Evaluating Spatio-temporal Data Models for Trajectories in PostGIS Databases

GI_Forum 2018, Issue.1
Page: 16 - 33
Full Paper
Corresponding Author:
anitagraser@gmx.at
DOI: 10.1553/giscience2018_01_s16

Anita Graser

AIT Austrian Institute of Technology, Vienna, Austria

Abstract

Research in transport, ecology, health and other fields stands to profit from an improved understanding of movement. As movement data availability improves, the need for appropriate movement data analysis increases. However, the limited support for modelling moving objects in GIS hampers data exploration and analysis. This paper discusses trajectory data models and their implementation in the open-source spatial database system PostGIS. We quantify the difference in performance between PostGIS default trajectory support, dedicated trajectory data models, and commonly used point-based data models. To the best of our knowledge, this is the first paper to evaluate PostGIS default trajectory support and compare it to a proposed dedicated trajectory data type from the literature. Our experiments include computing trajectory duration and length, temporal and spatial filters, extracting positions at a certain time, and visualizing trajectories in desktop GIS. We also discuss the limitations of, and potential for, contextual trajectories and moving area object trajectories. Our results show that PostGIS functions for moving point object trajectories are fast, reduce query complexity, and provide good indexing integration, especially concerning multi-dimensional indices; the results also reveal that trajectory data models outperform commonly used point-based data models.

Keywords:

trajectories, moving objects, spatial databases, PostGIS, PostgreSQL

1 Introduction

Analysis of movement data is an increasingly popular topic in the GIScience literature, with applications ranging from research in movement ecology (for example, understanding migration patterns, monitoring species distribution) (Wang et al., 2016), to transport (for example, detecting travel behaviour, monitoring traffic quality) (Siła-Nowicka et al., 2016), and health (for example, monitoring physical activity) (Doherty et al., 2014). Movement can be described using spatial, temporal and spatio-temporal parameters. These parameters are the building blocks of different movement patterns. Dodge et al. (2008) list parameters for individual moving objects. Beyond single moving object trajectories, a large number of measures (including similarity, density and formation stability) have already been developed

for a single trajectory amidst other trajectories, as well as between groups of trajectories (Wiratma et al., 2017).

Thanks to improved data collection technology, many issues that previously suffered from insufficient data are now being addressed. Nonetheless, GIS functionality to analyse data from moving objects is still limited. The lack of support for handling time in current GIS hampers data exploration and analysis. The de facto standard GIS data model is 'Simple Features', as covered by ISO 19125 'Geographic information – Simple feature access' (ISO, 2004) and 'OpenGIS® Implementation Standard for Geographic information – Simple feature access' (OGC, 2017a). Even though other models do exist (for example, topological data models), many GIS data formats only support simple features. The OGC Simple Feature Access - Common Architecture deals only with at most 2-dimensional geometric objects, whereas the ISO Spatial schema handles up to 3-dimensional geometric objects (OGC, 2011). Temporal information is not, therefore, considered in the simple features specification. Consequently, trajectories are commonly stored as points or segments with time stamp attributes to preserve temporal information. For example, the CSV (comma separated value) data model used by the Movebank Data Repository (Wikelski & Kays, 2017) stores points with time stamps, while the OGC® Moving Features standard (OGC, 2017b) models segments with start- and end-time stamps.

Beyond the field of GIS, the database community has developed the concept of moving objects databases (MODs). One of the first projects in this domain was 'Databases fOr MovINg Objects tracking' (DOMINO) (Wolfson et al., 1997). MODs provide specific functionality for handling moving objects and can be seen as extensions of temporal or spatial databases (Güting and Schneider, 2005, pp. 26–27). MODs can be optimized to deal with either historic or live movement data. Much work has focused on indexing, in particular the issue of how to index a large number of trajectories including the time dimension, in order to perform efficient querying and updating. For example, Frentzos (2008) discusses indexing of unconstrained and network-constrained trajectory data in MODs. As in GIS, common data modelling approaches are either primal (where each trajectory segment is represented as a line) or dual (where the trajectory segment is represented as a point) (Wolfson, 2002). To the best of our knowledge, there exist only research prototypes of moving objects databases, including SECONDO and HermesMOD. HermesMOD (n.d.) is an extension to PostgreSQL and Oracle by researchers at the University of Piraeus. SECONDO (2009) is a database management system (DBMS) 'suitable for research prototyping and teaching'. Since moving objects databases are still in the prototype phase, a practical solution for the problem of handling moving objects in GIS environments is to lean on a mature spatial DBMS such as PostGIS.

The remainder of this paper is structured as follows. Section 2 introduces approaches for modelling moving point objects in PostGIS. Section 3 presents experiments comparing the custom PG-Trajectory data model (Kucuk et al., 2016) to default PostGIS trajectory support and commonly used point-based data models. Section 4 discusses advantages and shortcomings of these approaches and concludes with an outlook on moving area objects, as well as on massive movement data.

2 Moving point objects in PostGIS

Güting and Schneider (2005, pp. 28–29) have already confirmed that spatial databases can handle moving objects, but cautioned that, in classical spatial databases, time is not managed intrinsically. An example of a PostGIS-based data model using a custom trajectory object is presented by Kucuk et al. (2016). Their model handles moving point and polygon features as an open source extension (DMLAB, 2016) for PostGIS. Kucuk et al. (2016) argue that '[w]hen the trajectories are represented [...] using three columns (traj id, timestamp, geometry), users are required to write complex queries and applications for implementing trajectory manipulating functions' (p. 83). They therefore propose a custom data type that 'makes these queries easier by wrapping up widely used trajectory functions' (ibid.) (see Table 1). Kucuk et al. (2016) indeed handle time explicitly. However, we find that this is not necessary – at least for moving point objects – in current versions of PostGIS, thanks to its built-in temporal support (version 2.2.0, released in October 2015, onwards (PostGIS, 2015)), as we will demonstrate in this paper.

