MobilityDB: A Mobility Database based on PostgreSQL and PostGIS

ESTEBAN ZIMÁNYI, Université libre de Bruxelles, Belgium
MAHMOUD SAKR, Université libre de Bruxelles, Belgium and Ain Shams University, Egypt
ARTHUR LESUISSE, Université libre de Bruxelles, Belgium

Despite two decades of research in moving object databases and a few research prototypes that have been proposed, there is not yet a mainstream system targeted for industrial use. In this paper, we present MobilityDB, a moving object database that extends the type system of PostgreSQL and PostGIS with abstract data types for representing moving object data. The types are fully integrated into the platform to reuse its powerful data management features. Furthermore, MobilityDB builds on existing operations, indexing, aggregation, and optimization framework. This is all made accessible via the SQL query interface.

CCS Concepts: • **Information systems → Database management system engines**.

Additional Key Words and Phrases: Moving Object Databases, Spatiotemporal data management, Mobility data management, SQL

ACM Reference Format:
Esteban Zimányi, Mahmoud Sakr, and Arthur Lesuisse. 2020. MobilityDB: A Mobility Database based on PostgreSQL and PostGIS. *ACM Trans. Datab. Syst.* 1, 1, Article 1 (January 2020), 43 pages. https://doi.org/10.1145/3406534

1 INTRODUCTION

Moving object data appear in many modern application domains including intelligent transportation systems, maritime safety, climate change, and autonomous vehicles. Industry and administration are on a quest to unlock its value. The right management of this data can contribute solutions to daily life problems. The advances in location tracking technologies solved the problem of data acquisition, including mass-production of GPS chips, GSM localization, automatic identification system (AIS), WiFi positioning, etc. There are however a lack of management tools, and a pressing need for them.

A moving object database (MOD), a.k.a. spatiotemporal database, provides data management for moving object data. Analogous to relational database systems, it is a middle layer between the data and the applications. Its main role is to allow the applications to manage the data using a standard query language, e.g., SQL. Research in MODs has been active since the early 2000s. There is already an extensive literature that covers the various aspects of data modeling, operations, indexing, etc. There are also a couple of research prototypes that are active in terms of new releases. Yet, a mainstream system is still missing, where by this we mean a system that builds on and exploits the functionality of widely accepted industrial-scale tools, and maximizes the reuse of their ecosystems. As a result, it becomes closer to end users, and easily adopted in the industry.

Authors' addresses: Esteban Zimányi, Université libre de Bruxelles, Belgium, ezimanyi@ulb.ac.be; Mahmoud Sakr, Université libre de Bruxelles, Belgium, Ain Shams University, Cairo, Egypt, mahmoud.sakr@ulb.ac.be; Arthur Lesuisse, Université libre de Bruxelles, Belgium, alesuiss@ulb.ac.be.

0362-5915/2020/1-ART1
https://doi.org/10.1145/3406534

In this paper we present MobilityDB, a MOD that builds on PostgreSQL and PostGIS. It extends their type system with abstract data types (ADTs) for representing moving object data. It defines, for instance, the `tgeompoint` type for representing moving geometry point objects. MobilityDB types are fully integrated into the platform to benefit from the existing data management features and the future improvements. For instance, the `tgeompoint` type builds on the PostGIS `geometry` type, restricted to 2D/3D points, and leverages its coordinate system transformation capabilities. The current implementation includes six types: temporal geometry point (`tgeompoint`), temporal geography point (`tgeogpoint`), `tint`, `tfloat`, `tbool`, and `ttext`. The type system is extensible, so that more types will be implemented in the future. These types come with a rich set of functions that can be used in SQL queries. The implementation of these functions uses the existing operations, indexing, and optimization framework. The direct result of this approach is that MobilityDB benefits from the enhancements done to PostgreSQL and PostGIS. For instance, recent versions of PostgreSQL come with new features that allow parallel processing of queries. In order to use these features in MobilityDB, we required to implement only a few functions. This effort was very small compared to an implementation of parallel queries from scratch. Finally, MobilityDB follows the ongoing OGC standards on Moving Features [11]. It is distributed as open source.[1]

The work in MobilityDB started four years ago, beginning 2016, with the vision of making the MOD research accessible to users in a mainstream DBMS: PostgreSQL and PostGIS. The practical constraints of efficiency, SQL integration, OGC standards, PostgreSQL and PostGIS integration motivated the development of new concepts that extend the state of the art. Section 2 reviews the MOD literature and system prototypes. Section 3 discusses a novel succinct representation of MOD types, the *sequence representation*. Section 4 describes the functions and operators for temporal types, and the applied algorithmic strategies. PostgreSQL and PostGIS have a big number of operations, which one would like to overload with temporal types (the temporal lifting concept). We take a disciplined approach towards temporal lifting, where an automatic algorithm is proposed in Section 4.2. Section 5 illustrates the use of the system and indicates its current performance on the 17 queries of the BerlinMOD benchmark [18]. Section 6 concludes the paper.

2 RELATED WORK

In its very beginning, research in spatiotemporal databases has separately studied the spatial and the temporal aspects. The general idea was to achieve spatiotemporal data management by crossing spatial and temporal data management. Accordingly, several proposals extended spatial databases with versioning as a mean of representing the temporal evolution (e.g., [40]). Other proposals went for extending temporal databases with spatial data support. Accordingly, spatial entities could be tagged with time instants or intervals that represent their valid or transaction time. Either of these two approaches can describe discrete movements, where the spatial data evolve in a sequence of discrete changes. For instance, TGRASS [24], a temporal extension to GRASS GIS, combines time with 2D and 3D spatial fields to provide spatiotemporal functionality. Spatial fields are tagged with timestamps, and organized as snapshots into space-time fields. Pelekis et al. [46] provide an extensive review of the early spatiotemporal database models.

Motivated by the need to represent the continuous evolution of moving objects, another line of research emerged centered around modeling spatiotemporal entities as first class citizens of the database. One notable example is based on the constraint database model. Constraint databases naturally support multidimensional data by adding constraints that reference all dimensions. Using a set of linear constraints, a multidimensional object can be described as an infinite set of points. Grumbach et al. [26] introduced a computationally efficient solution that performs relational algebra

[1] https://github.com/ULB-CoDE-WIT/MobilityDB

operations on spatiotemporal data represented using constraints. The complexity of performing queries over 3D objects, i.e., 2D space and 1D time, is reduced to a separated application of operations on the space and time components. As a proof of concept, the DEDALE system has implemented this model [25].

On the other hand, the abstract data type (ADT) approach implements types and operations for spatiotemporal data in extensible database systems. This is the approach that is followed in this work. Since ADTs encapsulate object attributes and functions, it is possible to compute sophisticated operations beyond the capabilities of the constraint database model. The MOD model that we adopt is the one proposed in [29]. There, Güting et al. have proposed an *abstract model* for representing moving objects. This model has been realized by a *discrete model* in [22], the so-called *sliced representation*, which has been implemented in the SECONDO system [27]. We propose a novel discrete model, called the *sequence representation*. As will be shown in Section 4.4, it achieves a smaller storage size of temporal data and a faster executions of the query operations.

In the literature, there are two main research prototypes for moving object databases: SECONDO [27] and HERMES [45]. SECONDO is the most extensive prototype by far and it is still being actively maintained. It is an open-source extensible database system that reuses the file systems of Berkeley DB and Cassandra, otherwise all the remaining functionality is implemented from scratch. It consists of three modules: the kernel, the optimizer, and the GUI. The kernel includes an extensible list of algebras, each defining database types and operators. One of them is the Temporal Algebra, which defines the types and operations of moving objects in [22, 29]. The *mpoint* type represents a moving point object, and the *mregion* represents a moving region. Additionally, the types *mreal*, *mbool*, etc. are defined to represent properties of moving objects, such as speed. SECONDO also includes algebras for spatial and spatiotemporal indexes such as RTree, TBTree, MMRTree, and MONTree. The kernel can be queried using a procedural language, which remains the main interface for using SECONDO. The optimizer module accepts an SQL like syntax. Finally, the GUI module offers an interface for visualizing the moving objects in a movie style.

For big data management, SECONDO provides two algebras. Parallel SECONDO [36] allows for building a cluster of standard SECONDO nodes, one of which is playing the role of a master. The communication between these nodes is done via Hadoop. Selection and transformation queries that are run on a tuple-by-tuple basis are simply split over the cluster nodes, as well as the data. Yet for joins, special parallel join operators have been implemented. Distributed SECONDO [42] follows a similar concept, yet without using Hadoop. Cassandra is used as a storage layer, where data is split into small units of work that are assigned to query processing nodes. The query processing is done using standard SECONDO.

HERMES [45] also follows the ADT approach of moving object databases. It integrates with OGC-compliant ORDBMS, currently with Oracle and PosgreSQL. HERMES type system extends on the base types, the time point and interval types, and the OGC geometry types (point, line, segment, and rectangle). For the spatiotemporal types, the sliced representation of [22] is adopted. The query language is SQL-like, with different syntax between the Oracle and the PostgreSQL implementations. It has been used to prototype many research works including the TB-tree for indexing moving point trajectories [48] and kNN queries [23].

Complementary to the SQL systems that have been surveyed above, several works have proposed extensions to Hadoop and Spark to manipulate moving object data in the context of NoSQL systems. UlTraMan [16] aims at building an in-memory trajectory data management system. It integrates Spark with Chronicle Map, and sketches an architecture that can accommodate trajectory types, indexes, and functions. SharkDB [54] proposes the novel idea of segmenting the trajectory points of the whole database into fixed time frames. Every frame is then considered a virtual column, and stored consecutively in main memory. Further ideas of compressed delta encoding and efficient

scans are discussed. HadoopTrajectory [5] is a Hadoop extension that adds spatiotemporal types and operators to the Hadoop core, so that they can be used to write MapReduce programs. Similar Hadoop extensions are ST-Hadoop [2] and Summit [1]. A similar line of work has also been done on Spark. TrajSpark [31] introduces an RDD structure to represent trajectory data in memory. These Hadoop and Spark extensions typically use a global index in the master node to distribute only the relevant data over the workers, thus reducing the data distribution overhead.

A lot of work exists that target specific aspects of MODs, e.g., modeling, operations, OLAP, indexing, visualization, standards, etc. A data model for moving objects on networks is given in [17, 30], where the coordinates are relative to the underlying network, e.g., a road identifier attached with the distance from the starting point of the road. Another data model that can represent both outdoor as well as indoor movements has been proposed in [55]. Complementary to these models, semantically enriched trajectories have been modeled in [43]. From an application perspective, in [44] was proposed a conceptual model for spatiotemporal applications called MADS (for Modeling Application Data with Spatiotemporal features). Similar to ER modeling and UML, MADS provides a simple yet expressive set of constructs for designing the applications. It is divided into four modeling dimensions; the data structures, the spatial, the temporal, and the multirepresentation features. MADS also provides a query language and data manipulation constructs. Different kinds of operations for moving objects have been studied including nearest neighbor search [28], spatiotemporal pattern matching [49, 50], and shortest path search [52]. OLAP querying on moving object data is studied in [53]. To this end, moving object data is defined as measures in a data warehouse fact table. Accordingly, it is possible to compute OLAP operations such as slice, dice, and rollup on moving object trajectories and their temporal attributes such as speed, distance to other objects, etc. The index structures and access methods of spatiotemporal data have been thoroughly surveyed in three parts [37, 38, 41] covering, respectively, the works before 2003, between 2003 and 2010, and between 2010 and 2017. V-Analytics (a.k.a. CommonGIS) [4] incorporates various visualization and data transformations techniques for analyzing spatiotemporal data. Many of these techniques are described in [3]. V-Analytics has tools to derive thematic attributes from trajectories, extract events, simplify trajectories, perform aggregations, interactive visual filtering of data according to their spatial, temporal, and thematic (attribute) components, and specific operations for movement data such as clustering of trajectories.

Both ISO and OGC have published standards for *moving features*. The ISO 19141 [34] schema for moving features defines a representation of a feature that moves as a point or as a rigid body. This schema has been adopted by OGC, and used as a basis for multiple exchange formats: OGC 18-075 for XML encoding [14], OGC 14-084r2 for CSV encoding [12], and OGC 19-045r3 for JSON encoding [15]. Additionally, the OGC 16-120r3 standard [13] defines and API for moving features access, that includes three types of operations: retrieval of feature attributes, operations between one moving object and one or more static geometry objects, and operations between pairs of moving objects.

All these works illustrate that the research in moving object databases has reached a good level of maturity. It is, however, not accessible to real-world applications because a production-ready moving object database system is still missing. This gap was the main motivation behind MobilityDB. It extends the existing state of art to realize an open-source moving object database system that can be used in industry.

3 MOBILITYDB TYPE SYSTEM

Temporal types are data types that represent the evolution in time of values of another type, called the *base type*. For instance, temporal integers may be used to represent the evolution in the number of employees of a department. In this case, *temporal integer* is the temporal type and *integer* is the base type. Similarly, a *temporal point* may be used to represent the evolution in time of

the location of a vehicle. In this case, the base type is a *spatial point*. Temporal types are useful because representing values that evolve in time is essential in many applications, for example in the mobility domain. Furthermore, the operations on the base types (such as arithmetic operators and aggregation for integers and floats, and spatial relationships and distance for spatial types) can be intuitively generalized when the values evolve in time. Intuitively, an instance of a temporal type represents a function from time to the base type. To this end, this section proposes a novel discrete data model called the *sequence representation*.

Temporal types are defined based on their base type and time type. The base types are, for instance, `bool`, `int`, `float`, `text`, `geometry(point)`, and `geography(point)`, where the latter denotes geometries/geographies restricted to 2D or 3D points. Recall that in PostGIS, the `geometry` and `geography` types store spatial values using, respectively, planar and spherical coordinate systems. The time type on the other hand can be `timestamp`, `timestampset`, `period`, or `periodset`. Every combination of a base type and a time type defines one temporal type. 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.

For example, by combining the time type `timestamp` with the base type `geography(point)` we define the temporal type `tgeogpointinst` which represents a pair of a time instant and a geography point, e.g., the location and time of a car accident in a road safety database. A set of such pairs can be represented by the `tgeogpointi` type which combines the time type `timestampset` with the base type `geography(point)`. This temporal type does not interpolate the coordinates between consecutive time instants. It can be used for instance to represent users' check-ins in social networks such as Facebook or Foursquare. Combining the time type `period` with the base type `geography(point)` defines the temporal type `tgeompointseq` which represents a continuous trajectory, such as the track of a vehicle. In this case, the location of the vehicle between consecutive time instants is interpolated. The `tgeompoints`, which uses the time type `periodset`, additionally allows for temporal gaps within the trajectory, e.g., when the GPS signal disappears.

Clearly, the time type `periodset` generalizes the other three time types, that is, a periodset can have a single period and a period can collapse into a time instant. The same holds for their corresponding temporal types. However, this specialization of types brings multiple gains. First, it explicitly encodes temporal continuity information in the type representation. As will be shown in Section 4.4, this reduces both the data size and the run time of operations. The algorithms for specialized time types are typically simpler and faster than their generic counterparts. The second gain is the expressiveness, as these specialized types represent constraints on the data, that the user can explicitly specify at the schema level.

Another aspect in the specification of a temporal type is the *representation at individual timestamps*. As temporal data is typically voluminous, compression techniques can be used to reduce the size. A typical practice is to use delta encoding since it leverages the fact that the value at a given timestamp is similar to the previous one. Delta encoding is used in most video formats such as MPEG, in version control systems such as svn or git, as well as in Gorilla [47], Facebook's In-Memory Time Series Database, which uses a delta-of-delta encoding for timestamp compression. Delta encoding is also a good candidate for moving non-deforming geometries since the coordinates of a complex geometry such as a polygon are encoded only once and the motion can be expressed by simply storing its rotation and displacement. Consider a temporal type T with its associated base type S. Two consecutive observations of a moving object can be either stored as is, or only the first

one is to be stored and the second one is stored as a delta relative to the first one. For this, we need to define a *delta type* S^{Δ} as well as its associated encoding and decoding functions. For example, if the base type S is a 2D geometry, the delta type is a tuple (o, θ, dx, dy) where o is a pointer to the original geometry object, θ is a rotation angle and dx and dy are float values representing the displacement along the x and y axis. The encoding function then takes two geometries and computes the rotation and displacement of the second geometry with respect to the first one, while the decoding function takes the delta tuple and produces the second geometry.

Finally, the specification of temporal types includes the *interpolation* between consecutive observations, which can in principle be any function. Currently, MobilityDB restricts to only two interpolation types, *step*, and *linear*, in order to balance ease of manipulation and expressiveness. We also think that these two interpolations cover most application requirements. In *step* interpolation the temporal value remains constant between timestamps. In *linear* interpolation the value evolves in a continuous linear manner. Some base types cannot evolve linearly, so their associated temporal types are restricted to *step* interpolation, e.g., `boolean`, `int`, and `text`. When the base type is continuous, it is up to the user/application to decide the interpolation, either *step* or *linear*. For example, one may use a temporal point with step interpolation to represent the location of the late night service pharmacy in a neighborhood. The point coordinates remain the same for the whole night, and gets changed in the next day.

Summarizing, a temporal type is, hence, fully specified by its time type, base type, delta type (if any), and interpolation. In the following subsections, we formally define this type system.

