly available AIS ship trajectory data that is provided by the Danish Maritime Authority[4]. We use two different sizes: 40GB (153M Points and 90K Trajectory), and 80GB (467M Points and 227K Trajectory).

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

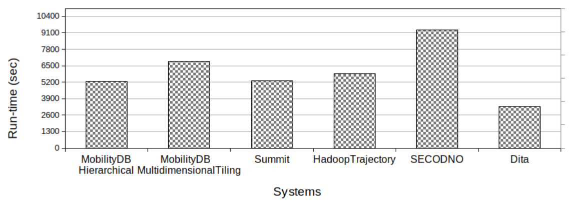


Figure 5: Data partitioning and indexing

System Setting. In MobilityDB, we experiment with the two partitioning methods: hierarchical and multidimensional tiling. The queries are executed five times and the average of the observed metrics are taken. The experiments with the other systems are done on the 40GB of the AIS data.

In Summit, we partition the data using its hierarchical partitioning method. The temporal partitioning is done at the granularity of days and the quadtree is used as the spatial index inside each day. We needed use a smaller extent than the spatial extent of the data. This is because apparently the quadtree index that Summit uses requires more memory than the cluster capabilities. After reducing the spatial extent, the index was built and stored successfully. For fairness of comparison, the reduced extent has also been loaded in MobilityDB. For executing the experiment queries, we built a python script that calls the command of every query five times.

For the HadoopTrajectory, the data was partitioned using the 3D R-tree index, and a local hash-map was used for accessing the full trajectories. The experimental queries were run using a python script, similar to the one used with Summit, modifying the query commands. In the Dita system, the data was partitioned and indexed using their pivoting strategy and a trie Index.

For the SECONDO Distributed2Algebra, the data was partitioned by 3D grid partitioning. After that, partitions were indexed using the 3D R-tree index. The experimental queries were translated into the SECONDO executable language.

In all systems, every query is executed five times, and the average runtime is recorded. Figure 5 shows the execution time for data partitioning and indexing. The multidimensional tiling partitioning in MobilityDB takes more time as it splits the trajectory multiple times. In contrast, the hierarchical partitioning does not take much time, as it deals only with the bounding box of the trips. In SECONDO, the local index took most of the execution time, because it was created on the units, i.e., trajectory segments. Partitioning part in the Dita system took most of the time as it requires to select several pivot points from each trajectory. For Summit and Hadoop-Trajectory, most of the time was taken in building the local indexes.

6.2 Query Performance Comparison

In this section, we compare the query run time in the five systems. As their capabilities are different, the comparison is based on the common query types: range query and kNN query. For the range query, we vary the range area and monitor the performance. The range type can be spatiotemporal, spatial, and temporal. We built a function that generates six random range sizes, for every range type, 0.05%, 3%, 6%, 12%, 25%, and 50% of the spatiotemporal extent of the data. For each range size, we randomly generate 5 ranges, then use them in queries to all systems. Per range size, the average of the 25 queries per system is taken.

Figures 6a, 6b, 6c show the run time comparison of range queries: spatial, temporal, and spatiotemporal respectively. MobilityDB is compared twice, using its two partitioning methods: hierarchical and multidimensional tiling. The Dita system does not have temporal query capabilities, so it appears only in the spatial range comparison. As illustrated in the three figures, MobilityDB, no matter the partitioning method, is slightly faster than SECONDO. Both are one order of magnitude faster than the two Hadoop systems. There is no significant difference in performance that can be attributed to the method of data partitioning in MobilityDB. In Summit, for the spatiotemporal and the temporal range queries, the results are quite similar because it filters data using the temporal index as a first step. Therefore, the mappers receive small part of the data. Moreover, the spatial index needs to scan more data, not like the database index. The spatial range queries are more expensive in Summit, clearly due to the hierarchical partitioning, combined with the less efficient spatial index. In general, the two Hadoop-based systems have overheads of starting the job and repartitioning the data between the mappers, which increases their run-time in disk-bound tasks. This overhead also exists in Dita, yet with a smaller value because it is a spark based, i.e., in memory. In general, they are more suited for analytic tasks that are CPU bound. The key comparison here is perhaps with SECONDO. Indeed, there is a minor performance advantage in MobilityDB, yet the main advantage is declarativeness. SECONDO uses a procedural language which is similar in its concept to map-reduce. In this work, the user interface is SQL, and the query distribution is transparently managed by a planner.

