# Querying Mobile Pollution Data using MobilityDB

Leticia Gómez Department of Systems and Digital Data Department of Systems and Digital Data Department of Decision Engineering Instituto Tecnológico de Buenos Aires Buenos Aires, Argentina lgomez@itba.edu.ar

Alejandro Vaisman Instituto Tecnológico de Buenos Aires Buenos Aires, Argentina avaisman@itba.edu.ar

Esteban Zimányi CODE Université Libre de Bruxelles Brussels, Belgium ezimanyi@ulb.be

Abstract—Air pollution monitoring requires a large number of expensive devices, especially in large cities. To reduce the cost of this process, the use of mobile devices has been proposed. Some proposals promote the use of cheap sensors on board of public buses instead of just a limited number of specialized vehicles. Analyzing the data provided by these mobile devices is computationally costly and complex with traditional database tools. Instead, we propose using MobilityDB, a novel database implemented as an extension to PostgreSQL and PostGIS, which provides support for storing and querying geospatial trajectory data and their time-varying properties (like air-pollution parameters), implementing persistent database types and query operations for such data. We use public data from the city of Delhi to show the viability and advantages of our approach and how analytical queries are expressed in a concise and elegant way using MobilityDB data types and functions.

Index Terms-MobilityDB, spatiotemporal databases, mobility analysis, air pollution

## I. MOTIVATION AND RELATED WORK

Air pollution has reached life-threatening levels in many highly-populated cities.<sup>1</sup> Therefore, public policies are carried out to protect public health, for example, restricting activities in zones with low air quality. Decision making requires monitoring air quality in urban and industrial areas in cities affected by this problem. Air quality is normally measured as a combination of parameters, like Particular Matter (PM), Nitrogen dioxide, Sulfur dioxide, Carbon monoxide and Ozone.<sup>2</sup> Particulate Matter (measured in  $\mu g/m^3$ ) measures the amount of solid particles and liquid droplets found in the air. High values of this parameter are especially dangerous. These particles are produced as a result of the action of sulfur dioxide and nitrogen oxides, pollutants emitted by power plants, industries and automobiles, and they are small enough to be inhaled by humans. PM is categorized according to the size of the particles. For instance PM1 represents particles with diameters that are generally 1 micron and smaller, PM2.5 particles with diameters less than 2.5 microns and smaller, and so on.

Given its importance, air pollution is measured regularly in most cities, through sensors that record multiple parameters. Multiple monitors are used for this, which, given the increasing

<sup>1</sup>https://waqi.info/

number of pollutants and locations, is an expensive procedure, especially (although not only) in developing countries. Therefore, a different approach has become popular, namely mobile air pollution monitoring. Here, air pollution monitoring equipment is mounted on a mobile platform. In this sense, two main approaches were adopted. One of them places quality air monitoring equipment on a mobile platform (e.g., a commercial van) [1]. Another approach promotes the use of small, portable low-cost sensors that may be attached to an individual or mounted on a bike [2]. In any case, mounting sensors on a mobile platform allows for observations in various locations.

As mentioned above, air monitoring in developing countries, where many highly-populated cites are at risk [3], is in most cases unaffordable. The case of Delhi, in India, is representative of this situation [4]. This is one of the most densely populated urban centers in the world, and air pollution there is also one of the highest in the world. The work reported in [5] is based on one of the approaches commented above. Low-cost sensors were installed on public buses, taking advantage of the fixed travel routes of the buses, and also of the additional equipment that the buses carry. There are many technical challenges in this method, but we are not interested in the technical part but in efficiently processing and analyzing the data produced by these devices. The work in [4] reports an experience where sensors were mounted on thirteen public buses in Delhi between November 1st, 2020 and January 31st, 2021, resulting in a dataset that is publicly available.<sup>3</sup> In the present work we use this dataset as a case study.

Mobile data collection typically results in a dataset which is spatially and temporally discontinuous. That means, unlike in the case where a sensor is placed in a fixed location, mobile sensors do not repeat sampling at such location with regularity, that means that there are spatial and temporal gaps. Statistical and mathematical techniques have been proposed to solve this problem, for example, determining a long-term mean from incomplete samples, which requires adjusting the mobile data based on the value at a fixed location monitor or incorporating both mobile and fixed location data into the air pollution model [6], [7]. Having many mobile low-cost sensors (instead of a few ones moving around) may mitigate this problem. In

<sup>3</sup>https://www.cse.iitd.ac.in/pollutiondata/details

<sup>&</sup>lt;sup>2</sup>https://www.epa.vic.gov.au/for-community/monitoring-your-environment /about-epa-airwatch/calculate-air-quality-categories

addition, we propose to take advantage of a spatiotemporal database to process and analyze the data produced by mobile sensors as we discuss in this work.