3.1 Time Types

Four *time types* are used to represent the time dimension of temporal types: `timestamp`, `period`, `timestampset`, and `periodset`. This section gives their formal definitions. The notation D_S shall be used to denote the domain of some type S. We assume that the null value is not included in D_S since null values are not allowed in any part of the representation of temporal types. The time type `timestamp` represents a time instant, and its domain $D_{\texttt{timestamp}}$ is isomorphic to $\mathbb{N}$, e.g., representing microseconds since a fixed reference instant in the history. In the implementation, the PostgreSQL type `timestamptz` (timestamp with time zone) is used so that timestamps are time zone aware. This is essential for building applications that can receive data from multiple time zones and synchronize them. It also allows the database to adapt to the daylight saving time changes.

A `timestampset` type is defined as a non-empty set of timestamps, that is:

$$D_{\texttt{timestampset}} = \{U \subseteq D_{\texttt{timestamp}} \mid U \neq \emptyset\}$$

The domain of the `period` type is defined as follows:

$$\begin{aligned} D_{\texttt{period}} = \{(l, u, li, ui) \mid l, u \in D_{\texttt{timestamp}} \wedge l \leq u \wedge li, ui \in \{true, false\} \wedge \\ (l = u \Rightarrow li = ui = true)\} \end{aligned}$$

that is, a value of the `period` type is a time interval with lower bound l and upper bound u. The lower bound is inclusive if li is true and the upper bound inclusive if ui is true. The last condition in the definition ensures that a period with equal lower and upper bounds is not empty. Given a period $p \in D_{\texttt{period}}$, we denote by $lower(p)$ and $upper(p)$, respectively, the lower and upper bounds of p. Similarly, we denote by $lower\_inc(p)$ and $upper\_inc(p)$ the Boolean values stating, respectively, whether the lower and upper bounds are inclusive. Given periods $p_1, p_2 \in D_{\texttt{period}}$, we define

- $adjacent(p_1, p_2)$ is true iff $(upper(p_1) = lower(p_2) \wedge upper\_inc(p_1) \neq lower\_inc(p_2)) \vee (upper(p_2) = lower(p_1) \wedge upper\_inc(p_2) \neq lower\_inc(p_1))$.

- $disjoint(p_1, p_2)$ is true iff $upper(p_1) < lower(p_2) \vee upper(p_2) < lower(p_1) \vee$
  $(upper(p_1) = lower(p_2) \Rightarrow \neg upper\_inc(p_1) \vee \neg lower\_inc(p_2)) \vee$
  $(upper(p_2) = lower(p_1) \Rightarrow \neg upper\_inc(p_2) \vee \neg lower\_inc(p_1))$

Finally, the domain of the `periodset` type is defined as follows:

$$D_{\texttt{periodset}} = \{U \subseteq D_{\texttt{period}} \mid U \neq \varnothing \wedge \forall p_1, p_2 \in U(disjoint(p_1, p_2) \wedge \neg adjacent(p_1, p_2))\}$$

that is, a value of the `periodset` type is a set of disjoint and nonadjacent periods. The last condition in the definition ensures a normal representation, so equal `periodset` values will have the same representation. These four time types will be used to represent the time dimension of temporal types as defined next.

3.2 Temporal Types

Temporal types are defined using type constructors. These provide a specification for generating types given other types. Type constructors are analogous to template classes in programming languages.

Let $\mathcal{B}$ be the set of base types, i.e., $\mathcal{B} = \{\texttt{bool}, \texttt{int}, \texttt{float}, \texttt{text}, \texttt{geometry(point)}, \ldots\}$. We start by defining the *INSTANT* type constructor. Given a base type $S \in \mathcal{B}$, the domain of *INSTANT*(S) is defined as follows:

$$D_{INSTANT(S)} = D_{metavalue(S)} \times D_{timestamp}$$

where *metavalue*(S) is the abstraction of a base type and its associated delta type, if any. The data model needs to know for every supported base type, whether it will be represented as is or is compressed as delta. For example, MobilityDB makes the following choices:

$$D_{metavalue(S)} = \begin{cases} D_S & S \in \{\texttt{bool}, \texttt{int}, \texttt{float}, \texttt{text}, \\ & \texttt{geometry(point)}, \texttt{geography(point)}\} \\ (o, \theta, dx, dy) & S = \texttt{geometry(polygon)} \\ \ldots & \end{cases}$$

For most of the base types the metavalue is a value of the domain of the type. For a rigid moving region, similar to [32], the base type is a `geometry(polygon)`, and the metavalue is a tuple of affine transformation parameters. Here we assume that the θ rotation is centered at the polygon centroid. It is out of the scope of this paper to discuss rigid moving regions. We only mention them here to explain the model expressiveness. An example of an `INSTANT(geometry(point))` value is illustrated in Figure 1a, it is a pair of a geometry point and a timestamp.

Next we define the *INSTANTS* type constructor. Given a base type S, the domain of *INSTANTS(S)* is defined as follows:

$$D_{INSTANTS(S)} = \{U \subseteq D_{INSTANT(S)} \mid U \neq \varnothing \wedge \forall (v_i, t_i), (v_j, t_j) \in U\ (t_i = t_j \Rightarrow v_1 = v_2)\}$$

that is, the constructed type represents set of instant values with distinct timestamps. The value of the moving object is known at these timestamps, and is not known otherwise. An example of an `INSTANTS(geometry(point))` value is illustrated in Figure 1b, which illustrates a set of instants, each of which is a pair of a geometry point and a timestamp.

To represent the continuous evolution of temporal types, the *SEQUENCE* type constructor is defined next. Given a base type S, the domain of *SEQUENCE(S)* is defined as follows:

$D_{SEQUENCE(S)} = \{(I, li, ui, interpolation) \mid$
(i) I is a non empty, temporally ordered list of values of *INSTANT*(S)
(ii) $li, ui \in \{true, false\}$
(iii) $interpolation \in \{step, linear\}$
(iv) $\forall i(\tau_i, \tau_{i+1}$ are not colinear)$\}$

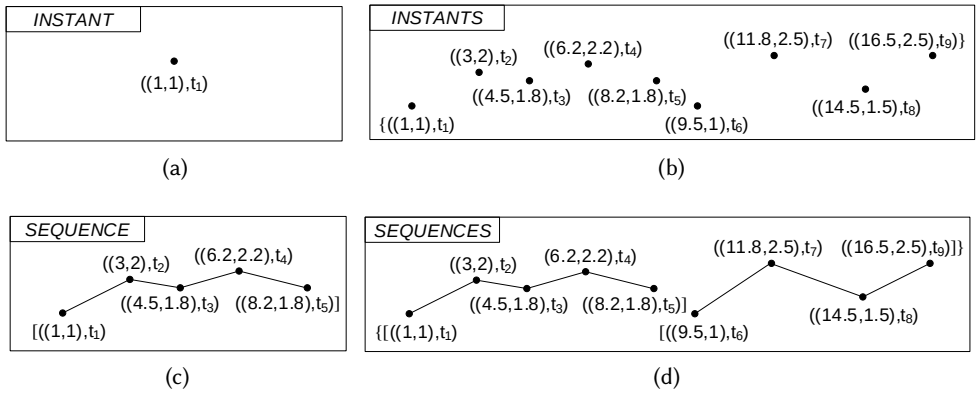


Fig. 1. Temporal types

where the moving object is described by a list of instants $I = [(v_1, t_1), \ldots, (v_n, t_n)]$. The object is continuously defined over the period (t_1, t_n, li, ui). Between two instants, intermediate instants are interpolated, as specified in the *interpolation* member, using the *step* or the *linear* interpolation function. In the last constraint, τ_i denotes the interpolation function between two instants (v_i, t_i) and (v_{i+1}, t_{i+1}). This constraint ensures that two consecutive pieces of the interpolation function cannot be the colinear, otherwise they must be merged. This ensures a minimal and unique representation, so that equal temporal values will have the same representation. An example of a `SEQUENCE(geometry(point))` value is illustrated Figure 1c. It consists of five instants. Both the start and end timestamps are included, i.e., $li = ui = true$. The interpolation at intermediate timestamps is linear.

The temporal types constructed by *SEQUENCE* are guaranteed to be defined at any time instant between their start and end times. In other words, temporal gaps are not allowed within their definition time. If the interpolation is *linear*, there is another guarantee that the piecewise function is continuous. This is because two consecutive pieces of τ share one end point. This continuity information, which is explicitly represented in the data model, is exploited in the algorithms to gain efficiency.

However, it is also needed to represent discontinuity in time or in value. It is possible that the definition time of a moving object includes temporal gaps, during which there is no information about its location or value (i.e., discontinuity in time). Such a temporal gap could represent, for instance, the event that the GPS signal was lost for a certain duration. It could also be the result of applying an operation on the moving object trajectory. For example, an operation that restricts the trajectory to the pieces during which it reached or exceeded a given speed limit, will generate temporal gaps in the periods where the speed is below the limit. It is also possible that the moving object trajectory undergoes a distinct location/value change (i.e., discontinuity in value), perhaps to represent the result of some operation. For these cases, the data model additionally defines the *SEQUENCES* type constructor.

Let q denote a value of a *SEQUENCE* type, i.e., $q = (I, li, ui, interpolation)$, with a list of instants $I = [(v_1, t_1), \ldots, (v_n, t_n)]$. To help the following definition, we first define $period(q)$ as the function that yields the bounding period of q as follows: $period(q) = period(t_1, t_n, li, ui)$. Further, we denote by $first(q)$ and last(q) the first and the last values of the moving object as follows: $first(q) = v_1$, $last(q) =$

v_n. Given a base type S, the domain of *SEQUENCES(S)* is defined as follows:

$$\begin{aligned} D_{SEQUENCES(S)} = \{U \subseteq D_{SEQUENCE(S)} \mid \\ (i) U \neq \emptyset \\ (ii) \forall q_i, q_j \in U\ (i \neq j \Rightarrow disjoint(period(q_i), period(q_j)) \\ (iii) \forall q_i, q_j \in U\ (q_i.interpolation = q_j.interpolation) \\ (iv) \forall q_i, q_{i+1} \in U\ (adjacent(period(q_i), period(q_{i+1})) \Rightarrow \\ \quad interpolation = linear \wedge last(q_i) \neq first(q_{i+1}))\} \end{aligned}$$

that is, it is a non-empty set of sequences that are disjoint on time and all have the same interpolation. The last condition in the definition ensures a minimal representation, where two adjacent sequences cannot be further joined into a single one. In other words, two consecutive sequences are split either by a temporal gap or by a value change. These types can hence represent discontinuities in both the time and the value dimensions. An example of a `SEQUENCES(geometry(point))` value is illustrated in Figure 1d. It consists of two SEQUENCE objects, with a temporal gap between t_5 and t_6.

Now we are able to define generic temporal types using the *TEMPORAL* type constructor. Given a base type S:

$$D_{TEMPORAL(S)} = D_{INSTANT(S)} \cup D_{INSTANTS(S)} \cup D_{SEQUENCE(S)} \cup D_{SEQUENCES(S)}$$

that is, when the type constructor *TEMPORAL* is instantiated by a base type, it constructs all the four temporal types that correspond to the given base type. The *TEMPORAL* type constructor thus represents the complete type system, i.e., the *sequence representation* of moving objects.

Recall that we have excluded null values in any component of a temporal value. However, a temporal value is considered as null at any timestamp at which it is not defined. Additionally, a null value is a valid value for any type in SQL. For this reason, for any of the temporal constructors defined above we defined the corresponding nullable version of the constructors as follows,

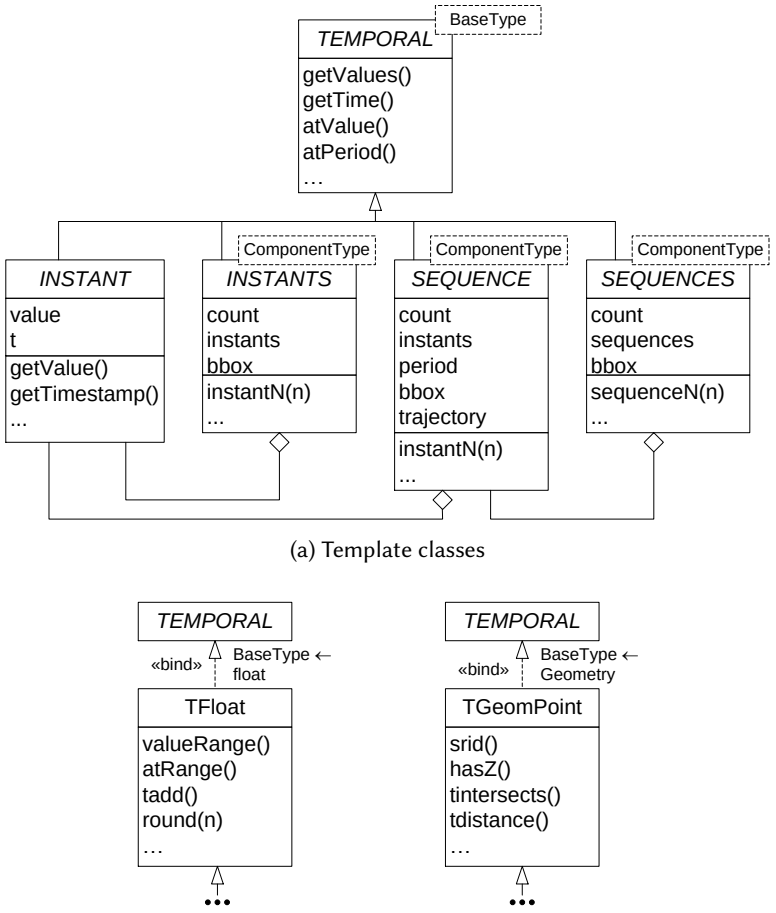
$$\begin{aligned} D'_{INSTANT(S)} &= D_{INSTANT(S)} \cup \{\text{NULL}\} \\ &\dots \\ D'_{TEMPORAL(S)} &= D_{TEMPORAL(S)} \cup \{\text{NULL}\} \end{aligned}$$

3.3 Implementation of Temporal Types

Figure 2 shows the conceptual UML class diagram of the type system described in Section 3.2. In Figure 2a, *TEMPORAL* is an abstract template class, which takes a base type as a parameter. It has four template subclasses, which inherit the same parameter: *INSTANT*, *INSTANTS*, *SEQUENCE*, and *SEQUENCES*. In addition, the three latter template subclasses have another parameter for the temporal type that participates in the aggregation, that is, *INSTANT* for both *INSTANTS* and *SEQUENCE*, and *SEQUENCE* for *SEQUENCES*. The temporal classes are obtained by instantiating these template classes as shown in Figure 2b.

We define polymorphic operations over these types. Some operations can be evaluated regardless of the underlying base and time types. Examples are the functions getValues and getTime, which return, respectively, the set of values and the time frame on which the temporal value is defined. Other operations can be evaluated regardless of the base type. Examples are those functions manipulating only the time dimension or the functions instantN or sequenceN, which return the n-th component of the temporal value. Finally, some operations depend on both the time type and the base type. Examples of the latter are arithmetic operators (+, −, *, and /) for temporal numbers or topological predicates (e.g., `intersects` or `disjoint`) for temporal points.

MobilityDB implements this type system in different programming languages. The user interacts with the database in SQL, while the database engine is implemented in C as an extension of



(a) Template classes

(b) Examples of temporal classes

Fig. 2. MobilityDB conceptual class hierarchy

PostgreSQL and PostGIS. The Python database connectivity driver is implemented in Python on top of the PostgreSQL drivers *psycopg2* and *asyncpg*. Therefore, the conceptual UML class diagram in Figure 2 is mapped into multiple physical models, according to the object-oriented capabilities of the used programming languages, if any. In the following, we describe the implementation of the database engine in C.

We define a generic type in C, *TEMPORAL*, and four concrete types, one for each time type, namely, *INSTANT*, *INSTANTS*, *SEQUENCE*, and *SEQUENCES*. These types store both the base type and the time type. The polymorphic operations over the temporal types are then implemented through dispatch functions where the arguments are of type *TEMPORAL*. Depending on the base and/or the time type, the dispatch functions call the specific function to be executed and pass the arguments with the appropriate casting.

In order to improve performance, especially for index operations, the temporal types store their *bounding box*. The bounding boxes of the various types are as follows:

- A time `period` for the `tbool` and `ttext` types, where only the temporal extent is considered.

- The `tbox` (temporal box) type for the `tint` and `tfloat` types, where the value extent is defined in the x dimension and the temporal extent in the t dimension.
- The `stbox` (spatiotemporal box) type for the `tgeompoint` and `tgeogpoint` and types, where the spatial extent is defined in the x, y, and z dimensions and the temporal extent in the t dimension.

For the *INSTANT* type, no bounding box is stored. For the *INSTANTS* and the *SEQUENCE* types, the minimum and maximum values of every dimension define the bounding box. For the *SEQUENCES* type, the bounding box is the union of the bounding boxes of the composing sequences, and thus it might include dead space. Similarly, also for performance reasons, temporal points of the *SEQUENCE* type store their spatial `trajectory`, i.e., a PostGIS `Linestring` value that represents all the values traversed by the temporal point.

