Distributed Moving Object Data Management in MobilityDB

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

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

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

ABSTRACT

The availability of moving object data that is being collected nowadays, and the demand of using them in applications, have generated the need for spatiotemporal data management systems. MobilityDB is an open source moving object database system. Its core function is to efficiently store and query moving object trajectories. It is engineered up from PostgreSQL and PostGIS, providing spatiotemporal data management via SQL. In order to store and analyze the massive datasets of trajectories, a scalable version is required. In this paper, we present a solution to distribute MobilityDB using Citus. Citus is a PostgreSQL extension for distributed query processing. We report on the integration architecture, and the types of queries that can be distributed out of the box. The experiments prove the feasibility of the solution, and show a significant speed up in queries.

CCS CONCEPTS

• **Information systems → Database management system engines**.

KEYWORDS

trajectory, sharding, distributed query planning

ACM Reference Format:
Mohamed Bakli, Mahmoud Sakr, and Esteban Zimanyi. 2019. Distributed Moving Object Data Management in MobilityDB . In *8th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial'19), November 5, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3356999.3365467

1 INTRODUCTION

Moving object data is being collected at scale in many domains. This includes mobility solutions in smart cities [14], crowd management [17], air traffic control [4], climate change [19], and ethology studies [15]. This data is characterized by being multidimensional and big in size. A single sensor generates thousands of observation per day. Existing big data management systems do not have the notion of trajectory. It is important to have it as first class citizen. This is offered by moving object database systems.

MobilityDB[1] [23] is an open source extension of PostgreSQL and PostGIS, which makes it easily adopted in industry and real life applications. It implements spatiotemporal and temporal database types inside PostgreSQL, and a rich set of query operations. By far, MobilityDB is the only SQL database for moving objects. It is a centralized system that can be run on a single server. It has features of high performance such as indexes and parallel query processing over the machine cores. However, the requirements of big data applications go beyond vertical scalability.

PostgreSQL has a large number of extensions that are maintained by separate communities. This includes Citus[2], which mainly provides three components inside a PostgreSQL database: sharding, distributed query planning, and distributed query execution. It thus enables distributing PostgreSQL databases, allowing for horizontal scalability. It has been acquired by Microsoft in January 2019 to be a solution for PostgreSQL managed databases on Azure [3].

As both MobilityDB and Citus are extensions to PostgreSQL, not forks, they can work together. This paper explores the potential of their integration. The main questions we aim to answer are: (1) Is it possible to distribute spatiotemporal query processing by integrating the two extensions, and do we gain performance?, (2) What are the queries that can be distributed out of the box?, and (3) What would be needed to distribute the remaining queries?. This assessment is done analytically, by discussing the architecture of the two extensions, and empirically by running the BerlinMOD benchmark [6] of moving object databases on the integrated solution.

2 BACKGROUND

2.1 MobilityDB

MobilityDB [23] defines abstract data types (ADT) for representing moving objects data, such as `tgeompoint` to represent the evolution of a geometry point over time, e.g., a vehicle trajectory, and `tfloat` to represent the evolution in time of a floating point number, e.g., describing the speed of a vehicle. It basically defines this set of temporal type: {`tgeompoint`, `tgeogpoint`, `tint`, `tfloat`, `tbool`, and `ttext`}, which are respectively the temporal counterparts of these PostgreSQL and PostGIS base types:{`geometry(point)`, `geography(point)`, `int`, `float`, `bool`, and `text`}.

The abstract representation of such temporal types is a continuous mapping from time into the domain of the base type. Because computers cannot efficiently process continuous representations, the so called *sequence representation* is used [23]. A temporal type is constructed from a pair of a base type, and a time type. The time types in MobilityDB are { `timestamp`, `timestampset`, `period`,

[1] https://github.com/ULB-CoDE-WIT/MobilityDB
[2] https://www.citusdata.com/
[3] https://www.citusdata.com/product/hyperscale-citus/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
BigSpatial'19, November 5, 2019, Chicago, IL, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6966-4/19/11...$15.00
https://doi.org/10.1145/3356999.3365467

periodset }. Respectively, MobilityDB defines four type constructors: INSTANT, INSTANTS, SEQUENCE, SEQUENCES. A type constructor is a function that constructs data types. The INSTANT type constructor receives a base type, and constructs a temporal type that represents a single pair of a time instant and a value of the base type. For instance, the type INSTANT(geometry) represents a pair of a timestamp and a geometry, which can be used to represent the location and time of a car accident. Similarly, the INSTANTS type constructor receives a base type and constructs a set of such pairs with distinct time instants, e.g., representing the four-square check-ins of one user. Agnate to the time type period, the SEQUENCE type constructor constructs temporal types that represents continuous mapping between time instants in the period and values of the base type, such as the continuous trajectory of a car. In contrast to the types constructed by INSTANTS, the types constructed by SEQUENCE imply linear interpolation between the consecutive time instants. Finally the type constructor SEQUENCES constructs types that represent a set of such continuous mappings with non-overlapping and non-adjacent time periods. Figure 1 illustrates the types that are constructed by the four type constructors, where v@t denotes a value v of the base type occurring at the time instant t.

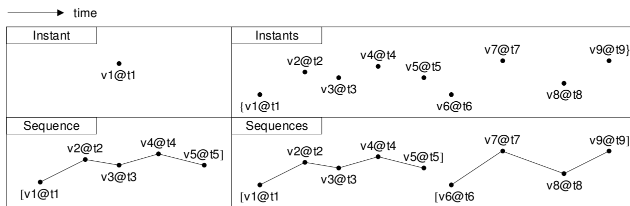


Figure 1: Sequence representation of moving objects

MobilityDB leverages the features of the PostgreSQL and PostGIS types. For instance, the timestamp type is time zone aware (a feature by PostgreSQL), while tgeompoint and tgeogpoint use the spatial framework provided by PostGIS. The same strategy is used for implementing the operations. The goal is to maximize the compatibility between MobilityDB and its underlying platform, so that it will benefit from the continuous development done by the community. This is one key enabler to the integration with Citus.

As illustrated in Figure 2, the aforementioned temporal types are supported by index access methods that extend on the PostgreSQL generalized search tree GiST, and the space partitioning search tree SP-GiST. Both indexes are implemented so that they support spatiotemporal, spatial only, and temporal only queries. They will share as many dimensions as available in the query argument. The b-tree index is also extended to support equality searches.

The query optimizer of MobilityDB collects statistics for temporal attributes, to use them in estimating the selectivity of the different query predicates, and selecting the optimal execution plan. The *type analyzer* is the function that collects the statistics. The general idea is to separately collect statistics for the base type, and for the time type that constitutes the temporal type in question. It hence reuses the PostgreSQL and PostGIS type analyze functions. Every selection predicate is then associated with a *selectivity estimation* function that uses the collected statistics to estimate the number of tuples that the predicate will return if used first.