MobilityDB [8] is a novel database that builds on PostGIS,<sup>4</sup> the spatial extension of the PostgreSQL database.<sup>5</sup> MobilityDB extends the type system of PostgreSQL and PostGIS with data types for representing spatiotemporal data. These data types are based on the notion of temporal types and their associated operations. MobilityDB defines temporal types that can represent the evolution over time of any kind of data, like integer, float, Boolean, and text. Thus, new data types like temporal integer (tint), temporal float (tfloat), and temporal Boolean (tbool) are defined, along with functions to manipulate them. Combining the support of spatial and temporal data types, MobilityDB is also a powerful tool for building so-called mobility data warehouses, as shown in [9]. We explain MobilityDB in detail in Section III.

## Contributions and Paper Organization

The temporal data types included in MobilityDB allow representing long sequences of geographic data (e.g., mobile stations' positions) and float data (like PM parameters), making the manipulation and analysis of such kinds of data, simpler and more efficient than the traditional database alternatives. In particular, regarding what was mentioned above, spatiotemporal interpolation is already built-in in the database. Further, complex queries can be written in a much simpler way than using SQL without the mobility extension. In this paper we show how we can take advantage of those features using the Delhi case study reported in [4]. As far as we are aware of, this is the first proposal for using spatiotemporal databases to manage and analyze mobile air pollution data (and mobile sensor data in general).

In the remainder, Section II introduces our case study. Section III presents the MobilityDB database, in order to make the paper self-contained, also using our case study to give some examples. In Section IV we show how the MobilityDB database for our case study is built, while in Section V we show the power of MobilityDB to write complex spatiotemporal queries in a concise and elegant way. A discussion concludes the paper in Section VI.

## II. RUNNING EXAMPLE

This section describes the running example and the public dataset we use throughout this paper. The dataset contains observations obtained by low-cost sensors mounted in the public transportation (buses) of Delhi city in India.<sup>6</sup>

The dataset is composed of different files, one per day from 1st November 2020 to 30th January 2021. Each file contains observations from thirteen different buses equipped with mobile sensors capable of measuring three type of pollutants:  $PM_{1.0}$ ,  $PM_{2.5}$  and  $PM_{10}$ .

<sup>4</sup>https://postgis.net/

<sup>6</sup>https://www.cse.iitd.ac.in/pollutiondata/details

Every file contains a header and each row represents an observation:

- Id (number): row identifier
- Uid (text): identifies univocally an observation in the dataset
- dateTime (text): instant when the observation was recorded.
- deviceID (text): identifier of the device mounted in the bus (there are 13 buses)
- lat (double): latitude coordinate (WGS84)
- lon (double): longitude coordinate (WGS84)
- pm1\_0 (double): PM<sub>1.0</sub> sampled value
- pm2\_5 (double): PM<sub>2.5</sub> sampled value
- pm10 (double): PM<sub>10</sub> sampled value

We first create a table for importing the data into a .csv file, which we call delhilnput, as follows.

CREATE TABLE delhiInput(seq int, uid text, deviceId text,

t timestamptz, lat float, long float, pm1\_0 float, pm2\_5 float, pm10 float);

To allow displaying the position of each measurement in a geographic information system (GIS) like QGIS,<sup>7</sup> we add a geom attribute as follows:

## ALTER TABLE delhilnput

ADD COLUMN geom geometry(Point, 4326);

UPDATE delhiInput SET geom = ST\_MakePoint(long, lat);

The final input database contains over twelve million records with a size of 12 GB.

Just as an introductory illustration, Fig. 1 shows a portion of the sensor measures of  $PM_{2.5}$  lower than 50 (in green) and higher than 200 (in red).



Fig. 1. Portion of sensor measures in the Delhi dataset.  $\mathsf{PM}_{2.5}>200$  (red);  $\mathsf{PM}_{2.5}<50$  (green).

# III. MOBILITYDB: A MOBILITY DATABASE

This section presents the Mobility $DB^8$  database. A more formal description can be found in [8].

<sup>7</sup>www.qgis.org <sup>8</sup>https://mobilitydb.com/

<sup>&</sup>lt;sup>5</sup>https://www.postgresql.org

As mentioned, MobilityDB is built on top of PostgreSQL. MobilityDB provides *set*, *span*, and *span set* types for representing finite subsets of the domains of base or time data types. *Set types* represent a set of distinct values. Examples are intset or dateset, which are defined over the int and date types provided by PostgreSQL. *Span types* represent ranges of base values and are defined by a lower and upper bounds, which may be inclusive or exclusive. Examples are intspan and datespan. *Span set types* represent set of disjoint spans. Examples are intspanset and datespanset.<sup>9</sup>

MobilityDB implements a set of operations on *time types*, which are defined at two *granularities*: date or timestamptz (timestamp with time zone). Four times types are used for defining finite subsets of the time domain at each granularity: date, dateset, datespan, and datespanset, as well as timestamptz, tstzset, tstzspan, and tstzspanset. We describe next these types using the date granularity.