PostGIS supports a superset of the simple features defined by the OGC. It supports 3D geometries as well as additional measure values. Using PostGIS temporal support enables a data model for moving objects that stores trajectories as LinestringM or LinestringZM, where Z stands for elevation and M stands for measure or m-value. In this model, the timestamp for each position is stored in the m-values of the vertices that make up the lines. In a valid trajectory, m-values have to increase from one vertex to the next. These m-values can be accessed both within the PostGIS database, and in desktop GISs such as QGIS. For example, we can use ST_M(ST_EndPoint(trajectory)) in PostGIS and m(end_point($geometry)) in QGIS to access the end-time of a trajectory.

To the best of our knowledge, this is the first paper to evaluate PostGIS default trajectory support. Our goal is to quantify the differences in performance between PostGIS default trajectory support, dedicated trajectory data models, and commonly used point-based data models. Specifically, we compare the trajectory data type PG-Trajectory by Kucuk et al. (2016) to default PostGIS without custom data types or functions. The comparison methodology is based on query run times as well as query complexity, from the perspectives of both the user writing the query (number of subqueries that have to be written) and the database system (steps in the execution plan). A comparison to other database systems or computing environments, such as R, is beyond the scope of this work. Table 1 provides an initial overview of relevant functions in both PG-Trajectory and default PostGIS. While not all aspects of functionality are available in both approaches, most basic trajectory functions overlap. This overlapping functionality is compared in Section 3.

Table 1: Comparing PG-Trajectory (Kucuk et al., 2016) and default PostGIS for moving point objects

	PG-Trajectory	**Default PostGIS**
Creating and editing		
Trajectory elements	tg_pair (timestamp-geometry pair)	PointM
Constructor	_trajectory(tg_pair[])	geometry(LineStringM,4326)
Add position to beginning / end	t_add_head(tg_pair,traj) / t_add_tail(tg_pair,traj) : trajectory	ST_AddPoint(traj,ptM,0) / ST_AddPoint(traj,ptM,-1) : geometry
Remove position from beginning / end	t_drop_head(traj) / t_drop_tail(traj) : trajectory	ST_RemovePoint(traj,0) / ST_RemovePoint(traj, ST_NPoints(traj) - 1) : geometry
Update location at certain time	t_update_geom_at(t,geom,traj) : trajectory	-
Accessing trajectory information		
Start time	tg_start_time(tg_pair[]): timestamp	ST_M(ST_StartPoint(traj)) : float
End time	tg_end_time(tg_pair[]) : timestamp	ST_M(ST_EndPoint(traj)) : float
Duration	t_duration(traj) : time interval	ST_M(ST_EndPoint(traj)) - ST_M(ST_StartPoint(traj)) : float
Length	t_distance(traj) : real	ST_Length(traj) : float
Position at certain time	t_geom_at(traj,t) : geometry * (code repo contains t_record_at_interpolated(traj, t): geometry)	ST_LocateAlong(traj,t) : geometry
Extract subtrajectory	-	ST_LocateBetween(traj,t1,t2) : geometry
Pair-wise trajectory analysis		
Spatio-temporal intersection	t_overlaps(traj1,traj2) : boolean * (code repo contains t_intersection(traj1,traj2) : traj)	ST_CPAWithin(traj1,traj2,maxdist) : bool
Closest point of approach (CPA)	-	ST_ClosestPointOfApproach(traj1,traj2) : float
Time-relaxed Euclidean distance	t_euclidean_distance(traj1,traj2) : real	-
Edit distance	t_edit_distance(traj1,traj2) : real	-
Euclidean distances	t_m_distance(traj1,traj2) : real[]	-
Hausdorff distance	-	ST_HausdorffDistance(traj1,traj2) : float

3 Experiments

Experiments were performed using the Geolife dataset published by Zheng, Li et al. (2008), Zhen, Zhang et al. (2009), and Zhen, Xie et al. (2010), which was also used in the point trajectory examples by Kucuk et al. (2016). The Geolife dataset contains 18,670 trajectories collected using 182 trackers, mostly in and around Beijing, with some trajectories including international travel. The run times of our experiments were measured on a Dell R730 database server with 8 cores (Intel Xeon E5-2637 v4) and 64 GB memory. The following experiments were carried out:

1. Computing total trajectory duration and length per tracker
2. Finding trajectories that occurred during a certain time frame (temporal filter)
3. Finding trajectories that originated in a certain spatial region (spatial filter)
4. Extracting positions at a certain point in time
5. Visualizing trajectories in desktop GIS

We found that the PG-Trajectory source code published by Kucuk et al. at https://bitbucket.org/gsudmlab/pg-trajectory had to be fixed at several locations before it was possible to import Geolife trajectories and use trajectory functions. The updated code is available at https://bitbucket.org/anitagraser/pg-trajectory.

In addition to the PG-Trajectory and default PostGIS trajectory data models, we also compared results with the commonly used basic point-based data model. Trajectory table definitions for all three approaches as well as an example of trajectory data insertion are provided in Table 2. The custom trajectory data type by Kucuk et al. (2016) stores an array of timestamp-geometry pairs, as well as trajectory start- and end-time stamps. Therefore, to enable a fair comparison, a table with a LineStringM and additional time-range column was chosen to represent PostGIS's temporal support.