The next step is to makes these C types accessible to the user via SQL. PostgreSQL types and functions, including the built-in ones, are defined through its catalog. This is how it is extensible. Additionally, PostgreSQL supports dynamic loading of the compiled/object code at runtime. Thus, it can be extended by user types and functions without requiring server restart. MobilityDB makes use of this, as it is packaged as a PostgreSQL extension. It consists of two main parts: the C code that defines the temporal types and operations, and an SQL part that adds them to the PostgreSQL catalog. A type is then declaratively created by specifying a group of functions for input, output, send, receive, etc. The operations on the temporal types are also programmed in C and added to the catalog via SQL wrappers.

Currently, MobilityDB instantiates the *TEMPORAL* class for the base classes `bool`, `int`, `float`, `text`, `geometry(point)`, and `geography(point)`. The constructed classes are exposed in SQL with the names `tbool`, `tint`, `tfloat`, `ttext`, `tgeompoint`, and `tgeogpoint`, respectively, where the prefix `t` denotes *temporal*. They are abstract data types, as they hide the complexity of the internal implementation and focus on usability. To users, it is natural to think of temporal counterparts of the non-temporal types. Temporal types can be used as any built-in data type in table definitions as shown next.

```
CREATE TABLE Department(DeptNo integer, DeptName varchar(25), NoEmps tint);
INSERT INTO Department VALUES
    (10, 'Research', tint '[10@2012-01-01, 12@2012-04-01, 12@2012-08-01)'),
    (20, 'Human Resources', tint '[4@2012-02-01, 6@2012-06-01, 6@2012-10-01)');
```

In this example, the column `NoEmps` of type `tint` keeps the evolution of the number of employees in a department.

In PostgreSQL, *type modifiers* specify additional type information for a column definition. Type modifiers can be used to restrict the values of a temporal type column to be of a specific time type. For example, in the following table definition:

```
CREATE TABLE Department(DeptNo integer, DeptName varchar(25), NoEmps tint(sequence));
```

the type modifier for the type `varchar` is the value 25, which indicates the maximum length of the values of the column, while the type modifier for the type `tint` is `sequence`, which specifies its time type. In other words, users have the choice to use the more generic *TEMPORAL* types, or one of the four subclasses, where for example `tint` = *TEMPORAL*`(int)` and `tint(sequence)` = *SEQUENCE*`(int)`. If a temporal attribute is defined using a type modifier, it will only accept values of the corresponding subclass. If no type modifier is specified, values of any subclass are allowed and the subclass will be decided at runtime during the assignment operation.

4 FUNCTIONS AND OPERATORS FOR TEMPORAL TYPES

The type system defined in Section 3 is supported by a rich set of functions and operators. The general idea in selecting them is to support most common operations. There is a minimum technical base that must be implemented including constructors, casts, input, and output. Then, there are functions specified or suggested by the ISO and OGC standards on moving features [11]. Many other functions come from analyzing the functions provided by SECONDO. In this case, we selected those that can be expressed in SQL. Finally, PostGIS has a big number of spatial functions, which we *lift* if they are relevant. The time types in Section 3.1 are also supported by functions that are available to users. In the following, Sections 4.1 and 4.2 describe the algorithms for synchronization, normalization, and lifting. These algorithms, which are essential internal operations, are involved in the computation of many of the functions. Then, Section 4.3 classifies the user functions into 11 categories, explains them and illustrates them with examples. A detailed list of the functions, their signatures, and query examples can be found in the MobilityDB manual[2].

4.1 Synchronization and Normalization

Synchronization and normalization are, respectively, essential pre- and post-processing steps of most temporal operations. *Synchronization* is relevant in the operations that process multiple temporal arguments. The result of the operation is only defined over the time intervals where all the arguments are defined. Moreover, the temporal function of the result can change whenever the temporal function of at least one of the argument changes. Figure 3 illustrates the temporal addition of the following `tint` values:

```
SELECT tint '{[0.5@2007-05-01, 1@2007-05-03, 0.5@2007-05-05, 0.5@2007-05-08)}' +
    tint '{[1.5@2007-05-02,1.5@2007-05-04], [1.5@2007-05-06,1.5@2007-05-07)}'
Result: tint '{[2@2007-05-02, 2.5@2007-05-03, 2.5@2007-05-04],
    [2@2007-05-06, 2@2007-05-07)}'
```

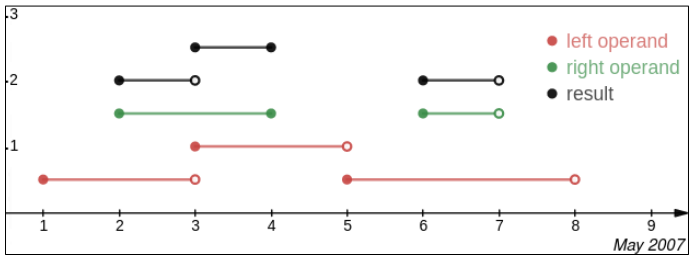


Fig. 3. Temporal addition of `tint`

The first argument is defined over the interval May 1st to 8th. The second argument is defined on the intervals May 2nd to 4th and 6th to 7th, with a temporal gap in between. Accordingly, the result of the addition is defined at the intersection of the definition times of the arguments, i.e., May 2nd to 4th and May 6th to 7th. Additionally, the temporal function of the result changes at May 3rd, because the temporal function of the first argument changes. Synchronization is the pre-processing operation that creates such temporal profile of the result.

We restrict the temporal synchronization operation to be binary. If a synchronization is required for more than two temporal values, the operation can be iteratively invoked for the arguments, as it is an associative operation. Algorithm 1 illustrates the synchronization operation for `SEQUENCES`.

[2]https://github.com/ULB-CoDE-WIT/MobilityDB#manual

Given a pair of SEQUENCES values $seqSet_1$, $seqSet_1$, it performs a parallel scan for their composing SEQUENCE lists and instantiates a call to Algorithm 2 to synchronize every pair.

Algorithm 1: synchronize($seqSet_1, seqSet_2, out\ syncInstants_1, out\ syncInstants_2$)

Input: $seqSet_1, seqSet_2$, the objects to synchronize, of type SEQUENCES
Output: $syncInstants_1, syncInstants_2$, the synchronized INSTANT[] corresponding to the two input parameters

```
begin
    syncInstants_1 ← ∅; syncInstants_2 ← ∅ ;
    i ← 0; j ← 0 ;
    while i < seqSet_1.count and j < seqSet_2.count do
        seq_1 ← seqSet_1.sequences[i]; seq_2 ← seqSet_2.sequences[j];
        if synchronize(seq_1, seq_2, sync_1, sync_2) then
            syncInstants_1 ← append(syncInstants_1, sync_1);
            syncInstants_2 ← append(syncInstants_2, sync_2);
        if seq_1.period = seq_2.period then i ← i + 1; j ← j + 1;
        else if seq_1.t < seq_2.t then i ← i + 1;
        else if seq_2.t < seq_1.t then j ← j + 1;
```

Algorithm 2 first finds the overlap interval of the two sequences. Restricted to this interval, it performs a parallel scan over their instants and produces a union of their timestamps. For each of the two sequences, the values at the additional timestamps are interpolated using the operation atTimestamp(., .). Together, the two algorithms will visit each of the instants of the two SEQUENCES arguments at most once. Therefore, the synchronization operation is $O(n + m)$, where n and m are the number of instants of the two arguments. In practice, only the instants that fall inside the intersection of the definition times of the arguments are visited, which can be much less than $m + n$ or even zero. The synchronization of INSTANT and INSTANTS values is straightforward. Since these types do not imply interpolation between the consecutive instants, the synchronization is the intersection of their timestamps. This is again an $O(n + m)$ operation.

When the result of an operation is a SEQUENCE or a SEQUENCES value, *normalization* is involved in the post-processing. It merges the consecutive co-linear segments of the piecewise temporal function. This is done in order to ensure a minimal and unique representation, as in the definitions in Section 3.2. The operation is a linear scan of all instants, hence $O(n)$.

4.2 Lifted Operations

Another algorithmic strategy is *temporal lifting*. It refers to overloading the query operations that are defined for non-temporal types to accept temporal types. Given a static operation *op* with the signature $\alpha \times \beta \rightarrow \gamma$, where α, β, and γ are type variables in the set of base types $\mathcal{B}$, *lifting* is a transformation that generates the lifted counterpart of *op*, denoted by LIFTED(*op*), which can have any of the following signatures:

LIFTED(*op*)

TEMPORAL(α)	×	TEMPORAL(β)	→	TEMPORAL(γ)
α	×	TEMPORAL(β)	→	TEMPORAL(γ)
TEMPORAL(α)	×	β	→	TEMPORAL(γ)

Algorithm 2: synchronize($seq_1, seq_2, out\ syncInstants_1, out\ syncInstants_2$)

Input: seq_1, seq_2, the objects to synchronize, of type SEQUENCE
Output: $syncInstants_1, syncInstants_2$, the synchronized INSTANT[] corresponding to the two input parameters
begin
$syncInstants_1 \leftarrow \varnothing$; $syncInstants_2 \leftarrow \varnothing$;
$inter \leftarrow seq_1.period \cap seq_2.period$;
$i \leftarrow$ binarySearch($seq_1.instants, inter.lower$);
$j \leftarrow$ binarySearch($seq_2.instants, inter.lower$);
$inst_1 \leftarrow$ atTimestamp($seq_1, inter.lower$); $inst_2 \leftarrow$ atTimestamp($seq_2, inter.lower$);
while $i < seq_1.count$ and $j < seq_2.count$ and ($inst_1.t < inter.upper$ or $inst_2.t < inter.upper$) **do**
  if $inst_1.t = inst_2.t$ **then** $i \leftarrow i + 1$; $j \leftarrow j + 1$;
  else if $inst_1.t < inst_2.t$ **then** $i \leftarrow i + 1$; $inst_2 \leftarrow$ atTimestamp($seq_1, inst_2.t$);
  else if $inst_2.t < inst_1.t$ **then** $j \leftarrow j + 1$; $inst_1 \leftarrow$ atTimestamp($seq_2, inst_1.t$);
  $syncInstants_1 \leftarrow$ append($syncInstants_1, inst_1$);
  $syncInstants_2 \leftarrow$ append($syncInstants_2, inst_2$);
  $inst_1 \leftarrow seq_1.instants[i]$; $inst_2 \leftarrow seq_2.instants[j]$;

For example, the temporal addition in Figure 3 is the lifted variant of the integer addition operator, with signature `tint` × `tint` → `tint`. Two other variants are also implemented, which accept a static `int` and a temporal `tint`. Semantically, LIFTED(op) is considered the lifted counterpart of op iff

$$\forall t, \text{LIFTED}(op)(a, b)(t) = op(a(t), b(t))$$

where at least one of a or b is a temporal value. In the formula above, $a(t)$ and $b(t)$ denote the static value (or snapshot) at the time instant t. When the argument is temporal, this is equivalent to calling `atTimestamp`. If the argument is non-temporal, then $\forall t, a(t) = a$. Hence, the temporal result of the lifted operation should describe at every time instant the result of applying the static operation on the snapshots of the temporal arguments at the same time instant. In a discrete data model, this equivalence is approximated, because the function on the right-hand side can be of any order while the function on the left-hand side is restricted to the discrete representation. In MobilityDB, the approximation guarantees that the two functions are equal at the synchronization time instants as well as at the turning points of the right-hand side. A turning point is where the function changes from increasing to decreasing or vice versa.

The concept of temporal lifting has been first introduced in [29]. It has however not been realized as an algorithm before. Instead, the lifting of every operation had to be individually implemented, e.g., in SECONDO. We propose an algorithm for automatically lifting a given static operation. That is, given a pair of temporal operands and a static operation, this algorithm evaluates the corresponding lifted operation on the two operands and yields a temporal value. The objectives of this algorithm are twofold:

(1) To achieve a consistent and automatic way of temporal lifting.
(2) To maximize the delegation as well as the separation between the MOD implementation and the underlying platform.

The first objective is clearly desirable, because it allows for lifting the existing operations as well as the ones that might be added in the future, without writing specific implementations.

The significance of the second objective is to make this algorithm agnostic to the underlying database platform, while allowing it to reuse its operations. For instance, in our implementation the spatiotemporal operations are lifted from the PostGIS operations. The lifting algorithm processes the temporal dimension, and delegates the spatial processing to PostGIS.

Lifting is straightforward when the temporal arguments of the lifted operation are of *INSTANT* or *INSTANTS* types. It translates into computing the static operation at every timestamp during which all the arguments are defined. Therefore, in the following we focus on the more sophisticated case when the arguments are of *SEQUENCE* type.

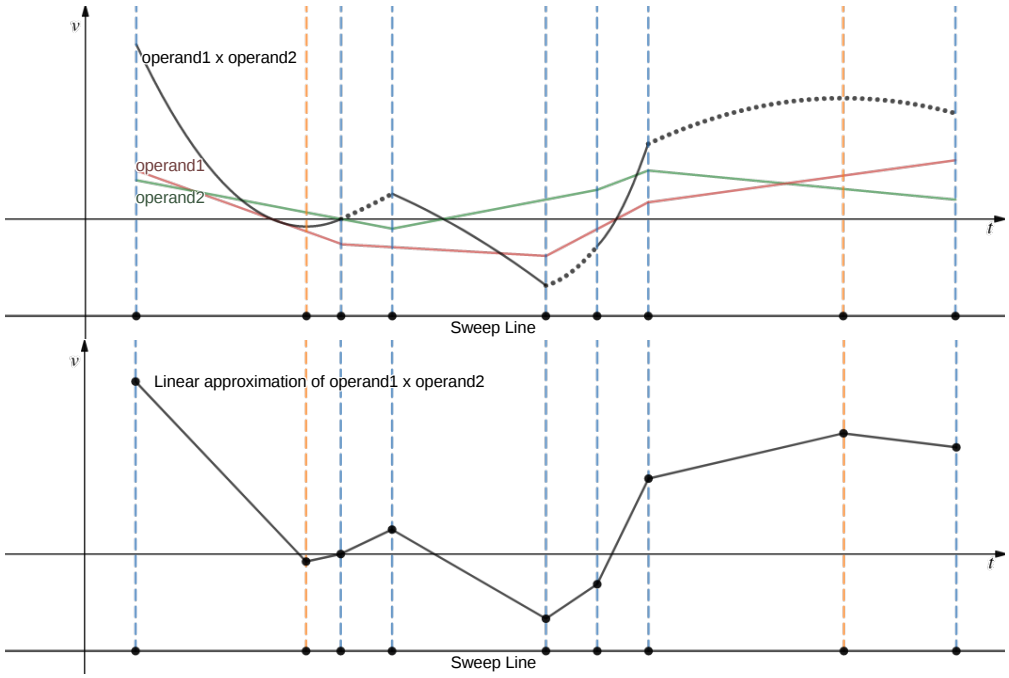


Fig. 4. Illustrating the lifting algorithm with multiplication of temporal values

Figure 4 illustrates the idea of the lifting algorithm using the example of temporal multiplication. On top, the two temporal operands are shown, respectively, in green and red. The multiplication result is shown in black and its composing segments are shown in alternating solid and dotted lines. As the two inputs are piecewise linear functions, their multiplication result is piecewise quadratic. On the bottom the multiplication result is shown after being linearly approximated according to the sequence representation. It is also the result that we want to return by the lifting algorithm. This approximation maintains all the local maxima and minima of the quadratic result. We want that the linear approximation preserves the turning points of the result.

The first step is to synchronize the two operands, as described in Algorithms 1–2. This is shown using the dashed blue vertical lines. In the algorithm, these time instants will serve as sweep line events, to notify a possible change in the output function. At each of these synchronization time instants, at least one of the two arguments change the linear function. Accordingly, the function that describes the multiplication result may also change.

Synchronizing the two arguments is not enough to preserve the turning points. Figure 4 shows that the first and the last pieces of the quadratic function of the result have turning points, indicated

Algorithm 3: LIFTED($seq_1, seq_2, op, \frac{d}{dt}op^L$)

Input: seq_1, seq_2, the argument of the lifted operation of type *SEQUENCE*, op the non-temporal operation to be lifted, $\frac{d}{dt}op^L$ the function that computes the zeros of the derivative of the lifted operation
Output: $result$, the outcome of applying the lifted operation on seq_1, seq_2 of type *SEQUENCE*

begin
$inter \leftarrow seq_1.period \cap seq_2.period$;
$i \leftarrow$ binarySearch($seq_1.instants, inter.lower$);
$j \leftarrow$ binarySearch($seq_2.instants, inter.lower$);
$inst_1 \leftarrow$ atTimestamp($seq_1, inter.lower$); $inst_2 \leftarrow$ atTimestamp($seq_2, inter.lower$);
$prev_1 \leftarrow \perp$; $prev_2 \leftarrow \perp$; $instants \leftarrow \varnothing$;
while $i < seq_1.count$ and $j < seq_2.count$ and ($inst_1.t < inter.upper$ or $inst_2.t < inter.upper$) **do**
  if $inst_1.t = inst_2.t$ **then** $i \leftarrow i + 1$; $j \leftarrow j + 1$;
  else if $inst_1.t < inst_2.t$ **then** $i \leftarrow i + 1$; $inst_2 \leftarrow$ atTimestamp($seq_1, inst_2.t$);
  else if $inst_2.t < inst_1.t$ **then** $j \leftarrow j + 1$; $inst_1 \leftarrow$ atTimestamp($seq_2, inst_1.t$);
  $instants \leftarrow instants \cup make\_instant(op(inst_1.value, inst_2.value), inst_1.t)$;
  if $prev_1 \neq \perp$ **then**
    $t \leftarrow \frac{d}{dt}op^L((prev_1, inst_1), (prev_2, inst_2))) = 0$;
    if $prev_1.t < t < inst_1.t$ **then**
      $value_1 \leftarrow$ valueAtInstant(seq_1, t); $value_2 \leftarrow$ valueAtInstant(seq_2, t);
      $instants \leftarrow instants \cup make\_instant(op(value_1, value_2), t)$;
  $prev_1 \leftarrow inst_1$; $prev_2 \leftarrow inst_2$;
  $inst_1 \leftarrow seq_1.instants[i]$; $inst_2 \leftarrow seq_2.instants[j]$;
$result \leftarrow make\_sequence(instants, seq.li, seq.ui, linear)$;
return $result$;

by the orange dashed line. These time instants must be added to the sweep line. They can be identified by finding the zeros of the derivative $\{t : (\frac{d}{dt}\text{LIFTED}(op)(a, b)) = 0\}$. Accordingly, the algorithm expects as an argument a function that computes the zeros of the derivative. Thus, the lifting algorithm evaluates the static operation $op(a(t), b(t))$ at every time instant t in the sweep line composed of both the synchronization and the turning point instants. The resulting *INSTANT* values are linearly connected, then normalized to produce the result.