A date value represents a time instant at a day granularity. A dateset value represents a set of distinct date values. It must contain at least one element, and the elements must be ordered. An example is as follows

SELECT dateset '{2021-01-01, 2021-01-03}';

A value of the datespan type has two bounds, the lower and the upper bounds, which are date values. The bounds can be inclusive (represented by "[" and "]"), or exclusive (represented by "(" and ")"). A datespan value with equal and inclusive bounds corresponds to a date value. An example of a datespan value is

SELECT datespan '[2021-01-01, 2021-01-03)';

A datespanset value represents a set of disjoint datespan values. It must contain at least one element and the elements must be ordered. An example is:

SELECT datespanset '{[2021-01-01, 2021-01-03), [2021-01-04, 2021-01-06)}';

MobilityDB also provides *temporal types* for representing values that evolve across time.<sup>10</sup> The temporal types tbool, tint, tfloat, and ttext are, respectively, based on the PostgreSQL types bool, int, float, and text. Temporal types tgeompoint and tgeogpoint are also provided. These types are based on the PostGIS types geometry and geography restricted to 2D and 3D points. Temporal types may be discrete or continuous depending on their base type. Discrete temporal types (such as tbool, tint, or ttext) evolve in a stepwise manner, while continuous temporal types (such as tfloat) evolve in either a linear or stepwise manner.

The *duration* of a temporal value states the time extent at which the evolution of values is recorded. Temporal values come in three durations, namely, *instant*, *sequence*, and *sequence set*. A temporal *instant* value represents the value at a time instant, such as

<sup>10</sup>Currently, MobilityDB provides temporal types only at the timestamptz granularity.

SELECT tfloat '17.1@2022-01-01 08:00:00';

A temporal *sequence* value represents the evolution of the value during a sequence of time instants, where the values between these instants are interpolated using either a discrete, stepwise or linear function. An example with *discrete* interpolation is as follows:

```
SELECT tint '{10@2022-01-01 08:00:00,
20@2022-01-01 08:05:00, 15@2022-01-01 08:10:00}';
```

where the value is defined at the given timestamps and undefined everywhere else (in other words, discrete interpolation means no interpolation). An example of a *temporal sequence* value with *step* interpolation is as follows:

```
SELECT tint '(10@2022-01-01 08:00:00,
20@2022-01-01 08:05:00, 15@2022-01-01 08:10:00]';
```

while an example of temporal sequence value with *linear* interpolation is as follows:

SELECT tfloat '(10@2022-01-01 08:00:00, 20@2022-01-01 08:05:00, 15@2022-01-01 08:10:00]';

The value of a temporal sequence is interpreted by assuming that the time period defined by every pair of consecutive values v1@t1 and v2@t2 is lower inclusive and upper exclusive. unless they are the first or the last instants of the sequence, in which case the bounds of the whole sequence apply. The value of the temporal sequence between two consecutive instants depends on whether the subtype is discrete or continuous. For example, the tint sequence above represents that the value is 10 during (2022-01-01 08:00:00, 2022-01-01 08:05:00), 20 during [2022-01-01 08:05:00, 2022-01-01 08:10:00), and 15 at the end instant 2022-01-01 08:10:00. On the other hand, the tfloat sequence above tells that the value evolved linearly from 10 to 20 during (2022-01-01 08:00:00, 2022-01-01 08:05:00) and evolved from 20 to 15 during [2022-01-01 08:05:00, 2022-01-01 08:10:00]. MobilityDB also allows representing sequences with stepwise interpolation when the subtype is continuous. An example is given next.

SELECT tfloat 'Interp=Step;(10.1@2022-01-01 08:00:00, 20.2@2022-01-01 08:05:00, 15.2@2022-01-01 08:10:00]';

Finally, a temporal *sequence set* value represents the evolution of the value at a set of sequences, where the values between them are unknown (like in the case of mobile sensors with no measures during long intervals), for example:

## SELECT tfloat

'{[17.2@2022-01-01 08:00:00, 17.5@2022-01-01 08:05:00], [18.2@2022-01-01 08:10:00, 18.5@2022-01-01 08:15:00]}';

Temporal types have an associated set of operations, which we use later in our queries. As an example, operations get-Time and getValues return, respectively, the projection of a temporal value into the time domain and the value range. Both operation results in a range value. Operations atTime and atValues restrict the function to a given subset of the time or base domain defined by a range value. For example, atTime applied to a trip returns the trip restricted to an interval which

<sup>&</sup>lt;sup>9</sup>The span and span set types in MobilityDB correspond to the range and multirange types in PostgreSQL, but they have a more efficient implementation.

is an argument of the function. Operations atMin and atMax restrict the function to the points in time when its value is minimal or maximal, respectively, while valueAtTimestamp gets the base value of the function at a given timestamp.

Local aggregate operators compute an aggregate base value from a *single* temporal value. For example, the **twavg** computes the time-weighted average of a base value (for example, in our case, a time-weighted average of a PM value, which is computed very efficiently, as we show later).

Finally, generalizing operations on base types for temporal types is called *lifting* [11]. The semantics of lifted operations is that the result is computed at each instant using the corresponding non-lifted operation. For example, the sum between two base types is lifted by allowing any of the arguments to be a temporal type and return a temporal type. Thus, we take two parameters represented as temporal floats, and compute their average at any time instant.