Table 2: Trajectory table creation, data insertion, and index creation for the three data model approaches compared in this paper

PG-Trajectory	Default PostGIS	Point-based
CREATE TABLE pgtrajectory.geolife(id serial NOT NULL, tracker integer, traj trajectory);	CREATE TABLE geolife.trajectory_ext(id serial NOT NULL, tracker integer, track geometry(LineStringM), time_range tstzrange);	CREATE TABLE geolife.trajectory_pt(id serial NOT NULL, sequence bigint, trajectory_id bigint, tracker integer, pt geometry(Point), t timestamp with time zone);
INSERT INTO pgtrajectory.geolife (tracker,traj) VALUES (010, _trajectory(ARRAY[ROW('2007-11-17 17:08:27+00',POINT(121.5	INSERT INTO geolife.trajectory_ext (tracker,track,time_range) VALUES (010, ST_SetSRID(ST_GeometryFromText('LINESTRINGM (121.574898	INSERT INTO geolife.trajectory_pt (sequence,trajectory_id,track er,pt,t) VALUES (1, 1, 010, ST_SetSRID(ST_GeometryFromTex t(

74898,31.195130))::tg_pair, ROW('2007-11-17 17:08:32+00',POINT(121.573458,31.197597))::tg_pair, ...]));	31.195130 1195319307, 121.573458 31.197597 1195319312, ...) '),4326), '[2007-11-17 18:08:27+01,2007-11-17 18:10:01+01]'::tstzrange);	'POINT(121.574898 31.195130)')), '2007-11-17 17:08:27+00');
CREATE INDEX idx_pgtrajectory_tracker ON trajectory_ext USING btree (tracker);	CREATE INDEX idx_trajectory_ext_time_range ON trajectory_ext USING gist (time_range); CREATE INDEX sidx_trajectory_ext_track3d ON trajectory_ext USING gist (track gist_geometry_ops_nd); CREATE INDEX idx_trajectory_ext_tracker ON trajectory_ext USING btree (tracker);	CREATE INDEX sidx_trajectory_pt_pt ON trajectory_pt USING gist (pt); CREATE INDEX sidx_trajectory_pt_trajectory_id ON trajectory_pt USING btree (trajectory_id); CREATE INDEX sidx_trajectory_pt_tracker ON trajectory_pt USING btree (tracker); CREATE INDEX sidx_trajectory_pt_t ON trajectory_pt USING btree (t);

Duration and length

The first experiment was to determine trajectory duration and length for all observed moving objects (identified by a tracker id in the case of the Geolife dataset). Queries and their run times are provided in Tables 3 and 4. Both duration and distance queries are fastest using default PostGIS trajectory types (default PostGIS variant a in Table 3). Only if temporal information has to be extracted from the LineStringM (variant b in Table 3) is the default trajectory approach slower than the PG-Trajectory. The basic point-based data model is significantly slower (up to 194 times) than either of the trajectory-based data models. This is because queries for the point-based trajectory data model result in a more complex execution plan, including a second time-consuming HashAggregate step.

Queries for the point-based data model are more complex than queries for the other models, since each requires an additional subquery. More specifically, to keep point-based queries readable, we use WITH queries, also known as Common Table Expressions (CTE). The WITH query in Table 3 prepares a temporary table, which groups points belonging to the same trajectory and extracts start and end times. It is worth noting that WITH queries can negatively impact query performance, but potential improvements gained by designing more complex nested queries are not investigated further in this paper.

Results show that the total duration of trajectories of tracker1 according to PG-Trajectory differs by one hour from the durations obtained using the other two approaches. We have not so far been able to determine the reason for this.

Table 3: Total trajectory duration for all trackers

PG-Trajectory	Default PostGIS	Point-based

SELECT tracker,
 sum(t_duration(traj))
FROM pgtrajectory.geolife
GROUP BY tracker
ORDER BY tracker
	SELECT tracker,
	 sum(upper(time_range) -
	 lower(time_range))
	FROM geolife.trajectory_ext
	GROUP BY tracker
	ORDER BY tracker
	
	SELECT tracker,
	 sum(to_timestamp(st_m(
	  st_endpoint(track)))
	  - to_timestamp(st_m(
	  st_startpoint(track))))
	FROM geolife.trajectory_ext
	GROUP BY tracker
	ORDER BY tracker
		WITH tmp AS (
		 SELECT trajectory_id,
		  tracker,
		  min(t) start_time,
		  max(t) end_time
		 FROM geolife.trajectory_pt
		 GROUP BY trajectory_id,
		         tracker
		)
		SELECT tracker,
		 sum(end_time - start_time)
		FROM tmp
		GROUP BY tracker
		ORDER BY tracker

Execution plan		

Sort (cost=5487.20..5487.66)
-> HashAggregate
   (cost=5478.55..5480.37)
   -> Seq Scan on geolife
      (cost=0.00..717.70)
	Sort (cost=5978.73..5979.18)
	-> HashAggregate
	   (cost=5970.07..5971.89)
	   -> Seq Scan on trajectory_ext
	      (cost=0.00..5736.70)
		Sort (cost=880610.37..880610.87)
		CTE tmp
		-> HashAggregate
		   (cost=829233.68..842931.56)
		   -> Seq Scan on trajectory_pt
		      (cost=0.00..580463.84)
		-> HashAggregate
		   (cost=37669.17..37671.17)
		   -> CTE Scan on tmp
		      (cost=0.00..27395.76)

Results (top 3 rows)		
tracker;sum 0;"2 days 879:43:29" 1;"314:05:30" 2;"700:49:38"	tracker;sum 0;"2 days 879:43:29" 1;"315:05:30" 2;"700:49:38"	tracker;sum 0;"2 days 879:43:29" 1;"315:05:30" 2;"700:49:38"
Run time		
1.8 sec	a) 31 ms, b) 2.1 sec	6.0 sec

Table 4 shows total trajectory-length queries and results. The point-based query is the most complex, requiring two CTEs. Even so, the point-based query run time is shorter than for PG-Trajectory due to the more efficient implementation of the length function. In all three cases, lengths are computed after casting from geometry to geography data type. The remaining minor differences in the resulting lengths are due to rounding.