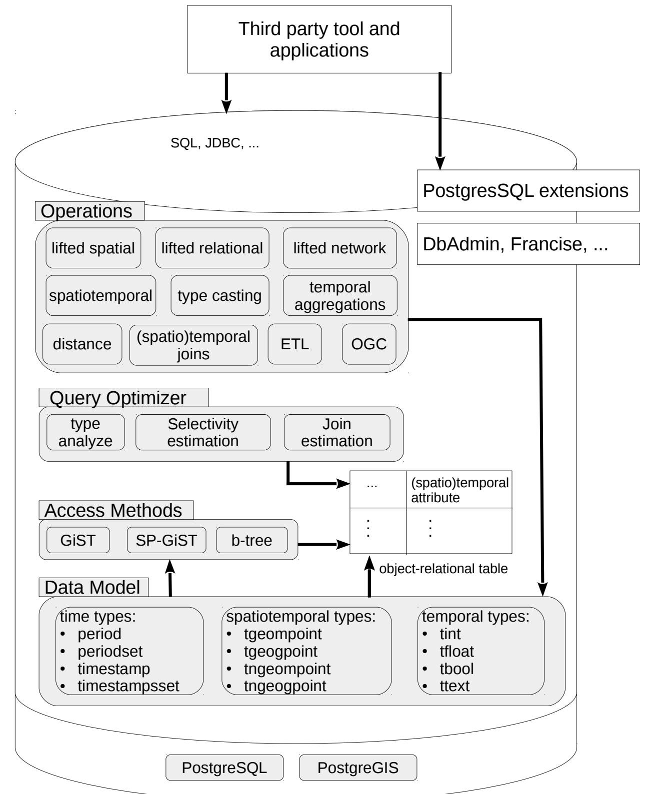


Figure 2: MobilityDB Architecture

Similarly, every join predicate is associated with a *join estimation* function to estimate the join cardinality.

MobilityDB defines over 2,300 query operations. Section 3 will list few of them. These operations are polymorphic, that is, their arguments may be of several types, and the result type depends on the types of the arguments. The arithmetic operations for instance have this signature:

$$+,-,* \quad \{\texttt{tfloat}, \texttt{tint}, \texttt{float}, \texttt{int}\} \times \{\texttt{tfloat}, \texttt{tint}, \texttt{float}, \texttt{int}\} \rightarrow \{\texttt{tfloat}, \texttt{tint}\}$$

These operations accept any combination of temporal and non-temporal numerical arguments, and thus the result is temporal. Clearly, the arithmetic of non-temporal types are excluded, as these are readily provided by the underlying DB system. Throughout the sequel, we will use this notation for defining the operation signatures, which starts in the left with the operation name, followed by the list of arguments, then the result type is placed in the right. When more than one temporal argument are accepted by an operation, the result is only defined on the *intersection* of their time spans. If the time spans are disjoint, then the result is null.

A major class of temporal operations is obtained by *lifting* the operations on non-temporal types, to also accept temporal types. This concept has been proposed in [11]. The *lifting* transformation can be illustrated as:

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

where *op* denotes a static operation, *LIFT(op)* denotes its lifted counterpart, and α, β are temporal arguments. Because the arguments of a lifted operation are temporal types, the result is also a temporal type. The notation $\alpha(t)$ denote the temporal functions of α, which yields its value at the given time instant. This equation reads as follows: sampling the result of a lifted operation at any time instant t yields the same value that one would get by sampling all the arguments at t and applying the corresponding static operation on the samples. For instance, the lifted predicate $\alpha > 0$, where α is a `tfloat` value, yields a `tbool` that is true over all the time instants/periods during which the value of α is positive, false during the rest of the definition time of α, and *undefined* otherwise.

In addition to *lifted operations*, MobilityDB defines other classes of operations including: ETL, aggregations, spatiotemporal, projection to space and to time, distance, etc. All its types and operations are available in SQL. It is compatible by default with the PostgreSQL extensions, because it leverages the PostgreSQL extensiblity features, rather than building from scratch.

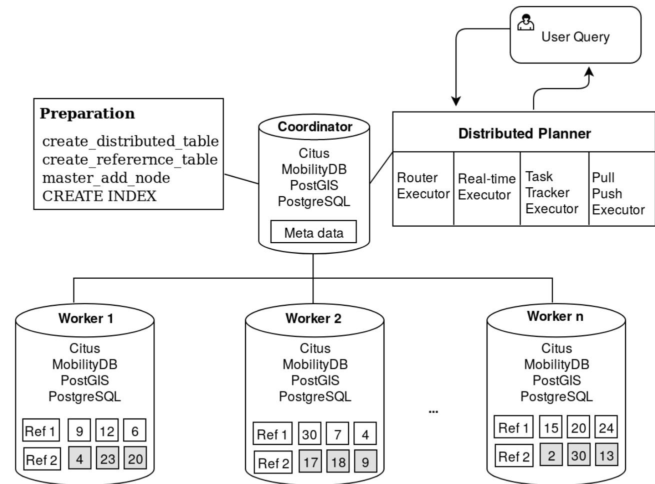


Figure 3: Distributed MobilityDB Cluster Architecture

2.2 Citus

Figure 3 illustrates the distribution architecture of MobilityDB using Citus. The hardware part consists of a controller node and multiple worker nodes. All the nodes have the same stack that consists of PostgreSQL, PostGIS, MobilityDB, and Citus. The distribution of the database is managed by Citus through the controller node. Building a cluster maps to iteratively adding worker nodes, i.e, `SELECT * from master_add_node(node-name, port)`. Citus distributes the work over the cores of the worker nodes. Additional worker nodes can be added during the run of the cluster without down time.

Big tables can then be sharded over the worker nodes using the `create_distributed_table` function, which expects a sharding key. The tuples are then routed by the controller to workers, where the data gets physically stored. The controller node maintains light weight metadata about the sharding so that it can distribute the queries, while the data storage and most of the query execution happen in the workers. Citus assigns shards to co-location groups, which ensure that rows with the same shard key are on the same worker node. It typically creates a number of shards that is more than the number of worker cores. Reference tables are replicated to all workers, using the function `create_reference_table`, such that they can be joined with distributed tables on any attribute. Citus also replicates the shards over workers to account for node failures (i.e., the gray blocks in Figure 3).

All queries issued to the cluster are executed via the coordinator. The coordinator generates a distributed query plan, where the user query gets partitioned into smaller query fragments that can be run independently on shards. The coordinator then assigns and monitors the execution on the workers. In case of a node failure, it re-assigns the execution to some replica. Finally, it merges their results, and returns the final result to the user.

The query processing engine in Citus consists of two components: the distributed query planner and multiple distributed query executors. PostgreSQL allows to extend the query planner via hooks, so that extension planners can be invoked before and after, in communication with the PostgreSQL planner. Citus uses this feature to implement its distributed query planner. It distinguishes four classes of queries: routable, push downable, recursive CTE, and complex queries.

Routable queries have a filter `sharding key = value`. Such queries are routed to the node that has the shard that corresponds to the value in the filter. Before routing, the planner rewrites the name of the distributed table in the FROM clause to reference the shard table instead of the original table. The router executor passes the rewritten query to the worker node, which then optimizes it using the regular PostgreSQL planner, executes it, and returns the result to the router executor.