Aggregate operations may also be lifted. Examples are tcount, tmin, tmax, and tavg, which combine several temporal values, yielding a new temporal value where the traditional aggregate functions count, min, max, and avg are computed at each time instant.

Finally, the unnest operation, commonly used in querying, transforms a temporal value into a set of (value,time) pairs.

It is important to remark that, as discussed in detail in [8], to ensure the closure of operations, when the operands of a lifted operation have a *linear* interpolation, the result of the operation must also be represented using linear interpolation. Since the result of a lifted operation over two linear functions may be quadratic, as is the case for temporal multiplication of real numbers, MobilityDB approximates this result by a linear function while keeping the *turning points* of the quadratic function in the result.

We close this section by briefly showing the operations for temporal types in MobilityDB. We use, as an example, a very simplified version of the table that we build in Section IV. The table pmtrips contains the identifier of the mobile device (e.g., a bus), and a temporal float, representing the measured values of the  $PM_{1.0}$  parameter at different times.

```
CREATE TABLE pmtrips(
deviceId varchar(5) PRIMARY KEY,
PM1_0 tfloat );
```

Tuples can be inserted in this table as follows:

## **INSERT INTO pmtrips VALUES**

```
('d1', tfloat '{[25.0@2021-01-01, 25.0@2021-07-01),
[30.0@2022-01-01, 30.0@2022-04-01)}'),
('d2', tfloat '{[45.0@2021-04-01, 45.0@2022-01-01),
[60.0@2022-01-01, 60.0@2022-04-01)}');
```

Given this table with the two tuples, the query

# SELECT deviceId, getTime(PM1\_0) FROM pmtrips

returns the following values

'd1' | {[2021-01-01, 2021-07-01), [2022-01-01, 2022-04-01)} 'd2' | {[2021-04-01 2022-01-01), [2022-01-01, 2022-04-01)} The second column of the result is of type tstzspanset since temporal types in MobilityDB have a timestamptz granularity.

Similarly, the query

```
SELECT deviceId,
```

```
valueAtTimestamp(PM1_0, timestamptz '2021-04-15'),
valueAtTimestamp(PM1_0, timestamptz '2021-07-15')
FROM pmtrips
```

returns the following values

'd1' | 25 | NULL

'd2' | 45 | 45

where the NULL value above represents the fact that the parameter for d1 is undefined on 2021-07-15.

The following query:

SELECT deviceld, atTime(PM1\_0, tstzspan '[2021-04-01, 2021-11-01)') FROM pmtrips

returns

'd1' | {[25@2021-04-01, 25@2021-07-01)}

'd2' | {[45@2021-04-01, 45@2021-11-01)}

Here, the temporal attributes have been restricted to the period given in the query.

The next query asks for the minimum and maximum values and the time when they occurred:

SELECT deviceId, atMin(PM1\_0), atMax(PM1\_0) FROM pmtrips

The result is:

```
'd1' | {[25@2021-01-01, 25@2021-07-01]} |
{[30@2022-01-01, 30@2022-04-01]}
'd2' | {[45@2021-04-01, 45@2021-10-01)} |
{[60@2021-10-01, 60@2022-07-01]}
```

The next query asks, given two devices 'd1' and 'd2', the time periods where the  $PM_{1.0}$  measured by 'd1' was lower than the one measured by 'd2'.

SELECT P1.PM1\_0 #< P2.PM1\_0 FROM pmtrips P1, pmtrips P2 WHERE P1.deviceId = 'd1' and P2.deviceId = 'd2'

The query returns the temporal Boolean value

{[t@2021-04-01, t@2021-07-01), [t@2022-01-01, t@2022-04-01)}

Note that no comparison is performed when a value is undefined.

As an example of temporal aggregation, the next query asks for the average  $PM_{1.0}$  value across time.

SELECT tAvg(PM1\_0) FROM pmtrips

This query returns

{[25@2021-01-01, 25@2021-04-01), [35@2021-04-01, 35@2021-07-01],

## [60@2022-04-01, 60@2022-07-01)]

Note that the second tuple in the result represents the average between the values 25 and 45, valid in the interval in which the two values are valid at the same time.

## IV. BUILDING THE MOBILITY DATABASE

Having introduced the running example, we are ready to build the mobility database. The idea is to transform the table Delhilnput, which contains one record for each measure obtained by the thirteen mobile sensors (that is, the twelve million tuples), into a database containing one tuple for each part of a trip of a bus carrying a sensor. These parts represent trajectories where the gaps between recordings are less than thirty minutes long (we explain this below). That means, a bus trajectory will be split into many trips. Each record of the new table contains four attributes of temporal type. First, there is a trip attribute, which is of type tgeompoint representing the continuous trip of the corresponding mobile sensor. Spatiotemporal interpolation is carried out automatically by the database engine when data are loaded and also when data are queried. That means, for each time instant we obtain a value, which can be the actual position or an interpolated one. Then, we do the same for each PM parameter, namely  $PM_{1.0}$ , PM<sub>2.5</sub> and PM<sub>10</sub>. For each one of them we define an attribute of type temporal float (tfloat). Again, interpolation is defined automatically. Details of the interpolation process can be found in [8]. Finally, we obtain a sequence for each one of the four attributes that were created. We detail the process next.