Table 4: Total trajectory length for each tracker

PG-Trajectory	Default PostGIS	Point-based

SELECT tracker,
 round(sum(
  t_distance(traj)
 ))
FROM pgtrajectory.geolife
GROUP BY tracker
ORDER BY tracker
	SELECT tracker,
	 round(sum(
	 ST_Length(track::geography)
	))
	FROM geolife.trajectory_ext
	GROUP BY tracker
	ORDER BY tracker
		WITH ordered AS (
		 SELECT trajectory_id,
		  tracker, t, pt
		 FROM geolife.trajectory_pt
		 ORDER BY t
		), tmp AS (
		 SELECT trajectory_id,
		  tracker,
		  st_makeline(pt) traj
		 FROM ordered
		 GROUP BY trajectory_id,
		         tracker
		)
		SELECT tracker,
		 round(sum(
		  ST_Length(traj::geography)
		))
		FROM tmp
		GROUP BY tracker
		ORDER BY tracker

Execution plan		

Sort (cost=5487.66..5488.11)
-> HashAggregate
   (cost=5478.55..5480.82)
   -> Seq Scan on geolife
      (cost=0.00..717.70)
	Sort
	(cost=10553.33..10553.79)
	-> HashAggregate
	   (cost=10544.22..10546.50)
	   -> Seq Scan on
	trajectory_ext
	      (cost=0.00..5736.70)
		Sort
		(cost=2604096.7..2604097.2)
		CTE ordered
		-> Index Scan using
		   sidx_trajectory_pt_t
		   (cost=0.44..1908369.55)
		CTE tmp
		-> HashAggregate
		
		(cost=684117.06..684617.06)
		   -> CTE Scan on ordered
		      (cost=0.00..497539.68)
		-> HashAggregate
		   (cost=11100.00..11102.50)
		   -> CTE Scan on tmp
		      (cost=0.00..800.00)

Results (top 3 rows)		
tracker;round 0;6520101 1;986594 2;3339309	tracker;round 0;6520101 1;986594 2;3339309	tracker;round 0;6520101 1;986594 2;3339309
Run time		
01:32 min	22.7 sec	43.0 sec

Temporal filters

In the second experiment, we extracted trajectories that occurred during a certain time frame, as shown in Table 5. The default PostGIS query illustrates how time ranges can be used in combination with the 'overlaps' operator && (default PostGIS variant a), thus providing a more succinct way to write temporal queries. The point-based query is the most complex and slowest, requiring two CTEs. Default PostGIS queries using time ranges (variant a in Table 5) or the n-dimensional index (variant b) are considerably faster than the PG-Trajectory approach, since their execution plans avoid sequential scans by leveraging appropriate indices. Even if temporal information had to be extracted from the LineStringM (variant c), the default approach would still be faster than PG-Trajectory.

Results show that all three approaches identify the same trajectories that overlap the specified time range. Trajectory ids vary between result sets, since these ids are not defined in the imported Geolife dataset but rather generated using auto-incrementing values on data import.

Table 5: Filtering trajectories by time

PG-Trajectory	Default PostGIS	Point-based

SELECT id, tracker,
 (traj).s_time,
 (traj).e_time
FROM pgtrajectory.geolife
WHERE (traj).e_time
      > '2008-11-26 11:00'
AND (traj).s_time
      < '2008-11-26 15:00'
	SELECT id, tracker,
	 time_range
	FROM geolife.trajectory_ext
	—
	WHERE
	time_range && '[2008-11-26 11:00+1,2008-11-26 15:00+01]'::tstzrange
	—
	WHERE track &&&
	ST_Collect(
	 ST_MakePointM(-180,-90,
	  extract(epoch from '2008-11-26 11:00'::timestamptz)),
	 ST_MakePointM(180,90,
	  extract(epoch from '2008-11-26 15:00'::timestamptz))
	)
		WITH tmp AS (
		 SELECT trajectory_id,
		  tracker,
		  min(t) start_time,
		  max(t) end_time
		 FROM geolife.trajectory_pt
		 GROUP BY trajectory_id,
		        tracker
		)
		SELECT trajectory_id,
		 tracker, start_time,
		 end_time
		FROM tmp
		WHERE end_time
		     > '2008-11-26 11:00'

	WHERE
	 to_timestamp(st_m(
	  st_endpoint(track)))
	  > '2008-11-26 11:00'
	AND to_timestamp(st_m(
	  st_startpoint(track)))
	  < '2008-11-26 15:00')
		AND start_time
		      < '2008-11-26 15:00'

	Execution plan	

Seq Scan on geolife
(cost=0.00..811.05)
 Filter
	Bitmap Heap Scan on
	trajectory_ext
	(cost=4.31..23.93)
	 -> Bitmap Index Scan on
	idx_trajectory_ext_time_range
	   (cost=0.00..4.31)
	
	Index Scan using
	sidx_trajectory_ext_track3d
	(cost=0.29..12.32)
	