Push downable queries are those that span multiple shards and use aggregates, GROUP BY, ORDER BY, and LIMIT, and are executed using the Citus real-time executor. The key feature that characterizes this class of queries is that they can be distributed in a single round (i.e., the coordinator pushes the query parts down to workers, then collects their results and produces the final result). Queries that contain complex joins and sub-queries may still fall into this class, as long as they join and group on the distribution column. The planner first creates a plan tree of the input query and transforms it into its commutative and associative form so it can be distributed. It also applies several optimizations, to push down filters and projections to the worker nodes, to ensure that network I/O is minimized. Similar to the MapReduce strategy, the planner next breaks the query into one part that runs at the coordinator (i.e., reduce), and one part that run on individual shards on the workers in parallel (i.e., map). The planner then assigns these query fragments (which are SQL queries) to the workers, and passes the control to the *Real-time executor*. The workers locally apply the PostgreSQL planner to optimize the execution of their fragments. Next, the workers return their results to the executor which triggers the query part of the coordinator to merge them, and produce the final result to the user. For instance the sum aggregate is commutative and associative, so the workers can perform partial sums over their shards, then the coordinator can sum these partial sums to produce the final result. The average aggregate is on the other hand, non associative, so it has to be distributed differently. Basically it is broken into a sum aggregate and a count aggregate.

Since both are commutative and associative, they are distributed, then the average is computed at the coordinator.

Other complex queries that require redistribution of data are handled by the *Task tracker executor*. This includes *Recursive CTE* queries that cannot be pushed down, and non-co-located joins. For recursive CTE, the planner is recursively called for the sub-queries. During the execution the coordinator pushes back the result of the sub-query to the worker nodes, and they get stored as intermediate results, which are used as reference tables in the evaluation of the main query.

Non-co-located joins are quite expensive as they involve a lot of network I/O for re-partitioning the data. The Citus planner rejects this class of queries by default. To activate it, an option needs to be set. The planner then applies relational algebra optimizations to reduce the number of iterations of re-partitioning, and to reduce the data size before re-partitioning. In our experiments, the queries that involved non-co-located joins always broke.

One query might be split into multiple parts, and different executors might be invoked for the parts. Citus chooses which to use depending on the structure of each query, and can use more than one at once for a single query, assigning different executors to different sub-queries/CTEs as needed to support the SQL functionality.

The two previous sections have described the out-of-the-box features of MobilityDB and Citus. In the next section, we describe their integration in order to provide big spatiotemporal data management. We assess the extent to which the API of MobilityDB can be distributed by Citus, and the effort that is required in the future to achieve a complete support.

3 DISTRIBUTED MOBILITYDB

As illustrated in Figure 3, managing a distributed database consists of two phases: the preparation phase, where the cluster is initialized and the distribution keys are defined, and the query phase. The preparation phase is agnostic to the attribute types in the distributed table. While distributing the data over shards, Citus treats MobilityDB types as binary objects. So moving the data from the coordinator to the workers (i.e., sharding) and among the workers (i.e., replication) is done in binary format. This is the mechanism used by Citus to cope with the type extensiblity of PostgreSQL. Insertions are routed by the coordinator to the shards in a way that guarantees co-location of shard keys and that tries to balance the partitioning over shards. Here is an INSERT statement (Line 1) and its execution plan generated using the PostgreSQL EXPLAIN command, then manually simplified for a better presentation (Lines 2-7).

```
1 INSERT INTO trips VALUES(...);
2 Custom Scan (Citus Router)
3   Task Count: 1
4   Tasks Shown: All
5   ->  Task
6         Node: host=pgxl3 port=5432 dbname=tripsddb
7         ->  Insert on trips_102041
```

Line 2 shows that the *Router executor* is in charge, mainly because the query hits a single worker node. Line 3 shows the number of parallel tasks, which is one in this case. Line 6 outputs the address of the worker that receives the task. Finally Line 7 outputs the task on the worker side. Notice that the table name has been rewritten into *trips_102014*, which is the name of the shard that receives the tuple.

In contrast to the preparation phase, answering spatiotemporal queries over a distributed schema might require a direct interaction between Citus and MobilityDB, which is currently not available. More specifically, the distributed query planners and executors of Citus do not understand the MobilityDB types and operations. Queries that require such an understanding cannot currently be distributed. Next we analyze this in depth, considering the Citus executor that handles the query, and the involved spatiotemporal operations.

The *Router executor* is involved in queries that can be fully evaluated on a single shard, i.e., query that contains a condition `shard key = value`. An example is given next:

```
1 SELECT * FROM trips WHERE carId= 100;
2 Custom Scan (Citus Router)
3   Task Count: 1
4   Tasks Shown: All
5   ->  Task
6         Node: host=pgxl4 port=5432 dbname=tripsddb
7         ->  Index Scan using
                trips33_carid_idx_carid_102031 on
                trips_102031 trips
8                Index Cond: (carid = 100)
```

This plan is similar to the previous one, because again it is handled by the Router executor. Citus planner used its metadata to locate the shard that contains the carId 100, which is *trips_102031* on worker node *pgxl4*. The shard key in this example is *carId*. It then rewrites the table name into the shard name *trips_102031*, and passes the modified query to the associated worker. At the worker node, which runs MobilityDB, the query gets optimized as if it is a local non-distributed query. Routable queries can hence use all the SQL features of MobilityDB. A common use case for this class of queries is in multi-tenant applications. For instance, a transportation company that has fleets in multiple cities would structure their database such that the city name is the shard key. In this way, the vehicle data of one city goes to one shard, and the queries that involve only one city will fall in the class of routable queries.

Excluding routable queries, all other classes of queries require that Citus performs a non-trivial split of the execution plan into coordinator part and workers part. This would require that Citus understands to some extent the operations in the query, so that it can decide an accurate splitting. PostgreSQL is however an extensible system, and Citus accounts for this. Moreover, MobilityDB operations are built using the extensibility features of PostgreSQL. Therefore, many of them can be distributed, despite the fact that Citus does not know their semantics. In the following we enumerate the classes of operations of MobilityDB, and assess whether the out-of-the-box Citus can distribute them.