An efficient version of this idea is illustrated in Algorithms 3 and 4, which integrate the synchronization and the lifting in a single pass. Algorithm 3 is used when the result is a *SEQUENCE* value, as in the lifted multiplication illustrated in Figure 4. In this case, the algorithm collects a list of instants in the main loop and constructs the resulting *SEQUENCE* from these instants at the end. On the other hand, Algorithm 4 is used when the result is a *SEQUENCES* value, as in the lifted topological predicates illustrated in Figure 5, which result in a temporal Boolean. In this case, each iteration of the main loop takes one segment of the synchronized arguments and produces either one or three sequences, depending on whether there is a turning point in the segment. If that case, one sequence is produced from the beginning of the segment to the turning point, one instantaneous sequence for the turning point, and one from the turning point to the end of the

segment. Notice that these algorithms can also perform the normalization in a single pass while producing the result, but this is not shown for simplicity reasons.

Algorithm 4: LIFTED_SEQUENCES($seq_1, seq_2, op, \frac{d}{dt}op^L$)

Input: seq_1, seq_2, the argument of the lifted operation of type *SEQUENCE*, op the non-temporal operation to be lifted, $\frac{d}{dt}op^L$ the function that computes the zeros of the derivative of the lifted operation

Output: $result$, the outcome of applying the lifted operation on seq_1, seq_2 of type *SEQUENCES*

begin
- $inter \leftarrow seq_1.period \cap seq_2.period$;
- $i \leftarrow$ binarySearch($seq_1.instants, inter.lower$);
- $j \leftarrow$ binarySearch($seq_2.instants, inter.lower$);
- $start_1 \leftarrow$ atTimestamp($seq_1, inter.lower$); $start_2 \leftarrow$ atTimestamp($seq_2, inter.lower$);
- $start \leftarrow make\_instant(op(start_1.value, start_2.value), start_1.t)$;
- $i \leftarrow i + 1$; $j \leftarrow j + 1$; $sequences \leftarrow \emptyset$;
- **while** $i < seq_1.count$ and $j < seq_2.count$ and ($start_1.t < inter.upper$ or $start_2.t < inter.upper$) **do**
  - $end_1 \leftarrow seq_1.instants[i]$; $end_2 \leftarrow seq_2.instants[j]$;
  - **if** $end_1.t = end_2.t$ **then** $i \leftarrow i + 1$; $j \leftarrow j + 1$;
  - **else if** $end_1.t < end_2.t$ **then** $i \leftarrow i + 1$; $end_2 \leftarrow$ atTimestamp($seq_1, end_2.t$);
  - **else if** $end_2.t < end_1.t$ **then** $j \leftarrow j + 1$; $end_1 \leftarrow$ atTimestamp($seq_2, end_1.t$);
  - $t \leftarrow \frac{d}{dt}op^L((start_1, end_1), (start_2, end_2))) = 0$;
  - **if** $start_1.t < t < end_1.t$ **then**
    - $end \leftarrow make\_instant(op(start_1.value, start_2.value), t)$;
    - $sequences \leftarrow sequences \cup make\_sequence(\{start, end\}, true, false)$;
    - $value_1 \leftarrow$ valueAtInstant(seq_1, t); $value_2 \leftarrow$ valueAtInstant(seq_2, t);
    - $start \leftarrow make\_instant(op(value_1, value_2), t)$;
    - $sequences \leftarrow sequences \cup make\_sequence(\{start\}, true, true)$;
    - $midt \leftarrow (end_1.t - t)/2$;
    - $value_1 \leftarrow$ valueAtInstant($seq_1, midt$); $value_2 \leftarrow$ valueAtInstant($seq_2, midt$);
    - $start \leftarrow make\_instant(op(value_1, value_2), t)$;
    - $end \leftarrow make\_instant(op(value_1, value_2), inst_1.t)$;
    - $sequences \leftarrow sequences \cup make\_sequence(\{start, end\}, false, false)$;
  - **else**
    - $end \leftarrow make\_instant(op(end_1.value, end_2.value), end_1.t)$;
    - $sequences \leftarrow sequences \cup make\_sequence(\{start, end\}, true, false)$;
  - $start_1 \leftarrow end_1$; $start_2 \leftarrow end_2$; $start \leftarrow end$;
- $result \leftarrow make\_sequences(sequences)$;
- **return** $result$;

4.3 Temporal Operations

This section describes the API that is available to users. It does not exhaustively list all functions (currently about 300). The up-to-date list is can be found on the MobilityDB manual[3]. In the following, we describe the function classes and give illustrative examples of functions in each class: (1) lifted topological predicates, (2) simple topological predicates, (3) distance operators, (4) mathematical, logical, and comparison functions, (5) restriction functions, (6) index comparison operators, (7) spatiotemporal attributes, (8) operations on bounding box and time types, (9) temporal/time/box casting, (10) utility functions, and (11) temporal aggregation.

Lifted Topological Predicates

This group of operations represents the lifted counterparts of the PostGIS spatial topological predicates, i.e., using the generic lifting of Algorithm 4. The PostGIS topological predicates implement the OGC standard Dimension Extended nine-Intersection Model (DE-9IM)[10], a model developed in [8, 9] based on the Egenhofer's nine intersection model [19]. Each of them supports geometry types, while some of them additionally support geography types. MobilityDB lifts these operations so that they accept a combination of a geometry/geography and a temporal point. If the semantics allows, they are also lifted to accept a pair of temporal geometries/geographies.

The lifting is automatically obtained using Algorithm 4. The static spatial predicate (i.e., from PostGIS) is called at every instant of the two arguments, and additionally at the turning points. Here the turning points occur at the timestamps where the two arguments intersect. The predicate value can be different before, at, and after the intersection, allowing for distinct changes in the resulting temporal Boolean. We illustrate this with the `twithin` temporal predicate and the following two examples (see also Figure 5).

```
SELECT twithin (tgeompoint '(Point(3 3)@2012-01-01, Point(3 7)@2012-01-03]'),
    box2d 'BOX(1 1,5 5)')
Result: "{(t@2012-01-01, f@2012-01-02, f@2012-01-03]}"

SELECT twithin (
    tgeompoint '(Point(3 3)@2012-01-01, Point(5 3)@2012-01-02, Point(3 4)@2012-01-03]'),
    box2d 'BOX(1 1,5 5)')
Result "{(t@2012-01-01, f@2012-01-02], (t@2012-01-02, t@2012-01-03]}"
```

The examples show a temporal point that has a single segment and a box geometry. In the first example, the temporal point starts inside the box and then crosses to outside. The result is then true for half the duration, then false. As the definition period of the `tgeompoint` is left open, so is also the resulting `tbool`. At the instant of intersection, the `within` predicate becomes false, and remains so until the end instant, which is inclusive. In the second example, the temporal point touches the box boundary and bounces back. So the result is always true except at the instant where the arguments touch each other. In summary, the predicate `twithin` yields true at each time point in which the temporal point is `within` the box, it returns false in the rest of the definition time of the temporal point, and it is undefined otherwise. This meaning has been formalized in [21], which has specialized the general concept of lifted predicates [29] for the topological predicates.

We list next the temporal predicates that are currently implemented. We denote by A and B, respectively, the first and the second argument.

- `tcontains`(`geometry`, `tgeompoint`) : `tbool`, whether B is in the interior A.
- `twithin`(`tgeompoint`, `geometry`) : `tbool`, whether A is in the interior B.

[3]https://github.com/ULB-CoDE-WIT/MobilityDB#manual

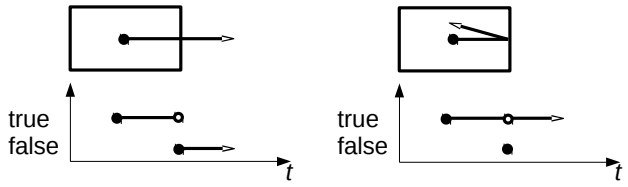


Fig. 5. The twithin temporal predicate

- tcovers(geometry, tgeompoint) : tbool, whether B is not in the exterior of A.
- tcoveredby(tgeompoint, geometry) : tbool, whether A is not in the exterior of B.
- ttouches(geometry, tgeompoint) : tbool, whether B lie in the boundary A. The other commutative signature is also implemented.
- tequals({geometry, tgeompoint}, {geometry, tgeompoint}) : tbool, whether A has the same geometry as B.
- tintersects({geometry, tgeompoint}, {geometry, tgeompoint}) : tbool), whether the two arguments share any portion of space.
- tdisjoint({geometry, tgeompoint}, {geometry, tgeompoint}) : tbool, whether the two arguments do not share any portion of space.
- tdwithin({geometry, tgeompoint}, {geometry, tgeompoint}, double) : tbool, whether the shortest distance between A and B is less than or equal the given threshold.

In the last four functions above, the notation {geometry, tgeompoint} means that the argument can take either of the types, with the restriction that at least one of the two arguments of the function must be temporal. It is a design choice not to implement the variant that accepts two temporal points for the first four functions, as these would be redundant with tequals. Actually, tintersects is also redundant with tequals, but we implement both since they are common operations that users expect to have.

The previous set of functions is not implemented for temporal geographies, because the current version of PostGIS does not provide the functionality to verify that a geographic point satisfies a topological predicate (e.g., intersects, dwithin) with an arbitrary geography. Accordingly, MobilityDB currently supports the following set of topological predicates on temporal geographies:

- tequals, tintersects(tgeogpoint, tgeogpoint) : tbool, whether A has the same geometry as B.
- tdisjoint(tgeogpoint, tgeogpoint) : tbool, whether the two arguments do not share any portion of space.
- tdwithin(tgeogpoint, tgeogpoint, double) : tbool, whether the shortest distance between A and B is less than or equal the given threshold.

Simple Topological Predicates

Consider an example where one wants to query for the moving objects in a table Trips that *ever* intersect a given geometry. The meaning of *ever* is clear, it yields true if the temporal Boolean has at least one instant in which the value is true. Alternatively, one would need to express whether a predicate has *always* been true. For example, we could ask about trips that were *always* covered by the geometry. The meaning of *always* is more tricky as it is relative to a time domain. Here we choose that *always* is evaluated with reference to the definition time of the temporal Boolean. Accordingly, two functions are defined for temporal Booleans:

- tbool ?= bool (ever equals) tests whether $\exists t \in getTime(A), A(t) = B$.

- `tbool %= bool` (always equals) tests whether $\forall t \in getTime(A), A(t) = B$.

Combining the lifted temporal predicates with the ever equals and always equals functions, one can express topological filter conditions in SQL. However, this nesting of two functions does not enable the query planner to make use of available spatiotemporal indexes. We thus define for every lifted topological predicate a simple version that yields a Boolean. Doing so, we choose the more probable meaning between *ever* and *always* for every predicate. This is as given next:

- `contains`(`geometry`, `tgeompoint`) : `bool`, equivalent to `tcontains`(A, B) `%=` TRUE
- `within`(`tgeompoint`, `geometry`) : `bool`, equivalent to `twithin`(A, B) `%=` TRUE
- `covers`(`geometry`, `tgeompoint`) : `bool`, equivalent to `tcovers`(A, B) `%=` TRUE
- `coveredby`(`tgeompoint`, `geometry`) : `bool`, equivalent to `tcoveredby`(A, B) `%=` TRUE
- `equals`({`geometry`, `tgeompoint`}, {`geometry`, `tgeompoint`}) : `bool`, equivalent to `tequals`(A, B) `%=` TRUE
- `touches`(`geometry`, `tgeompoint`) : `bool`, equivalent to `ttouches`(A, B) `?=` TRUE
- `intersects`({`geometry`, `tgeompoint`}, {`geometry`, `tgeompoint`}) : `bool`, equivalent to `tintersects`(A, B) `?=` TRUE
- `disjoint`({`geometry`, `tgeompoint`}, {`geometry`, `tgeompoint`}) : `bool`, equivalent to `tdisjoint`(A, B) `%=` TRUE
- `dwithin`({`geometry`, `tgeompoint`}, {`geometry`, `tgeompoint`}, `double`) : `bool`, equivalent to `tdwithin`(A, B, C) `%=` TRUE

Some of these predicates are also implemented for temporal geographies, as follows:

- `equals`, `intersects`(`tgeogpoint`, `tgeogpoint`) : `tbool`
- `disjoint`(`tgeogpoint`, `tgeogpoint`) : `tbool`
- `dwithin`(`tgeogpoint`, `tgeogpoint`, double) : `tbool`

These predicates are supported by both the GiST and the SP-GiST indexes. The formal semantics of this class of simple topological predicates has been discussed in [21], as a temporal lifting followed by an aggregation over the time domain.

Distance Operators

The spatial distance operator is lifted using Algorithm 3 into the following temporal distance operator:

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

The resulting `tfloat` gives the exact distance at the synchronized time instants and at the turning points of the Euclidean distance function. Between these instants, the value is a linear approximation. This restriction is imposed by the `tfloat` representation in MobilityDB, as described in Section 3. Additionally, the following simple distance operators are implemented:

- {`geometry`, `tgeompoint`} `|=|` {`geometry`, `tgeompoint`}: `float`, the nearest approach distance between A and B.
- `nearestApproachInstant`({`geometry`, `tgeompoint`}, {`geometry`, `tgeompoint`}) : `INSTANT(geometry)`, the instant at which A and B come to their closest distance.
- `shortestLine`({`geometry`, `tgeompoint`}, {`geometry`, `tgeompoint`}) : `geometry(linestring)`, yields the shortest line that connect A, B at their nearest approach instant.

These three functions are defined in OGC Moving Features Access standard [13]. The `|=|` operator is equivalent to finding the minimum value of the lifted distance `<->`. It has the additional merit of index support when used in a k-nearest neighbor (knn) query. The following query finds the three nearest trips (i.e., temporal points) to a given geometry:

```
SELECT Trip |=| geometry 'Point(1 1)' AS Distance
```

```
FROM Trips
ORDER BY Distance
LIMIT 3
```

The query plan of the above query is as follows

```
Limit (cost=0.14..1.73 rows=3 width=8)"
  -> Index Scan using trips_gist_idx on trips (cost=0.14..950.08 rows=1797 width=8)"
       Order By: (trip |=| '0101000000000000000000F03F000000000000F03F'::geometry)"
```

PostgreSQL understands the combination of distance, order, and limit as a knn query and triggers an index if available, as illustrated above. Note that this is different from temporal nearest neighbor queries, e.g., [6, 28, 39, 51], which are not yet implemented in MobilityDB.

Mathematical, Logical, and Comparison Functions

These are also lifted operations:

- {int, tfloat, tint, float} {+, -, *, /} {int, float, tint, tfloat} : {tint, tfloat}, lifted addition, subtraction, multiplication, and division. For the lifted division, if the denominator *ever* has a zero value, an error is thrown, as this is also the behavior of PostgreSQL in standard division.
- {int, tfloat, tint, float, text, ttext} {#=, #<>, #<, #>, #<=, #>=} {int, float, tint, tfloat, text, ttext} : tbool, lifted comparisons.
- {bool, tbool} {&, |} {bool, tbool} : {tbool}, lifted logical AND, OR operations. One use case is to aggregate the result of multiple temporal predicates.
- tbool : {tbool}, lifted logical NOT.

In these operators, only the combinations of arguments that make sense and that contain at least one temporal type are supported. For instance, the temporal comparison operators do not mix numbers with text. The temporal comparison operators are distinguished from the non-temporal counterparts by the # prefix. This is because the non-temporal comparisons are already overloaded with the temporal types to comply with the B-tree index requirements as will be shown later.

Restriction Functions

These functions select parts of the temporal value by restricting either the domain or the range of the underlying temporal function. There are two variants of these functions depending on whether the restriction is performed with respect to either (1) a value/time extent or (2) the complement of a value/time extent. Functions of the first category have the prefix at as follows: atValue, atValues, atRange, atRanges, atMin, atMax, atGeometry, atTimestamp, atTimestampSet, atPeriod, and atPeriodSet. Functions of the second category have the prefix minus as follows: minusValue, minusValues, minusRange, etc.

