MobilityDB: A Mainstream Moving Object Database System

Esteban Zimányi
ezimanyi@ulb.ac.be
CoDE. Université Libre de Bruxelles
Belgium

Mahmoud Sakr
mahmoud.sakr@ulb.ac.be
CoDE. Université Libre de Bruxelles, Belgium
FCIS. Ain Shams University, Egypt

Arthur Lesuisse
Arthur.Lesuisse@ulb.ac.be
CoDE. Université Libre de Bruxelles
Belgium

Mohamed Bakli
mohamed.bakli@ulb.ac.be
CoDE. Université Libre de Bruxelles
Belgium

ABSTRACT

This paper demonstrates the MobilityDB moving object database system. It is an extensive implementation on top of PostgreSQL and PostGIS with multiple novel aspects. MobilityDB defines multiple spatiotemporal types for moving geometry and geography points, as well as for temporal integers, reals, Booleans, and strings. It also defines a rich set of operations on these types. The types are supported with spatiotemporal index access methods by extending GiST (Generalized Search Tree) and SP-GiST (Space Partitioning GiST). The query interface is SQL. MobilityDB thus extends the PostgreSQL optimizer with statistics collectors and selectivity estimation functions. It is available as open source. The demonstration includes a scenario with multiple queries, and a publicly accessible query interface on the Web.

CCS CONCEPTS

• **Information systems** → *Spatial-temporal systems.*

KEYWORDS

Spatiotemporal Trajectories, Mobility Analytics, Map Matching, Floating Car Data.

ACM Reference Format:
Esteban Zimányi, Mahmoud Sakr, Arthur Lesuisse, and Mohamed Bakli. 2019. MobilityDB: A Mainstream Moving Object Database System. In *16th International Symposium on Spatial and Temporal Databases (SSTD '19), August 19–21, 2019, Vienna, Austria.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3340964.3340991

1 INTRODUCTION

Moving objects are objects that change their value or location with time. These can be vehicles, persons, animals, aircarft, the air temperature of a city, the fuel price in a certain gas station, etc. The ubiquity of tracking devices and IoT technologies has resulted in collecting massive amounts of data that describe the temporal evolution of such objects and values. This creates opportunities to build applications on this data, which in turn calls for building a moving object database system (MOD). A MOD is a database system that provides the functionality of storing and querying moving object data via a query interface. Building a MOD requires extending a database system vertically at all its levels: storage, access methods, operations, SQL interface, and query optimization.

The long established research in moving object databases has so far resulted in few research prototypes, e.g., SECONDO [3] and HERMES [5]. However, an industry-scale implementation that can be used in real-world applications is not yet available. MobilityDB is envisioned to fill in this gap. It is built on top of PostgreSQL and PostGIS, which are, respectively, a widely used open source database system and its spatial database extension. These two systems are actively supported by a big community of companies and individuals. Targeting both the academia and the open source communities, we aim that MobilityDB become a mainstream MOD, open for contributions from the experts in the two sides. The current implementation is quite extensive as will be demonstrated next. It is built as a PostgreSQL extension and consists of about 66,000 lines of C code, about 18,000 lines of SQL code to define the user API, and about 31,000 lines of SQL code for testing.

2 GENERAL ARCHITECTURE

MobilityDB uses the abstract data type approach of MOD implementations [4]. In an extensible relational database system this amounts to adding user-defined types that can be used as attribute types inside relations. MobilityDB defines in PostgreSQL and PostGIS the temporal types: `tgeompoint`, `tgeogpoint`, `tfloat`, `tint`, `ttext`, and `tbool`. These types encode functions from the time domain to their corresponding base type domains. For instance, the `tgeompoint`, `tgeogpoint`, and `tfloat` types encode, respectively, functions from time to the base types `geometry(point)`, `geography(point)`, and `float`[1]. At the abstract level, these temporal functions can be of any order. At the representation level, a discrete approximation must be chosen. MobilityDB implements a novel discrete representation called the *sequence representation.* Assume a sequence of input events, as illustrated in Figure 1, where one event is a pair of a value of the base type and a time instant. The sequence representation defines a piece-wise linear function that interpolates the value between the consecutive timestamps of the input events. It can also represent gaps in the definition time, e.g., such as the gap between t5 and t6. Compared to the *sliced*

SSTD '19, August 19–21, 2019, Vienna, Austria

ACM ISBN 978-1-4503-6280-1/19/08.
https://doi.org/10.1145/3340964.3340991

[1]These base types are provided by PostgreSQL and PostGIS.

representation in [2], this representation achieves less storage and faster execution of operations.