3.1 Spatiotemporal Joins

MobilityDB has multiple predicates that can be used in joins. They can be classified in two types: predicates that yield *bool*, and *lifted predicates* that yield a temporal Boolean *tbool*. The latter can be quantified using the two operations *ever equals* (`&=`) and *always equals* (`@=`), to yield a *bool* value that can be accepted by SQL as a join condition. The signature of these two operations are as follows:

&=, @= TEMPORAL(S) × S → bool

They accept a temporal value and a value of its base type, and yield *bool*. One can use them to express a predicate such as *tintersects(trip, place) &= TRUE*, which computes the temporal intersection of a vehicle trip (`tgeompoint`) and a spatial region (`geometry`) in the form of a temporal boolean `tbool`, then checks whether it has some true values, yielding a `bool`. Another example is *(speed(trip) < 50) @= TRUE*, which checks whether the speed of the vehicle has always been less than 50 km/h. Thus these two operations allow *lifted predicates* to be used in joins. MobilityDB has a big number of *lifted predicates*, as it basically implements the temporal versions of all PostgreSQL and PostGIS operations that can be made time-dependant. This covers the relationships between a value of a spatiotemporal type and another value of a spatiotemporal, a spatial, or a base type. The signatures of some examples are given next:

tintersects	tgeompoint	×	geometry	→	tbool
	tgeogpoint	×	geography		
tdwithin	tgeompoint	×	geometry × float	→	tbool
	tgeogpoint	×	geography × float		
#=, #<>, #<, #>, #<=, #>=	{float, tint, float, int}	×	{tfloat, int, float, int}	→	tbool
speed	tgeompoint			→	tfloat

where `tintersects` checks the time dependent intersection between a temporal geometry/geography with a geometry/geography objects respectively. It is the lifted version of the `st_intersects` predicate of PostGIS, that check the intersections between a pair of geometries/geographies. Similarly `tdwithin` is the lifted version of the `st_dwithin` which checks whether two geometries/geographies are within the distance threshold given in the last argument. The temporal comparison operators #=, #<>, etc lift their corresponding comparison operators =, <>, etc. Finally the operator speed computes the time dependant speed of a trajectory. It is not a predicate, yet it is put here as an example of the operators that can be used to compose temporal boolean expressions.

Additionally, a number of Boolean predicates are available for temporal types. The *overlaps operator* && checks whether the bounding boxes of its two arguments have non-empty overlap, and returns a `bool`. The two arguments can be temporal geometry/geography, then the operator will compare their 3D spatiotemporal boxes. If one of the arguments is a non-temporal geometry/geography, then the operator compares the spatial bounding box of the temporal argument with the bounding box of the non-temporal argument. This operator is of specific interest, as it triggers the query optimizer to use the available spatiotemporal indexes. The other bounding box comparison operators (e.g., left, right, before, etc) are also available.

All the lifted predicates and the Boolean predicates described above can be used in join expressions that relate temporal attributes with temporal or non-temporal attributes. Citus does not know the semantic of these join operations. Its distributed query planner classifies such joins into only two classes: (1) co-located joins, and (2) non-co-located joins. In co-located, every workers can perform the join on its shards, independently from other workers, and the final result is simply the union of these results. It can then be executed in a single round using the *Real-time executor*. In other words, co-located joins do not require redistributing the data. One clear example is a join between the distributed table and a reference table. Since the reference table is replicated on all workers, it is guaranteed to be a co-located join, for example:

```
1  SELECT *
2  FROM trips t, regions r
3  WHERE intersects(t.trip, r.geom)
4
5  Custom Scan (Citus Real-Time)
6    Task Count: 32
7    Tasks Shown: One of 32
8    ->  Task
9      Node: host=pgxl2 port=5432 dbname=tripsdb
10     ->  Nested Loop
11       ->  Seq Scan on regions_102303 r
12       ->  Bitmap Heap Scan on trips_102239 t
13         Recheck Cond: (trip && r.geom)
14         Filter: _intersects(trip, r.geom)
15         ->  Bitmap Index Scan on trips_spgist_idx_102239
16           Index Cond: (trip && r.geom)
```

The query joins the *trips* and the *regions* relations. The schema will be given in Section 4. The join condition is that the trip trajectory spatially intersects the geometry. The *trips* relation is distributed, and *regions* is a reference relation. The join statement is not known to Citus. Yet, the distributed planner can decide that it is a co-located join, and assigns it to the *Real time executor*. The worker nodes locally optimize the queries, since they have MobilityDB. Therefore, it was possible in this query to invoke the distributed SP-GiST index of the spatiotemporal *trip* attribute (Line 15).

The non-co-located joins, on the other hand, require redistribution of the data, because data from one shard needs to be joined with other shards. An example would be a self join of the *trips* relation. For this kind of joins, Citus only supports the equi-joins. It rejects all the spatiotemporal predicates described above, mainly because it does not know how to redistribute the data, and it rejects to perform a complete cross product.

Enabling non-co-located spatiotemporal joins would require an extension that utilizes the semantic of the join operation, to do better than a full cross product. For instance, the proximity predicates such as overlaps, intersects, contains, etc, only needs to join only the shards whose spatiotemporal extents do not violate the proximity condition. To efficiently perform this, the relation needs to be distributed in a way that preserves the proximity [20], e.g., spatial grid, spatiotemporal grid, temporal partitioning, Hilbert curve partitioning, etc.

3.2 Temporal aggregations

MobilityDB implements the following temporal aggregation functions:

tcount	set(TEMPORAL)	→	tint
tmin, tmax, tsum, tavg	set(tfloat/tint)	→	tfloat/tint
tand, tor	set(tbool)	→	tbool
tcentroid	set(tgeompoint)	→	tgeompoint

Figure 4 illustrates the tsum(`tint`)→ `tint` aggregate. The top two `tint` values are input, and the bottom one is the aggregation output.

Citus has a white list of aggregate functions that it supports. Clearly, the temporal aggregates of MobilityDB are not in this white list, so they are rejected. This white list approach is too restrictive. It is possible to leverage the PostgreSQL parallel aggregation framework, and agnostically support a wide class of aggregates. PostgreSQL has a native support of parallel aggregation. To enable

it, the user aggregate operator needs to define a *combine function* and a *final function*. The *combine function* is run by every process participating in the parallel execution. It consumes the input tuples, and generates an intermediate result. These intermediate results are transferred to the leader process, which then runs the *final function* to generate the result.

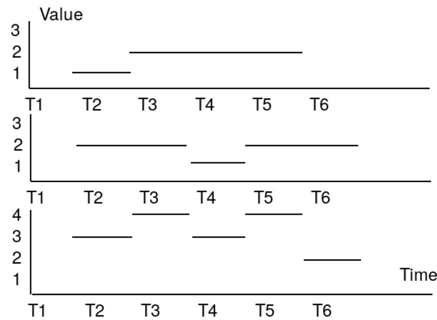


Figure 4: Temporal aggregation

Clearly this aggregation framework can be adapted for distributed processing, such that the *combine function* is run by the workers, and the *final function* is run by the coordinator node. Citus is unable to do this, because it distributes the SQL, rather than distributing the internal processing. The combine and the final functions are invisible to it because they are not exposed to SQL. A solution that uses the two functions will need to replace the executor of the parallel aggregations of PostgreSQL by a distributed executor. Such an extension would then be able to run the *combine function* on workers, rather than on the cores of the server machine.

Alternatively, distributing the SQL query would require that the aggregate function be both commutative and associative. To specify this, the `CREATE AGGREGATE` statement of PostgreSQL needs to be extended to allow the user to specify these two properties. For instance, all the temporal aggregates listed above, except `tavg` and `tcentroid`, are commutative and associative. It is correct, for instance, that the temporal sum of a set of temporal integers equals to the temporal sum of partial temporal sums of this set. Based on the verification of these properties, the aggregate can be partially calculated on the shards. The coordinator would then collect these partial aggregates, and run the same aggregation function on them. This is all achievable through SQL.