	Seq Scan on trajectory_ext
	(cost=0.00..6110.10)
	 Filter
		CTE Scan on tmp
		(cost=842931.56..877176.26)
		 Filter
		 CTE tmp
		 -> HashAggregate
		(cost=829233.68..842931.56)
		   -> Seq Scan on
		trajectory_pt
		(cost=0.00..580463.84)

	Results (top 3 rows)	
id;tracker;s_time;e_time 1197;5;"2008-11-25 23:26:34";"20 2089;14;"2008-11-26 11:07:04";"2 7872;68;"2008-11-26 12:59:36";"2	id;tracker;time_range 1347;5;"["2008-11-25 23:26:34+01 2252;14;"["2008-11-26 11:07:04+0 10494;68;"["2008-11-26 12:59:36+	trajectory_id;tracker;start_time 1347;5;"2008-11-25 23:26:34+01"; 2252;14;"2008-11-26 11:07:04+01" 10494;68;"2008-11-26 12:59:36+01
	Run time	
2.5 sec	a) with index: 12 ms, without: 21 ms, b) 12 ms, c) 1.8 sec	6.0 sec

Spatial filters

In the third experiment, we extracted trajectories that originated in a certain spatial region, as shown in Table 6. It is straightforward to adjust these queries to find trajectories that either end in a specific region, or both start and end within defined regions. The point-based query is the most complex and slowest (45 times slower than default PostGIS), requiring three CTEs instead of just one. Again, PG-Trajectory requires a sequential scan, while the default PostGIS leverages the index. Results show that all three approaches return the same trajectories that start within the buffer designated as areaA.

Table 6: Filtering trajectories' start locations

PG-Trajectory	Default PostGIS	Point-based

WITH my AS (SELECT
 ST_Buffer(ST_SetSRID(
 ST_MakePoint(116.31894,
 39.97472),4326),0.0005)
 areaA
)
SELECT id, tracker,
 traj
FROM pgtrajectory.geolife
JOIN my
ON ST_Within(ST_SetSRID(
 (tg_head((traj).tr_data)).g
 ,4326),
 areaA)
	WITH my AS (SELECT
	 ST_Buffer(ST_SetSRID(
	 ST_MakePoint(116.31894,
	 39.97472),4326),0.0005)
	 areaA
	)
	SELECT id, tracker,
	 ST_AsText(track)
	FROM geolife.trajectory_ext
	JOIN my
	ON areaA && track
	AND ST_Within(
	 ST_StartPoint(track),
	 areaA)
		WITH my AS (SELECT
		 ST_Buffer(ST_SetSRID(
		ST_MakePoint(116.31894,39.97
		472),4326),0.0005) areaA
		), ordered AS (
		 SELECT trajectory_id,
		  tracker, t, pt
		 FROM geolife.trajectory_pt
		 ORDER BY t
		), tmp AS (
		 SELECT trajectory_id,
		  tracker,
		  st_makeline(pt) traj
		 FROM ordered
		 GROUP BY trajectory_id,
		          tracker
		)
		SELECT tracker,
		 ST_AsText(traj)
		FROM tmp
		JOIN my
		ON ST_Within(
		 ST_StartPoint(traj),
		 my.areaA)

Execution plan		

Nested Loop
(cost=0.01..10286.1)
 Join Filter
 CTE my
 -> Result (cost=0.00..0.01)
-> CTE Scan on my
   (cost=0.00..0.02)
-> Seq Scan on geolife
   (cost=0.00..717.70)
	Nested Loop
	(cost=0.29..12.87)
	 CTE my
	 -> Result (cost=0.00..0.01)
	-> CTE Scan on my
	   (cost=0.00..0.02)
	-> Index Scan using
	   sidx_trajectory_ext_track
	   (cost=0.28..12.83)
	   Filter
		Nested Loop
		(cost=2592986.62..2604486.67
		)
		 Join Filter
		 CTE my
		  -> Result
		(cost=0.00..0.01)
		 CTE ordered
		  -> Index Scan using
		    sidx_trajectory_pt_t
		    (cost=0.44..1908369.55)
		 CTE tmp
		  -> HashAggregate
		
		(cost=684117.06..684617.06)
		    -> CTE Scan on ordered
		
		(cost=0.00..497539.68)

		-> CTE Scan on my (cost=0.00..0.02) -> CTE Scan on tmp (cost=0.00..800.00)
Results (top 3 rows)		
id;tracker;traj 9803;92;"("2008-07-14 09:44:03", 10031;95;"("2011-05-01 17:29:38" 10253;100;"("2011-07-31 12:26:29	id;tracker;st_astext 9293;92;"LINESTRING M (116.31869 9365;95;"LINESTRING M (116.3185 9511;100;"LINESTRING M (116.3193	tracker;st_astext 92;"LINESTRING M (116.318692 39. 95;"LINESTRING M (116.3185 39.9 100;"LINESTRING M (116.31937 39.
Run time		
1.8 sec	488 ms	21.9 sec

Extracting positions at a certain point in time

In the fourth experiment, we extracted positions from trajectories to determine where moving objects were located at a certain point in time, as shown in Table 7. The point-based query is the most complex and slowest, and requires two CTEs. Trajectory data models and indexed time ranges are up to 2,000 times faster than point-based data models. Results are not consistent between PG-Trajectory and default PostGIS, because PG-Trajectory's t_record_at_interpolated() is hard-coded to return the average position between the positions immediately before and after the specified point in time.