Functions that restrict the temporal dimension make use of the total ordering property of the time dimension. Therefore, they are generally $O(\log n)$. For example, Algorithm 5 for the atPeriod function uses binary search to locate the period bounds within the list of instants of a temporal SEQUENCE. It then interpolates the values at the bounds, and returns a copy of the restricted part.

On the other hand, the functions that restrict to the value domain have to iterate over all instances. Therefore, they are generally $O(n)$.

Index Comparison Operators

The operators =, <>, <, >, <=, and >= compare two temporal values and return a Boolean. These operators require that the types of the left and right arguments be identical, that is, of the same

Algorithm 5: atPeriod(Seq, P)

begin
$i \leftarrow$ BinarySearch($Seq.instants, P.lower$);
$j \leftarrow$ BinarySearch($Seq.instants, P.upper$);
if $i = -1$ **then** $i \leftarrow 0$;
if $j = -1$ **then** $j \leftarrow Seq.count - 1$;
$s \leftarrow$ interpolate the value at $P.lower$ on the segment from $Seq.instants[i]$ to $Seq.instants[i + 1]$;
$e \leftarrow$ interpolate the value at $P.upper$ on the segment from $Seq.instants[j - 1]$ to $Seq.instants[j]$;
$result \leftarrow$ create a temporal sequence from the instants $s \cup Seq.instants[i + 1, ..., j - 1] \cup e$;
return $result$;

time type and the same base type. Apart from = and <>, the other operators are not useful in the real world but allow B-tree indexes to be constructed on temporal types. For *INSTANT* types, they compare first the timestamps and only if those are equal, compare the value(s). For other temporal types, they compare the first n instants, where n is the minimum of the number of composing instants of both values. The equality and inequality operators compare the binary representation of the two values. Thanks to the normalization of temporal types, equal values will have equal binary representations.

Spatiotemporal Functions

These functions apply transformations and derive attributes of temporal types:

- `trajectory(tpoint)` : {`geometry`, `geography`}, yields the spatial trajectory (a geometry) of A. As already mentioned in Section 3.3, temporal points of the *SEQUENCE* type precompute and cache the spatial trajectory for optimization reasons.
- `transform(tpoint, SRID)` : `tpoint`, changes the spatial projection. SRID (spatial reference identifier) is an integer that uniquely identifies a projection. This OGC compliant function comes from PostGIS, which supports more than 3000 known SRID and the transformation between them.
- `speed(tgeompoint)` : `tfloat`, yields the time-varying speed of A. The spatial unit depends on the projection of the temporal geometry, e.g., meters or feet. The calculated speed is in spatial units per second.
- `twAvg(tfloat)` : `tfloat`, time-weighted average. It yields the area under the curve of the temporal function of A. This operation can be used to summarize a `tfloat`, where values that exist for longer durations will have a bigger effect on the average.
- `twCentroid(tgeompoint)` : `geometry(point)`, independently applying `twAvg` for the x, y, and z (if present) coordinates.
- `length(tgeompoint)` : `float`, the total length of A in the units of the projection.
- `cumulativeLength(tgeompoint)` : `tfloat`, the distance travelled by A as a function in time. This is an OGC standard function [11].
- `azimuth(tgeompoint)` : `tfloat`, the time-varying heading of A. The angle is referenced from North, and is positive clockwise: North = 0, East = $\frac{\pi}{2}$, South = π, West = $\frac{3\pi}{2}$.

- asMFJSON(tpoint) : JSON, exports A into the OGC Moving Features JSON standard format. This function enables data exchange between OGC-compliant tools. Similarly, more data exchange functions exist: asWKT, asWKB, fromMFJSON, fromWKT, etc.

All these functions require a single scan over the instants of the temporal value. Therefore, their complexity is $O(n)$.

Operations on Bounding Box and Time Types

As stated in Section 3, the bounding box of a temporal type depends on its base type: It is the `period` type for the `tbool` and `ttext` types, the `tbox` type for the `tint` and `tfloat` types, and the `stbox` type for the `tgeompoint` and `tgeogpoint` types. The box types and the time types (`timestamp`, `timestampset`, `period`, `periodset`) presented in Section 3.1 are part of the type system and can thus exist independently of temporal types. They can be used as any other type and are further supported by the GiST and SP-GiST indexes. Most of the operations consider a time type as a one-dimensional bounding box.

A first set of operators considers the relative position of the bounding boxes. The operators << (strictly left of), >> (strictly right of), &< (does not extend to the right of), and &> (does not extend to the left of) consider the value dimension for `tint` and `tfloat` types and the x coordinates for the `tgeompoint` and `tgeogpoint` types. The operators <<| (strictly below of), |>> (strictly above of), &<| (does not extend above of), and |&> (does not extend below of), <</ (strictly front), />> (strictly back), /&< (does not extend in front of), and /&> (does not extend at back of) consider, respectively, the y and z coordinates for the `tgeompoint` and `tgeogpoint` types. Finally, the operators <<# (strictly before), #>> (strictly after), #&< (does not extend after), and #&> (does not extend before) consider the time dimension for all temporal types.

Another set of operators consider the topological relationships between the bounding boxes. There are four topological operators: overlaps (&&), contains (@>), contained (<@), and equals (~=). The arguments of these operators can be a base type, a range, a time type, a box, or a temporal type. These operators are polymorphic and verify the topological relationship taking into account the value and/or the time dimension depending on the type of the arguments. Bounding box predicates are meant for invoking indexes in the query plans. The other functions in the API that can benefit from indexes will implicitly call a bounding box predicate. The signature of these two sets of operators is as follows, where *TIME* refers to any time type, and *BOX* refer to any box type:

$$\{\mathit{TEMPORAL}, \mathit{TIME}, \mathit{BOX}\} \times \{\mathit{TEMPORAL}, \mathit{TIME}, \mathit{BOX}\} \rightarrow \texttt{bool}$$

Additionally, the following set operations are defined for time and box types:

- *TIME* + *TIME* : *TIME*, *BOX* + *BOX* : *BOX* yields the union of A and B.
- *TIME* * *TIME* : *TIME*, *BOX* * *BOX* : *BOX* yields the intersection of A and B.
- *TIME* - *TIME* : *TIME*, *BOX* - *BOX* : *BOX* yields the difference of A and B.

In the case of boxes, an error is raised if the result of a union or difference is discontinuous, as this cannot be represented.

Temporal/Time/Box Casting

While the operations on box and time types are mainly used for index support, there are situations where they are useful in user queries. These situations are further enabled by casting methods from a temporal type into a period or a box type. Casts can be implicit, which allows automatic conversion of a function argument to a type supported by the function.

```
SELECT tgeompoint '[Point(0 1)@2012-01-01, Point(1 1)@2012-01-03)'&&
       period'[2012-01-02, 2012-01-04]'
```

```
Result: TRUE
```

In the example, the overlaps operator (&&) forces an implicit cast of the `tgeompoint` into a period, i.e., 1D box. It then performs a period overlap test, which succeeds in this case. A similar casting into a spatial 2D or 3D box is possible. The casting can also be explicitly called using the :: operator.

```
SELECT tgeompoint '[Point(0 1)@2012-01-01, Point(1 1)@2012-01-03)'::stbox
Result: 'STBOX T((0,1,2012-01-01 00:00:00+00),(1,1,2012-01-03 00:00:00+00))'
```

Utility Functions

The complex structure of the temporal types requires functions to access the different elements of their structures, to construct temporal values, and to cast between the different temporal types. Some metadata about the temporal attribute is accessed by the functions `memSize`(.) that returns the memory size and `temporalType`(.) that returns the temporal type (*INSTANT*, *INSTANTS*, *SEQUENCE*, *SEQUENCES*). The array of timestamps and the array of values that constitute the temporal type are respectively accessed by the functions `instants`(.) and `getValues(.)`. There are also functions to access the time or the value of a specific instant, the minimum value, the maximum value, and the bounding range/box for the time dimension and the value dimension.

Each temporal type has a constructor function with the same name as the type, so that it is possible to create temporal values in the user query. The constructor functions for the temporal instant types have two arguments, a value of the base type and a timestamp. The constructor functions of the temporal instant set types have one argument, an array of values of the corresponding instant type. The constructor functions for the temporal sequence types have four arguments, an array of the corresponding temporal instant type and three Boolean values determining whether the first and last instants are inclusive or not and whether the interpolation is linear or not. The constructor functions of the temporal sequence set types have two arguments, an array of values of the corresponding unit type and a Boolean value determining whether the interpolation is linear or not. As stated in Section 3, temporal types are converted into a normal form so that equivalent values have identical representations.

Both the time type and the base type of a temporal value can be casted into other compatible types. Accordingly, the temporal type also changes. The time type can be casted using the functions `temporalinst`(.), `temporali`(.), `temporalseq`(.), and `temporals`(.). It is possible to cast the time type as follows: `timestamp` to `timestampset`, `period`, or `periodset`, `timestampset` to `periodset`, and `period` to `periodset`. The base type can be casted. The possible conversions are `int` to/from `float` and `geometry` to/from `geography`.)

Temporal Aggregation

A temporal aggregation function processes a set of values of some temporal type and returns a single temporal value. In contrast to the other temporal functions, temporal aggregates are defined over the *union* of the definition times of the input. This is shown in the following example, which aggregates two `tfloat` values using the temporal `tmin` function:

```
SELECT tmin(seq) FROM (VALUES
    (tfloat '[4@2001-01-01, 1@2001-01-04]'),
    (tfloat '[1@2001-01-02, 4@2001-01-05]')) AS T(seq);
Result: "{[4@2001-01-01, 3@2001-01-02), [1@2001-01-02, 2@2001-01-03, 1@2001-01-04],
    (3@2001-01-04, 4@2001-01-05]}"
```

The definition time of the result is the union of the definition times of the inputs. In contrast to standard aggregation, temporal aggregation may return a result which is of a bigger size than the

input. For this reason, the temporal aggregate functions should be extremely optimized in order to perform efficiently. In the example above, while the temporal function of both input values has a single piece, the function of the resulting value has 4 pieces. For computing the aggregation, the segments of the input values must be temporally synchronized by breaking them with the end points of one another. In the example above, the segments of the two input values were broken in the periods [2001-01-01, 2001-01-02), [2001-01-02, 2001-01-04), and [2001-01-04, 2001-01-05]. Moreover, the aggregation function can further break the segments according to its logic. In the example above, the segments of the input values were further broken into the periods [2001-01-02, 2001-01-03) and [2001-01-03, 2001-01-04) since the functions of the input values intersect at time 2001-01-03.

MobilityDB implements the following temporal aggregation functions:

- `tcount`(*TEMPORAL*) : `tint`, counts the number of values that exist at every point in time, over the union of the definition time of all the aggregated values.
- {`tmax`, `tmin`, `tsum`, `tavg`} (`tfloat`) : `tfloat`, temporal versions of the traditional SQL aggregates. They apply the corresponding traditional aggregate at every point in time over the union of the definition time of all the aggregated values.
- {`tand`, `tor`} (`tbool`) : `tbool`, temporal Boolean aggregation.
- `tcentroid`(`tpoint`) : `tpoint`, computes the center of mass of a set of temporal points at every time instant. It can be used, for example, to find a representative trajectory for a flock of birds (i.e., by computing the temporal centroid of their individual trajectories).

These temporal aggregates are different from the classes of aggregates in the temporal database literature, e.g., [35, 56]. These works are centered around defining temporal grouping and partitioning methods over timestamped tuples. The aggregation over these groups and partitions is then done using the standard SQL functions. Accordingly, the expressiveness covers only discrete value evolution. This limitation has been partially addressed in [7], where attributes were allowed to a have linear proportionality relationship with the time interval, so called malleable attributes. An attribute can thus partially participate in multiple groups. The class of temporal aggregates mentioned here is different because it aggregates temporal functions. Moreover, the aggregation is orthogonal to the SQL grouping and partitioning. For instance, one can write a query that combines a standard aggregate, and a temporal one:

```
SELECT count(Trip), tcount(Trip)
FROM Trips
```

where `Trips` is a relation that includes a `tgeompoint` attribute `Trip`. The first is a simple count of the number of tuples in the relation. The second is a temporal count of number of defined trips at every time instant, i.e., a `tint`.

A detailed discussion of temporal aggregation is out of the scope of this paper.

4.4 Comparison of the Sequence and the Sliced Representations

Section 3 has described the *sequence representation* of moving objects, which is used as the MobilityDB data model. The alternative to this model is the *sliced representation*, which is well rooted in the moving object database literature. The latter has been proposed in [22] as a discrete representation of moving objects. Since then, it has been repeatedly used in several works, for instance in [45]. It encodes the evolution of values of a temporal type in time periods, called *units*. The *UNIT* type constructor is a pair of a time interval and a function that describes the evolution of the object during this time interval, i.e., a temporal function. The moving object is represented as a *MAPPING*, which is set of units that do not temporally overlap.

Because the temporal functions of the units are independent, it is possible to represent distinct changes in the object value at the boundary between two units. It is also possible to introduce temporal gaps between units. One unit can represent a single time instant, if its time interval has equal start and end timestamps. The sliced representation can uniformly represent all these scenarios using the two type constructors: *UNIT*, and *MAPPING*.

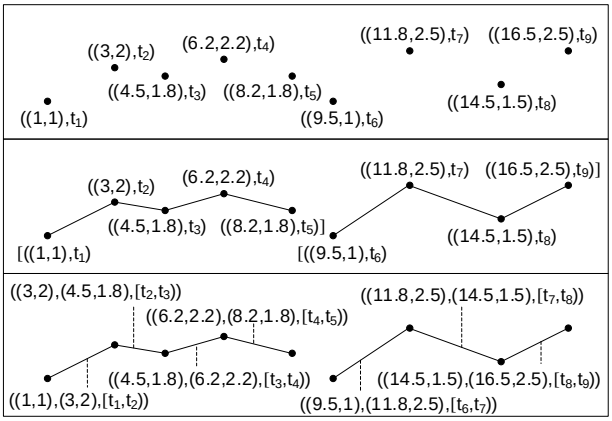


Fig. 6. Sequence versus sliced representation

The sequence representation achieves the same expressiveness by defining more type constructors to address the different evolution scenarios: *INSTANT*, *INSTANTS*, *SEQUENCE*, and *SEQUENCES*. As such, the specificities of every class of types can be used to optimize the storage and the algorithms. Figure 6 shows an example comparison between the two representations. The top rectangle in the figure shows the input events coming from some sensor, where an event is a pair of a point coordinate and a timestamp t_i. The example assumes that there is a break in the input between t_5 and t_6, e.g., a new recording is started. The sequence representation creates two sequences out of these input events: during $[t_1, t_5]$ and $[t_6, t_9]$. The sliced representation creates a list of *units*, each representing a segment between two consecutive events. It thus duplicates the intermediate events, e.g., $((3, 2), t_2)$ is stored once as the right bound of the first unit and once more as the left bound of the second unit.

The first clear advantage of the sequence representation is that it achieves a storage reduction of about half the storage required for the sliced representation. This is because it avoids duplicating the intermediate events, it stores a single timestamp instead of a time interval per segment, and it does not store artificial close-open interval bounds in the intermediate events. This size reduction improves the query performance because of the reduced I/O and the reduced memory management. The second advantage is that the algorithms can be optimized, because the types encode more information. Specifically, the temporal continuity information is encoded in the representation. In the following, we elaborate this algorithmic comparison.

Algorithm 6 shows the general algorithmic strategy used for a binary temporal operation using the sliced representation, e.g., the addition of a pair of temporal floats. The algorithm consists basically of three steps. The first step is to temporally synchronize the two temporal arguments, producing their synchronized intervals *syncPeriods*. The second step is that for every synchronized period, the corresponding segments (or subsegments) of the two objects are fetched. For the sliced representation it must be tested whether the two objects are defined during the synchronized interval. This amounts to $O(n)$ checks, where n is the smaller number of segments in the two

arguments α_1 and α_2. The third step is to merge the consecutive segments in the result whenever possible, to maintain a unique minimal representation. In this step, the temporal continuity over the two segments must be checked. This amounts to another $O(n)$ checks, where n is the number of segments in the result. When the sequence representation is used these two $O(n)$ continuity checks (underlined in the algorithm) are not needed. This is because the continuity is guaranteed by the representation.

Algorithm 6: Generic strategy for a binary temporal operation on two temporal arguments α_1, α_2

```
begin
    result ← ∅;
    syncPeriods ← synchronize(α1, α2);
    foreach period p ∈ syncPeriods do
        if α1 is defined during p AND α2 is defined during p then
            resultUnit ← apply the operation on α1 and α2 both restricted to the period p;
            result ← result ∪ resultUnit;
    foreach consecutive pair of units u_i, u_{i+1} ∈ result do
        if end timestamp of u_i = start timestamp of u_{i+1} AND
        the temporal functions of u_i and u_{i+1} are equivalent then
            merge u_i, u_{i+1} into one unit;
    return result;
```

A similar conclusion can be drawn for unary operations. Unary operations need to perform $O(n)$ continuity checks to maintain the normal form of the result. Some of them additionally need $O(n)$ checks during the processing. For instance, the `trajectory`(.) operator produces the spatial projection of a temporal point as a `geometry`. In the sequence representation, the operator iterates over all the points and produce a line string from all of them. In the sliced representation, it produces a line segment for each unit. To integrate these line segments into a `LineString`, every pair of consecutive segments are checked whether they share a common end point (i.e., continuity check).