3.3 Spatiotemporal index access

The distribution of the Spatiotemporal GiST and SP-GiST is supported out-of-the-box. The CREATE INDEX statement on distributed tables gets pushed down to individual shards, because one shard is one PostgreSQL table. Moreover, workers independently optimize the query parts that they receive from the coordinator. This allows the invocation of these local indexes.

3.4 LIMIT

The ADT model that is implemented in MobilityDB encapsulates the complete trajectory inside an object, that can be stored in a single attribute. Using the LIMIT clause on tables that contain spatiotemporal attributes shall limit the number of tuples, thus the number of spatiotemporal objects that are retrieved. Limiting the number of instants inside the trajectories is not done using the LIMIT clause. In contrast to the ADT model, the temporal database model (e.g., [5]), would use the LIMIT clause to limit the number of instants. This semantic of limiting the instants inside the trajectories is implemented in MobilityDB using many functions, e.g.,:

`startInstant, endInstant`	`TEMPORAL`	→ `INSTANT(S)`
`instantN`	`TEMPORAL(S)` × `int`	→ `INSTANT(S)`
`instants`	`TEMPORAL(S)`	→ `ARRAY(S)`

These functions allow to access the individual instants inside the trajectory, choose a specific instant, or return all of them into an SQL array allowing for random access, splice, and ordering. These operations can be distributed, by pushing them down, because they process attributes, rather then tuples.

4 EXPERIMENTAL EVALUATION

This section provides a comprehensive experimental evaluation of the scalability of MobilityDB using Citus. All experiments are conducted on a two clusters of 4 and 28 nodes running PostgreSQL[4], PostGIS[5], Citus[6] and MobilityDB[7] over Ubuntu 18.04. The first cluster consists of 4 nodes, having the same specs: 4 cores per socket (2 sockets and 2 threads per core) Intel E5520 2.27 GHz processor, 2 TB of disk space and 24 GB RAM. The coordinator node of the second cluster is one of the first cluster nodes but with many worker nodes and low specs. Every worker node is equipped with an 4 cores (1 socket and 1 thread per core) Intel i5-4590 3.30 GHz processor, 20 GB of disk space and 8 GB RAM. The coordinator node works for management and partitioning, and the other nodes work for storage and processing.

4.1 Dataset and queries

BerlinMOD [6] is a benchmark for spatiotemporal data that is used for measuring the performance of queries on moving objects data. It contains a data generator that uses SECONDO [10] for generating trips of moving vehicles within Berlin. A scenario is simulated where a set of cars move within the road network of Berlin's city. It simulates drives to and from the work during the day time on workdays, as well as leisure trips in the evening and on the weekend. It generates datasets with different sizes that are configured with a parameter called scale factor. The generated schema is as follows:

```
cars <moid: int, licence: string, type: string, model:
     string>
trips: <moid: int, tripid: int, trip: tgeompoint>
regions: <id: int, region: geometry>
points: <id: int, pos: geometry>
periods: <id: int, p: period>
licences: <licence: string, id: int>
```

We generated data with scale factors from 1 until 19 as described in Table 1. The workload is constitute form the 17 BerlinMOD/R queries, that come with the benchmark. They cover range queries of four categories: object-based, temporal, spatial, and spatiotemporal.

[4]https://github.com/postgres/postgres
[5]https://github.com/postgis/postgis
[6]https://github.com/citusdata/citus
[7]https://github.com/ULB-CoDE-WIT/MobilityDB

Table 1: Datasets description

SF	Trips	Points	size (GB)	SF	Trips	Points	(GB)
1	300k	56M	5	3	870k	166M	14
5	1.4M	280M	24	7	2M	394M	34
9	2.6M	500M	43	11	3.2M	615M	53
13	3.8M	730M	63	15	4.4M	830M	72
17	5M	945M	82	19	5.6M	1000M	93

Queries Q.5, Q.6, Q.10,and Q.15 could not be distributed. The Citus planner rejects them with the following error message:
`ERROR: complex joins are only supported when all distributed tables are joined on their distribution columns with equal operator.`
The remaining queries, despite their complexity, reduce to push downable queries. We illustrate next one query as an example. The SQL of Q. 17 is as follows:

```
1 WITH pointCount AS (
2     SELECT p.pointId, COUNT(DISTINCT t.carId) AS visits
3     FROM trips t, points p
4     WHERE intersects( t.trip, p.pos )
5     GROUP BY p.pointId
6 )
7 SELECT pointId, visits
8 FROM pointCount AS p
9 WHERE p.visits = ( SELECT MAX(visits) FROM pointCount );
```

It consists of a CTE that aggregates the count of the vehicle that visited/intersected each of the query points. The intersection is evaluated using the temporal `intersects` predicate in Line 4. The main query, then, finds the points with the maximum number of visits, which is again a CTE. Citus breaks this query into three part, one part for every CTE, and one part for the main query. The plan of the query is given next:

```
1 Custom Scan (Citus Router)
2  ->  Distributed Subplan 54_1
3  ->  GroupAggregate, Group Key: remote_scan.pointid
4    ->  Sort, Sort Key: remote_scan.pointid
5      ->  Custom Scan (Citus Real-Time)
6        Task Count: 32
7        Tasks Shown: One of 32
8        ->  Task
9          Node: host=pgxl2 port=5432 dbname=sf21_0
10         ->  HashAggregate, Group Key: p.pointid, t.carid
11           ->  Nested Loop
12             ->  Seq Scan on points_102041 p
13             ->  Bitmap Heap Scan on trips_102008 t
14               Recheck Cond: (trip && p.geom)
15               Filter: _intersects(trip, p.geom)
16               ->  Bitmap Index Scan on
                       trips_spgist_idx_carid_102008
17                 Index Cond: (trip && p.geom)
```

This is the first sub plan, which executes the *pointCount* CTE. It is executed by the *Real-time* executor (Line 5), which distributes 32 copies of it over the 32 cores in the worker nodes. Each task is an aggregate query on one shard that counts for every combination of (*pointId, carId*) the number of intersections. The task shown here is on shard *trips_102008*, and there are 31 similar tasks of the other shards. The PostgreSQL planner on the worker decides for the shown task to use the SP-GiST index on the trips table to optimize the intersection predicate (Lines 16,17). The executor then orders one worker to aggregate these partial aggregates, and counts the total visits per point (Lines 3,4). The result of this aggregation gets stored in an intermediate result file. The following subplan reads this file, and evaluates the second CTE (`SELECT MAX(visits) FROM pointCount`) in Lines 1-8. The *Router executor* evaluates it on the worker *pgx12*, which maintains the intermediate result (Lines 7,8). The result of the aggregate (i.e., the maximum number of visits) gets again stored in an intermediate result file.