Figure 6d shows the performance of trajectory kNN queries in MobilityDB and Summit. The x-axis represents the number of random trajectories that are sampled from the input data (1, 5, 10, 20), and the k which ranges from 2 to 60. The y-axis represents the query run-time. For doing a fair comparison, we execute the kNN for each trajectory in the experiment in an individual query and we take the summation of the total for each group. This same is done in Summit, where we call the operation *dtwknn* for each trajectory in a separate command. The query performance in Distributed MobilityDB is better than Summit due to two main reasons: the local index performance of MobilityDB and the overhead caused by starting the query in Summit. This is consistent with the results obtained with range queries.

Figure 6e shows the performance of the trajectory query in MobilityDB and SECONDO, for a varying number of queried trajectories. The multidimensional tiling gives a faster performance. This is expected as the query predicates only run on smaller segments, rather than complete trajectories. The data transfer cost is also smaller. Some of this gain is lost in the trajectory merge step at the coordinator. The overall performance is still however in favor of splitting trajectories.

Figure 6f analyzes the time spent inside MobilityDB for the predicate execution, compared to the time spent in transferring the results to the coordinator. In the predicate execution, there is no data transfer from the workers to the coordinator. In contrast, when the user asks for retrieving trajectories, after the predicate is executed the data is transferred to the coordinator. The transfer cost using the multidimensional tiling partitioning is better than the hierarchical partitioning as the former stores shorter trajectory segments in workers. The coordinator node receives these short trajectories and merges them before sending the results to the user. Clearly transferring complete trajectories will consume more disk and networking time.

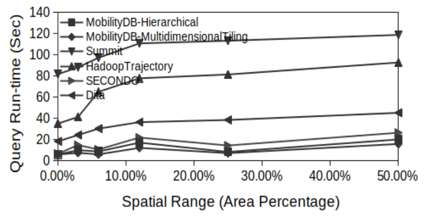
6.3 Scalability

To assess the scalability of Distributed MobilityDB, we measure the effect of changing the cluster parameters on the query performance. The cluster parameters are the number of workers, the dataset size, and the size of the replicated tables that are used in a join query. We use two kinds of queries: range and broadcast join.

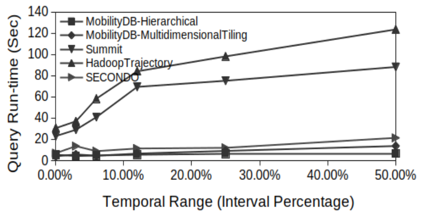
As shown in Figure 7a, we vary the number of workers (from 5 to 30) to test the performance of the spatiotemporal range query. Note that the number of workers is equal to the number of table partitions, where each worker is responsible for executing a query on one partition. We use two different ranges of the total volume: 6% and 25%. The run-time performance of the two partitioning methods improves linearly with the increase in the number of workers. The reason behind this depends on the local index size, where reducing the size of the partitions gives a higher possibility for the index to be fit in memory. Figure 7b shows the scalability wrt data size. We used two different AIS data sizes: 40 GB and 80 GB. The figure shows that multidimensional tiling partitioning consistently improves the range query performance. It also shows that the effect of the dataset size starts to be visible when the range size is big, i.e., 25% of the extent or more, which is not typical in range queries. For smaller ranges, the data size has a much smaller effect. This result is attributed to the indexes that will quickly filter the trajectories outside the range, which reduces the effect of the data size. In Figure 7c, we show the performance of the broadcast join query. The trips table is distributed and the ports table is replicated as a reference table. We test the query performance using different number of rows of the replicated table (50, 200, 400, 1000). The query returns the number of ships that visited each port, during an interval of 10 days.

7 CONCLUSION AND FUTURE WORK

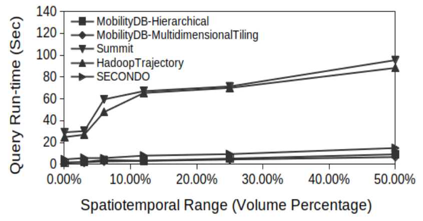
This paper presented an architecture of a distributed moving object database system and its implementation in MobilityDB. The distribution of data and queries is done by a distribution manager, which is implemented around the PostgreSQL query planner. The queries are done in SQL and the query distribution is transparent to users. The paper presented four kinds of queries that the distribution manager can currently support: range, broadcast joins, trajectory, and kNN. In the future, we will be working on supporting non-co-located spatiotemporal joins. These are the queries that require joining multiple distributed tables, where the partitions can be located in different worker nodes. This type of queries is particularly challenging as it involves data redistribution. We will also be working in methods to deal with node failures during the distributed query processing.