It is perhaps for these reasons that the state of practice has implementations that share the scheme of the proposed sequence representation. The PostGIS `LineStringM` type stores a polyline (2D or 3D), as well as additional measure values attached to vertices. One use case for the M measure is to store a timestamp per vertex. As such, the `LineStringM` object represents a trajectory. PostGIS additionally provides four basic trajectory processing functions for this representation[4]. It is possible to represent temporal gaps and distinct value changes by using the `MultiLineStringM` type, which is a set of `LineStringM`.

A second example is the ISO 19141:2008 Geographic information - Schema for moving features standard[5]. It defines a data model for moving point and rigid moving region objects. One *leaf* represents the geometry of the moving object at a given time. The overall movement is described as a set of leaves, called a *foliation*. The position along the trajectory (i.e., between leaves) can be described using a linear reference system. Again, to represent gaps and distinct value changes, a *feature collection* object shall be used.

The disadvantage of the sequence representation is that it uses more types than the sliced representation. This increases the development effort.

[4]https://postgis.net/docs/reference.html#Temporal
[5]https://www.iso.org/standard/41445.html

4.5 Indexing

Among the several types of indexes provided by PostgreSQL, MobilityDB implements B-Tree, GiST, and SP-GiST indexes for the `timestampset`, `period`, and `periodset` time types, the box types, and the temporal types. B-Tree indexes support balanced trees that handle equality and range queries on data that can be sorted into some ordering. In MobilityDB, these are primarily meant to allow sorting internally in queries (e.g., in the ORDER BY clause) rather than for creating actual indexes. GiST (Generalized Search Tree) [33] indexes support balanced trees and act as a template for implementing various indexing schemes. In MobilityDB, the GiST indexes implement an R-tree over the indexed type. Finally, SP-GiST (Space-Partitioned GiST) [20] indexes support unbalanced trees that repeatedly divide the search space into partitions that need not be of equal size and also act as a template for implementing various indexing schemes. In MobilityDB, the SP-GiST indexes implement a quadtree/octree depending on the number of dimensions of the indexed type.

An essential characteristic of B-trees, GiST, and SP-GiST indexes is that they are generic. In the case of B-trees, this means that they support any data type that provides comparison operators such as = and <. In the case of GiST and SP-GiST indexes, this means that they support any data type but also that the operators that can be used with the index are specified in an operator class. For example, the left bounding box operator << can be used by an index for both temporal floats and temporal points, while the below bounding box operator `<<|` is only used for temporal points. Furthermore, in GiST and SP-GiST indexes, the data type actually stored in the index may be different from the column data type. In MobilityDB we store bounding boxes/periods for these indexes. As a consequence of this, the index may be lossy, in the sense that the rows returned by the index do not necessarily satisfy the predicate. A refinement step that applies the predicate on the original value, rather than its bounding box, would then be required.

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 as follows:

- A 2-dimensional point to represent the lower and upper bounds of the bounding period for the time types and the `tbool` and `ttext` types.
- A 4-dimensional point to represent the minimum and the maximum values of the 2-dimensional bounding box for the temporal number types `tint` and `tfloat`.
- An 8-dimensional point to represent the minimum and the maximum values of the 4-dimensional bounding box for the temporal point types `tgeompoint` and `tgeogpoint`.

The tree structure of an SP-GiST index splits every dimension into two halves at every tree level. Thus, while the SP-GiST index for the time types and the `tbool` and `ttext` types implements a traditional quadtree in 2D space, the index for the `tint` and `tfloat` types splits the tree into 16 quadrants in 4D space. This can be conceptualized as splitting space two times into four quadrants, that is, one for the lower bound corner and another for the upper bound corner of the 2D bounding box. Similarly, the index for temporal points uses an octree analogy where the tree is split into 256 octants in 8D space. This can be conceptualized as splitting space two times into 64 octants, one for the lower bound corner and another for the upper bound corner of the 4D bounding box. The performance tests reveal that this technique is especially beneficial when there are many overlapping objects, that is, with so-called "spaghetti data".

A GiST or SP-GiST index can accelerate queries involving bounding box predicates (see Section 4) and nearest neighbor queries involving the `|=|` operator. Our spatiotemporal indexes considers as many dimensions as they are shared in the indexed column and in the query argument. If the search argument is a temporal point, then both the value and the time dimensions are considered

for querying the index, while if the right argument is a `geometry`/`geography` or a time type, then either the value or the time dimension will be used.

The aim of this polymorphic behaviour of the index is to support a wider range of user queries without creating multiple indexes (i.e., a spatiotemporal, a spatial-only, and a temporal-only index). One possible approach to achieve such polymorphism is to extend the missing dimensions of the arguments of the index predicate with a default value such as $\pm\infty$. However, this approach only works for the overlaps (`&&`) and the contained by (`<@`) operators. When the missing dimensions are extended with $\pm\infty$, the other operators such as equals (`~=`), contains (`@>`), etc. will always evaluate to false, giving a wrong result. For this reason, in MobilityDB the indexes have different functions that accept spatiotemporal types, spatial types, and temporal types. Note however that querying a spatiotemporal index with spatial or temporal arguments might not perform as good as a spatial-only or a temporal-only index.

5 BENCHMARK

In this section we assess the performance of MobilityDB and compare it to SECONDO using the BerlinMOD [18] benchmark of moving object databases. The benchmark contains a data generator and representative queries. The data generator aims to model a person's trips to and from work during the week as well as some leisure trips at evenings or weekends. It can be customized with a scale factor that determines both the number of vehicles and the number of simulation days. For instance, scale factor 1.0 simulates 2000 vehicles for 28 days, starting on a Monday. With any other scale factor, these values are scaled by the square root of the chosen scale factor. The schema that is generated and used by BerlinMOD is as follows:

```
CREATE TABLE Cars (
    CarId integer PRIMARY KEY,
    Licence varchar(32),
    Type varchar(32),
    Model varchar(32)
);
CREATE TABLE Points (
    PointId integer PRIMARY KEY,
    Geom Geometry(Point)
);
CREATE TABLE Regions (
    RegionId integer PRIMARY KEY,
    Geom Geometry(Polygon)
);
```

```
CREATE TABLE Trips (
    CarId integer REFERENCES Cars,
    TripId integer,
    Trip tgeompoint,
    PRIMARY KEY (CarId, TripId)
);
CREATE TABLE Instants (
    InstantId integer PRIMARY KEY,
    Instant timestamptz
);
CREATE TABLE Periods (
    PeriodId integer PRIMARY KEY,
    Period period
);
```

```
CREATE UNIQUE INDEX Cars_CarId_Idx ON Cars USING btree (CarId);
CREATE INDEX Trips_CarId_Idx ON Trips USING btree (CarId);
CREATE UNIQUE INDEX Trips_CarId_TripId_Idx ON Trips USING btree (CarId, TripId);
CREATE INDEX Trips_Trip_Idx ON Trips USING gist (Trip);
CREATE INDEX Points_Geom_Idx ON Points USING gist (Geom);
CREATE INDEX Regions_Geom_Idx ON Regions USING gist (Geom);
CREATE INDEX Instants_Instant_Idx ON Instants USING btree (Instant);
CREATE INDEX Periods_Per_Idx ON Periods USING gist (Per);

CREATE VIEW Points1 AS SELECT * FROM Points LIMIT 10;
```

```
CREATE VIEW Regions1 AS SELECT * FROM Regions LIMIT 10;
CREATE VIEW Instants1 AS SELECT * FROM Instants LIMIT 10;
CREATE VIEW Periods1 AS SELECT * FROM Periods LIMIT 10;
CREATE VIEW Licences1 AS SELECT * FROM Licences LIMIT 10;
CREATE VIEW Licences2 AS SELECT * FROM Licences LIMIT 10 OFFSET 10;
```

Table `Cars` stores static information about the cars while the information about their trips is stored in table `Trips`. Tables `Points`, `Regions`, `Instants`, and `Periods` define locations and times used for expressing query conditions with respect to the trips. Each of these query tables has 100 tuples, irrespective of the scale factor. Some queries need to combine multiple of these query tables. In this case, BerlinMOD does not use that whole 100 tuples, and rather uses the top 10 tuples of each. For these queries, we additionally define the views that select the 10 rows of the corresponding tables. The tables have indexes on traditional, spatial, temporal, or spatiotemporal attributes.

BerlinMOD comes with two sets of queries, namely, BerlinMOD/R and BerlinMOD/NN, which are, respectively, range queries and nearest neighbour (NN) queries for historical moving object databases. The BerlinMOD/NN part is preliminary and it is not implemented neither in SECONDO [18] nor in MobilityDB. In this comparison, we will only cover the BerlinMOD/R queries, which are 17 queries of different types that involve query points, regions, instants, periods, and vehicle identities/licences as parameters. The queries are classified, as detailed in [18], according to five criteria: whether or not the object identity is given, the query dimensionality (spatial, temporal, spatiotemporal), the query interval (time instant, duration, unbounded), the condition type (single object, object relation), and whether or not the query computes aggregations. Every query addresses a different combination of these criteria.

In the following we first show the 17 BerlinMOD/R queries, and how they are expressed in SQL in MobilityDB. Then, we show the performance comparison with SECONDO using these queries in multiple scale factors. In SECONDO the queries are written in a special procedural language called *SECONDO executable*. Their scripts will not be shown here, as they can be found on the BerlinMOD Webpage[6].

Query 1. What are the models of the vehicles with licence plate numbers from `Licences`?

```
1 SELECT DISTINCT L.Licence, C.Model AS Model
2 FROM Cars C, Licences L
3 WHERE C.Licence = L.Licence;
```

This is a traditional equijoin query that does not involve spatiotemporal operations.

Query 2. How many vehicles exist that are “passenger" cars?

```
1 SELECT COUNT (Licence)
2 FROM Cars C
3 WHERE Type = 'passenger';
```

This is also a traditional aggregation query that does not involve spatiotemporal operations. As mentioned in the benchmark design [18], some DBMS might partially access Trip data, which is not needed, but shall result in performance losses. Queries 1,2 assess this.

Query 3. Where have the vehicles with licences from Licences1 been at each of the instants from Instants1?

```
1 WITH ClippedTrips AS (
2     SELECT DISTINCT T.CarId, I.InstantId, I.Instant,
```

[6] http://dna.fernuni-hagen.de/secondo/BerlinMOD/BerlinMOD.html

```
3     valueAtTimestamp(T.Trip, I.Instant) AS Pos
4     FROM Trips T, Instants1 I
5     WHERE T.Trip @> I.Instant AND valueAtTimestamp(T.Trip, I.Instant) IS NOT NULL )
6   SELECT L.Licence, T.InstantId, T.Instant, T.Pos
7   FROM ClippedTrips T, Licences1 L
8   WHERE T.CarId = L.CarId
9   ORDER BY L.Licence, T.InstantId
```

In this query and in the following ones, the `WITH` statement will often be used. It defines a temporary named result set (i.e., common table expression or CTE) that can be further used within the query. We use it to help the query optimizer in choosing an efficient execution plan. The idea is to split a query with many conjunctive filter conditions into a sequence of more simple queries. Through this splitting, we hint the query optimizer to invoke the right combination of indexes. The `WITH` statement in this query, for instance, filters the trips to only those that include time instants from `Instants1` and restricts them to these time instants. The predicate `T.Trip @> I.Instant` triggers the use of the spatiotemporal index on `Trip` (`Trips_Trip_Idx`). Since the argument of the predicate is a time instant, only the temporal dimension of the index is traversed. As described in Section 4.5, the spatiotemporal indexes in MobilityDB have this polymorphic behaviour, that allows them to also act as spatial only and temporal only indexes. The main query joins the view with the `Licences1` table and produces the required output.

To further illustrate the making of the queries, we compare next two versions of Query 3; the one shown above, and a flat version without the `WITH` statement. The execution plan that is produced by PostgreSQL for the above version is given next. The upper part shows the execution plan of the `WITH` statement, which indeed invokes the GiST index `trips_gist_idx` using the predicate `T.Trip @> I.Instant`. The lower part shows the execution plan of the main query, which joins the CTE with `Licences1`.

```
1   Sort  (cost=646.28..646.51 rows=89 width=52)
2     Sort Key: l.licence, t.instantid
3     CTE ClippedTrips
4       ->  HashAggregate  (cost=577.17..599.52 rows=1788 width=48)
5             Group Key: t_1.carid, instantid, instants.instant, valueattimestamp(trip,
                  instant)
6             ->  Nested Loop  (cost=3.88..559.29 rows=1788 width=48)
7                   ->  Limit  (cost=0.00..0.20 rows=10 width=12)
8                         ->  Seq Scan on instants  (cost=0.00..2.00 rows=100 width=12)
9                   ->  Bitmap Heap Scan on trips t_1  (cost=3.88..55.27 rows=18 width=175)
10                        Recheck Cond: (trip @> instants.instant)
11                        Filter: (valueattimestamp(trip, instants.instant) IS NOT NULL)
12                        ->  Bitmap Index Scan on trips_gist_idx  (cost=0.00..3.88 rows=18
                              width=0)
13                              Index Cond: (trip @> instants.instant)
14    ->  Hash Join  (cost=0.53..43.88 rows=89 width=52)
15          Hash Cond: (t.carid = l.carid)
16          ->  CTE Scan on ClippedTrips t  (cost=0.00..35.76 rows=1788 width=48)
17          ->  Hash  (cost=0.40..0.40 rows=10 width=12)
18                ->  Subquery Scan on l  (cost=0.00..0.40 rows=10 width=12)
19                      ->  Limit  (cost=0.00..0.30 rows=10 width=16)
20                            ->  Seq Scan on licences  (cost=0.00..3.00 rows=100 width=16)
```

Alternatively, Query 3 could be expressed in SQL as follows:

```
SELECT DISTINCT L.Licence, I.InstantId, I.Instant AS Instant,
   valueAtTimestamp(T.Trip, I.Instant) AS Pos
FROM Trips T, Licences1 L, Instants1 I
WHERE T.CarId = L.CarId AND valueAtTimestamp(T.Trip, I.Instant) IS NOT NULL
ORDER BY L.Licence, I.InstantId
```

This version is twice slower than the previous one. It leaves the query optimizer without hints. The optimizer then chooses the execution plan based on the statistical information available to it and the accuracy of selectivity estimation. In practice, query optimizers can find good plans, but they often miss the optimal ones, specially for complex queries. The execution plan of this version is as follows:

```
Unique (cost=269.31..276.89 rows=607 width=52)
  -> Sort (cost=269.31..270.82 rows=607 width=52)
       Sort Key: licences.licence, instants.instantid, instants.instant, (
           valueattimestamp(t.trip, instants.instant))
       -> Nested Loop (cost=0.28..241.25 rows=607 width=52)
            Join Filter: (valueattimestamp(t.trip, instants.instant) IS NOT NULL)
            -> Nested Loop (cost=0.28..230.25 rows=61 width=179)
                 -> Limit (cost=0.00..0.30 rows=10 width=16)
                      -> Seq Scan on licences (cost=0.00..3.00 rows=100 width=16)
                 -> Index Scan using trips_carid_idx on trips t
                      Index Cond: (carid = licences.carid)
                 -> Limit (cost=0.00..0.20 rows=10 width=12)
                      -> Seq Scan on instants (cost=0.00..2.00 rows=100 width=12)
```

In contrast to the previous version, this one uses the btree index `trips_carid_idx`. This is not an optimal choice by the optimizer, since this plan is twice slower than the previous one. The conclusion of this illustration is that MobilityDB queries, as it is also the case with regular SQL queries, require tweaking to hint the optimizer towards optimal execution plans. The following queries have been optimized as such.

Query 4. Which vehicles have passed the points from Points?

```
SELECT DISTINCT P.PointId, P.geom, T.CarId
FROM Trips T, Points P
WHERE st_intersects(trajectory(T.Trip), P.geom)
ORDER BY P.PointId, T.CarId
```

This is a spatial range query as it ignores the temporal dimension. The `trajectory` operator is part of MobilityDB, and it computes the spatial projection of the temporal point. The `st_intersects` predicate is provided by PostGIS. Before computing the intersection, it internally performs a bounding box comparison `trajectory(T.Trip) && P.Geom` using the spatial index on table `Points`.

Query 5. What is the minimum distance between places, where a vehicle with a licence from Licences1 and a vehicle with a licence from Licences2 have been?

```
SELECT L1.Licence AS Licence1, L2.Licence AS Licence2,
   MIN(st_distance(trajectory(T1.Trip), trajectory(T2.Trip))) AS MinDist
FROM Trips T1, Licences1 L1, Trips T2, Licences2 L2
WHERE T1.CarId = L1.CarId AND T2.CarId = L2.CarId AND T1.CarId < T2.CarId
```

```
5  GROUP BY L1.Licence, L2.Licence
6  ORDER BY L1.Licence, L2.Licence
```

The distance in this query is a spatial distance between the spatial projections of the trips. The query computes this distance for two trips `T1` and `T2` of distinct cars and then aggregates for the minimum. The eqijoin conditions on `CarId` invoke the btree index of the `Trips` table.