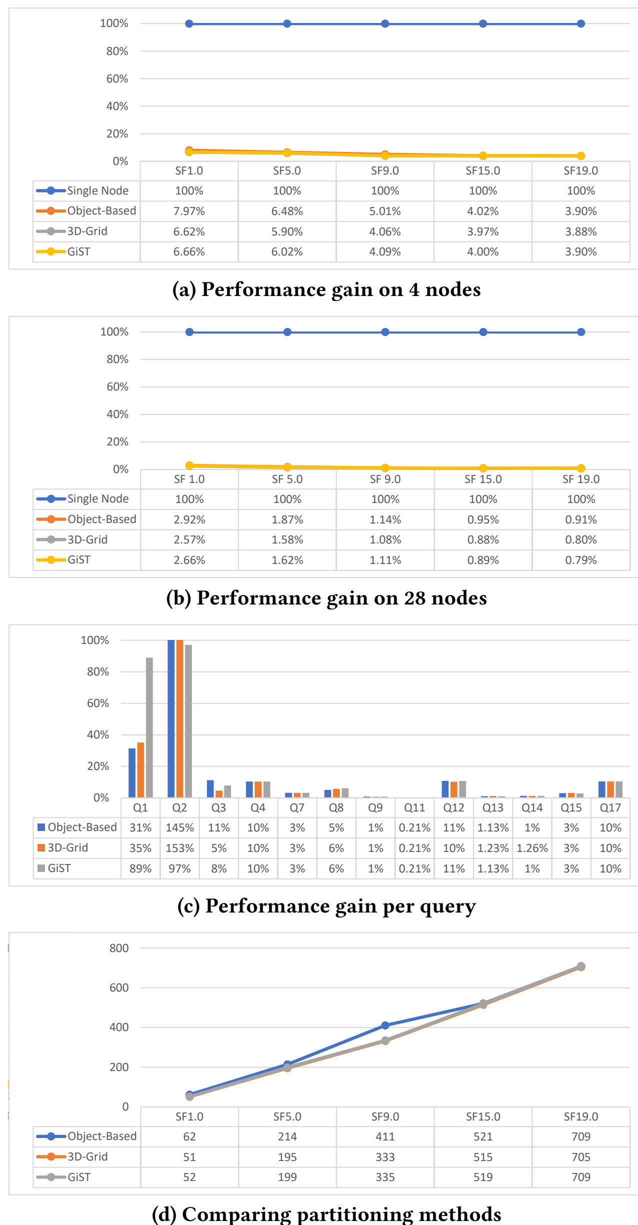
```
1 -> Distributed Subplan 54_2
2  -> Custom Scan (Citus Router)
3   Task Count: 1
4   Tasks Shown: All
5   -> Task
6    Node: host=pgxl2 port=5432 dbname=sf21_0
7    -> Aggregate
8     -> Function Scan on read_intermediate_result
          intermediate_result
9
10 Task Count: 1
11 Tasks Shown: All
12  -> Task
13   Node: host=pgxl3 port=5432 dbname=sf21_0
14   -> Function Scan on read_intermediate_result
         intermediate_result
15    Filter: (visits = $0)
16    InitPlan 1 (returns $0)
17      -> Function Scan on read_intermediate_result
            intermediate_result_1
```

Next, one task is assigned for the filter of the main query. It reads the intermediate result, and executes the filter. Finally the *Router executor* performs a scan to collect this result and returns it to the user.

4.2 Evaluation results

Every scale factor is loaded into a separate database. On the cluster setting, the distributed relation is *trips*. We experiment with three data partitioning methods: object based, 3D grid partitioning, and GiST partitioning. In object based partitioning, the shard key is *carId*. In 3D-Grid partitioning, a regular grid is created on the whole extent to the data. The grid cell size is defined based on the size of the table in KB and the page size of PostgreSQL. We divide the two values and take the cube root of the result, to be the number of cells in each of the three dimensions. Every trip gets assigned to the cell that contains its first most instant. This grid cell number is stored in an extended attribute, which is then used as the shard key of the relation. In GiST partitioning, we use the cluster operation of PostgreSQL to redistribute the data of the table using the GiST index. This operation exploits the index to store the trips that are close to each other together in the same table. Next, we assign a number to each row based on the new order of the data and use it as a sharding key.

Normally the partitioning method should not affect the query performance, because the distributable queries all belong to the push downable class, which is only affected by the distribution balance. Still, we want to confirm this analysis, and assess whether the distribution method affects other aspects such as the index efficiency. The chosen partitioning methods are examples of different partitioning methods: object-based partitioning, spatiotemporal space and data partitioning. The remaining tables are replicated in



(a) Performance gain on 4 nodes

(b) Performance gain on 28 nodes

(c) Performance gain per query

(d) Comparing partitioning methods

Figure 5: Experimental results

all workers (in the cluster setting) as reference tables. We measure the query performance of the 13 distributable queries on the all the scale factors. Every query is run 5 times and the average response time is calculated.

Figure 5a illustrates the speedup gained on the 4 nodes cluster (one coordinator and three workers) versus a single machine of the same configuration as any of the cluster nodes. The figure is normalized by the runtime of the single node. For instance, in SF1.0, the distributed query using GiST partitioning costs 6.66% of the run time of the same query on a single node. The gain is significant, as the runtime on such a small cluster is between 3.8% and 8% of the single node runtime. The gain slightly increases with the increase of data size, because the single node gets more congested. These results are confirmed on the 28 nodes cluster, Figure 5b. It shows more gain than the smaller cluster, but not proportional to the increase in the cluster size, mainly because the data is not big for such a cluster size. Figure 5c shows the gain per query. Except for queries Q1, Q2 all other queries have similar gain. This is mainly because all of them fall into the class of co-located joins, as discussed in Section 3. The exception of Q1, Q2 is of no significance, because the run times of these queries on the single node as well as on the cluster is in terms of milliseconds. So this exception can be contributed to other minor factors such as the networking cost. The three figures do not show significant differences between the data partitioning methods. The 3D-Grid is slightly better than the other two methods, but with a small margin. This is further confirmed in Figure 5d, which shows the total runtime of all queries in seconds for every configuration.

5 RELATED WORK

This work is part of the MobilityDB project, which has the goal of filling the gap between moving object database research and practice. On the one hand, there is a long standing research on moving object databases, dating back to early 2000s, with quite mature results. There are also few system prototypes [8, 10, 18]. On the other hand, there is no industrial-scale MOD available. Therefore, these research results are not accessible to end users. MobilityDB is engineered up from PostgreSQL and PostGIS, providing spatiotemporal data management in SQL. It implements the state of art, developed by the research community, into a widely used open source DBMS. While the core features have been developed by the MobilityDB team, it has recently been released as open source [8], with detailed documentation [9], to encourage community contributions. So it can be seen as a platform for implementing MOD features and research results.

As aforementioned, there are few MOD prototypes in the literature; namely: HERMES [18], SECONDO [10], DEDALE [8], and DOMINO [21]. SECONDO[10] and HERMES [18] follow the ADT approach of moving object databases, which is also followed in MobilityDB. HERMES can be run on top of Oracle and PostGIS, hence accessible in SQL. It did not however exploit their type system. For instance, for defining the moving point type, it builds on a custom-defined x, y pair, rather than using the PostGIS/Oracle point type. Accordingly, the functions provided by the underlying DBMS are not reused. The spatiotemporal operations in HERMES are evaluated over the bounding boxes of the objects, rather than their exact coordinates.