Each sequence is created from data that contains a value and a time instant. Thus, we must add, in the input table delhilnput, one column for the spatiotemporal position of the sensor at each recording instant. The spatial position is built using the PostGIS function ST\_MakePoint and the time instant is taken from column t. In this way we create the column trip\_inst which is of type tgeompoint, which is the building block for the sequences of spatiotemporal positions of the sensors. We proceed analogously for each parameter. The procedure is as follows (we only show one of the PM parameters, the other ones are analogous).

```
ALTER TABLE delhilnput

ADD COLUMN trip_inst tgeompoint;

UPDATE delhilnput SET

trip_inst = tgeompoint(ST_MakePoint(long, lat), t);

ALTER TABLE delhilnput

ADD COLUMN pm1_0_inst tfloat;

UPDATE delhilnput SET

pm1_0_inst = tfloat(pm1_0, t);

ALTER TABLE delhilnput

ADD COLUMN pm2_5_inst tfloat;

...
```

We can now create and populate the table containing the sequence of spatiotemporal positions and parameters. We call this table delhiTrips. We next show how we create the table and produce the sequences using the tgeompoint and tfloat constructors.

CREATE TABLE delhiTrips (Id integer, deviceId text, trip tgeompoint, pm1\_0 tfloat, pm2\_5 tfloat, pm10 tfloat, PRIMARY KEY (deviceid, Id) );

The trips of the buses, therefore the corresponding measurements, have gaps. For example, a bus may run from 6AM through 8PM and start over at 6AM. We cannot leave this gap and consider a continuous sequence, since the interpolation would be wrong. There are many ways for partitioning trajectories. In this case, we assume that if there is a gap of at least thirty minutes, then we consider that a new trips starts. This explains why the primary key of the table is the pair (deviceid, ld)

Due to space limitations we do not include the complete script which splits the complete trajectory of each bus into a collection of sequences, one per record. We only show the statements that create the sequences. The statements below are executed in a loop, one time for each device ID.

# **INSERT INTO** delhiTrips

# SELECT myid, deviceId,

```
tgeompoint_seq(array_agg(trip_inst ORDER BY trip_inst)),
tfloat_seq(array_agg(pm1_0_inst ORDER BY pm1_0_inst)),
tfloat_seq(array_agg(pm2_5_inst ORDER BY pm2_5_inst)),
tfloat_seq(array_agg(pm10_inst ORDER BY pm10_inst))
FROM delhilnput
GROUP BY deviceId;
```

Since we will most likely need to display in a GIS the spatial trajectories of the sensors (and GISs do not understand spatiotemporal data types), we created one additional column of LineString type, that is populated using the MobilityDB trajectory function, which computes the spatial projection of the trip attribute, as follows:

# ALTER TABLE delhiTrips

ADD COLUMN trajectory geometry(LineString, 4326); UPDATE delhiTrips SET trajectory = trajectory(trip);

We finally deleted the sequences with a duration of less than ten minutes. This is easily done with the duration function provided by MobilityDB as follows:

DELETE FROM delhiTrips WHERE duration(trip) < '10 minute';

The resulting table contains 3352 tuples. Note that, since there are thirteen buses, each bus trajectory has been split many times. As additional information, we report that the maximum number of trips (i.e., sequences) for a device is 401, while the minimum number of trips for a device is 119.

One of the immediately visible advantages of this approach, refers to the database size. MobilityDB implements a data compression strategy that is applied when data are loaded (this also occurs in querying). This procedure removes redundant data which can be inferred using interpolation. Intuitively, if a bus moves in a straight line and the input database contains, says, one hundred position records, only the first and last positions are kept. This results in a dramatic size reduction, as we can see in our example, where the 12 GB of the delhilnput (including the geom column) was reduced to 680MB for the delhiTrips table (including the trajectory column).

Figure 2 shows a portion of the trajectories of three mobile sensors, represented with different colors.

#### V. USING THE MOBILITY DATABASE FOR ANALYSIS

In the previous section we showed how the novel MobilityDB database allows a fast data loading process, reducing the size of the original database, in this case, by a factor



Fig. 2. Mobile sensors in Delhi.

of ten. Now we show how complex and useful analytical queries can be expressed in SQL using MobilityDB. We do this through typical queries that an analyst interested in the effect of pollution over human health would likely want to express. We remark that we do not aim at reporting performance results, but at showing the functionality and expressiveness of MobilityDB queries. Nevertheless, it is worth noting that all the queries shown here ran in a few seconds over standard laptop computers equipped with the PostgreSQL database extended with MobilityBD functionality.

Given the high pollution levels that can be observed in the raw data, bus drivers and passengers are exposed to different pollutants during their trips. The following queries aim at evaluating the length and duration of such exposures.