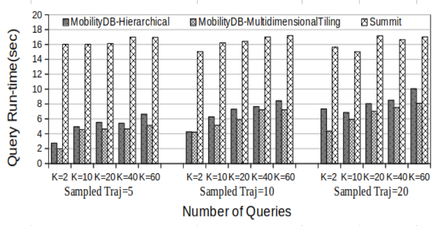
(a) Spatial Range Query



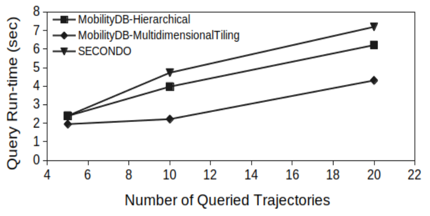
(b) Temporal Range Query



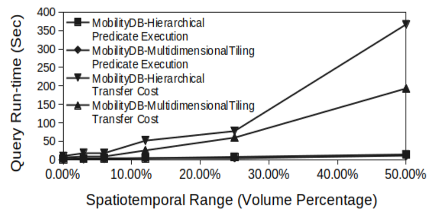
(c) Spatiotemporal Range Query



(d) kNN Query

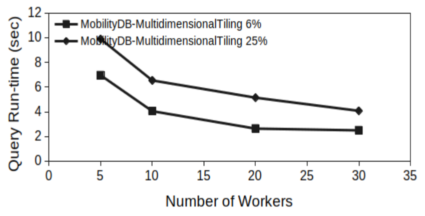


(e) Trajectory Query

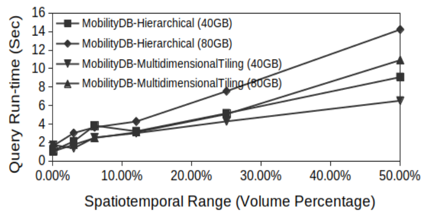


(f) Predicate Execution Vs Transfer Cost

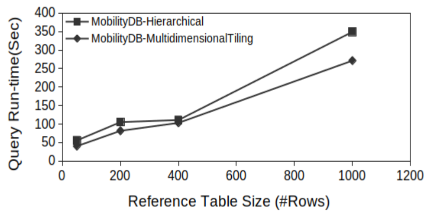
Figure 6: Performance assessment and comparison



(a) Spatiotemporal Range Query



(b) Different Dataset size



(c) Broadcast-Join Query

Figure 7: Scalability

8 ACKNOWLEDGEMENT

We acknowledge Marco Slot and the Citus engineering team for fruitful discussions around the practical aspects of the proposed system.
This work has been partially funded by Innoviris, Brussels, within the MobiPulse project.

REFERENCES

[1] L. Alarabi. 2019. Summit: A Scalable System for Massive Trajectory Data Management. *SIGSPATIAL Special* 10, 3 (2019), 2–3.
[2] M. Bakli, M. Sakr, and T.H.A. Soliman. 2018. A Spatiotemporal Algebra in Hadoop for Moving Objects. *Geo-spatial Information Science* 21, 2 (2018), 102–114.
[3] M. Bakli, M. Sakr, and T.H.A. Soliman. 2019. HadoopTrajectory: A Hadoop Spatiotemporal Data Processing Extension. *Journal of Geographical Systems* 21, 2 (2019), 211–235.
[4] Alberto Belussi, Sara Migliorini, and Ahmed Eldawy. 2018. Detecting Skewness of Big Spatial Data in SpatialHadoop. In Proc. of the 26th ACM SIGSPATIAL, New York, NY, USA, 432–435.
[5] 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.
[6] 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 ICDE*. IEEE, Dallas, TX, USA, 2002–2005.
[7] R. Li, S. Ruan, J. Bao, Y. Li, Y. Wu, L. Hong, and Y. Zheng. 2020. Efficient Path Query Processing Over Massive Trajectories on the Cloud. *IEEE Transactions on Big Data* 6, 1 (2020), 66–79.
[8] S. Migliorini, A. Belussi, E. Quintarelli, and D. Carra. 2020. A Context-based Approach for Partitioning Big Data. In Proc. of the 23rd EDBT, 431–434.
[9] J. Qin, L. Ma, and J. Niu. 2019. THBase: A Coprocessor-Based Scheme for Big Trajectory Data Management. *Future Internet* 11, 1 (2019), 10.
[10] R.H.Güting, T.Behr, and J.K.Nidzwetzki. 2020. *Distributed Arrays–An Algebra for Generic Distributed Query Processing*. Technical Report 381–05. FernUniversität in Hagen.
[11] Z. Shang, G. Li, and Z. Bao. 2018. DITA: Distributed In-Memory Trajectory Analytics. In *Proc. of SIGMOD*. ACM, Houston, TX, USA, 725–740.
[12] P. Tampakis, C. Doulkeridis, N. Pelekis, and Y. Theodoridis. 2020. Distributed Subtrajectory Join on Massive Datasets. *ACM Transaction on Spatial Algorithms Systems* 6, 2 (2020), 8:1–8:29.
[13] H. Wang, K. Zheng, J. Xu, B. Zheng, X. Zhou, and S. Sadiq. 2014. SharkDB: An In-Memory Column-Oriented Trajectory Storage. In *Proc. of the 23rd ACM International Conference on Conference on Information and Knowledge Management (CIKM '14)*. ACM, Shanghai, China, 1409–1418.
[14] Shengxun Yang, Zhen He, and Yi-Ping Phoebe Chen. 2018. GCOTraj: A Storage Approach for Historical Trajectory Datasets Using Grid Cells Ordering. *Information Sciences* 459 (2018), 1–19.
[15] 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*. Springer, Beijing, China, 11–26.
[16] E. Zimányi, M. Sakr, and A. Lesuisse. Accepted in June 2020. MobilityDB: A Mobility Database based on PostgreSQL and PostGIS. *ACM Transactions on Database Systems* (Accepted in June 2020), to appear.