SECONDO [10] is an extensive implementation of moving object types and operations. In contrast to HERMES and DEDALE, it is still an active project, with recent releases. SECONDO reuses the file systems of Berkeley DB and Cassandra. The remaining functions otherwise are implemented from scratch. It consists of three modules: the kernel, the optimizer, and the GUI. The kernel has the query processor, and the Algebras. Every Algebra defines database types and operators. For instance, the Temporal Algebra defines

[8] https://github.com/ULB-CoDE-WIT/MobilityDB
[9] https://docs.mobilitydb.com/nightly/

the types and operations of moving objects in [11]. SECONDO also implements indexes such as RTree, TBTree, MMRTree, and MONTree. The kernel can be queried using a procedural language called *SECONDO executable language*. The optimizer module accepts a syntax similar to SQL, and generates plan in the *SECONDO executable language*. Not all SECONDO operations are available in this SQL-like language. Finally, the GUI module offers an interface for visualizing the moving objects in a movie style.

DEDALE [8], and DOMINO [21] are a proof of concept implementations, rather than systems. Both are not anymore maintained. DEDALE implements a constraint database model for spatiotemporal data management [9], where a spatiotemporal trajectory is represented using a set of linear constraints. DOMINO (Database for Moving Object tracking) implements the concept of *dynamic attributes* to model the current location of a moving object that is being tracked. It stores the last observed location (as fact), and the speed of the object (for prediction). Location updates are pushed by the moving object, in a way that balances the update cost, the deviation cost, and the uncertainty cost. On top of this model, DOMINO answers probabilistic range queries. For big spatiotemporal data management, there are two main lines of work: object-relational databases, and NoSQL systems.

5.1 Object-Relational Databases

Up to our knowledge, the system implementations in this category are all SECONDO extensions: Parallel SECONDO [12], Distributed SECONDO [16], and SECONDO Distributed2 Algebra [3]. Parallel SECONDO [12] is a scalable version of SECONDO that uses Hadoop as a communication manager for scheduling and managing the tasks between the cluster nodes. Every node in the cluster runs a regular SECONDO instance and contains a full copy of the data. One node is playing the role of a master. The coming query is partitioned into small parallel queries in the master node and Hadoop distributes the parallel queries over the worker nodes to be executed by every SECONDO instance. Then, Hadoop sends the results back to the master node to be aggregated. 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 [16] follows a similar concept, yet without using Hadoop. Cassandra is used as a storage layer, where data is split into small units that are assigned to query processing nodes. The query processing is done using standard SECONDO. The cluster consists of three types of nodes: management nodes (MNs), storage nodes (SNs), and query processing nodes (QPNs). Both MNs and QPNs are running instances of SECONDO. SNs run Cassandra and store the data. QPNs also run a helper tool for executing the queries, called QueryExecutor. It determines which part of the data need to be processed by the local SECONDO installation. Reading and writing operations to and from the SNs are done by MNs and QPNs. MNs are used to import and export the data into and from Distributed SECONDO. Query processing is done by SECONDO nodes where the QueryExecutor is the link between the SNs and the QPNs. QPNs request the data from SNs to execute the specified query on the data chunks and write the result back to the SNs.

The SECONDO Distributed2 Algebra [3] allows the user to explicitly distribute the data and the processing in the query. Again the cluster is built of regular SECONDO nodes, one of which is marked as the master. Mainly two types are defined: *darray* distributed array, and *dfarray* distributed file array. The later is for distributing a SECONDO relation, while the former is for distributing any collection of SECONDO objects. The definition of the arrays explicitly stores how the cells are mapped to the SECONDO nodes. The user query, which is written in the *SECONDO executable language*, consists of three parts, that use the operations of the Algebra. The *distribute* part explicitly defines how the data is sharded over the array slots, hence over the worker nodes. The *map* part defines a function, in the form of a SECONDO query, that every worker node will execute on the array slots assigned to it. Finally the *collect* part defines how the master collects back the results of the workers. The *map* part, being a SECONDO query, is very flexible. It can process the local array slot, join different slots, or even redistribute the data over the workers. The SECONDO Distributed2 Algebra is thus a powerful tool for experimenting with data and processing distribution methods.

Alas, we could not compare MobilityDB results with these three extensions for technical reasons. Parallel SECONDO and Distributed SECONDO are not maintained with the recent SECONDO and Hadoop versions. While Distributed2 Algebra requires that the queries are expressed using the Algebra operations, that we do not master. We would like to do so in the future, though. We learn a lot from these three works, and exploit them further in the context of MobilityDB. The added value is that these distribution methods will be accessible to end users in PostgreSQL.

5.2 NoSQL Systems

STARK [13] provides a platform for analyzing big spatiotemporal data by adding different modes of indexing and spatial partitioners on the top of the Apache Spark's core. The representation of the spatial object and the index structure are done by using JTS Java library. The spatial operators are implemented inside the SpatialRDDFunction class, that are join and kNN. SpatialRDDFunction is a spatial extension of the original RDD. SpatialRDDFunction partitions the data based on one of these types: spatial GridPartitioner, BinarySpacePartitioner, and PartitioningPolygons. GridPartitioner divides the space into grid of equal cell size. BinarySpacePartitioner divides the space into grid based on the maximum cost for every partition and it is better than GridPartitioner because there are no empty partitions in this approach. The last partitioning approach is PartitioningPolygons, every polygon only exists in one partition based on it's centroid. After partitioning the spatiotemporal data, the index can be built using any in-memory spatial index structure that are provides by JTS library such as R-tree.

Summit [1] is a spatiotemporal data management extension of SpatialHadoop [7]. It supports two levels of partitioning: temporal-based and spatial-based. The temporal-based partitioning is based on equi-width or equi-depth. For every temporal partition, the spatial data is partitioned using spatial-based or segmentational-based. Spatial-based technique supports range and join queries while segmentational-based technique supports similarity kNN queries. Summit does not support the notion of trajectory because

it is only limited to process a set of discrete temporal points. Also, it can not express the trajectory operations such as speed and intersects. Summit provides two types: STPoint to represent spatiotemporal point and trajectory to link the STPoints that are related to the same moving object.

TrajSpark [22] is a trajectory data management framework based on Apache Spark. It provides RDD extensions for managing trajectory segments, called TRDD and IndexTRDD. TRDD is used to scan the whole partitions without filtering while IndexTRDD has the ability to filter the partitions by incorporating a global and local indexing strategy. Partitioning the data is done using three constraints: Data locality, load balancing (fixed partition size), and STPartitioner which contains a spatial quad-tree or kd-tree index. STPartitioner uses the bounding box of the segments to partition points, then the trajectory points located in the same bounding box are grouped together. A global index is built over the partitions to filter them from the beginning of the query and bring only the relevant partitions. TrajSpark supports three kinds of queries: (1) (Single Object)-based query, (2) (Spatiotemporal range)-based query, and (3) KNN-based query.