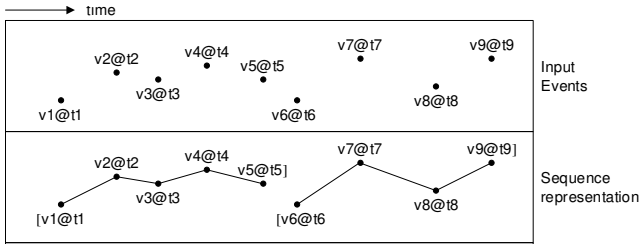

Figure 1: Sequence representation

MobilityDB leverages PostgreSQL's extensibility capabilities to support the new data types. It also leverages the features of their base types. For instance, time instants are time zone aware (a feature by PostgreSQL), while `tgeompoint` and `tgeogpoint` use the spatial framework provided by PostGIS. The same strategy is used for implementing the operations. The goal is to maximize the compatibility between MobilityDB and its underlying platform, so that it will benefit from the continuous development done by the community, which supports the goal of a mainstream MOD. For instance, PostgreSQL versions 9.6 and 10.1 came with new features that allow parallel processing of queries. Most of the MobilityDB operations that were already implemented before these versions, could make use of the new parallelization, without requiring to change their implementation. The next section explains how the different components of MobilityDB use the extensibility aspects of PostgreSQL.

3 MOBILITYDB COMPONENTS

3.1 Temporal Types

The temporal types in MobilityDB are defined in accordance with their base type and time type. As mentioned above the base types are `geography(point)`, `geometry(point)`, `float`, `int`, `text`, and `bool`. MobilityDB respectively defines the temporal types `tgeogpoint`, `tgeompoint`, `tfloat`, `tint`, `ttext`, and `tbool`. Each of them represents a function from time to its base type. On the other hand, the time type, that is used to represent the time dimension, can be `timestamp`, `timestampset`, `period`, or `periodset`. When the time type is `timestamp`, the temporal type represents a single pair of a time instant and a value of the base type. When the time type is `timestampset`, the temporal type represents a set of such pairs with distinct time instants. When the time type is `period`, the temporal type represents a continuous mapping between time instants in the period and values of the base type. Finally when the time type is `periodset`, the temporal type represents a set of such mappings with non-overlapping and non-adjacent time periods, e.g., as illustrated in Figure 1.

The aforestated specification of the time type can be explicitly set using type modifiers. This is a mechanism by PostgreSQL to specialize types into subtypes. For instance, `tgeompoint(timestampset)` is the subtype of `tgeompoint` whose time type is a `timestampset`. It represents a mapping from a set of discrete time instants into geometry points. It can be used for instance to store the set of check-ins of a foursquare user. This subtype does not interpolate the coordinates between consecutive time instants. On the other hand, the subtype `tgeompoint(period)` represents a continuous trajectory, i.e., a mapping of the time instants inside the period to geometry points that are interpolated. When no type modifier is given during declaration, the attribute can accept any subtype. In other words, the subtype will be automatically decided during the value assignment (i.e., dynamic type binding).

Clearly, the time type `periodset` generalizes the other three time types because a period set can have a single period and a period can collapse into time instant. Therefore, formally, it is enough to define subtypes for the `periodset` type modifier. While this is formally true, in practice it is beneficial to explicitly define all the subtypes for two reasons. The first is the efficiency of operations, since the algorithms for certain subtypes can be more efficient than the others. The second is the expressiveness, as the subtype declaration represents constraints, that the user can explicitly specify in the schema declaration (i.e., static type binding).

3.2 Operations on Temporal Types

MobilityDB implements over 2,300 operations on temporal types. These functions and operators are polymorphic, that is, their arguments may be of several types, and the result type may depend on the types of the arguments. When more than one temporal argument are accepted by an operation, the result is only defined on the *intersection* of their time spans. If the time spans are disjoint, then the result is null.