A APPENDIX

A.1 MobilityDB Query Operations

MobilityDB supports a large set of query operations. In this section, we will explain the signature and usage of the query operations that were used in the queries in the paper. The trajectory data is represented using a data type, called `tgeompoint`, for temporal geometry point. It represents the continuous motion of a moving point object. That is, it linearly interpolates the location between the spatiotemporal point observations.

The function speed computes the speed of the object in meters/second, feet/second, etc, according to the spatial projection of the underlying map. The signature of the function and its output type are as follows:

```
speed(geompoint): tfloat
```

The output of the function is of type `tfloat` which represents the speed as a float value at every timestamp. It can be used in a predicate, called ? > (ever greater than), as in the trajectory query in Section 5. This predicate returns true if the `tfloat` value was greater than a given value atleast for a single time instant during the whole trip. Other similar operators are ? =, ? <, ? <>, respectively read as ever equals, ever less than, and ever not equal.

```
tfloat {?>, ?=, ?<, ?<>} numeric: bool
```

The function intersects a temporal version of the PostGIS function st_intersects. It accepts two arguments, where at least one of them is of type `tgeompoint`, and the other is either a `geometry` (e.g., polygon, linestring, etc), or a `tgeompoint`. If the case is that the two arguments are of type `geometry`, then MobilityDB delegates the execution to the PostGIS function. The signature of the function is as follows:

```
intersects({tgeompoint, geometry},
           {tgeompoint, geometry}): bool
```

The semantic is *ever intersects*. That is, the function returns true when there is a non empty intersection between the two argument at least one time instant during their definition time. Note that there is another temporal intersects function with the signature:

```
tintersects({tgeompoint, geometry},
            {tgeompoint, geometry}): tbool
```

This function returns a temporal boolean, which represents the intersection at every time instant in the common definition time of the two arguments. Similarly, there are other topological functions: contains, tcontains, touches, ttouches, etc.

The distance operator < − > is a temporally lifted version of the PostGIS distance operator. The left and right arguments can be of type `tgeompoint` or `geometry`. For example, It can be used to calculate the spatiotemporal distance between two trajectories. The signature of this operator is as follows:

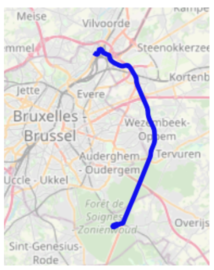
```
{tgeompoint, geometry} <-> {tgeompoint, geometry}: tfloat
```

The output is of type `tfloat` and can be used in a predicate to check a given distance threshold. It also can be passed to the function twavg, a time-weighted average. This function receives a `tfloat` and summarizes it into a `float` value. The signature is as follows:

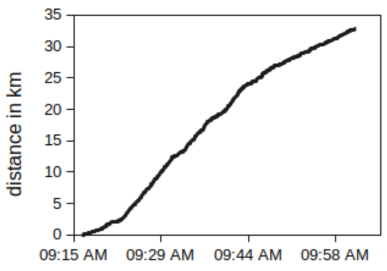
```
twAvg(tfloat): tfloat
```

A.2 Distributed Functions

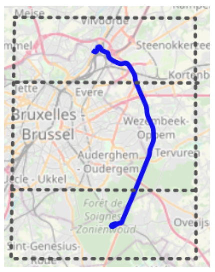
This annex illustrates the concept of distributed functions in Section 5.3. An example is illustrated in Figure 8 for the cumulativeLength function. Given the trajectory in Figure 8a, this function should return a `tfloat` value that represents the travelled distance, as a function in time (Figure 8b). When the trajectory is split over multiple partitions the cumulative length will independently be computed by workers for every segment, starting from zero. In this example, the moving object trajectory is split in three segments as in Figure 8c. Three workers will compute the three partial results in Figure 8d. The coordinator will then need to combine these partial results, and generate the final result.



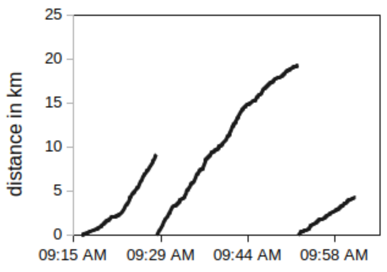
(a)



(b)



(c)



(d)

Figure 8

To achieve this, we define the cumulativeLength function as a *distributed function*. This is a mechanism that we implement in the distribution manager to help distributing queries. A distributed function is defined in terms of three functions: worker, combiner, and final. For instance, we define cumulativeLength as a distributed function, using the following statement:

```
CREATE DISTFUNCTION cumulativeLength(tgeompoint) (
        WORKERFUNC = cumulativeLength_worker,
        COMBINERFUNC = cumulativeLength_combiner,
        FINALFUNC = cumulativeLength_final);
```

The worker function cumulativeLength_worker processes the trajectory segments at worker nodes:

```
CREATE FUNCTION cumulativeLength_worker(traj tgeompoint)
  RETURNS tfloat[] AS $$
DECLARE
  result tfloat[];
BEGIN
  SELECT ARRAY(cumulativeLength(traj)) INTO result;
  RETURN result;
END;
$$ LANGUAGE PLPGSQL IMMUTABLE STRICT;
```

It computes the cumulativeLength for the given trajectory segment, and puts it into a single element SQL array. The combiner function cumulativeLength_combiner, which is evaluated in the coordinator node, will do a simple concatenation of the arrays resulting from the multiple workers, as follows:

```
CREATE FUNCTION cumulativeLength\_combiner(res1 tfloat[],
      res2 tfloat[])
  RETURNS tfloat[] AS $$
BEGIN
  RETURN array_cat(res1, res2);
END;
$$ LANGUAGE PLPGSQL IMMUTABLE STRICT;
```

The final function cumulativeLength_final will sort the array of all partial results by their start time, and merge them. Lines 13, 14 illustrate how the last value of the cumulative length of one segment is added to the next segment, in order to construct a single function out of the partial results. Line 16 calls the merge function which constructs a tfloat result from the array.

```
1  CREATE FUNCTION cumulativeLength_final(partialResults
       tfloat[])
2    RETURNS tfloat AS $$
3  DECLARE
4    cnt integer;
5    i integer;
6    len float= 0;
7    sorted tfloat[];
8    merged tfloat[] = '{}';
9  BEGIN
10   SELECT array_sort(partialResults) INTO sorted;
11   cnt= array_length(sorted, 1);
12   FOR i IN 1..cnt LOOP
13     merged= array_append(merged, arr_sort[i] + len);
14     len = endValue(merged[i]);
15   END LOOP;
16   RETURN merge(merged); //convert array into tfloat
17 END;
18 $$ LANGUAGE PLPGSQL IMMUTABLE STRICT;
```

As such, we allow for future extensiblity, where developers can add new distributed functions and define the way they will be evaluated. The list of such functions is stored in the distribution catalogue, and used by the planner. This mechanism of implementing distributed functions would work for other methods of tiling, and for multidimensional data in general. Assume that for application needs, it is required to partition data by administrative boundaries, e.g., multi-tenant applications. Then the spatial boundaries of the partitions can be described by polygons, and the data gets partitioned and split accordingly. Note that distributed functions are not relevant if the data partitioning method does not split the trajectories.