*Query 1:* Compute the time-weighted average of pollutants to which each driver was exposed during her trips occurred in the morning of 21th and 22nd of December of 2020

This query illustrates the benefit of working with a mobility database. The twAvg function computes the time-weighted average of a temporal float data type, in this case, of the pollutants that were measured by the sensors carried by the bus. The query reads in SQL:

## WITH time AS (

SELECT tstzspanset '{[21-12-2020 09:00, 21-12-2020 12:00], [22-12-2020 09:00, 22-12-2020 12:00]}' AS period ) SELECT deviceid, id, twAvg(atTime(PM1\_0, time.period )) AS PM1\_0, twAvg(atTime(PM2\_5, time.period )) AS PM2\_5, twAvg(atTime(PM10, time.period )) as PM10 FROM delhiTrips, time WHERE PM1\_0 && spanN(period, 1) OR PM1\_0 && spanN(period, 2) OR PM2\_5 && spanN(period, 2) OR

## PM10 && spanN(period, 1) OR PM10 && spanN(period, 2)

We can see how simple results to express in MobilityDB such a complex query. The first Common Table Expression (CTE) computes the table time, with only one tuple: a span set (a set of intervals) corresponding to one day's morning and the morning after. This avoids embedding the same literal in different parts of the query. Then, we use the table delhiTrips and keep the tuples that have temporal float pollutant measurements (i.e.  $PM_{1.0}$ ,  $PM_{2.5}$  and  $PM_{10}$ ) that overlap the bounding box of the period of interest (this is performed in the WHERE clause). Nevertheless, since our period of interest is a spanset, in this case composed of two spans, we need to split it and calculate each bounding box separately. The subexpressions spanN(period, 1) and spanN(period, 2) return the first and second part of the spanset. The operator '&&' checks if there is an overlap between each span and the measure of interest.

Finally, in the SELECT clause, the expressions at-Time(pollutant, period) restrict the pollutant to the period of interest, and then the aggregate function twAvg is applied.

We now further elaborate on the expressions in the WHERE clause. We remark that these expressions are aimed at speeding-up the query performance, taking advantage of indices that MobilityDB can build over temporal types. Typically, this is the case of the computation of the functions that project the temporal types over the value/spatial and/or time dimensions. GiST and SP-GiST indexes can be created for table columns of temporal types. The GiST index implements an R-tree and the SP-GiST index implements an n-dimensional quad-tree. The GiST and SP-GiST indices store the bounding boxes, actually a minimum bounding rectangle (MBR), for the temporal types. Expressions like PM1\_0 && spanN(period, 1) perform the intersection between the time intervals and the temporal value, using the index. If we do not want to force the use of the index, in the WHERE clause we would only write atTime(Trip, time.period) IS NOT NULL.

Indices are created as follows:

```
CREATE INDEX delhiTrips_trip_ldx
ON delhiTrips USING SPGist(Trip);
CREATE INDEX delhiTrips_pm1_0_ldx ON
delhiTrips USING Gist(PM1_0);
```

The next query shows how MobilityDB queries can be used to write analytical aggregate queries à la OLAP (Online Analytical Processing).

*Query 2:* Compute the average air quality at which the bus drivers were exposed during the mornings (from 9AM to 12PM).

This query expresses what we call a roll-up, in OLAP jargon. To avoid redundancy, we limit ourselves to just use  $PM_{2.5}$  and  $PM_{10}$  in the query.

WITH eachDate AS ( SELECT DISTINCT deviceid, id, unnest(timestamps(PM2\_5))::date AS dd FROM delhiTrips WHERE duration(PM2\_5) < '12 hour'::interval),

| morning AS (                                     |
|--|
| SELECT deviceid, id,                             |
| span(dd+'09:00'::interval, dd+'12:00'::interval, |
| true, true) interval                             |
| FROM eachdate)                                   |
| SELECT delhiTrips.deviceid, delhiTrips.id,       |
| twAvg(atTime(PM2_5, morning.interval)) as PM2_5, |
| twAvg(atTime(PM10, morning.interval)) as PM10    |
| FROM delhiTrips, morning                         |
| WHERE delhiTrips.deviceid = morning.deviceid AND |
| delhiTrips.id = morning.id AND                   |
| PM2_5 && morning.interval AND                    |
| PM10 && morning.interval                         |
| GROUP BY delhiTrips.deviceid, delhiTrips.id,     |
| morning.interval                                 |
| ORDER BY PM2_5 desc, PM10 desc                   |
|  |

First we compute the eachdate CTE, which calculates the date of each trip of each device. Given that the timestamps are arrays, we use the unnest function to produce the corresponding tuples. We use DISTINCT to consider the devices that span through different dates. The morning CTE builds, for each trip of each device, the three-hour morning intervals. The final query computes the time-weighted averages and aggregates them by trip and morning intervals. Figure 3 shows the result of the query, where we can see the high level of contamination due to these pollutants.