Table 7: Extracting moving object positions at a certain point in time

PG-Trajectory	Default PostGIS	Point-based
SELECT id, tracker, ST_AsText(t_record_at_interpolated(traj, '2008-11-26 13:00')) FROM pgtrajectory.geolife WHERE (traj).e_time >= '2008-11-26 13:00' AND (traj).s_time <= '2008-11-26 13:00'	SELECT id, tracker, ST_AsText(ST_LocateAlong(track, extract(epoch from '2008-11-26 13:00'::timestamptz))) FROM geolife.trajectory_ext WHERE time_range @> '2008-11-26 13:00'::timestamptz	WITH ordered AS (SELECT trajectory_id, tracker, t, pt FROM geolife.trajectory_pt ORDER BY t), tmp AS (SELECT trajectory_id, tracker, min(t) start_time, max(t) end_time, st_makeline(pt) traj FROM ordered GROUP BY trajectory_id, tracker

<table>
<tr><td></td><td></td><td>)
SELECT tracker,
ST_AsText(ST_LocateAlong(
traj,
extract(epoch from '2008-11-26 13:00'::timestamptz)))
FROM tmp
WHERE end_time
>= '2008-11-26 13:00'
AND start_time
<= '2008-11-26 13:00'</td></tr>
<tr><td colspan="3">Execution plan</td></tr>
<tr><td>Seq Scan on geolife
(cost=0.00..1334.74)
Filter</td><td>Index Scan using
idx_trajectory_ext_time_range
(cost=0.28..8.30)</td><td>CTE Scan on tmp
(cost=2717371.53..2718404.86)
Filter
CTE ordered
-> Index Scan using
sidx_trajectory_pt_t
(cost=0.44..1908369.55)
CTE tmp
-> HashAggregate
(cost=808501.98..809001.98)
-> CTE Scan on ordered
(cost=0.00..497539.68)</td></tr>
<tr><td colspan="3">Results</td></tr>
<tr><td>id;tracker;st_astext
7872;68;"POINT(116.414505 39.984
9003;84;"POINT(116.388098 39.968
11494;126;"POINT(116.3715395 39.
15458;153;"POINT(116.435101 39.9
18270;167;"POINT(116.3715395 39.</td><td>id;tracker;st_astext
10494;68;"MULTIPOINT M (116.4116
8469;84;"MULTIPOINT M (116.44798
11286;126;"MULTIPOINT M (116.422
15332;153;"MULTIPOINT M (116.399
18448;167;"MULTIPOINT M (116.422</td><td>tracker;st_astext
68;"MULTIPOINT M (116.411612 39.
84;"MULTIPOINT M (116.447984 39.
126;"MULTIPOINT M (116.422563384
153;"MULTIPOINT M (116.399335 39
167;"MULTIPOINT M (116.422563384</td></tr>
<tr><td colspan="3">Run time</td></tr>
<tr><td>2.5 sec</td><td>11 ms</td><td>22.2 sec</td></tr>
</table>

Visualization

Visualization is straightforward for default PostGIS trajectories. A GIS that supports PostGIS data sources renders LineStringM features as lines. The open-source desktop GIS QGIS provides functionality to access m-values and use them for visualization purposes. For example, to visualize speed along a trajectory as shown in Figure 1, we can leverage QGIS's geometry generator feature to split the trajectory into individual segments for rendering on the fly. (For more details, see https://anitagraser.com/2016/10/09/movement-data-in-gis-2-visualization/.) Speed is calculated using the length of the segment and the time between the segment's start and end points. Speed values from 0 to 50 km/h are then mapped to a red-yellow-blue colour ramp as follows:

```
ramp_color('RdYlBu',
  scale_linear(
length(transform(geometry_n($geometry,@geometry_part_num),'EPSG:4326','EPSG:5402
7'))
    / (
      m(end_point(geometry_n($geometry,@geometry_part_num))) -
      m(start_point(geometry_n($geometry,@geometry_part_num)))
    ) * 3.6,
    0,50,
    0,1
  )
)
```

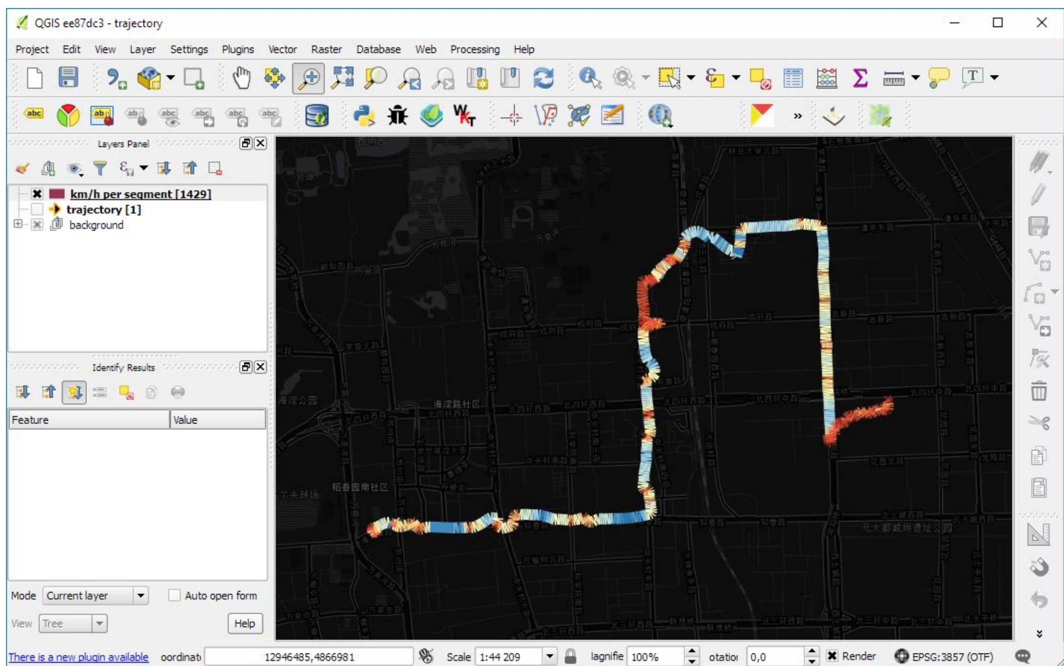

Figure 1: Visualization of speed variation along a trajectory

In contrast, there is no straightforward way to visualize trajectories stored as PG-Trajectory objects, since GIS systems do not support this custom data type.