The operations on base types that make sense for temporal types are *lifted* [4]. The *lifting* transformation can be illustrated as:

$$(LIFT(op)(\alpha, \beta))(t) = op(\alpha(t), \beta(t))$$

where *op* denotes a static operation, *LIFT(op)* denotes its lifted counterpart, and α, β are temporal arguments. The notations $\alpha(t)$ denote the temporal functions of α, which yield its values at the given time instant. A lifted operation hence returns a temporal value. The snapshot of this temporal value at any given time instant is equal to the evaluation of the corresponding static operation on the snapshots of the temporal arguments at he same time instant. For instance, the lifted predicate $\alpha > 0$, where α here is a `tfloat` value, yields a `tbool` that is true over all the time instants/periods during which the value of α is positive, false during the rest of the definition time of α, and *undefined* otherwise.

Another class of operations, are those that only make sense for temporal arguments. These include:

trajectory(tgeompoint) → geometry(line),	**min/max(tfloat) → float**
speed(tgeompoint) → tfloat,	**atmin/atmax(tfloat) → tfloat**
cumulativeLength(tgeompoint) → tfloat,	**length(tgeompoint) → float**

There are also operations to perform comparisons and arithmetic over temporal types, to input and output temporal values, to compute the distance, and to access the properties of the temporal types. Finally, MobilityDB defines temporal aggregations such as `tmin(stream(tfloat))`, which returns a `tfloat` that represents at every time instant the minimum value of all the input. Similarly, there are aggregate operations for temporal maximum, sum, and average for temporal integers and floats, as well as temporal centroid for temporal geometry points.

3.3 Indexing

MobilityDB extends the GiST and SP-GiST indexes provided by PostgreSQL, so that they can work for the temporal types. The GiST index implements an R-tree while the SP-GiST index implements a quadtree. The storage type is always a bounding box where the dimensionality depends on the type: 1D box (i.e., a period) for `tbool` and `ttext`, 2D box (value and time) for `tint` and `tfloat`, 3D or 4D box for `tgeogpoint` and `tgeompoint` depending on whether the underling points are 2D or 3D points.

Since an SP-GiST index is a space-partitioning index, an entry can exist in only one leaf node of the index. For this reason, we transform the bounding boxes as points in a higher dimensional space to avoid overlaps. For using a point to represent a box, we basically concatenate the coordinates of the lower bound corner w.r.t all dimensions and the coordinates of the upper bound corner. The resulting point hence has double the dimensionality of the box points, e.g., a 4D point to represent the minimum and the maximum values of the 2-dimensional bounding box for the temporal types `tint` and `tfloat`.

Our spatiotemporal indexes consider as many dimensions as they are shared in the indexed column and in the query argument. For example, a spatiotemporal index on a `tgeompoint` attribute can be used to optimize an intersection predicate between this attribute and a spatial region. The access functions of the index will traverse the tree based on the spatial extent of the region argument, without requiring a temporal extent. The aim of this polymorphic behaviour of the index is to support a wider range of user queries without the need to create multiple indexes (i.e., a spatiotemporal, a spatial-only, and a temporal-only indexes). Clearly, a spatial-only index on the spatial projection of the `tgeompoint` attribute will be more efficient for such a predicate. Depending on the queries, users need to choose between maintaining such a polymorphic index, or multiple indexes.

3.4 Optimizer Statistics

The PostgreSQL planner estimates predicate selectivity in order to choose the most efficient execution plans for queries, i.e., cost-based optimization. It decides, for instance, whether to perform index scans and on which tables, whether to parallelize the query execution, and the order of computing the predicates. To this end, it collects statistics about the contents of tables using the `ANALYZE` command and stores them the `pg_statistic` catalog table. These statistics include slots for predefined values such as the percentage of nulls and the average size. They also include five generic slots for type-specific statistics, each of which can accommodate an array. The content of the five slots is decided by the developer of the user-defined type. For large tables, a random sample of the table contents is taken, rather than examining all rows. This enables large tables to be analyzed in a small amount of time.

We base the statistics collected for temporal types on those collected by PostgreSQL for scalar types, array types, and range types. For scalar types, like `float`, the following statistics are collected:

(1) fraction of values that are `NULL`,
(2) value width in bytes (for variable-length types),
(3) number of distinct values in the column,
(4) an array of most common values together with an array of their frequencies,
(5) histogram bounds of the values in the column, where the most common values are excluded,
(6) correlation between physical row ordering and logical ordering.