| deviceid<br>[PK] text | id<br>[PK] integer <b>*</b> | interval<br>tstzspan                             | pm2_5<br>double precision | pm10<br>double precision |
|-----------------------|-----------------------------|--|---------------------------|--------------------------|
| 000000038861c77       | 2                           | [2021-01-01 09:00:00+01, 2021-01-01 12:00:00+01] | 678.7488589685075         | 756.0245321770881        |
| 00000000980cc94b      | 3                           | [2021-01-01 09:00:00+01, 2021-01-01 12:00:00+01] | 614.3502857142857         | 687.6045714285714        |
| 0000000029c36345      | 3                           | [2021-01-01 09:00:00+01, 2021-01-01 12:00:00+01] | 614.1090799517879         | 669.4689299584841        |
| 000000024568afd       | 2                           | [2021-01-01 09:00:00+01, 2021-01-01 12:00:00+01] | 554.2151480310434         | 610.2062374245473        |
| 00000000d4bc37f2      | 2                           | [2021-01-01 09:00:00+01, 2021-01-01 12:00:00+01] | 539.8479427549195         | 584.0536113595707        |
| 00000000980cc94b      | 2                           | [2021-01-01 09:00:00+01, 2021-01-01 12:00:00+01] | 533.7560137457044         | 581.6813669339443        |
| 1000000dc5bb76b       | 1                           | [2021-01-01 09:00:00+01, 2021-01-01 12:00:00+01] | 521.3859595607461         | 599.3530590901167        |

## Fig. 3. Results for Query 2.

Our last query shows the full power of MobilityDB to address spatiotemporal problems in a very simple and elegant way. We first compute a grid that contains all the bus trips (the CTE mbr below), and for each square in the grid (the CTE grid below), we compute the average value of the  $PM_{2.5}$  parameter. Figure 4 shows the result in a Choropleth map.

## WITH mbr AS (

SELECT ST\_Transform(ST\_SetSRID(ST\_Extent(trajectory), 4326), 3857) as tripgeom FROM delhiTrips), grid AS ( SELECT (ST\_SquareGrid(2500, MBR.tripgeom)).geom AS squares FROM mbr), pm25 AS ( SELECT avg(PM2\_5) AS pm2\_5avg, ST\_Transform(grid.squares, 4326) AS gridsquares FROM delhiInput AS pts, grid g ON ST\_Contains(ST\_Transform(g.squares, 4326), ST\_Transform(pts.geom, 4326)) GROUP BY g.squares ) SELECT \* FROM pm25

*Query 3:* We want to compute, for each bus, the total amount of time that the bus driver has been exposed in the darker zones



Fig. 4. Grid illustrating the number of measurements.

of the  $PM_{2.5}$  grid. These zones have an average value between 221 and 290, which is considered as very poor quality.<sup>11</sup>

The MobilityDB query to compute this, uses the pm25 CTE above. The query is written as:

```
WITH ...
pm25 AS (...),
highPoITrips AS (
SELECT deviceid, id,
atGeometry(transform(delhiTrips.trip,4326),
PM25.gridsquares) AS dangPart
FROM delhiTrips, PM25
WHERE pm2_5avg > 221 )
SELECT deviceId, id, duration(getTime(dangPart))
FROM highPoITrips
WHERE dangPart IS NOT NULL
```

The atGeometry function, restricts the trip to a given geometry. Note that this is much powerful than a simple intersection, since it computes the intersection between a spatiotemporal trajectory and a geometry. In this case, the geometries are the squares in the grid where the  $PM_{2.5}$  parameter is higher than 221 (the darker squares previously computed). This is stored in the highPolTrips CTE. The final query uses the MobilityDB function atTime to restrict the intersection between each trip and the corresponding squares (the intersections are called dangPart in the highPolTrips CTE), to the time spans when they occurred. Then, the MobilityDB duration function computes the sum of all the parts of each trip occurred within the dangerous zones.

<sup>11</sup>https://www.epa.vic.gov.au/for-community/monitoring-your-environment /about-epa-airwatch/calculate-air-quality-categories

## VI. DISCUSSION AND CONCLUSION

We claimed throughout this paper that, using the MobilityDB database to address problems like the ones in this paper, has many advantages over using traditional database tools to manage and analyze the data measured by the mobile sensors. *First*, since continuous data (in the delhiTrips table) are produced from the raw data (in our example, the delhiInput table), a query asking for temporal data type attribute, returns a value for every point in the trajectory of the sensor, and users do not need to care about interpolation. Further, although in the case study, mobile sensors were placed on fixed-route buses, in other situations domain experts may design specific routes that cover all areas of interest, optimizing the routes.

*Second*, we showed that MobilityDB built-in data compression strategies result in order of magnitude reduction in the database size. In our case, we went from 12GB in the input data, down to less than 700MB in the delhiTrips table.

*Third*, queries in MobilityDB are written in a very concise and elegant way, as shown in Section V. Below, we elaborate on this topic, discussing the effort that writing analogous queries in classic SQL would require.