4 Discussion and outlook

Our experiments reveal that trajectory data models outperform commonly used point-based data models. Our comparison of PG-Trajectory and default trajectory support shows that PG-Trajectory performs worse than default PostGIS for moving point objects. PG-Trajectory query run times are longer than comparable default PostGIS run times (duration queries: 58 times longer; length queries: 4,052 times longer; temporal filters: 208 times longer; spatial filters: 3.7 times longer; extracting positions: 227 times longer). The custom trajectory data type cannot be used with multi-dimensional indices, and there is no straightforward way to visualize the data. PG-Trajectory implements a series of distance measures for trajectory pairs that are not available in the default PostGIS. There is no obvious reason, however, that would prevent porting those functions to the default PostGIS trajectory model. We therefore argue that using default functions is (1) faster and (2) no more complex, (3) provides better integration, especially concerning multi-dimensional indices, and (4) is more sustainable, since these functions are maintained as part of the core project.

In contrast to PG-Trajectory, PostGIS trajectory support does not currently cover moving area trajectories. By transferring the moving point object trajectory approach (where moving points are modelled as lines) to areas, one approach would be 3-dimensional geometries. Similar to LineStringM data types for moving points, moving areas could be modelled as PolyhedralSurface data types. PolyhedralSurfaceM types would be consistent with the LinestringM approach. On the other hand, PolyhedralSurfaceZ data can already be rendered in 3D viewers, such as QGIS 3. As an example, Figure 2 shows a moving square in 2D (left) and 3D map view (right) that is modelled as follows:

```
INSERT INTO area_trajectory (traj) VALUES (
ST_GeometryFromText('POLYHEDRALSURFACE Z (
 ((0 0 0, 0 2 0, 2 2 0, 2 0 0, 0 0 0)),
 ((1 1 1, 1 3 1, 3 3 1, 3 1 1, 1 1 1)),
 ((2 2 2, 2 4 2, 4 4 2, 4 2 2, 2 2 2))
)'))
```

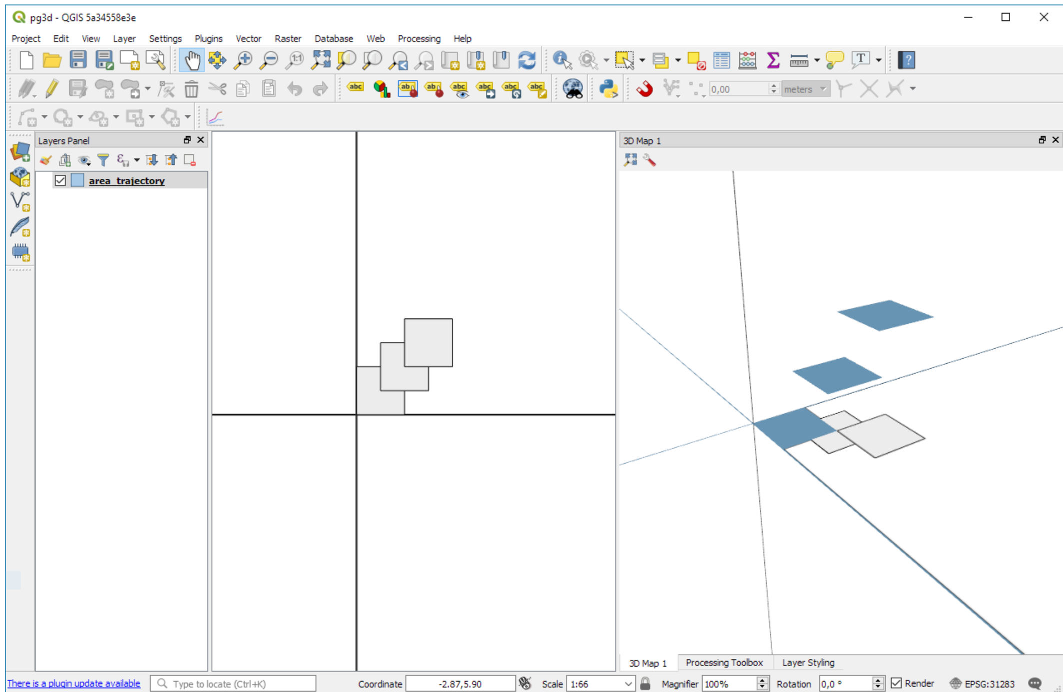

Figure 2: Exemplary moving area object trajectory as PolyhedralSurfaceZ in QGIS

Another limitation of using PostGIS as a moving object database is the lack of explicit spatio-temporal indices. Existing multi-dimensional indices are a step in this direction, but explicit spatio-temporal approaches could take advantage of the constraint that time values in trajectory objects are continuously increasing.