For array types, like `float[]`, in addition to the above, the following statistics are collected:

(7) an array of most common elements together with an array of their frequencies,
(8) a distinct element counts histogram.

For range types, like `period`, additional histograms are collected:

(9) a length histogram of all the non-empty ranges,
(10) a bounds histogram, which are actually two histograms of lower and upper bounds.

For spatial types offered by PostGIS, like `geometry(point)`:

(11) a grid of counts is collected, where a regular grid is constructed to cover the layer extent. Every grid cell stores the number of gemoetries that intersect its extent.

The statistics collected for temporal types are a combination of the above statistics (1)–(11) depending on their time type and base type. In addition to statistics (1)–(3) that are collected for all the temporal values, the following statistics are collected.

- When the time type is `timestamp`, the statistics (4) and (5) are collected for both the values and the timestamps.
- When the base type is discrete, like `tint`, the statistics (4) and (5) are collected for the values and the statistics (9) and (10) are collected for the periods.
- For `tfloat` with time type `period` or `periodset`, the statistics (9) and (10) are collected for both the float ranges and the periods.
- For the spatiotemporal types `tgeogpoint`, `tgeompoint`, the statistics (9) and (10) are collected for the periods and the grid (11) is collected for the spatial points.

Given a query predicate, and according to the attribute type, the selectivity over the time dimension, and the selectivity over the base type dimensions are estimated then multiplied. By this, we make the assumption that these dimensions are non-correlated, which is clearly a simplification that can be improved in the future. Join predicate are way more complex, and their selectivity functions are not yet implemented in MobilityDB.

4 DEMONSTRATION SCENARIO

The demonstration focuses on the expressiveness of MobilityDB. A set of representative queries based on the BerlinMOD benchmark of moving object databases [1] will be presented. The benchmark generates car trips in Berlin that simulate people movement for home, work, and leisure. The table *Cars* stores static information about the vehicles, while their spatiotemporal trips are stored in table *Trips*. One car normally has multiple trips. Tables *Points*, *Regions*, *Instants*, and *Periods* define locations and times used for expressing query conditions with respect to the trips. The tables have indexes on traditional, spatial, temporal, or spatiotemporal attributes. The data generator is controlled by a *scale factor* to control the size of the data in terms of the number of cars and simulation days.

The demonstration queries are:

(1) List the cars that have passed at a point from Points.

(2) List the cars that were within a region from Regions during a period from Periods
(3) List the pair of cars that were both located within a region from Regions during a period from Periods.
(4) List the first time at which a car visited a point in Points.
(5) Compute how many cars were active at each period in Periods.
(6) For each region in Regions, give the window temporal count of trips with a 10-minute interval.
(7) Count the number of trips that were active during each hour in May 29, 2007.
(8) List the overall traveled distances of the cars during the periods from Periods.
(9) List the minimum distance ever between each car and each point from Points.
(10) List the minimum temporal distance between pairs of cars that ever come closer than 100 meters to one another.
(11) List the nearest approach time, distance, and shortest line between each pair of trips.
(12) List when and where a pairs of cars have ever been at 10 m or less from each other.

Here, as the space is limited, we only show Query (10):

```
1  SELECT T1.CarId AS Car1Id, T2.CarId AS Car2Id,
2         tmin( distance (T1.Trip , T2.Trip )) AS MinDistance
3  FROM Trips T1, Trips100 T2
4  WHERE T1.CarId < T2.CarId
5    AND tdwithin(T1.Trip, T2.Trip , 100.0) &= true
6    AND T1.Trip && expandSpatial(T2.Trip, 100)
7  GROUP BY T1.CarId, T2.CarId
```