These systems do not represent trajectories as first class citizens. HadoopTrajectory [2] tries to fill in this gap by extending Hadoop with moving object types and operations. HDFS is extended by readers and splitters that understand the new types. Similarly the Hadoop MapReduce is extended with trajectory processing operations. User can access these types and operations in their MapReduce programs.

6 CONCLUSIONS

In order to address the increasingly large-scale moving object query processing, we proposed an integration between two PostgreSQL extensions MobilityDB for managing moving object data, and Citus for distributing the query processing across a cluster of nodes. The integrated solution could distribute most of the spatiotemporal operations of MobilityDB. It was not possible to distribute temporal aggregations because Citus hard-codes a white list of aggregations that it supports. It was also not possible to distribute the non-co-located spatiotemporal joins, because these require that the distributed query planner knows about the spatiotemporal extents of the shards, and about the semantic of the join expression. These two classes of operations would require further extensions, to support their distribution. Ideas for such extensions have been discussed.

The experiments were done using the BerlinMOD benchmark of moving object databases. The results show multiple order of magnitude gain in the performance of the distributed queries, over single node queries. Thirteen out of the seventeen BerlinMOD queries could be distributed out-of-the-box. We comment, though, that BerlinMOD query is not designated to distributed MOD. Almost all the fourteen queries fall in the class of push-downable queries. A specific benchmark is hence required to assess the performance of the different classes of distributed MOD queries.

ACKNOWLEDGMENTS

This work is supported by the MobiPulse project, funded by Innoviris, Brussels.

REFERENCES

[1] Louai Alarabi. 2019. Summit: A Scalable System for Massive Trajectory Data Management. *SIGSPATIAL Special* 10, 3 (jan 2019), 2–3.
[2] 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.
[3] Thomas Behr and Ralf Güting. 2016. *Distributed Query Processing in Secondo*. Technical Report. FernUniversity in Hagen. http://dna.fernuni-hagen.de/Secondo.html/files/Documentation/General/DistributedQueryProcessinginSecondo.pdf
[4] Lucio Bianco and Maurizio Bielli. 1992. Air traffic management: Optimization models and algorithms. *Journal of Advanced Transportation* 26, 2 (1992), 131–167. https://doi.org/10.1002/atr.5670260205
[5] Michael Böhlen, Johann Gamper, and Christian S. Jensen. 2006. Multi-dimensional Aggregation for Temporal Data. In *Proceedings of the 10th International Conference on Advances in Database Technology (EDBT'06)*. Springer-Verlag, Berlin, Heidelberg, 257–275. https://doi.org/10.1007/11687238_18
[6] Christian Düntgen, Thomas Behr, and Ralf Güting. 2009. BerlinMOD: a benchmark for moving object databases. *The VLDB Journal* 18, 6 (2009), 1335–1368. https://doi.org/10.1007/s00778-009-0142-5
[7] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce Framework for Spatial Data. In *Proc. of the 31st IEEE International Conference on Data Engineering, ICDE 2015*. Seoul, South Korea, 1352–1363.
[8] 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-Verlag, London, UK, 38–59.
[9] Stéphane Grumbach, Philippe Rigaux, and Luc Segoufin. 2001. Spatio-Temporal Data Handling with Constraints. *GeoInformatica* 5, 1 (01 Mar 2001), 95–115. https://doi.org/10.1023/A:1011464022461
[10] Ralf Güting, Thomas Behr, and Christian Düntgen. 2010. SECONDO: A Platform for Moving Objects Database Research and for Publishing and Integrating Research Implementations. *IEEE Data Eng. Bull.* 33 (06 2010), 56–63.
[11] Ralf 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.
[12] Ralf Güting and Jiamin Lu. 2015. Parallel SECONDO: Scalable Query Processing in the Cloud for Non-standard Applications. *SIGSPATIAL Special* 6, 2 (2015).
[13] Stefan Hagedorn, Philipp Goïze, and Kai-Uwe Sattler. 2017. The STARK Framework for Spatio-Temporal Data Analytics on Spark. In *Datenbanksysteme fúr Business, Technologie und Web, BTW 2017*. Gesellschaft fúr Informatik, Bonn.
[14] Rida Khatoun and Sherali Zeadally. 2016. Smart Cities: Concepts, Architectures, Research Opportunities. *Commun. ACM* 59, 8 (July 2016), 46–57.
[15] Zhenhui Li, Jiawei Han, Ming Ji, Lu-An Tang, Yintao Yu, Bolin Ding, Jae-Gil Lee, and Roland Kays. 2011. MoveMine: Mining Moving Object Data for Discovery of Animal Movement Patterns. *ACM Trans. Intell. Syst. Technol.* 2, 4, Article 37 (July 2011), 32 pages. https://doi.org/10.1145/1989734.1989741
[16] Jan Kristof Nidzwetzki and Ralf Güting. 2015. Distributed SECONDO: A Highly Available and Scalable System for Spatial Data Processing. In *Proc. of the International Symposium on Spatial and Temporal Databases, SSTD 2015*. Springer.
[17] Bei Pan, Yu Zheng, David Wilkie, and Cyrus Shahabi. 2013. Crowd Sensing of Traffic Anomalies Based on Human Mobility and Social Media. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'13)*. ACM, New York, NY, USA, 344–353. https://doi.org/10.1145/2525314.2525343
[18] Nikos Pelekis, Elias Frentzos, Nikos Giatrakos, and Yannis Theodoridis. 2015. HERMES: A Trajectory DB Engine for Mobility-Centric Applications. *Int. J. Knowledge-Based Organ.* 5, 2 (April 2015), 19–41. https://doi.org/10.4018/ijkbo.2015040102
[19] Michael Steinbach, Pang-Ning Tan, Vipin Kumar, Steven Klooster, Christopher Potter, and Alicia Torregrosa. 2001. Clustering Earth Science Data: Goals, Issues and Results. (2001).
[20] Randall T. Whitman, Bryan G. Marsh, Michael B. Park, and Erik G. Hoel. 2019. Distributed Spatial and Spatio-Temporal Join on Apache Spark. *ACM Trans. Spatial Algorithms Syst.* 5, 1, Article 6 (June 2019), 28 pages. https://doi.org/10.1145/3325135
[21] Ouri Wolfson, Liqin Jiang, A. Prasad Sistla, Minglin Deng, Sam Chamberlain, and Naphtali Rishe. 1999. Databases for Tracking Mobile Units in Real Time. In *Database Theory — ICDT'99*, Catriel Beeri and Peter Buneman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–186.
[22] Zhigang Zhang, Cheqing Jin, Jiali Mao, Xiaolin Yang, and Aoying Zhou. 2017. TrajSpark: A Scalable and Efficient In-Memory Management System for Big Trajectory Data. In *Proc. of the APWeb-WAIM Joint Conference on Web and Big Data*. Springer, 11–26.
[23] Esteban Zimányi, Mahmoud Sakr, Arthur Lesuisse, and Mohamed Bakli. 2019. MobilityDB: A Mainstream Moving Object Database System. *Proc. of the 16th International Symposium on Spatial and Temporal Databases, SSTD 2019* (2019).