Consider Query 1. This query uses a tstzspanset, to restrict the analysis to specific (closed) time intervals. Then, overlaps between these intervals and the temporal functions are computed by using the '&&' operator. In MobilityDB it is easy to construct open, half-open and closed intervals. However, in native PostgreSQL there is no function allowing to express close intervals, only half-open intervals can be built. Therefore, in SQL, to express complex time constraints we need to build complex expressions combining equalities and inequalities between the timestamps' endpoints.

Query 1 also computes the parts of the pollutant continuous evolution (not just samples like in the delhilnput table), that lie within the interval of interest, using the function atTime. In native PostgreSQL we would need to generate a LineString as a sequence of pollutant values ordered by time. The points in this linestring will be composed by pairs (pollutant-value, number) where number is a mapping from the time instant at which every sample was recorded, to a number (that is, number will not be a timestamp). The following expression sketches the idea:

SELECT ST\_MakeLine(ARRAY( SELECT ST\_MakePoint(PM2\_5, CAST(EXTRACT(EPOCH from t) AS integer)) FROM delhilnput d2 ORDER BY t)) PM2\_5

Further, MobilityDB is equipped with a large toolset of functionalities and *lifting* (Section III) and aggregate operators, that make it simple to write complex queries. Queries 1 and 2 use the time-weighted average function, twAvg. Internally, this function encodes a complex computation of weighted averages according to the time span of each value of the variable. To compute this in SQL, PSM functions must be coded.

Finally, MobilityDB includes GiST and SP-GiST indexes for temporal types. The former implements an R-tree and the latter implements an n-dimensional quad-tree. GiST and SP- GiST indexes store the bounding box for the temporal types and are the basis for speeding-up query performance.

Query 3 also shows how the temporal intersection between a spatiotemporal trajectory and a geometry, is computed using the atGeometry function. Note that this intersection is not just the geometric intersection provided by PostGIS (which returns a geometry), but (intuitively) the computation of this intersection at every point in time, therefore returning a spatiotemporal type. Due to space restrictions we omit the details, although the reader can guess that writing this in classic SQL would require expert SQL knowledge.

MobilityDB is in constant evolution and being applied in many real-world cases, like air traffic control and public transport. The problem presented in this paper opens a new application field that can help in performing large cities monitoring tasks at low cost.

## REFERENCES

- J. Wallace, D. Corr, P. Deluca, P. Kanaroglou, and B. McCarry, "Mobile monitoring of air pollution in cities: the case of hamilton, ontario, canada," *J. Environ. Monit.*, vol. 11, pp. 998–1003, 2009. [Online]. Available: http://dx.doi.org/10.1039/B818477A
- [2] B. Elen, J. Peters, M. V. Poppel, N. Bleux, J. Theunis, M. Reggente, and A. Standaert, "The aeroflex: A bicycle for mobile air quality measurements," *Sensors*, vol. 13, no. 1, pp. 221–240, 2013. [Online]. Available: https://www.mdpi.com/1424-8220/13/1/221
- [3] P. M. Mannucci and M. Franchini, "Health effects of ambient air pollution in developing countries," *International journal of environmental research and public health*, vol. 14, no. 9, p. 1048, 2017.
- [4] C. Tripathi, P. Baredar, and L. Tripathi, "Air pollution in delhi: biomass energy and suitable environmental policies are sustainable pathways for health safety," *Current Science*, vol. 117, no. 7, pp. 1153–1161, 2019.
- [5] S. K. Chauhan, S. Ranu, and R. Sen, "Fine-grained spatio-temporal particulate matter dataset from delhi for ml based modeling," unpublished. [Online]. Available: https://www.cse.iitd.ac.in/~rijurekha/p apers/neurips\_2023.pdf
- [6] T. Larson, S. B. Henderson, and M. Brauer, "Mobile monitoring of particle light absorption coefficient in an urban area as a basis for land use regression," *Environmental Science & Technology*, vol. 43, no. 13, pp. 4672–4678, 2009. [Online]. Available: https: //doi.org/10.1021/es803068e
- [7] M. D. Adams and P. S. Kanaroglou, "Mapping real-time air pollution health risk for environmental management: Combining mobile and stationary air pollution monitoring with neural network models," *Journal of Environmental Management*, vol. 168, pp. 133–141, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/ S030147971530428X
- [8] E. Zimányi, M. Sakr, and A. Lesuisse, "MobilityDB: A mobility database based on PostgreSQL and PostGIS," ACM Transactions on Database Systems, vol. 45, no. 4, pp. 19:1–19:42, 2020.
- [9] A. Vaisman and E. Zimányi, "Mobility data warehouses," *ISPRS International Journal of Geo-Information*, vol. 8, no. 4, p. 170, 2019.
- [10] M. D. Adams and D. Corr, "A mobile air pollution monitoring data set," *Data*, vol. 4, no. 1, 2019. [Online]. Available: https: //www.mdpi.com/2306-5729/4/1/2
- [11] R. Güting, M. Böhlen, M. Erwig, C. Jensen, N. Lorentzos, M. Schneider, and M. Vazirgiannis, "A foundation for representing and quering moving objects," ACM Transactions on Database Systems, vol. 25, no. 1, pp. 1– 42, 2000.