Query 6. What are the pairs of trips from Licences 1 that have ever been as close as 10m or less to each other?

```
1  SELECT DISTINCT C1.Licence AS Licence1, C2.Licence AS Licence2
2  FROM Trips T1, Licences1 C1, Trips T2, Licences1 C2
3  WHERE T1.CarId = C1.CarId AND T2.CarId = C2.CarId
4      AND T1.CarId < T2.CarId
5      AND T1.Trip && expandSpatial(T2.Trip, 10)
6      AND tdwithin(T1.Trip, T2.Trip, 10.0) ?= true
7  ORDER BY C1.Licence, C2.Licence
```

The main predicate in this query is in Line 6. The `tdwithin` lifted predicate returns a `tbool` which is *true* when the the two trips have a distance within 10 meters. The ever equals operator `?=` checks whether this `tbool` has some true values. The remaining predicates join only pair of trips `T1` and `T2` of distinct cars, and perform a bounding box comparison with the `&&` operator. This uses the spatiotemporal index on the `Trips` table, where the bounding box of `T2` is expanded by 10 m.

Query 7. What are the licence plate numbers of the “passenger" cars that have reached the points from Points1 first of all “passenger" cars during the complete observation period?

```
1  WITH Timestamps AS (
2      SELECT DISTINCT C.Licence, P.PointId, P.geom,
3      MIN(startTimestamp(atValue(T.Trip, P.geom))) AS Instant
4      FROM Trips T, Cars C, Points1 P
5      WHERE T.CarId = C.CarId AND C.Type = 'passenger'
6          AND T.Trip && P.geom
7          AND st_intersects(trajectory(T.Trip), P.geom)
8      GROUP BY C.Licence, P.PointId, P.geom
9  )
10 SELECT T1.Licence, T1.PointId, T1.geom, T1.Instant
11 FROM Timestamps T1
12 WHERE T1.Instant <= ALL(
13     SELECT T2.Instant FROM Timestamps T2 WHERE T1.PointId = T2.PointId )
14 ORDER BY T1.PointId, T1.Licence
```

The CTE finds, for every pair of a passenger car and a query point, the first time instant where the car traversed the point. The main query then finds for every point the car that traversed it first. Two indexes are invoked in the CTE. The btree index on `C.CarId` is used in the equijoin in Line 5. Additionally, the GiST index on `T.Trip` is triggered by the overlap operator (`&&`) in Line 6. Only the spatial dimension of this index is traversed, since the query argument `P.geom` is spatial.

Query 8. What are the overall travelled distances of the vehicles with licence plate numbers from Licences1 during the periods from Periods1?

```
1  SELECT L.Licence, P.PeriodId, P.Period,
2      SUM(length(atPeriod(T.Trip, P.Period))) AS Dist
3  FROM Trips T, Licences1 L, Periods1 P
```

```
4  WHERE T.CarId = L.CarId AND T.Trip && P.Period
5  GROUP BY L.Licence, P.PeriodId, P.Period
6  ORDER BY L.Licence, P.PeriodId
```

The query performs a bounding box comparison with the && operator using the spatiotemporal index on the Trips table. Here it acts as a temporal index. The query then projects the trip to the period, computes the length of the projected trip, and sums the lengths of all the trips of the same car during the period.

Query 9. What is the longest distance that was travelled by a vehicle during each of the periods from Periods1?

```
1  WITH Distances AS (
2      SELECT P.PeriodId, P.Period, T.CarId,
3          SUM( length( atPeriod( T.Trip, P.Period ) ) ) AS Dist
4      FROM Trips T, Periods P
5      WHERE T.Trip && P.Period
6      GROUP BY P.PeriodId, P.Period, T.CarId
7  )
8  SELECT PeriodId, Period, MAX( Dist ) AS MaxDist
9  FROM Distances
10 GROUP BY PeriodId, Period
11 ORDER BY PeriodId
```

This query is similar to the previous one, except that it groups the distance summation by period. The CTE is similar to the previous query, so it is optimized in the same way. The main query groups the CTE results by period, and produces the final result.

Query 10. When and where did the vehicles with licence plate numbers from Licences1 meet other vehicles (distance < 3m) and what are the latters' licences?

```
1  WITH License1Trips AS (
2      SELECT T.CarId, T.Trip, L.Licence
3      FROM Trips T, Licences1 L
4      WHERE T.CarId = L.CarId ),
5  Meetings AS (
6      SELECT T1.Licence AS Licence1, T2.CarId AS Car2Id, T1.Trip AS Trip,
7          getTime( atValue( tdwithin( T1.Trip, T2.Trip, 3.0 ), TRUE ) ) Periods
8      FROM License1Trips T1, Trips T2
9      WHERE T2.Trip && expandspatial( T1.trip, 3.0 ) )
10 SELECT DISTINCT T.Licence1, L2.Licence AS Licence2, Periods AS meetingTime,
11     atPeriodset(T.Trip, Periods) meetingPlace
12 FROM Meetings T, Cars L2
13 WHERE T.Car2Id = L2.CarId AND Periods IS NOT NULL
```

This query has two CTEs. The License1Trips CTE joins the Trips and the Licences1 tables using the btree index on Trips.CarId. The Meetings CTE joins this result further with the Trips table. For every pair of trips of distinct cars it performs a bounding box comparison with the && operator using the spatiotemporal index on the Trips table, where the bounding box of T2 is expanded by 10 m. Then, the expression in Line 7 computes the periods during which the cars were within 10 m from each other. The atPeriodset function in Line 11 projects the trips to those periods.

Query 11. Which vehicles passed a point from Points1 at one of the instants from Instants1?

```
1  WITH Passes AS (
2      SELECT P.PointId, P.geom, I.InstantId, I.Instant, T.CarId
3      FROM Trips T, Points1 P, Instants1 I
4      WHERE T.Trip @> gbox(P.geom, I.Instant) AND
5          st_equals( valueAtTimestamp( T.Trip, I.Instant ), P.geom ) )
6  SELECT T.PointId, T.geom, T.InstantId, T.Instant, C.Licence
7  FROM Passes T JOIN Cars C ON T.CarId = C.CarId
8  ORDER BY T.PointId, T.InstantId, C.Licence
```

The main predicate in this query is in Line 5. It accepts only the trips that pass a point from `Points1` at an instant from `Instants1`. The predicate in Line 4 helps optimizing it by triggering the spatiotemporal index on `T.Trip` using the bounding box containment predicte `@>`. The main query joins this result with the table `Cars` to fetch the licence numbers, and produces the final result.

Query 12. Which vehicles met at a point from Points1 at an instant from Instants1?

```
1  WITH Passes AS (
2      SELECT DISTINCT P.PointId, P.geom, I.InstantId, I.Instant, T.CarId
3      FROM Trips T, Points1 P, Instants1 I
4      WHERE T.Trip @> gbox(P.geom, I.Instant) AND
5          st_equals( valueAtTimestamp( T.Trip, I.Instant ), P.geom ) )
6  SELECT DISTINCT C1.Licence AS Licence1, C2.Licence AS Licence2
7  FROM Passes T1 JOIN Cars C1 ON T1.CarId = C1.CarId JOIN
8      Passes T2 ON T1.CarId < T2.CarId AND T1.PointID = T2.PointID AND T1.InstantId = T2.
           InstantId
9      JOIN Cars C2 ON T2.CarId = C2.CarId
10 ORDER BY C1.Licence, C2.Licence
```

The `Passes` CTE in this query is the same as in the previous query. The main query self joins the CTE result to combine pairs of cars that met at the same point and the same instant. It then joins twice with the `Cars` table to fetch the licence plates of the two cars.

Query 13. Which vehicles travelled within one of the regions from Regions1 during the periods from Periods1?

```
1  WITH Passes AS(
2      SELECT DISTINCT R.RegionId, P.PeriodId, P.Period, T.CarId
3      FROM Trips T, Regions1 R, Periods1 P
4      WHERE T.trip && gbox( R.geom, P.Period )
5          AND st_intersects( trajectory( atPeriod( T.Trip, P.Period ) ), R.geom )
6      ORDER BY R.RegionId, P.PeriodId
7  )
8  SELECT DISTINCT T.RegionId, T.PeriodId, T.Period, C.Licence
9  FROM Passes T, Cars C
10 WHERE T.CarId = C.CarId
11 ORDER BY T.RegionId, T.PeriodId, C.Licence
```

The query is again similar to the previous two queries, except that it restricts the trips using regions and periods rather than points and time instants. Therefore, the `st_intersects` predicate

is used here instead of st_equals. Yet it is optimized in the same way, using the bbox overlap predicate in Line 4, that invokes the index.

Query 14. Which vehicles travelled within one of the regions from Regions1 at one of the instants from Instants1?

```
1 WITH Passes AS (
2     SELECT DISTINCT R.RegionId, I.InstantId, I.Instant, T.CarId
3     FROM Trips T, Regions1 R, Instants1 I
4     WHERE T.Trip && gbox(R.geom, I.Instant) AND
5         st_contains( R.geom, valueAtTimestamp( T.Trip, I.Instant ) ) )
6 SELECT DISTINCT T.RegionId, T.InstantId, T.Instant, C.Licence
7 FROM Passes T JOIN Cars C ON T.CarId = C.CarId
8 ORDER BY T.RegionId, T.InstantId, C.Licence
```

The main predicate is in Line 5. It restricts the trips to one time instant, and checks whether they have been inside one of the query regions at this time instant. The bbox overlap in Line 4 is to optimize this predicate by invoking the index.

Query 15. Which vehicles passed a point from Points1 during a period from Periods1?

```
1 WITH Passes AS(
2     SELECT DISTINCT PO.PointId, PO.geom, PR.PeriodId, PR.Period, T.CarId
3     FROM Trips T, Points1 PO, Periods1 PR
4     WHERE T.Trip && gbox(PO.geom, PR.Period) AND
5         intersects(atPeriod( T.Trip, PR.Period ) , PO.geom) )
6 SELECT DISTINCT T.PointId, T.geom, T.PeriodId, T.Period, C.Licence
7 FROM Passes T, Cars C
8 WHERE T.CarId = C.CarId
9 ORDER BY T.PointId, T.PeriodId, C.Licence
```

This query aims at restricting the trips using temporal periods and spatial points. The atPeriod operator, in Line 5, does the restriction to period. The resulting subtrajectory is then checked for interesection with the spatial point, using the intersects operator. Recall form Section 4.5 that this version of the operator has the semantics of *ever intersects*. Thus all trip that ever intersects a point from Points1 during a period from Periods1 will be further forwarded to the main query, which fetches their licence numbers and prepares the final result.

Query 16. List the pairs of licences for vehicles, the first from Licences1, the second from Licences2, where the corresponding vehicles are both present within a region from Regions1 during a period from Period1, but do not meet each other there and then.

```
1 SELECT P.PeriodId, P.Period, R.RegionId, L1.Licence AS Licence1, L2.Licence AS Licence2
2 FROM Trips T1, Licences1 L1, Trips T2, Licences2 L2, Periods1 P, Regions1 R
3 WHERE T1.CarId = L1.CarId AND T2.CarId = L2.CarId AND L1.Licence < L2.Licence AND
4     getTime(T2.Trip) && P.Period AND
5     T1.Trip && gbox(R.geom, P.Period) AND T2.Trip && gbox(R.geom, P.Period) AND
6     st_intersects(trajectory(atPeriod(T1.Trip, P.Period)), R.geom) AND
7     st_intersects(trajectory(atPeriod(T2.Trip, P.Period)), R.geom) AND
8     NOT intersects(T1.Trip, T2.Trip)
9 ORDER BY PeriodId, RegionId, Licence1, Licence2
```

This query joins many tables. Therefore, there are many optimization opportunities using index. Apparently the most selective among them is the btree index on CarId that restricts the Trips

table to only those in `Licence1` and `Licence2`. Nevertheless, we sill add the bbox predicates in Line 5, because they act as a quick filter for the more expensive predicates in Lines 7–8.

Query 17. Which points from Points have been visited by a maximum number of different vehicles?

```
1 WITH PointCount AS (
2     SELECT P.PointId, COUNT(DISTINCT T.CarId) AS Hits
3     FROM Trips T, Points P
4     WHERE st_intersects( trajectory( T.Trip ), P.geom )
5     GROUP BY P.PointId )
6 SELECT PointId, Hits
7 FROM PointCount AS P
8 WHERE P.Hits = ( SELECT MAX(Hits) FROM Point Count )
```

In this query we found that the use of the spatial index on `Points` is more efficient than the use of the spatiotemporal index on `Trips`. Therefore, in the CTE we use the PostGIS predicate `st_intersects`, which internally issues a bbox overlaps using the spatial index. The CTE counts per point, the trips that traversed it. The main query then finds the maximum.

The performance of these 17 queries has been benchmarked versus SECONDO on different scale factors. We used a machine with the following configuration:

- CPU: 2x Intel Xeon E5-2640 v4
- RAM: 128GB
- HDD: 500GB NVMe SSD
- OS: Debian 9 “stretch" (kernel 4.9.0-8-amd64)

All queries were executed five times and the average execution time was computed. We used MobilityDB V1.0 Beta, and SECONDO 4.1.3 - Patch 1. For SECONDO, the TBA-CR (trip-based approach/compact representation) were used, which means that one trip is stored as a single temporal point, which is equivalent to what we do in MobilityDB. Four major distinctions that need to be considered while interpreting the benchmark results are: (1) The SECONDO queries are written in the so-called SECONDO executable language, which is a procedural language. Therefore, the queries are optimized manually, i.e., the execution plan is defined in the query. We balance this in MobilityDB by tweaking the queries to guide the SQL planner, e.g., by using CTEs. (2) The spatiotemporal index in SECONDO stores the bounding boxes of the segments that compose the trajectory. This is in contrast to GiST and SP-GiST indexes in MobilityDB that stores the bounding box for the whole trajectory. The dead space in the later is bigger, meaning that the index filter is less restrictive. These two distinctions are in favor of SECONDO. (3) On the other hand, MobilityDB precomputes and stores the spatial projection of the moving point, i.e., a linestring, which is used to speed up topological operations. (4) PostgreSQL, hence also MobilityDB, has advanced parallel query processing feature; including parallel scan, parallel index scan, parallel aggregates, and parallel joins. According to the query load and internal thresholds, it decides the number of cores to use, and distributes the query processing. We will have a closer look at the effect of parallelism later in this section.

First, we wish to have a global assessment of the performance of MobilityDB. Figures 7-9 show the query runtime of all queries on multiple the scale factors: SF 1.0 (300K trips), SF 3.0 (870K trips), SF 5.0 (1.5M trips), and SF 7.0 (2M trips). We split the queries over multiple figures for better visualization. In all of them, the x-axis shows the BerlinMOD/R queries, and the y-axis shows their runtimes in seconds. Queries 1, 2, 3, and 8 run in less than one second on both systems, and in all scale factors, so we exclude them form the figures. Query 5 crashes in SECONDO, so we exclude it too. In MobilityDB, it responds in 100 seconds on scale factor SF 7.0. Figure 7 shows the fast

responding queries, Figure 8 shows the more time consuming queries in SF 1.0 and SF 3.0, and Figure 9 shows them in SF 5.0 and SF 7.0. Query 9 takes very long time in SECONDO, indicating a possible implementation problem. Excluding it as well as Query 5, the overall runtime of all queries on the four scale factors is 5.7K seconds in MobilityDB, and 7.3K seconds in SECONDO.

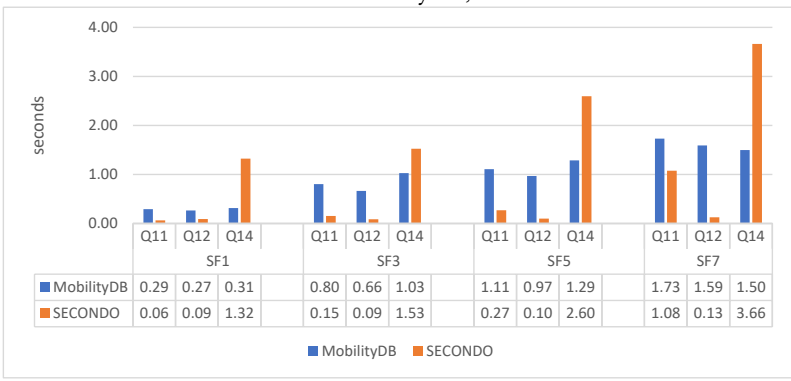


Fig. 7. Fast responding queries.