The *Trips100* table is a sample of 100 tuples from the *Trips* table. This sampling is done to keep the runtime of the query suitable for the demo. The query joins the *Trips* and the *Trips100* tables. The first join condition (line 4) is to ensure that the two trips are different, and that their distance is computed only once. The *tdwithin* in the second join condition (line 5) is a lifted predicate, yielding a `tbool` result, which is true whenever the distance between its first two arguments is within 100 meters, the third argument. The `&=` predicate reads as *ever equals*. Here it checks whether the `tbool` result of *tdwithin* has some true intervals/instants. The third join condition (line 6) is an overlap predicate between the bounding box of *T1.Trip* and the bounding box of *T2.Trip* expanded 100 meters from every side. This predicate is index supported. It is added to the query to trigger the optimizer to invoke the spatiotemporal index defined on T1.Trip. Finally the query computes the temporal distance between the filtered pairs of trips, and then computes for every pair of cars, the temporal minimum aggregate of the distances between their trips. This query finishes in 4.7 seconds on scale factor 0.05 (15,045 trips), in 6.6 seconds on scale factor 0.2 (62,510 trips), and in 10.4 seconds on scale factor 1 (292,940 trips).

The execution plan that PostgreSQL generates for this query is:

```
1  GroupAggregate
2    Group Key: t1 . carid , t2 . carid
3    -> Sort
4      Sort Key: t1 . carid , t2 . carid Sort Method: quicksort
5      -> Nested Loop
6        -> Seq Scan on trips100 t2
7        -> Index Scan using trips_spgist_idx on trips t1
8          Index Cond: ( trip && expandspatial(t2 . trip , 100))
9          Filter : (( carid < t2 . carid ) AND
10           (tdwithin( trip , t2 . trip , 100) &= true))
```

The spatiotemporal index scan is indicated in lines 9,10. As explained above, and as indicated in line 10, it is triggered by the bounding box overlap operator &&. The index scan result is then joined with the *Trips100* table using the other two predicates in the query. Finally the groups are constructed, and the temporal aggregation on every group is computed.

MobilityDB has been compared with SECONDO using these BerlinMOD queries, on scale factors 0.005 (1.7k trip) up to scale factor 1 (300k trips). In summary, MobilityDB came faster in 63% of the queries. The total run time of all queries in MobilityDB required 13% of the time required in SECONDO.

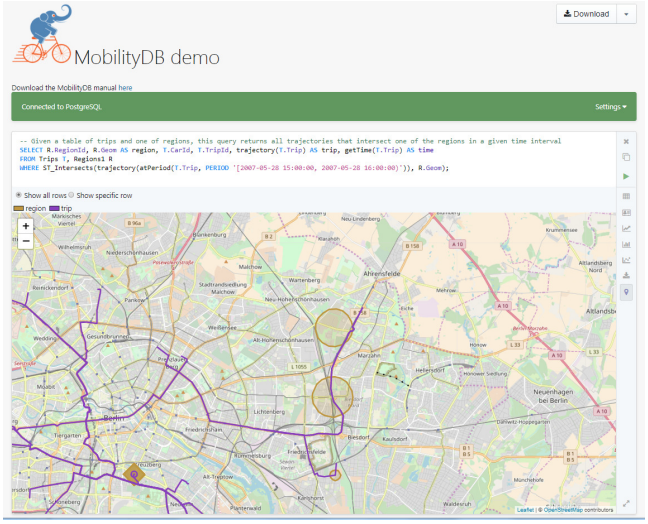

Figure 2: MobilityDB Demo Website

As shown in Figure 2, we have created a web interface[2] to write queries and visualize query results on a database with the above schema that contains the BerlinMOD benchmark data generated at scale 0.005 (1797 trips). This interface is publicly accessible.

REFERENCES

[1] Christian Düntgen, Thomas Behr, and Ralf Hartmut Güting. 2009. BerlinMOD: a benchmark for moving object databases. *The VLDB Journal* 18, 6 (2009), 1335–1368. https://doi.org/10.1007/s00778-009-0142-5
[2] Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. 2000. A data model and data structures for moving objects databases. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 319–330. https://doi.org/10.1145/342009.335426
[3] 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 *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 1115–1116.
[4] Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. 2000. A foundation for representing and querying moving objects. *ACM Trans. Database Syst.* 25, 1 (2000), 1–42. https://doi.org/10.1145/352958.352963
[5] Nikos Pelekis, Elias Frentzos, Nikos Giatrakos, and Yannis Theodoridis. 2015. HERMES: A Trajectory DB Engine for Mobility-Centric Applications. *International Journal of Knowledge-Based Organizations* 5, 2 (2015), 19–41.

[2] Available at https://github.com/ULB-CoDE-WIT/franchise-mobilitydb/