Further performance improvements could make it possible to handle increasingly massive trajectory datasets. PostgreSQL has traditionally (up to version 9.5, released on 2016-01-07 (PostgreSQL, 2016)) run queries in a single thread of execution. This is now changing, with more and more functionality being parallelized (Ramsey, 2017b). There is no parallel processing in PostGIS so far, but it is under development (Ramsey, 2017a). Other avenues that are being explored include the use of GPUs to accelerate calculations. The GPU extension pg-strom is currently being developed but does not yet support PostGIS (Ramsey, 2017b). Another approach is horizontal data partitioning provided by the citus extension. This allows the scaling-up of write and read operations, so that large amounts of data can be processed across multiple worker nodes (Ramsey, 2017b). Moving object applications will profit from appropriate trajectory data models that simplify and speed up interacting with the data.

Acknowledgements

This work was supported by the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT) within the programme 'IKT der Zukunft' under Grant 861258 (project MARNG).

References

DMLAB (Data Mining Lab). (2016). PG-Trajectory. Georgia State University. URL: http://pg-trajectory.dmlab.cs.gsu.edu (accessed 2017-12-15).

Dodge, S., Weibel, R., and Lautenschütz, A.-K. (2008). Towards a taxonomy of movement patterns. Information visualization, 7(3-4):240–252.

Doherty, S. T., Lemieux, C. J., & Canally, C. (2014). Tracking human activity and well-being in natural environments using wearable sensors and experience sampling. Social Science & Medicine, 106, 83-92.

Frentzos, E. (2008). Trajectory data management in moving object databases. PhD thesis, PhD thesis, University of Piraeus.

Güting, R. H. and Schneider, M. (2005). Moving objects databases. Elsevier.

HermesMOD (n.d.) HERMES MOD. UNIPI-InfoLab University of Piraeus Information Management Lab. URL: http://infolab.cs.unipi.gr?page_id=1999 (accessed 2017-12-15).

ISO (2004). ISO 19125-1:2004 Preview Geographic information - Simple feature access - Part 1: Common architecture. URL: https://www.iso.org/standard/40114.html (accessed 2017-12-15).

Kucuk, A., Hamdi, S. M., Aydin, B., Schuh, M. A., & Angryk, R. A. (2016). PG-TRAJECTORY: A PostgreSQL/PostGIS based Data Model for Spatiotemporal Trajectories. In Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom), 2016 IEEE International Conferences on (pp. 81-88). IEEE. doi:10.1109/BDCloud-SocialCom-SustainCom.2016.23

OGC Open Geospatial Consortium (2011). OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture. URL: http://portal.opengeospatial.org/files/?artifact_id=25355 (accessed 2017-12-12).

OGC Open Geospatial Consortium (2017a). Simple Feature Access - Part 1: Common Architecture. URL: http://www.opengeospatial.org/standards/sfa (accessed 2017-12-12).

OGC Open Geospatial Consortium (2017b) OGC® Moving Features. URL: http://www.opengeospatial.org/standards/movingfeatures (accessed 2017-12-12).

PostGIS Development Group. (2015). PostGIS 2.2.7dev Manual. URL: http://postgis.net/docs/manual-2.2/reference.html#Temporal (accessed 2017-12-15).

PostgreSQL Global Development Group. (2016). Release 9.5. URL: https://www.postgresql.org/docs/9.5/static/release-9-5.html (accessed 2017-12-15).

Ramsey, P. (2017a). Parallel PostGIS II. URL: http://blog.cleverelephant.ca/2017/10/parallel-postgis-2.html (accessed 2017-12-15).

Ramsey, P. (2017b). PostGIS Scaling. URL: http://blog.cleverelephant.ca/2017/12/postgis-scaling.html (accessed 2017-12-15).

SECONDO. (2009). Secondo. FernUniversität Hagen. URL: http://dna.fernuni-hagen.de/Secondo.html/index.html (accessed 2017-12-15).

Siła-Nowicka, K., Vandrol, J., Oshan, T., Long, J. A., Demšar, U., & Fotheringham, A. S. (2016). Analysis of human mobility patterns from GPS trajectories and contextual information. International Journal of Geographical Information Science, 30(5), 881-906.

Wang, Y., Luo, Z., Takekawa, J., Prosser, D., Xiong, Y., Newman, S., ... & Yan, B. (2016). A new method for discovering behavior patterns among animal movements. International Journal of Geographical Information Science, 30(5), 929-947.

Wikelski, M., and Kays, R. (2017). Movebank: archive, analysis and sharing of animal movement data. Hosted by the Max Planck Institute for Ornithology. URL: http://www.movebank.org/ (accessed 2017-12-15).

Wiratma, L., van Kreveld, M., and Löffler, M. (2017). On Measures for Groups of Trajectories. In International Conference on Geographic Information Science, 311–330. Springer.

Wolfson, O. (2002). Moving objects information management: The database challenge. In International Workshop on Next Generation Information Technologies and Systems, 75–89. Springer.

Wolfson, O., Chamberlain, S., Dao, S., and Jiang, L. (1997). Location management in moving objects databases. In WoSBIS, volume 97, 7–13.

Zheng, Y., Li, Q., Chen, Y., Xie, X., and Ma, W.-Y. (2008). Understanding mobility based on GPS data. In Proceedings of the 10th international conference on Ubiquitous computing, 312–321. ACM.

Zheng, Y., Xie, X., and Ma, W.-Y. (2010). GeoLife: A Collaborative Social Networking Service among User, Location and Trajectory. IEEE Data Eng. Bull., 33(2):32–39.

Zheng, Y., Zhang, L., Xie, X., and Ma, W.-Y. (2009). Mining interesting locations and travel sequences from GPS trajectories. In Proceedings of the 18th international conference on World wide web, 791–800. ACM.