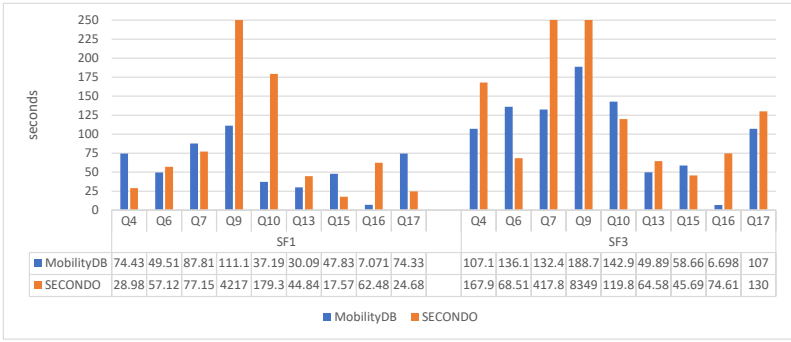


Fig. 8. Scale factors 1 (300K trips) and 3 (870K trips)

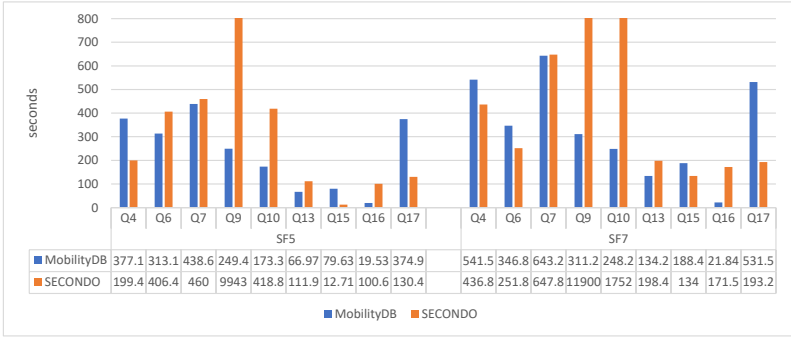
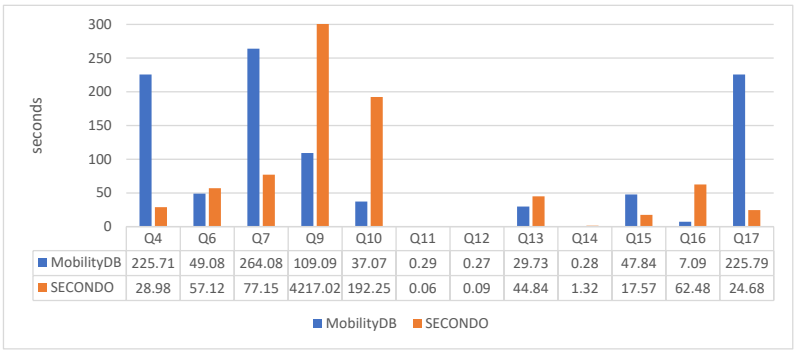


Fig. 9. Scale factors 5 (1.5M trips) and 7 (2M trips)

It is worth noting that the two systems implement similar data models, and algorithms. An algorithmic difference in the performance is not to be expected. The difference here is mainly due to different system overheads. One would expect that MobilityDB has a higher overhead, because it

builds on a bigger stack of database mechanisms such as concurrency and transaction management, statistics collection, etc. The overall view in the figures show that in contrary, the performance is rather good.



	Q4	Q6	Q7	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17
MobilityDB	225.71	49.08	264.08	109.09	37.07	0.29	0.27	29.73	0.28	47.84	7.09	225.79
SECONDO	28.98	57.12	77.15	4217.02	192.25	0.06	0.09	44.84	1.32	17.57	62.48	24.68

Fig. 10. Benchmark results on scale factor 0.02 (62K trips)

The previous experiment was done using the default parallel query processing settings of PostgreSQL. Next, we disable the parallel query processing feature of MobilityDB, to assess its performance on a single core. This is done on scale factor 1.0. As shown in Figure 10, most of the queries were not affected, except for Queries 4, 7, and 17. By analyzing the query plans, we observed that, except for these three queries, all are disk bound. That is, most of the response time is spent in reading the trajectory data from disk. If one admits that this benchmark is representative of real-world queries, it would be recommended to use faster storage, e.g., SSD.

Queries 4, 7, and 17 spend more time in CPU, and thus they benefit from parallelism. Next, we look closer at Query 4, as an example of the three queries, and assess its scalability with varying number of cores (1-12) and various scale factors (1-5). Figure 11 illustrates it. The gain of using 2, then 4 cores is big, but then we gain less with 6-12 cores, because the query becomes disk bound.

What can be concluded is that the performance of MobilityDB is generally good. One should pay attention to the query bounds, as I/O can be a bottleneck. With SSD and random access disk drives, one would like to configure MobilityDB to use as many cores. On contrary, with spinning disks it is advisable to use fewer cores, as the swapping of processes over the disk handler can increase the response time.

6 CONCLUSIONS AND FUTURE WORK

This paper presented MobilityDB, a mainstream moving object database based on PostgreSQL and PostGIS. MobilityDB leverages the functionality of these tools and maximizes the reuse of their ecosystems. It uses an abstract data type approach for implementing time-varying data types using a novel discrete data model called the *sequence representation*. The new types come with a rich set of operators that are exposed to the user as SQL functions. MobilityDB builds on existing operations and uniformaly lifts them into temporal operations using a novel algorithm. It also builds on the indexing and optimization framework of the underlying systems, which results in an efficient and expressive moving object database. MobilityDB is aligned with the ongoing OGC standards on Moving Features.

While all our development is done on PostgreSQL, most of this work can be done on other extensible DBMSs. Of a special interest here is the disciplined approach to temporal lifting, that we have implemented in MobilityDB. The proposed lifting algorithm manages the temporal part, and delegates the processing of the base or spatial part to the underlying platform. The sequence

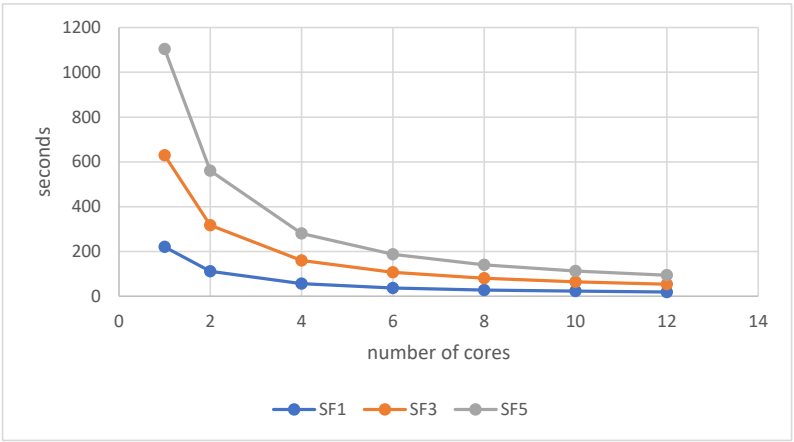


Fig. 11. Performance w.r.t. number of cores

representation and the ability of encoding continuity information inside the data model has shown advantages in the efficiency of queries, and in the simplicity of the algorithms. These ideas are platform agnostic.

The MobilityDB project has the goal to make the state of art in moving object databases accessible to users. Thus, the road map of future work includes numerous possibilities. We are currently working on a distributed version of MobilityDB. In many mobility applications the amount of data to be processed may be extremely large. Although the current version leverages PostgreSQL's features for parallel processing (which is based on scaling up) this is not enough for many applications and a cluster-based implementation (i.e., scaling out) is needed to comply with high-performance requirements. We base our implementation on Citus[7], an extension to scale-out PostgreSQL.

Visualization is an important aspect for real-world applications. PostGIS is well supported by the desktop visualization client QGIS, as well as by online mapping servers such as MapServer. It is natural in future work to integrate with these tools.

Finally, there are interesting operations in the literature that we wish to include such as kNN, spatiotemporal pattern matching, and similarity functions. We also plan to develop a model for network-constrained temporal points. The current version of MobilityDB assumes that objects move freely on space. However, this is not always the case, e.g., when objects move within spatially embedded networks such as transportation networks.

Acknowledgements. The work in MobilityDB would not have been possible without the support of many colleagues and students. We would like to thank Mohamed Bakli for the invaluable help in programming as well as in performing the benchmark, and for the work in his PhD Thesis towards developing a distributed database version of MobilityDB. We would like to thank Xinyang Li, who in the course of his Master's Thesis developed the initial implementations of the GiST and SP-GiST indexes for temporal types as well as the initial implementation of the network-constrained temporal points. We are grateful to Maxime Schoemans who in the course of his Master's Thesis developed the initial implementations of the rigid moving regions. We would also like to thank Darafei Praliaskouski for our fruitful discussions concerning the definition of the API for MobilityDB.

[7]https://www.citusdata.com/

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] Gennady Andrienko, Natalia Andrienko, Peter Bak, Daniel Keim, and Stefan Wrobel. 2013. *Visual Analytics of Movement*. Springer.

[4] Gennady Andrienko, Natalia Andrienko, and Stefan Wrobel. 2007. Visual Analytics Tools for Analysis of Movement Data. *SIGKDD Exploration Newsletter* 9, 2 (2007), 38–46.

[5] 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.

[6] Rimantas Benetis, Christian S. Jensen, Gytis Karciauskas, and Simonas Saltenis. 2002. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *Proceedings of the 2002 International Symposium on Database Engineering & Applications, IDEAS'02*. IEEE Computer Society, Washington, DC, USA, 44–53.

[7] Michael Böhlen, Johann Gamper, and Christian S. Jensen. 2006. Multi-dimensional Aggregation for Temporal Data. In *Advances in Database Technology - EDBT 2006*. Springer Berlin Heidelberg, Berlin, Heidelberg, 257–275.

[8] Eliseo Clementini and Paolino Di Felice. 1996. A model for representing topological relationships between complex geometric features in spatial databases. *Information Sciences* 90, 1 (1996), 121–136.

[9] Eliseo Clementini, Jayant Sharma, and Max J. Egenhofer. 1994. Modelling topological spatial relations: Strategies for query processing. *Computers & Graphics* 18, 6 (1994), 815–822.

[10] OGC Open Geospatial Consortium. 2010. Simple Feature Access - Part 1: Common Architecture. https://www.opengeospatial.org/standards/sfa.

[11] OGC Open Geospatial Consortium. 2013. *OGC Moving Features*. https://www.opengeospatial.org/standards/movingfeatures

[12] OGC Open Geospatial Consortium. 2014. OGC Moving Features Encoding Extension: Simple Comma Separated Values (CSV). http://docs.opengeospatial.org/is/14-084r2/14-084r2.html

[13] OGC Open Geospatial Consortium. 2016. OGC Moving Features Access. http://docs.opengeospatial.org/is/16-120r3/16-120r3.html

[14] OGC Open Geospatial Consortium. 2018. OGC Moving Features Encoding Part I: XML Core. http://docs.opengeospatial.org/is/18-075/18-075.html

[15] OGC Open Geospatial Consortium. 2019. OGC Moving Features Encoding Extension - JSON. http://docs.opengeospatial.org/is/19-045r3/19-045r3.html

[16] Xin Ding, Lu Chen, Yunjun Gao, Christian S. Jensen, and Hujun Bao. 2018. UlTraMan: A Unified Platform for Big Trajectory Data Management and Analytics. *Proc. VLDB Endow.* (2018), 787–799.

[17] Zhiming Ding and Ke Deng. 2011. Collecting and Managing Network-Matched Trajectories of Moving Objects in Databases. In *Proceedings of the 22nd International Conference on Database and Expert Systems Applications, DEXA 2011*. Springer, Toulouse, France, 270–279.

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

[19] Max J. Egenhofer, Eliseo Clementini, and Paolino Di Felice. 1994. Topological Relations Between Regions with Holes. *International Journal of Geographical Information Systems* 8, 2 (1994), 129–142.

[20] M. Y. Eltabakh, R. Eltarras, and Walid G. Aref. 2006. Space-Partitioning Trees in PostgreSQL: Realization and Performance. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE'06*. IEEE.

[21] Martin Erwig and Markus Schneider. 2002. Spatio-Temporal Predicates. *IEEE Transactions on Knowledge and Data Engineering* 14, 4 (2002), 881–901.

[22] Luca Forlizzi, Ralf Hartmut Güting, Enrico Nardelli, and Markus Schneider. 2000. A data model and data structures for moving objects databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD'00*. ACM, Dallas, TX, USA, 319–330.

[23] Elias Frentzos, Kostas Gratsias, Nikos Pelekis, and Yannis Theodoridis. 2007. Algorithms for Nearest Neighbor Search on Moving Object Trajectories. *Geoinformatica* 11, 2 (2007), 159–193.

[24] Sören Gebbert and Edzer Pebesma. 2017. The GRASS GIS temporal framework. *International Journal of Geographical Information Science* 31, 7 (2017), 1273–1292.

[25] Stéphane Grumbach, Philippe Rigaux, Michel Scholl, and Luc Segoufin. 1998. DEDALE, A Spatial Constraint Database. In *Proceedings of the 6th International Workshop on Database Programming Languages, DBLP-6*. Springer, London, UK, 38–59.

[26] Stéphane Grumbach, Philippe Rigaux, and Luc Segoufin. 2001. Spatio-Temporal Data Handling with Constraints. *GeoInformatica* 5, 1 (2001), 95–115.

[27] 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 *Proceedings of the 21st International Conference on Data Engineering, ICDE'05*. IEEE Computer Society, Washington, DC, USA, 1115–1116.

[28] Ralf Hartmut Güting, Thomas Behr, and Jianqiu Xu. 2010. Efficient k-nearest neighbor search on moving object trajectories. *VLDB Journal* 19, 5 (2010), 687–714.

[29] 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 Transactions on Database Systems* 25, 1 (2000), 1–42.

[30] Ralf Hartmut Güting, Teixeira de Almeida, and Zhiming Ding. 2006. Modeling and Querying Moving Objects in Networks. *The VLDB Journal* 15, 2 (2006), 165–190.

[31] Stefan Hagedorn, Philipp Götze, and Kai-Uwe Sattler. 2017. The STARK Framework for Spatio-Temporal Data Analytics on Spark. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*. Gesellschaft für Informatik, Bonn, 123–142.

[32] Florian Heinz and Ralf Hartmut Güting. 2018. A data model for moving regions of in databases. *International Journal of Geographical Information Science* 32, 9 (2018), 1737–1769.

[33] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. 1995. Generalized Search Trees for Database Systems. In *Proceedings of the 21th International Conference on Very Large Data Bases, VLDB '95*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 562–573.

[34] ISO. 2008. ISO 19141:2008 Geographic information âĂŤ Schema for moving features. https://www.iso.org/standard/41445.html

[35] Nick Kline and Richard T. Snodgrass. 1995. Computing temporal aggregates. *Proceedings of the Eleventh International Conference on Data Engineering* (1995), 222–231.

[36] 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.

[37] Ahmed R. Mahmood, Sri Punni, and Walid G. Aref. 2019. Spatio-temporal access methods: a survey (2010 - 2017). *GeoInformatica* 23, 1 (2019), 1–36.

[38] Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. 2003. Spatio-temporal Access Methods. *IEEE Data Engineering Bulletin* 26 (2003), 40–49.

[39] Kyriakos Mouratidis, Dimitris Papadias, and Marios Hadjieleftheriou. 2005. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data, SIGMOD'05*. ACM, Baltimore, Maryland, USA, 634–645.

[40] Richard G. Newell, David Theriault, and Mark Easterfield. 1992. Temporal GIS: Modeling the Evolution of Spatial Data in Time. *Computational Geoscience* 18, 4 (1992), 427–433.

[41] Long-Van Nguyen-Dinh, Walid G. Aref, and Mohamed F. Mokbel. 2010. Spatio-Temporal Access Methods: Part 2 (2003 - 2010). *IEEE Data Eng. Bull.* 33, 2 (2010), 46–55.

[42] 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.

[43] Christine Parent, Stefano Spaccapietra, Chiara Renso, Gennady Andrienko, Natalia Andrienko, Vania Bogorny, Maria Luisa Damiani, Aris Gkoulalas-Divanis, Jose Macedo, Nikos Pelekis, Yannis Theodoridis, and Zhixian Yan. 2013. Semantic Trajectories Modeling and Analysis. *ACM Computer Surveys* 45, 4 (2013), 42:1–42:32.

[44] Christine Parent, Stefano Spaccapietra, and Esteban Zimányi. 2006. *Conceptual Modeling for Traditional and Spatio-Temporal Applications: The MADS Approach*. Springer.

[45] 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.

[46] Nikos Pelekis, Babis Theodoulidis, Ioannis Kopanakis, and Yannis Theodoridis. 2004. Literature Review of Spatio-temporal Database Models. *Knowledge Engineering Review* 19, 3 (2004), 235–274.

[47] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, in-Memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (2015), 1816–1827.

[48] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. 2000. Novel Approaches in Query Processing for Moving Object Trajectories. In *Proceedings of the 26th International Conference on Very Large Data Bases, VLDB'00*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 395–406.

[49] Mahmoud Sakr and Ralf Hartmut Güting. 2011. Spatiotemporal pattern queries. *GeoInformatica* 15, 3 (2011), 497–540.

[50] Mahmoud Sakr and Ralf Hartmut Güting. 2014. Group spatiotemporal pattern queries. *GeoInformatica* 4 (2014), 699–746.

[51] Zhexuan Song and Nick Roussopoulos. 2001. K-nearest neighbor search for moving query point. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases, SSTD'01*. Springer, London, UK, 79–96.

[52] Yufei Tao and Dimitris Papadias. 2005. Historical Spatio-temporal Aggregation. *ACM Trans. Inf. Syst.* 23, 1 (2005), 61–102.

[53] Alejandro A. Vaisman and Esteban Zimányi. 2019. Mobility Data Warehouses. *ISPRS Int. J. Geo-Information* 8, 4 (2019), 170.

[54] 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*. Association for Computing Machinery, New York, NY, USA, 1099–1104.

[55] Jianqiu Xu and Ralf Hartmut Güting. 2013. A Generic Data Model for Moving Objects. *Geoinformatica* 17, 1 (2013), 125–172.

[56] Jun Yang and Jennifer Widom. 2003. Incremental computation and maintenance of temporal aggregates. *The VLDB Journal* 12, 3 (2003), 262–283.